Small Bisimulations for Reasoning About Higher-Order Imperative Programs

Vasileios Koutavas
Northeastern University
vkoutav@ccs.neu.edu

Mitchell Wand
Northeastern University
wand@ccs.neu.edu

Abstract

We introduce a new notion of bisimulation for showing contextual equivalence of expressions in an untyped lambda-calculus with an explicit store, and in which all expressed values, including higher-order values, are storable. Our notion of bisimulation leads to smaller and more tractable relations than does the method of Sumii and Pierce [31]. In particular, our method allows one to write down a bisimulation relation directly in cases where [31] requires an inductive specification, and where the principle of local invariants [22] is inapplicable. Our method can also express examples with higher-order functions, in contrast with the most widely known previous methods [4, 22, 32] which are limited in their ability to deal with such examples. The bisimulation conditions are derived by manually extracting proof obligations from a hypothetical direct proof of contextual equivalence.

Categories and Subject Descriptors F.3.2 [*Logic and Meanings of Programs*]: Semantics of Programming Languages—operational semantics; D.3.3 [*Programming Languages*]: Language Constructs and Features—procedures, functions and subroutines; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—semantics

General Terms theory, languages

Keywords contextual equivalence, bisimulations, lambda-calculus, higher-order procedures, imperative languages

1. Introduction

In the classic notion of contextual equivalence, two expressions e and e' are *contextually equivalent* if and only if $C[e]$ and $C[e']$ produce the same answer for any program context C. Contextual equivalence is the "gold standard" of equivalences. If e and e' are contextually equivalent, then a compiler or other program optimizer can always safely replace e with e' or vice versa, wherever it appears in any program. The idea of contextual equivalence goes back to Morris [21], and is a key notion in much of denotational semantics, e.g. [2, 17, 19, 23, 24].

Proving the contextual equivalence of expressions is generally difficult, since it involves reasoning about all possible contexts. One way of easing this burden is to find a way to cut down the set of contexts that need to be considered (e.g. [16]).

POPL'06 January 11–13, 2006, Charleston, South Carolina, USA.

Another approach is to use a denotational semantics (*e.g.*, [18]). In a denotational semantics, two programs with the same denotation are guaranteed to be contextually equivalent. Unfortunately, denotational equality is a very strong property. Two expressions that manipulate the store differently will generally have different denotations. For example, consider

```
class Cell {
  private int y;
  Cell (int x) {y = x;}
  public void set (int z) {y = z;}
  public int get () {return y;}
}
```

and

```
class Cell {
  private int y1, y2, p;
  Cell (int x) {p = 0; y1 = x; y2 = x;}
  public void set (int z) {p = p+1; y1 = z; y2 = z;}
  public int get () {if ((p % 2) == 0)
                       {return y1;}
                       {return y2;}}
}
```

Here we have two different implementations of a cell containing an integer. The first keeps a single instance variable `y`, but the second keeps three instance variables: a counter `p` and two copies of the cell, `y1` and `y2`. It reads from either one copy or the other, depending on how many times the `set` method has been executed.

These fragments are indistinguishable (at least in the absence of reflection), but they will have different denotations because they allocate different numbers of locations in the store. There are more complex denotational approaches that attempt to deal with this kind of situation (*e.g.*, [29]), but this example demonstrates that a straightforward denotational approach is insufficient for the kind of examples we wish to handle.

A more promising technique is that of *bisimulation*. In the bisimulation approach, one defines a relation R between pairs of configurations, such that

1. If two configurations are related by R, and each takes a step (or perhaps a sequence of steps), then the resulting configurations are again related by R.
2. If two final configurations are related by R, then they represent the same answer.

For any such R, if two configurations are related by R, then they will produce the same final answer.

This approach seems more in keeping with how programmers think. In the example above, we can imagine the sequence of method calls on a `Cell` object. At each step, we can imagine the contents of the instance variable `y` on one hand, and the instance

variables `y1`, `y2`, and `p` on the other. It's clear that corresponding calls to `get` will always return the same value. The relation R is a kind of invariant that connects the two computations.

Bisimulation techniques are widely used for process calculi [20]. They are also an important technique in analysis-based program transformation [8, 34, 35].

Although bisimulations seem to capture a programmer's intuition, they have two significant shortcomings:

1. First, as described above, bisimulations are formulated in terms of small-step (or rewriting) semantics. This formulation makes it difficult to reason directly about procedures, especially recursive procedures, for which a big-step (or evaluation) semantics is more natural.[1] In previous work, we have shown how to adapt bisimulation ideas to handle procedures using a big-step semantics [30, 35].
2. More importantly, bisimulation is a *whole-program* property. It is concerned with entire program configurations; it does not talk directly about program fragments. The key step, therefore, is to formulate bisimulation as a term relation and show that it is a congruence. There are many such results for process calculi [9].

In the sequential realm, Sumii and Pierce [31] have shown how to define bisimulations for a higher-order language with dynamic sealing. Sealing is of interest because generating a new seal is a primitive effect. They replaced a single bisimulation with a set of partial bisimulations, each representing a piece of a bisimulation under different conditions of knowledge. Similar ideas have also been used in concurrency [6].

In this paper, we extend and improve the Sumii-Pierce method. This work makes four contributions:

- We introduce a new notion of bisimulation that leads to smaller and more tractable relations. In particular, our method allows one to write down the bisimulation relations directly in cases where [31] requires an inductive specification, and where the principle of local invariants [22] is inapplicable.
- We handle a calculus with full-fledged store that may contain any expressed value.
- Our method can express examples with higher-order functions. The most widely known previous methods [4, 22, 32] are limited in their ability to deal with examples containing higher-order functions.
- Last, we derive our bisimulation by a more structured method than [31], which relied more on intuition for its derivation. This method allows us to identify weaker proof obligations, and sugests the possibility of systematically defining notions of bisimulation for other languages.

The remainder of the paper is organized as follows: in Section 2, we present the syntax and operational semantics of the language we will use. In Section 3, we define contextual equivalence. In Section 4 we consider how one might go about proving contextual equivalence directly. From this proof we extract proof obligations that serve to define the requirements for a bisimulation. In Section 5 we define our notion of bisimulation, and state its correctness. The proofs are deferred to Appendix A. In Section 6 we present some well-known examples using this method. In Section 7 we compare the proof obligations generated by our method with those of [31]. We conclude with a discussion of related work and conclusions.

2. The language

The term language is the untyped, call-by-value lambda calculus augmented with constants, primitive operations, a conditional construct, tuples, and store operations. The syntax of the language is shown in Figure 1.

The meta-variable c ranges over the set of natural numbers and booleans; op ranges over the arithmetic and boolean operators. Tuple construction is done by the language constructor $(e_1, \ldots, e_n)$, and projection of the i-th element of a tuple by $\#_i(e)$. Constants, abstractions, tuples of values, and locations are values.

The expression $\nu x.\, e$ creates a new location in the store, and binds x to that location inside e; expressions $!e$ and $(e := e)$ return and update, respectively, the contents of a location in the store.

The domain of locations is an infinite countable set; meta-variables l, k range over this set. A *store* is a finite map from locations to closed values; s and t range over stores. The value in location l of store s is given by $s(l)$; $s[l \mapsto v]$ places the value v in location l of store s, extending the domain of s, if necessary. $Dom(s)$ denotes the domain of store s; $Locs(e)$ denotes the set of locations that occur inside e. $Lclosed(s)$ is the predicate:

$$\forall l \in Dom(s).\ (FV(s(l)) = \emptyset) \wedge (Locs(s(l)) \subseteq Dom(s))$$

Definition 2.1. *A* configuration *is a pair of a store s and expression e, written $\langle s,\, e\rangle$.*

Definition 2.2. *A configuration $\langle s,\, e\rangle$ is called* well-formed *iff it satisfies the following properties:*

$$Lclosed(s) \wedge (FV(e) = \emptyset) \wedge (Locs(e) \subseteq Dom(s))$$

The big-step operational semantics of the language is defined for well-formed configurations, and is shown in Figure 2. Each big-step reduction $\langle s,\, e\rangle \Downarrow \langle t,\, w\rangle$ consists of an initial well-formed configuration that contains an expression e and the store s, under which e evaluates, and a final configuration that contains the reduced value w and the final store t. If the derivation tree defined by a big-step evaluation $\langle s,\, e\rangle \Downarrow \langle t,\, v\rangle$ has height less than k then we write $\langle s,\, e\rangle \Downarrow^{<k} \langle t,\, v\rangle$.

Lemma 2.3. *If $\langle s,\, e\rangle \Downarrow \langle t,\, w\rangle$ then $Dom(s) \subseteq Dom(t)$.*

Proof. By straightforward induction on the derivation of $\langle s,\, e\rangle \Downarrow \langle t,\, w\rangle$. □

Lemma 2.4. *If $\langle s, e\rangle$ is a well-formed configuration and $\langle s,\, e\rangle \Downarrow \langle t,\, w\rangle$, then $\langle t,\, w\rangle$ is also well-formed.*

Proof. By straightforward induction on the derivation of $\langle s,\, e\rangle \Downarrow \langle t,\, w\rangle$, using Lemma 2.3, and the definition of well-formed configuration. □

We say that a well-formed configuration $\langle s,\, e\rangle$ *terminates*, and we write $\langle s,\, e\rangle \Downarrow$, iff there is a finite derivation tree from the rules in Figure 2 that has $\langle s,\, e\rangle$ as initial configuration, and some $\langle t,\, w\rangle$ as final configuration. Conversely, if there is no such derivation, we say that $\langle s,\, e\rangle$ *diverges*, and we write $\langle s,\, e\rangle \Uparrow$.

In the following sections we will use $\texttt{let}\, x = e_1\, \texttt{in}\, e_2$ and $(e_1; e_2)$ as syntactic sugar for $((\lambda x.e_2)\ e_1)$, depending on whether x is free in e_2.

We will also use an overbar as a syntactic meta-operator to denote a comma-separated sequence of syntax fragments:

$$\overline{\sigma} = \sigma_1, \ldots, \sigma_n$$

where σ is a syntax fragment and σ_i is the same fragment with i-subscripts on all meta-identifiers it contains. For example we will write $[\overline{u/x}]e$ instead of $[u_1/x_1, \ldots, u_n/x_n]e$, and $\overline{(v, v')} \in R$ instead of $(v_1, v'_1), \ldots, (v_n, v'_n) \in R$. For the purposes of this paper the size of a sequence in the overbar notation is considered arbitrary, and left implicit (see [15] for a more rigorous treatment).

[1] For an example of the complications caused by this mismatch, see [34].

$$
\begin{array}{rrcll}
\textsc{Expressions}: & d, e & ::= & x & \text{Identifiers} \\
& & | & c & \text{Constants} \\
& & | & \lambda x.e & \text{Abstraction} \\
& & | & (e\ e) & \text{Application} \\
& & | & op(e, \ldots, e) & \text{Primitive Operations} \\
& & | & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e & \text{Conditional} \\
& & | & (e, \ldots, e) & \text{Tuple Construction} \\
& & | & \#_i(e) & \text{Projection} \\
& & | & l & \text{Locations} \\
& & | & \nu x.e & \text{New Location} \\
& & | & !e & \text{Dereferencing} \\
& & | & (e := e) & \text{Assignment} \\
\textsc{Values}: & u, v, w & ::= & c \mid \lambda x.e \mid (v, \ldots, v) \mid l & \\
\textsc{Locations}: & l, k & & & \\
\textsc{Stores}: & s, t & \in & \textsc{Locations} \xrightarrow{\text{fin}} \textsc{Values} & \\
\textsc{Configurations}: & \langle s, e\rangle & \in & \textsc{Stores} \times \textsc{Expressions} &
\end{array}
$$

Figure 1. Syntactic Domains

$$
\textsc{Eval-Const}\ \ \langle s, c\rangle \Downarrow \langle s, c\rangle \qquad \textsc{Eval-Abstr}\ \ \langle s, \lambda x.e\rangle \Downarrow \langle s, \lambda x.e\rangle
$$

$$
\textsc{Eval-App}\ \ \frac{\langle s, e_1\rangle \Downarrow \langle s_1, \lambda x.e\rangle \quad \langle s_1, e_2\rangle \Downarrow \langle s_2, v\rangle \quad \langle s_2, [v/x]e\rangle \Downarrow \langle t, w\rangle}{\langle s, (e_1\ e_2)\rangle \Downarrow \langle t, w\rangle}
$$

$$
\textsc{Eval-Op}\ \ \frac{\langle s, e_1\rangle \Downarrow \langle s_1, c_1\rangle \ \ \ldots \ \ \langle s_{n-1}, e_n\rangle \Downarrow \langle t, c_n\rangle \quad op^{arith}(c_1, \ldots, c_n) = c}{\langle s, op(e_1, \ldots, e_n)\rangle \Downarrow \langle t, c\rangle}
$$

$$
\textsc{Eval-If}\ \ \frac{\langle s, e_1\rangle \Downarrow \langle s_1, w_1\rangle \quad \langle s_1, e_i\rangle \Downarrow \langle t, w\rangle \quad (w_1, i) \in \{(\texttt{true}, 2), (\texttt{false}, 3)\}}{\langle s, \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\rangle \Downarrow \langle t, w\rangle}
$$

$$
\textsc{Eval-Tuple}\ \ \frac{\langle s, e_1\rangle \Downarrow \langle s_1, w_1\rangle \quad \ldots \quad \langle s_{n-1}, e_n\rangle \Downarrow \langle t, w_n\rangle}{\langle s, (e_1, \ldots, e_n)\rangle \Downarrow \langle t, (w_1, \ldots, w_n)\rangle} \qquad \textsc{Eval-Proj}\ \ \frac{\langle s, e\rangle \Downarrow \langle t, (w_1, \ldots, w_n)\rangle \quad 1 \le i \le n}{\langle s, \#_i(e)\rangle \Downarrow \langle t, w_i\rangle}
$$

$$
\textsc{Eval-Loc}\ \ \langle s, l\rangle \Downarrow \langle s, l\rangle \qquad \textsc{Eval-New}\ \ \frac{\langle s[l \mapsto 0], [l/x]e\rangle \Downarrow \langle t, w\rangle \quad l \notin Dom(s)}{\langle s, \nu x.e\rangle \Downarrow \langle t, w\rangle}
$$

$$
\textsc{Eval-Deref}\ \ \frac{\langle s, e\rangle \Downarrow \langle t, l\rangle \quad t(l) = w}{\langle s, !e\rangle \Downarrow \langle t, w\rangle} \qquad \textsc{Eval-Assign}\ \ \frac{\langle s, e_1\rangle \Downarrow \langle s_1, l\rangle \quad l \in Dom(s_1) \quad \langle s_1, e_2\rangle \Downarrow \langle t, w\rangle}{\langle s, (e_1 := e_2)\rangle \Downarrow \langle t[l \mapsto w], ()\rangle}
$$

Figure 2. Operational Semantics

3. Contextual Equivalence

We begin with the standard notion of contextual equivalence:

Definition 3.1 (Standard Contextual Equivalence). $(e, e') \in \equiv_{\text{std}}$ *if and only if for all stores* s *and expression contexts* $C[]$ *such that* $\langle s, C[e]\rangle$ *and* $\langle s, C[e']\rangle$ *are well-formed configurations,*

$$\langle s, C[e]\rangle \Downarrow \iff \langle s, C[e']\rangle \Downarrow$$

This definition is difficult to work with for several reasons:

- Since expression contexts can bind variables, contexts are not equal up to α-renaming, which makes quantification over expression contexts hard. Furthermore a proof must deal with the complications of binding in e and e'.

- It deals with expressions in the same store, but since closures are storable, any inductive proof must somehow involve generalization to programs operating in different stores.

Following Sumii and Pierce, we eliminate the first of these complications by invoking the following theorem.

Theorem 3.2 (Value Restriction). *For any expressions e, e', and identifier x,*

$$(e, e') \in \equiv_{\mathsf{std}} \iff (\lambda x.e, \lambda x.e') \in \equiv_{\mathsf{std}}$$

Proof. Appendix A.1. □

As a result, instead of reasoning about the equivalence of two open expressions e and e', we can close them under an appropriate number of abstractions and reason about the equivalence of those closed values. If e and e' are closed, then we can reason about the equivalence of $\lambda x.e$ and $\lambda x.e'$, for some identifier x.

We will therefore reason primarily about *value relations*:

Definition 3.3. *A* value relation *R is a set of pairs of closed values.*

Restricting the equivalent expressions to closed values permits the use of β-substitution $[v/x]C$, instead of the capturing substitution $C[v]$, where x is a free identifier in the hole of C. This allows us to define the set of all instances of expression contexts, with R-related values in their holes:

Definition 3.4. *If R is a value relation, then R^* is the set:*

$$R^* = \{([\overline{u/x}]d, [\overline{u'/x}]d) \mid \overline{(u, u')} \in R,\ FV(d) \subseteq \{\overline{x}\},\ Locs(d) = \emptyset\}$$

The generalization to contexts with multiple holes is needed by the induction principle we will introduce later. It is also useful to separately define the largest subset of R^* which contains only values:

Definition 3.5. *If R is a value relation, then R^*_{val} is the set:*

$$R^*_{\mathsf{val}} = \{(v, v') \mid (v, v') \in R^*, \text{ and } v, v' \text{ are values}\}$$

Both $(\ )^*$ and $(\ )^*_{\mathsf{val}}$ are closure operations:

Lemma 3.6. *If R and S are value relations, then:*

1. $R \subseteq R^*_{\mathsf{val}} \subseteq R^*$.
2. *If $R \subseteq S$, then $R^* \subseteq S^*$ and $R^*_{\mathsf{val}} \subseteq S^*_{\mathsf{val}}$.*
3. $(R^*)^* = R^*$ *and* $(R^*_{\mathsf{val}})^*_{\mathsf{val}} = R^*_{\mathsf{val}}$.
4. *If $R \subseteq S \subseteq R^*$, then $S^* = R^*$. If $R \subseteq S \subseteq R^*_{\mathsf{val}}$, then $S^*_{\mathsf{val}} = R^*_{\mathsf{val}}$.*

Proof. The first is immediate by the definitions of R^* and R^*_{val}. The rest follow by elementary properties of substitution. □

The next complication is that of expressions in different, but equivalent, stores. Therefore, we will reason not just about value relations, but *states*:

Definition 3.7. *A* state *is a triple (s, s', R), where s and s' are stores, and R is a value relation.*

A state (s, s', R) represents pairs of values that are intended to be equivalent in stores s and s', respectively. We call these "states" by analogy with the states of a Kripke structure: they represent possible worlds in which expressions may be evaluated. Since here R contains pairs of presumably equivalent values, the expressions to be evaluated cannot simply be the values in R. Instead, they are expressions that are R-related instances of a common root. This leads to the definition of contextual equivalence which we will use:

Definition 3.8. Contextual equivalence $\equiv$ *is the largest set of states (s, s', R), such that:*

1. $Lclosed(s)$ *and* $Lclosed(s')$,
2. *if $(u, u') \in R$, then $Locs(u) \subseteq Dom(s)$, and $Locs(u') \subseteq Dom(s')$,*
3. *if $(e, e') \in R^*$, then $\langle s, e\rangle \Downarrow \iff \langle s', e'\rangle \Downarrow$*

This definition is similar to the one in [31].

The first two conditions of the definition restrict to well-formed configurations; the third condition demands equivalence of termination under *related*, rather than equal, stores.

The role of R in each tuple of $\equiv$ is twofold. It contains pairs of equivalent values, where all the left-hand sides are evaluated in contexts with store s, and the right-hand sides in contexts with store s'. Furthermore, it is the means of accessing existing locations by the contexts; any location l not mentioned in R is inaccessible by elements of R^*, since the context d in Definition 3.4 may not contain any locations.

We extend $\equiv$ to open expressions by the following definition.

Definition 3.9. *$e \equiv e'$ iff for all stores s, there exists a value relation R such that:*

$$((s, s, R) \in \equiv) \wedge ((\lambda\overline{x}.e, \lambda\overline{x}.e') \in R)$$

for some $\overline{x}$ such that $FV(e) \subseteq \{\overline{x}\}$ and $FV(e') \subseteq \{\overline{x}\}$.

We show that $\equiv$ coincides with the standard notion of contextual equivalence:

Theorem 3.10. *$e \equiv e'$ iff $e \equiv_{\mathsf{std}} e'$.*

Proof. Appendix A.2. □

4. Deriving Obligations for an Equivalence Proof

Let $\mathcal{X}$ be a set of states of the form (s, s', R). How would we go about proving that $\mathcal{X} \subseteq (\equiv)$? For each state $(s, s', R) \in \mathcal{X}$, we must show:

1. $Lclosed(s)$ and $Lclosed(s')$,
2. for all $(u, u') \in R$, $Locs(u) \subseteq Dom(s)$, and $Locs(u') \subseteq Dom(s')$,
3. if $(e, e') \in R^*$, then $\langle s, e\rangle \Downarrow \iff \langle s', e'\rangle \Downarrow$

The first two of these are generally straightforward. The last one, of course, is the difficult one. In order to have any hope of carrying out an induction, we will need to show not merely that if $\langle s, e\rangle \Downarrow$ then $\langle s', e'\rangle \Downarrow$ (and vice versa), but also that the terminal configurations are related in some state of $\mathcal{X}$. More precisely, we will want to prove that

$$\begin{aligned}&((s, s', R) \in \mathcal{X}) \wedge ((e, e') \in R^*) \wedge \langle s, e\rangle \Downarrow \langle t, w\rangle \\ &\Longrightarrow (\exists t', w', Q : \langle s', e'\rangle \Downarrow \langle t', w'\rangle \\ &\qquad \wedge (Q \supseteq R) \wedge ((t, t', Q) \in \mathcal{X}) \\ &\qquad \wedge ((w, w') \in Q^*_{\mathsf{val}}))\end{aligned} \tag{1}$$

The obvious way to do this is by induction on the size of the derivation of $\langle s, e\rangle \Downarrow \langle t, w\rangle$. This leads to the following induction hypothesis:

$$\begin{aligned}IH(k) = \ &((s, s', R) \in \mathcal{X}) \wedge ((e, e') \in R^*) \wedge \langle s, e\rangle \Downarrow^{<k} \langle t, w\rangle \\ &\Longrightarrow (\exists t', w', Q : \langle s', e'\rangle \Downarrow \langle t', w'\rangle \\ &\qquad \wedge (Q \supseteq R) \wedge ((t, t', Q) \in \mathcal{X}) \\ &\qquad \wedge ((w, w') \in Q^*_{\mathsf{val}}))\end{aligned} \tag{2}$$

We can now imagine a proof of (2) for all k by induction on k. The proof would proceed by cases on e. Most of the cases follow directly from the induction hypothesis. The ones that don't would remain as proof obligations on $\mathcal{X}$.

To demonstrate this we consider the case of application ($e = (e_1\ e_2)$) in this proof: We assume (2) for k and prove it for $k+1$.

Let $(s, s', R) \in \mathcal{X}$, $((e_1\ e_2), e') \in R^*$, and $\langle s, e\rangle \Downarrow^{<k+1} \langle t, w\rangle$. We have to show that

$$\begin{array}{l}\exists t', w', Q\ :\ \langle s', e'\rangle \Downarrow \langle t', w'\rangle \\ \qquad \wedge\ (Q \supseteq R) \wedge ((t, t', Q) \in \mathcal{X}) \\ \qquad \wedge\ ((w, w') \in Q^*_{\mathsf{val}})\end{array}$$

By the definition of R^*, $e' = (e'_1\ e'_2)$, and $(e_i, e'_i) \in R^*$, for $i = 1, 2$. The big-step rule for application (EVAL-APP) gives for $(e_1\ e_2)$:

$$\frac{\begin{array}{l}\langle s, e_1\rangle \Downarrow^{<k} \langle s_1, \lambda x.e_3\rangle \\ \langle s_1, e_2\rangle \Downarrow^{<k} \langle s_2, v\rangle \\ \langle s_2, [v/x]e_3\rangle \Downarrow^{<k} \langle t, w\rangle\end{array}}{\langle s, (e_1\ e_2)\rangle \Downarrow^{<k+1} \langle t, w\rangle} \tag{3}$$

By the first premise of (3), and the induction hypothesis at e_1 and (s, s', R) we get:

$$\begin{array}{l}\exists s'_1, e'_3, R_1\ :\ \langle s', e'_1\rangle \Downarrow \langle s'_1, w'_1\rangle \\ \qquad \wedge\ (R_1 \supseteq R) \wedge ((s_1, s'_1, R_1) \in \mathcal{X}) \\ \qquad \wedge\ ((\lambda x.e_3, w'_1) \in {R_1}^*_{\mathsf{val}})\end{array}$$

By Lemma 3.6 (2) we get $R^* \subseteq {R_1}^*$. Thus, by the second premise of (3), and the induction hypothesis at e_2 and (s_1, s'_1, R_1):

$$\begin{array}{l}\exists s'_2, v', R_2\ :\ \langle s'_1, e'_2\rangle \Downarrow \langle s'_2, v'\rangle \\ \qquad \wedge\ (R_2 \supseteq R_1) \wedge ((s_2, s'_2, R_2) \in \mathcal{X}) \\ \qquad \wedge\ ((v, v') \in {R_2}^*_{\mathsf{val}})\end{array}$$

For the third premise of EVAL-APP we distinguish two cases:

• *Case 1:* $\lambda x.e_3$ and w'_1 are R_1-related contexts; i.e., there exist e'_3, d, and $(u, u') \in R_1$ such that $FV(d) \subseteq \{x, \overline{y}\}$, $Locs(d) = \emptyset$, $w'_1 = \lambda x.e'_3$, $e_3 = \overline{[u/y]}d$, $e'_3 = \overline{[u'/y]}d$. By the properties of β-substitution, and because ${R_1}^* \subseteq {R_2}^*$ and $(v, v') \in {R_2}^*_{\mathsf{val}} \subseteq {R_2}^*$ we have:

$$([v/x]e_3, [v'/x]e'_3) \in {R_2}^*$$

Thus, by the third premise of (3) and the induction hypothesis at $[v/x]e_3$ and (s_2, s'_2, R_2):

$$\begin{array}{l}\exists t', w', Q\ :\ \langle s'_2, [v'/x]e'_3\rangle \Downarrow \langle t', w'\rangle \\ \qquad \wedge\ (Q \supseteq R_2) \wedge ((t, t', Q) \in \mathcal{X}) \\ \qquad \wedge\ ((w, w') \in Q^*_{\mathsf{val}})\end{array}$$

• *Case 2:* $(\lambda x.e_3, w'_1) \in R_1$

To handle this case we need to show that w'_1 is also an abstraction (otherwise the application on the right-hand side would not succeed). Therefore one of the conditions on $\mathcal{X}$ must be:

> *if $(s, s', R) \in \mathcal{X}$, and $(v, v') \in R$, then v and v' have the same top-level language constructor.*

Assuming that $\mathcal{X}$ satisfies the above proof obligation, we have $w'_1 = \lambda x.e'$ and $(\lambda x.e_3, \lambda x.e') \in R_1 \subseteq R_2$. The third premise of (3) is equivalent to:

$$\langle s_2, (\lambda x.e_3\ v)\rangle \Downarrow^{<k+1} \langle t, w\rangle$$

Therefore, to finish this case of the proof, it is sufficient to show that

$$\begin{array}{l}\langle s_2, (\lambda x.e_3\ v)\rangle \Downarrow^{<k+1} \langle t, w\rangle \\ \Longrightarrow\ \exists t', w', Q\ :\ \langle s'_2, (\lambda x.e'_3\ v')\rangle \Downarrow \langle t', w'\rangle \\ \qquad \wedge\ (Q \supseteq R_2) \wedge ((t, t', Q) \in \mathcal{X}) \\ \qquad \wedge\ ((w, w') \in Q^*_{\mathsf{val}})\end{array}$$

assuming the induction hypothesis for derivations less than k. This leaves the following condition on $\mathcal{X}$:

> *if $(s, s', R) \in \mathcal{X}$, and $(\lambda x.e_3, \lambda x.e'_3) \in R$, and the induction hypothesis holds for derivations of height less than k, then for all $(v, v') \in R^*_{\mathsf{val}}$:*
>
> $$\begin{array}{l}\langle s_2, (\lambda x.e_3\ v)\rangle \Downarrow^{<k+1} \langle t, w\rangle \\ \Longrightarrow\ \exists t', w', Q\ :\ \langle s'_2, (\lambda x.e'_3\ v')\rangle \Downarrow \langle t', w'\rangle \\ \qquad \wedge\ (Q \supseteq R_2) \wedge ((t, t', Q) \in \mathcal{X}) \\ \qquad \wedge\ ((w, w') \in Q^*_{\mathsf{val}})\end{array}$$

If $\mathcal{X}$ satisfies the above proof obligations, then we get for both cases:

$$\frac{\begin{array}{l}\langle s', e'_1\rangle \Downarrow \langle s'_1, \lambda x.e'_3\rangle \\ \langle s'_1, e'_2\rangle \Downarrow \langle s'_2, v'\rangle \\ \langle s'_2, [v'/x]e'_3\rangle \Downarrow \langle t', w'\rangle\end{array}}{\langle s', (e'_1\ e'_2)\rangle \Downarrow \langle t', w'\rangle}$$

and also $Q \supseteq R$, $(t, t', Q) \in \mathcal{X}$, $(w, w') \in Q^*_{\mathsf{val}}$, and we are done.

By a similar treatment of the rest of the cases for e, we get all the conditions on $\mathcal{X}$, which are summarized in the definition of bisimulation in the following section.

5. Small Bisimulations

We now reformulate the proof obligations derived above into conditions on $\mathcal{X}$. We define a bisimulation to be any set $\mathcal{X}$ of states that satisfy these closure conditions.

Definition 5.1 (k-approximation). *We will write $(s, s', R) \vdash_{\mathcal{X}} e \sqsubseteq_k e'$ to mean:*

$$\begin{array}{l}\forall t, w.\ (\langle s, e\rangle \Downarrow^{<k} \langle t, w\rangle \\ \qquad \Longrightarrow \exists t', w', Q\ :\ (\langle s', e'\rangle \Downarrow \langle t', w'\rangle \\ \qquad\qquad \wedge\ (t, t', Q) \in \mathcal{X} \\ \qquad\qquad \wedge\ (w, w') \in Q^*_{\mathsf{val}} \\ \qquad\qquad \wedge\ R \subseteq Q))\end{array}$$

We will also write $(s, s', R)\ \vdash_{\mathcal{X}}\ e \sqsupseteq_k e'$ to mean:

$$\begin{array}{l}\forall t', w'.\ (\langle s', e'\rangle \Downarrow^{<k} \langle t', w'\rangle \\ \qquad \Longrightarrow \exists t, w, Q\ :\ (\langle s, e\rangle \Downarrow \langle t, w\rangle \\ \qquad\qquad \wedge\ (t, t', Q) \in \mathcal{X} \\ \qquad\qquad \wedge\ (w, w') \in Q^*_{\mathsf{val}} \\ \qquad\qquad \wedge\ R \subseteq Q))\end{array}$$

We use two flavors of k-approximation, one that corresponds to the forward direction of the proof (shown in the preceding section), and one for the opposite direction. Note that these two relations are not converses, since $(t, t', Q) \in \mathcal{X}$, and $(w, w') \in Q^*_{\mathsf{val}}$ appear in both.

The full induction hypothesis of the proof is given by $IH^L_{\mathcal{X}}(k)$ and $IH^R_{\mathcal{X}}(k)$, the first for the forward direction and the second for the opposite:

Definition 5.2. *We will write $IH^L_{\mathcal{X}}(k)$ to mean:*

$$\forall (s, s', R) \in \mathcal{X}.\ \forall (e, e') \in R^*.\ (s, s', R) \vdash_{\mathcal{X}} e \sqsubseteq_k e'$$

and $IH^R_{\mathcal{X}}(k)$ to mean:

$$\forall (s, s', R) \in \mathcal{X}.\ \forall (e, e') \in R^*.\ (s, s', R)\ \vdash_{\mathcal{X}}\ e \sqsupseteq_k e'$$

We now give the definition of bisimulation.

Definition 5.3. *A* bisimulation $\mathcal{X}$ *is a set of states such that for any state $(s, s', R) \in \mathcal{X}$, the following conditions are satisfied:*

1. *$Lclosed(s)$ and $Lclosed(s')$,*
2. *If $(v, v') \in R$, then $Locs(v) \subseteq Dom(s)$ and $Locs(v') \subseteq Dom(s')$.*

3. *If $(v, v') \in R$, then v and v' have the same top-level language constructor.*
4. *If $(c, c') \in R$, then $c = c'$.*
5. *If $((v_1, \ldots, v_n), (v'_1, \ldots, v'_m)) \in R$, then $n = m$, and for all $1 \leq i \leq n$, there exists a value relation Q such that $R \subseteq Q$, $(v_i, v'_i) \in Q^*_{\mathsf{val}}$, and $(s, s', Q) \in \mathcal{X}$.*
6. *For any l, l', with $l \notin Dom(s)$ and $l' \notin Dom(s')$, there exists a value relation Q such that $R \subseteq Q$, $(l, l') \in Q$, and $(s[l \mapsto 0], s'[l' \mapsto 0], Q) \in \mathcal{X}$.*
7. *If $(l, l') \in R$, then*
   (a) *there exists a value relation Q such that $R \subseteq Q$, $(s(l), s'(l')) \in Q^*_{\mathsf{val}}$, and $(s, s', Q) \in \mathcal{X}$,*
   (b) *for all $(v, v') \in R^*_{\mathsf{val}}$, we have $(s[l \mapsto v], s'[l' \mapsto v'], R) \in \mathcal{X}$*
8. *For each $(\lambda x.e, \lambda x.e') \in R$, and for any values $(v, v') \in R^*_{\mathsf{val}}$, we have*

$$IH^L_{\mathcal{X}}(k) \Longrightarrow (s, s', R) \vdash_{\mathcal{X}} (\lambda x.e\ v) \sqsubseteq_{k+1} (\lambda x.e'\ v')$$
$$IH^R_{\mathcal{X}}(k) \Longrightarrow (s, s', R) \vdash_{\mathcal{X}} (\lambda x.e\ v) \sqsupseteq_{k+1} (\lambda x.e'\ v')$$

Each of the conditions of Definition 5.3 addresses one of the proof obligations of the direct proof discussed in Section 4. The first two conditions are identical to the first two conditions of $\equiv$. Condition 3 allows us to conclude throughout the direct proof that, if a value v has a specific top-level language constructor, and it is related to a value v', then v' also has the same language constructor. This is necessary, for example, in the case of application, when the operator is one of the related values; if the operator in the left-hand side is an abstraction, then the operator of the right-hand side is also an abstraction (otherwise they would not necessarily have the same terminating behavior).

Condition 4 captures the proof obligation for the case of applying a primitive operator, Condition 5 the proof obligation for the case of projection, Condition 6 the proof obligation for the case of the ν-expression, and Conditions 7 (a), (b) capture the proof obligations for the cases of dereferencing and assignment, respectively.

Condition 8 captures the proof obligation in the case of application. As shown in Section 4, in this case the only sub-case that doesn't go through directly by the induction hypothesis is when there are related abstractions in operator position. To prove this subcase we may assume the induction hypothesis for shorter derivation trees. This induction hypothesis is very useful to have when reasoning about higher-order procedures, as we will see in the examples.

Next we show that our bisimulations are sound and complete and that a maximal bisimulation exists.

Theorem 5.4 (Completeness). *Contextual equivalence is a bisimulation.*

Proof. Appendix A.3. □

Theorem 5.5 (Soundness). *Any bisimulation is included in contextual equivalence.*

Proof. The proof recapitulates the derivation of the preceding section, and is given in Appendix A.4. □

Theorem 5.6. *A maximal bisimulation, called* bisimilarity $(\sim)$, *exists and coincides with contextual equivalence.*

Proof. By Definition 3.8 and Theorems 5.4 and 5.5 we get that $\equiv$ is the largest bisimulation. Thus $(\sim)$ exists and coincides with contextual equivalence. □

6. Examples

The following examples illustrate our techniques. The first two are due to Meyer and Sieber [17]. The rest of the examples in [17] can be done in similar fashion. These examples will also illustrate the added power given by the induction hypotheses in the last condition of Definition 5.3.

6.1 Local Store

This example shows that allocation of local store does not affect computation.

$$M = \lambda g.\nu x.g \qquad N = \lambda g.g$$

Proof. To show $M \equiv N$, it will suffice to construct a bisimulation $\mathcal{X}$ that relates the two expressions at empty stores; Conditions 6 and 7 (b) of Definition 5.3 ensure that the relation covers all equal stores. We define the parameterized relation:

$$Q(\overline{k, k'}) = \{(M, N), \overline{(k, k')}\}$$

and the set:

$$\begin{aligned}\mathcal{X} = \big\{(s, s', R) \,\big|\, & \exists \overline{k, k'}, \overline{l} \\ & : \ R = Q(\overline{k, k'}) \\ & \wedge \ \{\overline{k}\} \cap \{\overline{l}\} = \emptyset \\ & \wedge \ \exists \overline{u, u'} \\ & \qquad : \ s \ = [\overline{k \mapsto u}, \overline{l \mapsto 0}] \\ & \qquad \wedge \ s' = [\overline{k' \mapsto u'}] \\ & \qquad \wedge \ \overline{(u, u')} \in R^*_{\mathsf{val}}\big\}\end{aligned}$$

Each state (s, s', R) of $\mathcal{X}$ relates M and N in R, as well as an arbitrary set of locations $\overline{k}$ and $\overline{k'}$. The stores s and s' consist of the related locations $\overline{k}$ and $\overline{k'}$, with related contents from R^*_{val}, and s has some more locations $\overline{l}$, created by calls to M, each of which contain the integer zero.

We will show that $\mathcal{X}$ satisfies the conditions for a bisimulation. Conditions 1, 3 and 7 are satisfied because $\overline{(u, u')} \in R^*_{\mathsf{val}}$, and M, N are closed abstractions; Condition 2 is satisfied because $Dom(s)$ and $Dom(s')$ contain all the related locations in R. Condition 6 is obviously satisfied; Conditions 4 and 5 are trivially satisfied.

Condition 8 applies only to (M, N). Let $(s, s', R) \in \mathcal{X}$. We have to show that for all $(v, v') \in R^*_{\mathsf{val}}$, the following is true:

$$IH^L_{\mathcal{X}}(k) \Longrightarrow (s, s', R) \vdash_{\mathcal{X}} (M\ v) \sqsubseteq_{k+1} (N\ v')$$
$$IH^R_{\mathcal{X}}(k) \Longrightarrow (s, s', R) \vdash_{\mathcal{X}} (M\ v) \sqsupseteq_{k+1} (N\ v')$$

For this example we will show directly that both $\langle s, (M\ v)\rangle$ and $\langle s', (N\ v')\rangle$ always terminate, and there exist t, t', w, w', S such that $\langle s, (M\ v)\rangle \Downarrow \langle t, w\rangle$, $\langle s', (N\ v')\rangle \Downarrow \langle t', w'\rangle$, $(t, t', S) \in \mathcal{X}$, $(w, w') \in S^*_{\mathsf{val}}$, and $R \subseteq S$.

It is easy to show that $\langle s, (M\ v)\rangle$ terminates:

$$\begin{aligned}& \langle s, (M\ v)\rangle \Downarrow \langle t, w\rangle \\ \Longleftrightarrow\ & \langle s, (\lambda g.\nu x.g)v\rangle \Downarrow \langle t, w\rangle \\ \Longleftrightarrow\ & \langle s, \nu x.[v/g]g\rangle \Downarrow \langle t, w\rangle \\ \Longleftrightarrow\ & \exists l_0 \notin Dom(s).\ \langle s[l_0 \mapsto 0], [l_0/x]v\rangle \Downarrow \langle t, w\rangle \\ \Longleftrightarrow\ & \exists l_0 \notin Dom(s).\ \langle s[l_0 \mapsto 0], v\rangle \Downarrow \langle t, w\rangle \text{ (since } v \text{ is closed)}\end{aligned}$$

Therefore $t = s[l_0 \mapsto 0]$ for some l_0, and $w = v$.

Similarly for $\langle s', (N\ v')\rangle$:

$$\begin{aligned}& \langle s', (N\ v')\rangle \Downarrow \langle t', w'\rangle \\ \Longleftrightarrow\ & \langle s', (\lambda g.g)v'\rangle \Downarrow \langle t', w'\rangle \\ \Longleftrightarrow\ & \langle s', [v'/g]g\rangle \Downarrow \langle t', w'\rangle \\ \Longleftrightarrow\ & \langle s', v'\rangle \Downarrow \langle t', w'\rangle\end{aligned}$$

and $t' = s'$, $w' = v'$. Moreover, by the definition of $\mathcal{X}$, $(s[l_0 \mapsto 0], s', R) \in \mathcal{X}$, and $(w, w') \in R$. □

6.2 Higher-Order Functions

The preceding example did not rely on the induction hypotheses in Condition 8 of Definition 5.3. The next example uses the induction hypotheses to reason about a call to an unknown procedure.

$$\begin{aligned} M &= \lambda g.\nu x. \\ &\qquad \texttt{let}\ f = \lambda z.(x := !x + 2) \\ &\qquad \texttt{in}\ ((g\ f); \\ &\qquad\quad \texttt{if}\ (!x \bmod 2 = 0)\ \texttt{then}\ ()\ \texttt{else}\ \Omega) \\ N &= \lambda g.((g\ \lambda x.()); ()) \end{aligned}$$

Since the only operation g can perform on the location bound to x is to increase it by two, invocation of g preserves the invariant that the contents of the location is even.

Proof. As we did in the previous example, we define a parameterized relation:

$$Q(\overline{k, k'}, \overline{l}) = \{(M, N), \overline{(\lambda z.(l := !l + 2), \lambda z.())}, \overline{(k, k')}\}$$

and the set

$$\begin{aligned} \mathcal{X} = \big\{(s, s', R) \,\big|\, &\exists \overline{k, k'}, \overline{l} \\ &: R = Q(\overline{k, k'}, \overline{l}) \\ &\wedge \{\overline{k}\} \cap \{\overline{l}\} = \emptyset \\ &\wedge \exists \overline{u, u'}, \overline{n} \\ &\qquad : s = [\overline{k \mapsto u}, \overline{l \mapsto 2n}] \\ &\qquad \wedge s' = [\overline{k' \mapsto u'}] \\ &\qquad \wedge \overline{(u, u')} \in R^*_{\mathsf{val}}\big\} \end{aligned}$$

Here the states of the bisimulation are of the form (s, s', R), where the stores s and s' consist of related locations $\overline{k}$ and $\overline{k'}$, containing related values $\overline{u}$ and $\overline{u'}$, and s contains some additional locations $\overline{l}$, allocated by calls to M, each of which contains some even number. The relation R relates the locations $\overline{(k, k')}$ and each instantiation of $\lambda z.(l_i := !l_i + 2)$ to $\lambda z.()$.

We will show that $\mathcal{X}$ satisfies the conditions for a bisimulation. As in the previous example, it is easy to check that Conditions 1 through 7 are satisfied by construction of $\mathcal{X}$.

It remains to prove that Condition 8 is satisfied. The related abstractions in R are (M, N) and $(\lambda z.(l_i := !l_i + 2), \lambda z.())$, for all $l_i \in \{\overline{l}\}$.

First we prove Condition 8 for $(\lambda z.(l_i := !l_i + 2), \lambda z.())$. Let $s = s_0[l_i \mapsto 2n_i]$. We have to show that for all $(v, v') \in R^*_{\mathsf{val}}$, the following is true:

$$\begin{aligned} \mathit{IH}^L_{\mathcal{X}}(k) \implies &(s_0[l_i \mapsto 2n_i], s', R) \vdash_{\mathcal{X}} \\ &(\lambda z.(l_i := !l_i + 2)\ v) \sqsubseteq_{k+1} (\lambda z.()\ v') \end{aligned}$$

Equivalently, we assume $\mathit{IH}^L_{\mathcal{X}}(k)$, and $\langle s_0[l_i \mapsto 2i], (\lambda z.(l_i := !l_i + 2)\ v)\rangle \Downarrow^{<k+1} \langle t, w\rangle$, and show that there exist t', w', S, such that $\langle s', (\lambda z.()\ v')\rangle \Downarrow \langle t', w'\rangle$, $(t, t', S) \in \mathcal{X}$, $(w, w') \in S^*_{\mathsf{val}}$, $R \subseteq S$. We have:

$$\begin{aligned} &\langle s_0[l_i \mapsto 2n_i], (\lambda z.(l_i := !l_i + 2)\ v)\rangle \Downarrow \langle t, w\rangle \\ \iff &\langle s_0[l_i \mapsto 2n_i], [v/z](l_i := !l_i + 2)\rangle \Downarrow \langle t, w\rangle \\ \iff &\langle s_0[l_i \mapsto 2n_i], (l_i := !l_i + 2)\rangle \Downarrow \langle t, w\rangle \end{aligned}$$

and by the evaluation rule for assignment we get that $t = s_0[l_i \mapsto 2(n_i + 1)]$, and $w = ()$. Obviously

$$\langle s', (\lambda z.()\ v')\rangle \Downarrow \langle s', ()\rangle$$

Furthermore, by the definition of $\mathcal{X}$, $(s_0[l_i \mapsto 2(n_i + 1)], s', R) \in \mathcal{X}$, and $((), ()) \in R^*_{\mathsf{val}}$. The second part of Condition 8 is analogous.

Now we need to prove Condition 8 for (M, N). Again we have to show that for all $(v, v') \in R^*_{\mathsf{val}}$:

$$\mathit{IH}^L_{\mathcal{X}}(k) \implies (s, s', R) \vdash_{\mathcal{X}} (M\ v) \sqsubseteq_{k+1} (N\ v')$$

or equivalently, if $\mathit{IH}^L_{\mathcal{X}}(k)$, and $\langle s, (M\ v)\rangle \Downarrow^{<k+1} \langle t, ()\rangle$, then there exist t', S, such that $\langle s', (N\ v')\rangle \Downarrow \langle t', ()\rangle$, $(t, t', S) \in \mathcal{X}$, $R \subseteq S$. We have:

$$\begin{aligned} &\langle s, (M\ v)\rangle \Downarrow^{<k+1} \langle t, ()\rangle \\ \implies &\exists w, l_0 : \langle s[l_0 \mapsto 0], (v\ \lambda z.(l_0 := !l_0 + 2))\rangle \Downarrow^{<k} \langle t, w\rangle \\ &\qquad \wedge (l_0 \notin Dom(s)) \end{aligned}$$

Let $R = Q(\overline{k, k'}, \overline{l})$ and $P = Q(\overline{k, k'}, \overline{l}, l_0)$. We have that $R \subseteq P$. By the definition of $\mathcal{X}$, we get $(s[l_0 \mapsto 0], s', P) \in \mathcal{X}$. Also, because $(v, v') \in R^*_{\mathsf{val}}$ and $(\lambda z.(l_0 := !l_0 + 2), \lambda z.()) \in R$, we have $((v\ \lambda z.(l_0 := !l_0 + 2)), (v'\ \lambda z.())) \in R^* \subseteq P^*$. Thus, from $\mathit{IH}^L_{\mathcal{X}}(k)$, we get that there exist t', w', S such that $\langle s', (v\ \lambda z.())\rangle \Downarrow \langle t', w'\rangle$, $(t, t', S) \in \mathcal{X}$, $(w, w') \in S^*_{\mathsf{val}}$, and $P \subseteq S$. Therefore $\langle s', (N\ v')\rangle \Downarrow \langle t', ()\rangle$, $(t, t', S) \in \mathcal{X}$, and $R \subseteq S$, as required. □

This example would be difficult to prove using the technique of Sumii and Pierce from [31], since that technique has no corresponding induction hypothesis. Pitts and Stark's method of Local Invariants [22] does not apply to this example either.[2]

6.3 Higher-Order Stored Values and Methods

Our technique can also deal with the structures underlying object-oriented programs. For example, here is a model of the `Cell` example on page 1. Here M and N each receive an initial value and return a setter and a getter.

$$\begin{aligned} M &= \lambda x.\nu y.(\lambda z.(y := z), \lambda z.!y) \\ N &= \lambda x.\nu y_1.\nu y_2.\nu p.(\lambda z.((p := !p + 1); (y_1 := z); (y_2 := z)), \\ &\qquad \lambda z.(\texttt{if}\ even(!p)\ \texttt{then}\ !y_1\ \texttt{else}\ !y_2)) \end{aligned}$$

M allocates one location and exports a setter and a getter. N allocates two locations. The setter stores a value in both locations and also increments a counter; the corresponding getter retrieves the value from one location when the counter is even and from the other when the counter is odd. This example is slightly more general than the one on page 1, since here the location can be used for higher-order values, not just integers.

In general, every application of M will allocate a location l_i. For each such location, the corresponding application of N will allocate three locations $l'_{1i}, l'_{2i}, l'_{pi}$. The locations l_i, l'_{1i}, l'_{2i} will always contain related values. These locations are not themselves going to be related in the eventual bisimulation (since they are not accessible by the context), but their setter and getter procedures will. In addition, the execution of the contexts, with M and N in their holes, will allocate some locations $\overline{k}$, $\overline{k'}$, respectively. So as before we define a parameterized relation, from which we build a bisimulation.

The construction is shown in Figure 3. As before, Q relates the starting terms M and N, the corresponding locations $\overline{(k, k')}$ allocated external to M and N, and the corresponding setter and getter methods for each object created by M or N (which we write as $set_M, get_M, set_N, get_N$, parameterized by the locations they access). We turn this into a bisimulation by specifying that the related locations contain related values, and the locations of the corresponding objects contain the correct values as well.

Proof. As in the previous examples, Conditions 1 through 7 are immediately satisfied. We have to show Condition 8 for each of

[2] Like the technique of [31], the Logical Relation of [22] is complete, but does not seem to lead to a useful proof.

$$
\begin{aligned}
&Q(\overline{k,k'},\overline{l,l'_1,l'_2,l'_p}) = \{(M,N),\overline{(set_M(l),set_N(l'_1,l'_2,l'_p))},\overline{(get_M(l),get_N(l'_1,l'_2,l'_p))},\overline{(k,k')}\}\\
&\mathcal{X} = \big\{(s,s',R)\,\big|\,\exists \overline{k,k'},\overline{l,l'_1,l'_2,l_p}\\
&\qquad\qquad : \; R = Q(\overline{k,k'},\overline{l,l'_1,l'_2,l'_p})\\
&\qquad\qquad \wedge\; \{\overline{k}\}\cap\{\overline{l}\} = \{\overline{k'}\}\cap\{\overline{l,l'_1,l'_2,l_p}\} = \emptyset\\
&\qquad\qquad \wedge\; \exists \overline{u_0,u'_0,u,u'},n\\
&\qquad\qquad\qquad : \; s \;= [\overline{k\mapsto u_0},\overline{l\mapsto u}]\\
&\qquad\qquad\qquad \wedge\; s' = [\overline{k'\mapsto u'_0},\overline{l'_1\mapsto u',l'_2\mapsto u',l'_p\mapsto n'}]\\
&\qquad\qquad\qquad \wedge\; \overline{(u_0,u'_0)}\in R^*_{\mathsf{val}}\\
&\qquad\qquad\qquad \wedge\; \overline{(u,u')}\in R^*_{\mathsf{val}}\big\}
\end{aligned}
$$

Figure 3. Bisimulation for Example 6.3

(M,N), $(set_M(l_i), set_N(l'_{1i},l'_{2i},l'_{pi}))$, $(get_M(l_i), get_N(l'_{1i},l'_{2i},l'_{pi}))$. It easy to see that all of the related abstractions terminate. Moreover the final states satisfy the closure requirements of Condition 8. □

7. Comparison with Sumii and Pierce

A direct adaptation of the operational bisimulation of Sumii and Pierce [31] to our language would give a definition of bisimulations that differs from Definition 5.3 in the following ways:

Condition 5: If $(s,s',R)\in\mathcal{X}$, and $((v_1,\ldots,v_n),(v'_1,\ldots,v'_n))\in R$, then $(s,s',R\cup\{(v_i,v'_i)\})\in\mathcal{X}$.

Condition 6: If $(s,s',R)\in\mathcal{X}$, and $(l,l')\in R$, then $(s,s',R\cup\{(s(l),s'(l'))\})\in\mathcal{X}$, and for all $(v,v')\in R^*_{\mathsf{val}}$, $(s[l\mapsto v], s'[l'\mapsto v'], R\cup\{(v,v')\})\in\mathcal{X}$.

Condition 8: For all $(s,s',R)\in\mathcal{X}$, and all $(\lambda x.e,\lambda x.e')\in R$, take any $\overline{l,l'}$, such that $\{\overline{l}\}\cap Dom(s)=\{\overline{l'}\}\cap Dom(s')=\emptyset$. Let $Q = R\cup\overline{(l,l')}$, and $\overline{(u,u')}\in Q^*_{\mathsf{val}}$, $(v,v')\in Q^*_{\mathsf{val}}$. Then

(a) $\langle s[\overline{l\mapsto u}],(\lambda x.e\ v)\rangle\Downarrow\langle t,w\rangle \iff \langle s'[\overline{l'\mapsto u'}],(\lambda x.e'\ v')\rangle\Downarrow\langle t',w'\rangle$

(b) and if they terminate, then $(t,t',Q\cup\{(w,w')\})\in\mathcal{X}$

There are three important differences between these conditions and the ones we give in Definition 5.3:

- We use a more relaxed closure condition in each of the above three rules. For example, in Condition 5, instead of requiring that $(s,s',R\cup\{(v_i,v'_i)\})\in\mathcal{X}$, we require that there exists a value relation Q such that $R\subseteq Q$, $(v_i,v'_i)\in Q^*_{\mathsf{val}}$, and $(s,s',Q)\in\mathcal{X}$. This is essentially an *up-to context* closure of our bisimulations, similar to the one in [25]. An up-to context technique is not applicable in [31] because the values of that calculus may contain an arbitrary nesting of seals not known to the context.
- In Condition 8 we do not require the arguments given to the functions to be from a larger value relation than R; i.e. instead of requiring the arguments $(v,v')\in Q^*_{\mathsf{val}}$, and $Q = R\cup\overline{(l,l')}$, for some fresh $\overline{l,l'}$, we just require that $(v,v')\in R^*_{\mathsf{val}}$. This simplification, along with the up-to context closure, permits direct construction of bisimulations, whereas the conditions shown here would require inductive constructions in some cases (e.g. for Example 6.1). Of course to restore the soundness of the bisimulation we have added Condition 6 to ensure soundness under any possible extension of the store.

  We could have achieved at least the same level of simplicity in constructing bisimulations by adding an *up-to store* closure of our bisimulations, instead of using Condition 6. We chose not to do this because (a) it would require more technical machinery to define a unified up-to store and context operator, which would make our technique (and especially proving Condition 8) more involved, and (b) the benefit from such an addition would be the elimination of a small and highly stylized part from each bisimulation (the explicit mention of any extension of the store).
- The last, and perhaps the most important, difference is also in Condition 8. As discussed in [32], the Condition 8 shown above is too strict to help reason about a class of higher-order functions; that is, the class of functions that apply their arguments internally (like in Example 6.2), and thus their termination depends on the termination of these applications.

  In order to prove that such higher-order functions co-terminate, one must prove that the applications of their arguments to related values co-terminate. Since these arguments are provided by the context, the problem is reduced to proving co-termination of R-related contexts, the very thing that we tried to avoid by defining a bisimulation. We solve this problem by introducing the induction hypothesis of Section 4 inside Condition 8, which directly implies the co-termination of applications of the arguments.

  In Section 7 of [31] a similar condition is proposed, but not studied thoroughly, as a possible way of reasoning about higher-order functions. That condition is also based on induction on the height of derivation trees.

8. Other Related Work

A wide variety of techniques have been used to prove the contextual equivalence of higher-order expressions in the presence of a store.

One obvious approach is denotational: if two expressions have the same denotation in an adequate compositional semantics, then they are guaranteed to be contextually equivalent. The difficulty is that most denotational models are not fully abstract: there are contextually equivalent expressions that have different denotations. Meyer and Sieber [17] give a set of such examples. We have used our system to verify the equivalence of all of the examples in [17].

Another approach is to attempt to restrict the set of contexts that must be considered in a direct proof. In this approach, one defines a set $\mathcal{C}$ of contexts such that if two expressions are equivalent in all contexts in $\mathcal{C}$, then they are equivalent in every context. Such theorems typically take the form of *ciu theorems*. For example, Mason and Talcott [16] prove such a theorem for a language similar to ours. Similar results were obtained by Felleisen [7].

Yet another family of approaches uses either implicit or explicit transformation to continuation-passing style. For example, Tiuryn and Wand [33] present a continuation-passing model of an untyped lambda-calculus with input-output, and prove that applicative ap-

proximation coincides with contextual approximation. Wand and Sullivan [36] give a denotation to a recursively-typed higher-order language with side effects by translation to a CPS calculus, and use the technique to prove the correctness of assignment elimination, an important step in the compilation of Scheme. Our technique does not require conversion to CPS.

Pitts and Stark in [22] present a reasoning technique for a simply-typed functional language with a store that contains only integers. Their method is based on considering relations over configurations of what is effectively a continuation-passing interpreter. They define an operational logical relation between expressions that is parameterized over store relations, and prove its soundness and completeness with respect to observational equivalence. The definition of this logical relation is done by induction on types, using a set of conditions that take into account the possible evaluation contexts in which related expressions may appear. Because the set of evaluation contexts is infinite, it is not possible to directly construct this relation and use it in concrete examples. Instead, they give a proof method called the principle of local invariants, which can be used to prove that two expressions are included in the logical relation. This method is shown to be sound but not complete. Our method can be used for all their examples, including ones for which their method does not apply (Example 6.2).

Honda, Berger, and Yoshida in [12, 5] give a compositional program logic for a language with effects and higher-order procedures. Their method can prove properties of programs written as Hoare triples. They also give a connection between their logic and contextual equivalence: two expressions M, N are equivalent if and only if for all pre- and post-conditions C, C', M satisfies C, C' iff N satisfies C, C'. Although the quantification on all pre- and post-conditions seems easier than a quantification over all possible program contexts, it is still hard enough to make their method impractical for proving contextual equivalence.

Bisimulation was originally introduced as a method of characterizing the behavior of non-deterministic systems [10, 11]; this work also considered the difference between bisimulation and contextual equivalence. Abramsky [1] applied this idea to an untyped lazy lambda-calculus, and proved by domain-theoretic methods that the bisimulation was a congruence; Howe [13] later provided a direct operational proof.

Up-to techniques were originally introduced in concurrency [26, 27, 28]. Their purpose was to reduce the size of bisimulations by closing them automatically up to context, β-equivalence, injective substitution, bisimilarity etc., hence making their construction easier. We incorporate an up-to context technique in the untyped lambda calculus augmented with higher-order procedures and a general store. As discussed in Section 7 an up-to store technique could also be used, but the added value from such a method would be limited compared to the technical complication that it would introduce.

9. Conclusions and Future Work

We have presented a method for reasoning about higher-order imperative programs by using bisimulations. Our technique, although inspired by and similar to the one in [31], follows a more technical path and reaches a better notion of bisimulation. Using our method we were able to overcome the difficulties of previous approaches, and successfully deal with higher-order functions and store.

Example 6.3, with its setter and getter procedures, illustrates how this technique can be used to relate objects or methods that manipulate private state. This suggests that our results should be applicable to imperative object-oriented languages to prove, for example, the correctness of refactoring transformations.

We hope to investigate whether our techniques can improve results for typed calculi, such as [32]. We also hope to apply our techniques to study contextual equivalence in aspect-oriented calculi, in which advice is typically kept in the store, as in [14]. Last, we would like to see whether this Kripke-style bisimilarity can help investigate contextual equivalence in languages with control effects, e.g. call/cc or exceptions.

References

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

[2] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000. Originally appeared as [3].

[3] Samson Abramsky, Pasquale Malacaria, and Radha Jagadeesan. Full abstraction for PCF. In *Theoretical Aspects of Computer Software*, pages 1–15, 1994.

[4] Nick Benton and Benjamin Leperchey. Relational reasoning in a nominal semantics for storage. In *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2005.

[5] Martin Berger, Kohei Honda, and Nobuko Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*. ACM Press, sept 2005. To appear.

[6] Yuxin Deng and Davide Sangiorgi. Towards an algebraic theory of typed mobile processes. In *Proc. Icalp '04*, volume 3142 of *Lecture Notes in Computer Science*, pages 445–456. Springer-Verlag, 2004.

[7] Matthias Felleisen. *The Calculi of Lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[8] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. In *Proceedings 22nd Annual ACM Symposium on Principles of Programming Languages*, pages 209–220, 1995.

[9] Matthew Hennessy. *Algebraic theory of processes*. MIT Press, Cambridge, MA, USA, 1988.

[10] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In *ICALP*, pages 299–309, 1980.

[11] Matthew Hennessy and Robin Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32:137–161, 1985.

[12] Kohei Honda, Martin Berger, and Nobuko Yoshida. An observationally complete program logic for imperative higher-order functions. In *Proceedings of the Twentieth Annual IEEE Symposium on Logic in Computer Science (LICS)*, June 2005. To appear.

[13] Douglas J. Howe. Equality in lazy computation systems. In *Proc. 4th IEEE Symposium on Logic in Computer Science*, pages 198–203, 1989.

[14] R. Jagadeesan, A. S. A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proceedings European Conference on Object-Oriented Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 415–427, Berlin, Heidelberg, and New York, 2003. Springer-Verlag.

[15] Eugene M. Kohlbecker and Mitchell Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings 14th Annual ACM Symposium on Principles of Programming Languages*, pages 77–84, 1987.

[16] Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.

[17] Albert R. Meyer and Kurt Sieber. Towards fully abstract semantics for local variables: Preliminary report. In *Proceedings 15th Annual ACM Symposium on Principles of Programming Languages*, pages 191–203, 1988.

[18] Robert Milne and Christopher Strachey. *A Theory of Programming Language Semantics*. Chapman and Hall, London, 1976. Also Wiley, New York.

[19] Robin Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[20] Robin Milner. Operational and algebraic semantics of concurrent processes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1201–1242. MIT Press/Elsevier, 1990.

[21] James H. Morris, Jr. *Lambda Calculus Models of Programming Languages*. PhD thesis, MIT, Cambridge, MA, 1968.

[22] Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In Andrew Gordon and Andrew Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Publications of the Newton Institute, Cambridge University Press, 1998.

[23] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[24] E. Ritter and A. M. Pitts. A fully abstract translation between a λ-calculus with reference types and Standard ML. In *2nd Int. Conf. on Typed Lambda Calculus and Applications, Edinburgh, 1995*, volume 902 of *Lecture Notes in Computer Science*, pages 397–413, Berlin, Heidelberg, and New York, 1995. Springer-Verlag.

[25] Davide Sangiorgi. Locality and true-concurrency in calculi for mobile processes. In *Theoretical Aspects of Computer Software*, pages 405–424, 1994.

[26] Davide Sangiorgi. On the bisimulation proof method. In J. Wiedermann and P. Háiek, editors, *Proc. MFCS'95*, volume 969 of *Lecture Notes in Computer Science*, pages 479–488. Springer-Verlag, 1995. Full version to appear in J. Math. Structures in Comp. Sci.

[27] Davide Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processes. *Theoretical Computer Science*, 155:39–83, 1996.

[28] Davide Sangiorgi and Robin Milner. The problem of "Weak Bisimulation up to". In W.R. Cleveland, editor, *Proc. CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer-Verlag, 1992.

[29] Kurt Sieber. Full abstraction for the second order subset of an algol-like language. *Theor. Comput. Sci.*, 168(1):155–212, 1996.

[30] Paul A. Steckler and Mitchell Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, January 1997. Original version appeared in Proceedings 21st Annual ACM Symposium on Principles of Programming Languages, 1994.

[31] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for dynamic sealing. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 161–172, New York, NY, USA, 2004. ACM Press.

[32] Eijiro Sumii and Benjamin C. Pierce. A bisimulation for type abstraction and recursion. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–74, New York, NY, USA, 2005. ACM Press.

[33] Jerzy Tiuryn and Mitchell Wand. Untyped lambda-calculus with input-output. In H. Kirchner, editor, *Trees in Algebra and Programming: CAAP'96, Proc. 21st International Colloquium*, volume 1059 of *Lecture Notes in Computer Science*, pages 317–329, Berlin, Heidelberg, and New York, April 1996. Springer-Verlag.

[34] Mitchell Wand and William D. Clinger. Set constraints for destructive array update optimization. *Journal of Functional Programming*, 11(3):319–346, May 2001.

[35] Mitchell Wand and Igor Siveroni. Constraint systems for useless variable elimination. In *Proceedings 26th Annual ACM Symposium on Principles of Programming Languages*, pages 291–302, 1999.

[36] Mitchell Wand and Gregory T. Sullivan. Denotational semantics using an operationally-based term model. In *Proceedings 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 386–399, 1997.

A. Proofs

Definition A.1. *$M \preceq N$ is the smallest congruence containing $(e, ((\lambda\overline{x}.e)\ \overline{x}))$, for all e.*

Note that if $e \preceq e'$ then $FV(e) = FV(e')$, and for all C, $C[e] \preceq C[e']$.

Lemma A.2. *Let $e \preceq e'$, let $\overline{x}$ be identifiers, and $\overline{v, v'}$ be closed values, such that $\overline{v \preceq v'}$, $FV(e) \subseteq \{\overline{x}\}$, and $FV(e') \subseteq \{\overline{x}\}$. Then for all stores s, s', with $Dom(s) = Dom(s')$, and for all $l \in Dom(s)$, $s(l) \preceq s'(l')$:*

$$\langle s, [\overline{v/x}]e\rangle \Downarrow \iff \langle s', [\overline{v'/x}]e'\rangle \Downarrow$$

Proof. Similar to the one in [32]. □

A.1 Proof of Theorem 3.2 (Value Restriction)

Proof. The forward direction is easy. If some store s and expression context $C[]$ distinguish $\lambda\overline{x}.e$ from $\lambda\overline{x}.e'$, then the same store and the expression context $C[\lambda\overline{x}.[]]$ distinguish e from e'.

The opposite direction follows from Lemma A.2 as follows. For all stores s and expression contexts $C[]$ we have $C[e] \preceq C[(\lambda\overline{x}.e\ \overline{x})]$, and therefore:

$$\begin{array}{ll} & \langle s, C[e]\rangle \Downarrow \\ \iff & \langle s, C[(\lambda\overline{x}.e\ \overline{x})]\rangle \Downarrow \quad \text{(by Lemma A.2)} \\ \iff & \langle s, C[(\lambda\overline{x}.e'\ \overline{x})]\rangle \Downarrow \quad (\lambda\overline{x}.e \equiv_{\mathsf{std}} \lambda\overline{x}.e) \\ \iff & \langle s, C[e']\rangle \Downarrow \quad \text{(by Lemma A.2)} \end{array}$$

□

A.2 Proof of Theorem 3.10 ($(\equiv) = (\equiv_{\mathsf{std}})$)

Proof. Let:

$$e \equiv_{\mathsf{std}} e'$$

or equivalently, by Theorem 3.2:

$$\begin{array}{l} \exists\overline{x}\colon \{\overline{x}\} \supseteq FV(e) \cup FV(e'). \\ \quad \lambda\overline{x}.e \equiv_{\mathsf{std}} \lambda\overline{x}.e' \end{array}$$

or equivalently, by the definition of $\equiv_{\mathsf{std}}$:

$$\begin{array}{l} \exists\overline{x}\colon \{\overline{x}\} \supseteq FV(e) \cup FV(e'). \\ \quad \forall s, C\colon \langle s, C[\lambda\overline{x}.e]\rangle, \langle s, C[\lambda\overline{x}.e']\rangle \\ \qquad \text{are well-formed configurations.} \\ \quad \langle s, C[\lambda\overline{x}.e]\rangle \Downarrow \iff \langle s, C[\lambda\overline{x}.e']\rangle \Downarrow \end{array}$$

or equivalently, by choosing the appropriate d, $\overline{y}$, and $\overline{l}$ for the forward direction (and by choosing C for the reverse direction), such that $C[y_0] = [\overline{l/y}]d$:

$$\begin{array}{l} \exists\overline{x}\colon \{\overline{x}\} \supseteq FV(e) \cup FV(e'). \\ \quad \forall s, d, \overline{l}, \overline{y}\ :\ (Locs(d) = \emptyset) \\ \qquad \wedge\ (\{\overline{l}\} \subseteq Dom(s)) \\ \qquad \wedge\ (FV(d) \subseteq \{y_0, \overline{y}\}). \\ \quad \langle s, [\lambda\overline{x}.e/y_0, \overline{l/y}]d\rangle \Downarrow \iff \langle s, [\lambda\overline{x}.e/y_0, \overline{l/y}]d\rangle \Downarrow \end{array}$$

or equivalently, by Definition 3.8:

$$\begin{array}{l} \exists\overline{x}\colon \{\overline{x}\} \supseteq FV(e) \cup FV(e'). \\ \quad \forall s, d, \overline{l}, \overline{y}\ :\ (Locs(d) = \emptyset) \\ \qquad \wedge\ (\{\overline{l}\} \subseteq Dom(s)) \\ \qquad \wedge\ (FV(d) \subseteq \{y_0, \overline{y}\}). \\ \quad \exists R\ :\ ((\lambda\overline{x}.e, \lambda\overline{x}.e') \in R) \\ \qquad \wedge\ (\overline{(l, l)} \in R) \\ \qquad \wedge\ ((s, s, R) \in \equiv) \end{array}$$

or equivalently, by Definition 3.9:

$$e \equiv e'$$

□

A.3 Completeness

We first show some useful properties of $\equiv$.

Lemma A.3. *For any state $(s, s', R) \in \equiv$:*

1. *If $(u, u') \in R$ and $(v, v') \in R$, such that $\langle s, (u\ v)\rangle \Downarrow \langle t, w\rangle$ and $\langle s', (u'\ v')\rangle \Downarrow \langle t', w'\rangle$, then $(t, t', R \cup \{(w, w')\}) \in \equiv$.*
2. *If $((v_1, \ldots, v_n), (v'_1, \ldots, v'_n)) \in R$ then $(s, s', R \cup \{(v_i, v'_i)\}) \in \equiv$.*
3. *If l, l' are locations with $l \notin Dom(s)$ and $l' \notin Dom(s')$, then $(s[l \mapsto 0], s'[l' \mapsto 0], R \cup \{(l, l')\} \cup \{(0, 0)\}) \in \equiv$.*
4. *If $(l, l') \in R$ then $(s, s', R \cup \{(s(l), s'(l'))\}) \in \equiv$.*
5. *If $(l, l') \in R$ and $(v, v') \in R^*_{\mathsf{val}}$ then $(s[l \mapsto v], s'[l' \mapsto v'], R \cup \{(v, v')\}) \in \equiv$.*
6. *If $(v, v') \in R^*_{\mathsf{val}}$ then $(s, s', R \cup \{(v, v')\}) \in \equiv$.*

Proof. Immediate from the definition of contextual equivalence. For case 1, if there was a context e that could distinguish value w from w', then the context $\texttt{let}\ x = (y\ z)\ \texttt{in}\ e$ would distinguish the values u, v from u', v'. Similarly, for the other cases we consider the contexts $\texttt{let}\ x = \#_i(y)\ \texttt{in}\ e$, $\nu x.e$, $\texttt{let}\ x = !y\ \texttt{in}\ e$, and $\texttt{let}\ x = ((y := d))\ \texttt{in}\ e$. The last part follows from Lemma 3.6. □

Proof of Theorem 5.4 (Completeness)

Proof. We prove that $\equiv$ is a bisimulation by showing that it satisfies the conditions for a bisimulation. Conditions 1, 2 of the bisimulation are immediately satisfied by clauses 1, 2 of the definition of $\equiv$. Condition 3 follows if we apply clause 3 of the definition of $\equiv$ to all contexts that use the related values according to their kind (e.g. they apply them if they are abstractions). Condition 4 follows if we consider the context $\texttt{if}\ x = c\ \texttt{then}\ ()\ \texttt{else}\ \bot$. Condition 5 follows from Lemma A.3 (2). Condition 6 follows from Lemma A.3 (3). Condition 7 follows from Lemma A.3 (4, 5). Condition 8 follows from Lemma A.3 (6) for constructing the arguments, and Lemma A.3 (1) and clause 3 of the definition of $\equiv$ to get $(s, s', R) \vdash_{\equiv} (\lambda x.e\ v) \sqsubseteq_{k+1} (\lambda x.e'\ v')$ and $(s, s', R) \vdash_{\equiv} (\lambda x.e\ v) \sqsupseteq_{k+1} (\lambda x.e'\ v')$. □

A.4 Soundness

Lemma A.4 (Pre-soundness). *Let $\mathcal{X}$ be a bisimulation. Then for any $(s, s', R) \in \mathcal{X}$ we have:*

1. *For all $(e, e') \in R^*$, $\langle s, e\rangle \Downarrow \langle t, w\rangle \iff \langle s', e\rangle \Downarrow \langle t', w'\rangle$,*
2. *and if they both terminate, there exists a relation $Q \supseteq R$ such that $(t, t', Q) \in \mathcal{X}$, and $(w, w') \in Q^*_{\mathsf{val}}$.*

Proof. We prove the forward direction for the first clause by induction on the derivation of the left-hand side of clause 1; the opposite direction follows from symmetry. Our induction hypothesis is:

$$\begin{aligned}((s, s', R) \in \mathcal{X}) \wedge ((e, e') \in R^*) \wedge \langle s, e\rangle \Downarrow^{<k} \langle t, w\rangle \qquad (4)\\ \implies (\exists t', w', Q : \langle s', e'\rangle \Downarrow \langle t', w'\rangle\\ \wedge (Q \supseteq R) \wedge ((t, t', Q) \in \mathcal{X})\\ \wedge ((w, w') \in Q^*_{\mathsf{val}}))\end{aligned}$$

We assume the induction hypothesis for derivations of height less than k, and we will prove it for derivations less than $k + 1$; we proceed by cases on e:

- $e = w$, where w is a value. By the definition of R^*, e' must also be a value. Thus (4) is immediately satisfied, since $t' = s'$, $w' = e'$, and $Q = R$.

- $e = (e_1\ e_2)$. By the definition of R^*, $e' = (e'_1\ e'_2)$, and $(e_i, e'_i) \in R^*$, where $i = 1, 2$. The big-step rule for application (EVAL-APP) on the left-hand side gives:

$$\frac{\langle s, e_1\rangle \Downarrow^{<k} \langle s_1, \lambda x.e_3\rangle \quad \langle s_1, e_2\rangle \Downarrow^{<k} \langle s_2, v\rangle \quad \langle s_2, [v/x]e_3\rangle \Downarrow^{<k} \langle t, w\rangle}{\langle s, (e_1\ e_2)\rangle \Downarrow^{<k+1} \langle t, w\rangle} \qquad (5)$$

By the first premise of (5) and the induction hypothesis at e_1 and (s, s', R) we get:

$$\begin{aligned}\exists s'_1, w'_1, R_1 : \langle s', e'_1\rangle \Downarrow \langle s'_1, w'_1\rangle\\ \wedge (R_1 \supseteq R) \wedge ((s_1, s'_1, R_1) \in \mathcal{X})\\ \wedge ((\lambda x.e_3, w'_1) \in {R_1}^*_{\mathsf{val}})\end{aligned}$$

By the second premise of (5) and the induction hypothesis at e_2 and (s_1, s'_1, R_1) we get:

$$\begin{aligned}\exists s'_2, v', R_2 : \langle s'_1, e'_2\rangle \Downarrow \langle s'_2, v'\rangle\\ \wedge (R_2 \supseteq R_1) \wedge ((s_2, s'_2, R_2) \in \mathcal{X})\\ \wedge ((v, v') \in {R_2}^*_{\mathsf{val}})\end{aligned}$$

For the third premise of EVAL-APP we distinguish two cases:

Case 1: There exist e'_3, d, $\overline{(u, u')} \in R_1$, such that $FV(d) \subseteq \{x, \overline{y}\}$, $Locs(d) = \emptyset$, $w'_1 = \lambda x.e'_3$, $e_3 = \overline{[u/y]}d$, $e'_3 = \overline{[u'/y]}d$.

By the properties of β-substitution, and because ${R_1}^* \subseteq {R_2}^*$ and $(v, v') \in {R_2}^*_{\mathsf{val}}$ we have $([v/x]e_3, [v'/x]e'_3) \in {R_2}^*$, and thus we can apply the induction hypothesis at $[v/x]e_3$ and (s_2, s'_2, R_2):

$$\begin{aligned}\exists t', w', Q : \langle s'_2, [v'/x]e'_3\rangle \Downarrow \langle t', w'\rangle\\ \wedge (Q \supseteq R_2) \wedge ((t, t', Q) \in \mathcal{X})\\ \wedge ((w, w') \in Q^*_{\mathsf{val}})\end{aligned}$$

Case 2: $\lambda x.e_3 = [\lambda x.e_3/y]y, (\lambda x.e_3, w') \in R_1 \subseteq R_2$. By Condition 3 we get $w'_1 = \lambda x.e'_3 = [\lambda x.e'_3/y]y$. We have:

$$\langle s_2, [v/x]e_3\rangle \Downarrow^{<k} \langle t, w\rangle \iff \langle s_2, (\lambda x.e_3\ v)\rangle \Downarrow^{<k+1} \langle t, w\rangle \qquad (6)$$

Because $(s_2, s'_2, R_2) \in \mathcal{X}$, $(\lambda x.e_3, \lambda x.e'_3) \in R_2$, and $(v, v') \in {R_2}^*_{\mathsf{val}}$, we get by Condition 8 of the definition of bisimulation:

$$IH^L_{\mathcal{X}}(k) \implies (s_2, s'_2, R_2) \vdash_{\mathcal{X}} (\lambda x.e_3\ v) \sqsubseteq_{k+1} (\lambda x.e'_3\ v')$$

$IH^L_{\mathcal{X}}(k)$ is true by the induction hypothesis. Therefore by (6) and the definition of k-approximation we get:

$$\begin{aligned}\exists t', w', Q : \langle s'_2, (\lambda x.e'_3\ v)\rangle \Downarrow \langle t', w'\rangle\\ \wedge (Q \supseteq R_2) \wedge ((t, t', Q) \in \mathcal{X})\\ \wedge ((w, w') \in Q^*_{\mathsf{val}})\end{aligned}$$

Thus for both of the above cases, the EVAL-APP rule for the right-hand side gives:

$$\frac{\langle s', e'_1\rangle \Downarrow \langle s'_1, \lambda x.e'_3\rangle \quad \langle s'_1, e'_2\rangle \Downarrow \langle s'_2, v'\rangle \quad \langle s'_2, [v'/x]e'_3\rangle \Downarrow \langle t', w'\rangle}{\langle s', (e'_1\ e'_2)\rangle \Downarrow \langle t', w'\rangle}$$

and also $Q \supseteq R_2$, $(t, t', Q) \in \mathcal{X}$, $(w, w') \in Q^*_{\mathsf{val}}$.

- $e = op(e_1, \ldots, e_n)$. By definition of R^*, $e' = op(e'_1, \ldots, e'_n)$ and $(e_i, e'_i) \in R^*_{\mathsf{val}}$, $1 \le i \le n$. The big-step rule for primitive operators (EVAL-OP) on the left-hand side gives:

$$\frac{\langle s, e_1\rangle \Downarrow^{<k} \langle s_1, c_1\rangle \quad \ldots \quad \langle s_{n-1}, e_n\rangle \Downarrow^{<k} \langle s_n, c_n\rangle \quad op^{arith}(c_1, \ldots, c_n) = c}{\langle s, op(e_1, \ldots, e_n)\rangle \Downarrow^{<k+1} \langle s_n, c\rangle} \qquad (7)$$

By premises of (7) and the induction hypothesis at e_i and $(s_{i-1}, s'_{i-1}, R_{i-1}) \in \mathcal{X}$ for all $1 \leq i \leq n$ we get:

$$\begin{array}{l}\exists s'_i, R_i \,:\, \langle s'_{i-1},\, e'_i\rangle \Downarrow \langle s'_i,\, w'_i\rangle \\ \qquad \wedge\, (R_i \supseteq R_{i-1}) \wedge ((s_i, s'_i, R_i) \in \mathcal{X}) \\ \qquad \wedge\, ((c_i, w'_i) \in R_{i\,\mathsf{val}}^*)\end{array}$$

where $R_0 = R$, $s_0 = s$, $s'_0 = s'$.

By Conditions 3, 4, we get $w_i = c_i$, for all $1 \leq i \leq n$. Thus by applying EVAL-OP on the right-hand side we get:

$$\frac{\langle s',\, e'_1\rangle \Downarrow \langle s'_1,\, c_1\rangle \quad \ldots \quad \langle s'_{n-1},\, e'_n\rangle \Downarrow \langle s'_n,\, c_n\rangle \quad op^{arith}(c_1, \ldots, c_n) = c}{\langle s',\, op(e'_1, \ldots, e'_n)\rangle \Downarrow \langle s'_n,\, c\rangle}$$

and $R_n \supseteq R$, $(s_n, s'_n, R_n) \in \mathcal{X}$, $(c, c) \in R_{n\,\mathsf{val}}^*$.

- $e = \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$. This case follows from straightforward applications of the induction hypothesis at e_1, e_2, and e_3.

- $e = (e_1, \ldots, e_n)$. This case also follows from applications of the induction hypothesis at each of the e_i ($1 \leq i \leq n$).

- $e = \#_i(e_0)$. By the definition of R^* we have $e' = \#_i(e'_0)$ and $(e_0, e'_0) \in R^*$. The big-step rule for projection (EVAL-PROJ) on the left-hand side gives:

$$\frac{\langle s,\, e_0\rangle \Downarrow^{<k} \langle t,\, (w_1, \ldots, w_n)\rangle \quad 1 \leq i \leq n}{\langle s,\, \#_i(e_0)\rangle \Downarrow^{<k+1} \langle t,\, w_i\rangle}$$

By the induction hypothesis at e_0 and (s, s', R) we get:

$$\begin{array}{l}\exists t', w', Q \,:\, \langle s',\, e'_0\rangle \Downarrow \langle t',\, w'\rangle \\ \qquad \wedge\, (Q \supseteq R) \wedge ((t, t', Q) \in \mathcal{X}) \\ \qquad \wedge\, (((w_1, \ldots, w_n), w') \in Q_{\mathsf{val}}^*)\end{array}$$

Case 1: $w' = (w'_1, \ldots, w'_n)$, and for all $1 \leq j \leq n$, $(w_j, w'_j) \in Q_{\mathsf{val}}^*$. By the evaluation rule EVAL-PROJ, we get for the right-hand side:

$$\frac{\langle s',\, e'_0\rangle \Downarrow \langle t',\, (w'_1, \ldots, w'_n)\rangle \quad 1 \leq i \leq n}{\langle s',\, \#_i(e'_0)\rangle \Downarrow \langle t',\, w'_i\rangle}$$

and also $Q \supseteq R$, $(t, t', Q) \in \mathcal{X}$, $(w_i, w'_i) \in Q_{\mathsf{val}}^*$.

Case 2: $(w_1, \ldots, w_n) = [(w_1, \ldots, w_n)/y]y$, $w' = [w'/y]y$, and $((w_1, \ldots, w_n), w') \in Q$. By Conditions 3 and 5 of the definition of bisimulation we get:

$$\begin{array}{l}\exists w'_1, \ldots, w'_n, P \,:\, (w' = (w'_1, \ldots, w'_n)) \\ \qquad \wedge\, (P \supseteq Q) \wedge ((w_i, w'_i) \in P_{\mathsf{val}}^*)\end{array}$$

Again the evaluation rule EVAL-PROJ for the right-hand side gives:

$$\frac{\langle s',\, e'_0\rangle \Downarrow \langle t',\, (w'_1, \ldots, w'_n)\rangle \quad 1 \leq i \leq n}{\langle s',\, \#_i(e'_0)\rangle \Downarrow \langle t',\, w'_i\rangle}$$

and also $P \supseteq Q \supseteq R$, $(t, t', P) \in \mathcal{X}$, $(w_i, w'_i) \in P_{\mathsf{val}}^*$.

- $e = \nu x.e_0$. By the definition of R^* we have $e' = \nu x.e'_0$ and $(e_0, e'_0) \in R^*$. The evaluation rule for reference allocation (EVAL-NEW), gives for the left-hand side:

$$\frac{\langle s[l \mapsto 0],\, [l/x]e_0\rangle \Downarrow^{<k} \langle t,\, w\rangle \quad l \notin Dom(s)}{\langle s,\, \nu x.\, e_0\rangle \Downarrow^{<k+1} \langle t,\, w\rangle}$$

We choose $l' \notin Dom(s')$. From Condition 6 of the definition of bisimulation we get:

$$\exists Q \supseteq R \cup \{(l, l')\} \colon (s[l \mapsto 0], s'[l' \mapsto 0], Q) \in \mathcal{X}$$

From the induction hypothesis at $[l/x]e_0$ and $(s[l \mapsto 0], s'[l' \mapsto 0], Q) \in \mathcal{X}$ we get:

$$\begin{array}{l}\exists t', w', P \,:\, \langle s'[l' \mapsto 0],\, [l'/x]e'_0\rangle \Downarrow \langle t',\, w'\rangle \\ \qquad \wedge\, (P \supseteq Q) \wedge ((t, t', P) \in \mathcal{X}) \\ \qquad \wedge\, ((w, w') \in P_{\mathsf{val}}^*)\end{array}$$

Combining these into EVAL-NEW, we get for the right-hand side:

$$\frac{\langle s'[l' \mapsto 0],\, [l'/x]e'_0\rangle \Downarrow \langle t',\, w'\rangle \quad l' \notin Dom(s')}{\langle s',\, \nu x.\, e'_0\rangle \Downarrow \langle t',\, w'\rangle}$$

- $e = !e_0$. By the definition of R^*, we have: $e' = !e'_0$ and $(e_0, e'_0) \in R^*$. The EVAL-DEREF evaluation rule gives for the left-hand side:

$$\frac{\langle s,\, e_0\rangle \Downarrow^{<k} \langle t,\, l\rangle \quad l \in Dom(t) \quad t(l) = w}{\langle s,\, !e_0\rangle \Downarrow^{<k+1} \langle t,\, w\rangle}$$

From the induction hypothesis at e_0 and (s, s', R) we get:

$$\begin{array}{l}\exists t', w'_0, P \,:\, \langle s',\, e'_0\rangle \Downarrow \langle t',\, w'_0\rangle \\ \qquad \wedge\, (Q \supseteq R) \wedge ((t, t', Q) \in \mathcal{X}) \wedge ((l, w'_0) \in Q_{\mathsf{val}}^*)\end{array}$$

By the definition of Q_{val}^*, and Conditions 2, 3, and 7 (a) of the definition of bisimulation we have $(l, w'_0) \in Q$, and there exists l' such that $w'_0 = l'$, $l' \in Dom(t')$, and there exists $P \supseteq Q$ such that $((t(l), t'(l')) \in P_{\mathsf{val}}^*)$ and $(t, t', P) \in \mathcal{X}$. Therefore, by EVAL-DEREF, we get for the right-hand side:

$$\frac{\langle s',\, e'_0\rangle \Downarrow \langle t',\, l'\rangle \quad l' \in Dom(t') \quad t'(l') = w'}{\langle s',\, !e'_0\rangle \Downarrow \langle t',\, w'\rangle}$$

and also $P \supseteq Q \supseteq R$, $(t, t', P) \in \mathcal{X}$, $(w, w') \in P_{\mathsf{val}}^*$.

- $e = (e_1 := e_2)$. By the definition of R^* we have $e' = (e'_1 := e'_2)$, and $(e_i, e'_i) \in R^*$, and $i = 1, 2$. From the EVAL-ASSIGN evaluation rule we get for the left-hand side:

$$\frac{\langle s,\, e_1\rangle \Downarrow^{<k} \langle s_1,\, l\rangle \quad l \in Dom(s_1) \quad \langle s_1,\, e_2\rangle \Downarrow^{<k} \langle t,\, w\rangle}{\langle s,\, (e_1 := e_2)\rangle \Downarrow^{<k+1} \langle t[l \mapsto w],\, ()\rangle}$$

By the induction hypothesis at e_1 and $(s, s', R) \in \mathcal{X}$ we get:

$$\begin{array}{l}\exists s'_1, w'_1, R_1 \,:\, \langle s',\, e'_1\rangle \Downarrow \langle s'_1,\, w'_1\rangle \\ \qquad \wedge\, (R_1 \supseteq R) \wedge ((s_1, s'_1, R_1) \in \mathcal{X}) \\ \qquad \wedge\, ((l, w'_1) \in R_{1\,\mathsf{val}}^*)\end{array}$$

By the definition of $R_{1\,\mathsf{val}}^*$, and Conditions 2, and 3, of the definition of bisimulation we get that $(l, w'_1) \in R_1$, and there exists l' such that $w'_1 = l'$, and $l' \in Dom(s'_1)$. Applying the induction hypothesis again at e_2 and (s_1, s'_1, R_1) gives:

$$\begin{array}{l}\exists t', w', Q \,:\, \langle s'_1,\, e'_2\rangle \Downarrow \langle t',\, w'\rangle \\ \qquad \wedge\, (Q \supseteq R_1) \wedge ((t, t', Q) \in \mathcal{X}) \wedge ((w, w') \in Q_{\mathsf{val}}^*)\end{array}$$

And by the EVAL-ASSIGN rule we get:

$$\frac{\langle s',\, e'_1\rangle \Downarrow \langle s'_1,\, l'\rangle \quad l' \in Dom(s'_1) \quad \langle s'_1,\, e'_2\rangle \Downarrow \langle t',\, w'\rangle}{\langle s',\, (e'_1 := e'_2)\rangle \Downarrow \langle t'[l' \mapsto w'],\, ()\rangle}$$

From Lemma 2.3 we have: $l \in Dom(t)$, $l' \in Dom(t')$. Moreover, we know that $(l, l') \in R_1 \subseteq Q$, $(w, w') \in Q_{\mathsf{val}}^*$, and $(t, t', Q) \in \mathcal{X}$. Thus, by Condition 7(b) of the definition of bisimulation we get:

$$(t[l \mapsto w], t'[l' \mapsto w'], Q) \in \mathcal{X}$$

□

Proof of Theorem 5.5 (Soundness)

Proof. To prove soundness, we need to show that $\sim$ satisfies the conditions of $\equiv$. The first two conditions are trivially satisfied, because they are also part of the definition of bisimulation. Condition 3 follows from Lemma A.4. □