A Theory of Non-monotone Memory (Or: Contexts for `free`)*

Eijiro Sumii

Tohoku University
sumii@ecei.tohoku.ac.jp

Abstract. We develop a general method of proving contextual properties—including (but not limited to) observational equivalence, space improvement, and memory safety *under arbitrary contexts*—for programs in untyped call-by-value λ-calculus with first-class, higher-order references (`ref`, := and !) and deallocation (`free`). The method significantly generalizes Sumii et al.'s environmental bisimulation technique, and gives a sound and complete characterization of each proved property, in the sense that the "bisimilarity" (the largest set satisfying the bisimulation-like conditions) equals the set of terms with the property to be proved. We give examples of contextual properties concerning typical data structures such as linked lists, binary search trees, and directed acyclic graphs with reference counts, all with deletion operations that release memory.

This shows the scalability of the environmental approach from contextual equivalence to other binary relations (such as space improvement) and unary predicates (such as memory safety), as well as to languages with non-monotone store, where Kripke-style logical relations have difficulties.

1 Introduction

1.1 Background

Memory management is tricky, be it manual or automatic. Manual memory management is notoriously difficult, leading to memory leaks and segmentation faults (or, even worse, security holes). Automatic memory management is usually more convenient. Still, real programs often suffer from performance problems—in terms of both memory and time—due to automatic memory management, and require manual tuning. In addition, implementing memory management routines—such as memory allocators and garbage collectors—is even harder than writing programs that use them.

To address these problems, various theories for safe memory management have been developed, including linear types [17], regions [16], and the capability calculus [6], just to name a few. These approaches typically conduct a sound and efficient static analysis—often based on types—on programs, and guarantee their memory safety. However, since static analyses are necessarily incomplete

* Extended abstract with appendices online [13].

G. Castagna (Ed.): ESOP 2009, LNCS 5502, pp. 237–251, 2009.

$$
\begin{array}{lll}
dag & = & \nu z := \mathtt{null}; \langle addn, deln, gc \rangle \\
addn & = & \lambda x.\, \lambda p.\, x + 0; map(\lambda y.\, y + 0)p; \\
 & & incr_x(!z)p; \nu n := \langle x, \mathtt{true}, 0, p, !z \rangle; z := n \\
incr_x & = & \mathtt{fix}\ f(n).\, \lambda p.\, \mathtt{ifnull}\ n\ \mathtt{then}\ \langle\rangle\ \mathtt{else} \\
 & & \mathtt{if}\ \#_1(!n) \stackrel{int}{=} x\ \mathtt{then}\ \mathtt{diverge}\ \mathtt{else} \\
 & & \mathtt{if}\ member(\#_1(!n))p\ \mathtt{then}\ \#_3^5(!n) \leftarrow \#_3(!n) + 1; f(n)(remove(\#_1(!n))p)\ \mathtt{else} \\
 & & f(\#_5(!n))p \\
deln & = & \lambda x.\, deln_x(!z) \\
deln_x & = & \mathtt{fix}\ g(n).\, \mathtt{ifnull}\ n\ \mathtt{then}\ \langle\rangle\ \mathtt{else} \\
 & & \mathtt{if}\ \#_1(!n) \stackrel{int}{=} x\ \mathtt{then}\ \#_2^5(!n) \leftarrow \mathtt{false}\ \mathtt{else} \\
 & & g(\#_5(!n)) \\
gc & = & \lambda x.\, z := decr(!z)[\,] \\
decr & = & \mathtt{fix}\ h(n).\, \lambda p.\, \mathtt{ifnull}\ n\ \mathtt{then}\ \mathtt{null}\ \mathtt{else} \\
 & & \mathtt{if}\ member(\#_1(!n))p\ \mathtt{then}\ \#_3^5(!n) \leftarrow \#_3(!n) - 1; h(n)(remove(\#_1(!n))p)\ \mathtt{else} \\
 & & \mathtt{if}\ \#_2(!n) \vee \#_3(!n) > 0\ \mathtt{then}\ \#_5^5(!n) \leftarrow h(\#_5(!n))p; n\ \mathtt{else} \\
 & & h(\#_5(!n))(append(\#_4(!n))p)\ \mathtt{before}\ \mathtt{free}(n)
\end{array}
$$

Fig. 1. Directed acyclic graph with garbage collection by reference counting

in the sense that some safe programs are rejected, the programs usually have to be written in a style that is accepted by the analysis.

1.2 Our Contributions

In this paper, we develop a different approach, originating from Sumii et al.'s environmental bisimulations [7, 12, 14, 15]. Unlike most static analyses, our method is not fully automated, but is (sound and) complete in the sense that all (and only) safe programs can potentially be proved safe. Moreover, it guarantees memory safety *under any context*, even if the context—or, in fact, the whole language—is untyped.

For instance, consider the triple *dag* in Figure 1, which implements a directed acyclic graph object with addition, deletion, and garbage collection by reference counting. (Details of this implementation are explained in Section 6. The formal syntax and semantics of our language are given in Section 3.) To verify the memory safety of this implementation, it makes no sense to consider the triple by itself; rather, we must consider all possible uses of it, i.e., put it under arbitrary contexts. Our method gives such a proof in many examples.

Because our method is based on a relational technique (namely, bisimulations), we can also prove binary properties such as observational equivalence, in addition to unary properties such as memory safety. Furthermore, we can prove more general binary properties like "the memory usage (i.e., number of locations) is the same on the left hand side and the right" or "the left hand side uses less memory than the right." Again, such properties between programs are preserved by contexts in the language, like contextual equivalence [10].

1.3 Our Approach

Environmental bisimulations. Suppose that we want to prove the equivalence of two programs e and e'.[1] The basic idea of our approach is to consider the set X of every possible "configuration" of the programs. A configuration takes one of the two forms: $(\mathcal{R}, s \triangleright e, s' \triangleright e')$ and $(\mathcal{R}, s, s')$. The former means that the compared programs e and e' are running under stores s and s', respectively. The latter means that the programs have stopped with stores s and s'. In both forms, $\mathcal{R}$ is a binary relation on values and represents the *knowledge* of a context, called an environment.

For instance, suppose that we have a configuration $(\mathcal{R}, s \triangleright e, s' \triangleright e')$ in X. (Typically, $\mathcal{R}$ is empty at first.) If $s \triangleright e$ reduces to $t \triangleright d$ in one step according to the operational semantics of the language, then it must be that $s' \triangleright e'$ also reduces to some $t' \triangleright d'$ in some number of steps, and the new configuration $(\mathcal{R}, t \triangleright d, t' \triangleright d')$ belongs to X again. Knowledge $\mathcal{R}$ does not change yet, because the context cannot learn anything from these internal transitions.

Now, suppose $(\mathcal{R}, s \triangleright e, s' \triangleright e') \in X$ and e has stopped running, i.e., e is a value v. Then, $s' \triangleright e'$ must also converge to some $t' \triangleright w'$, and the context learns the resulting values v and w'. Thus, $\mathcal{R}$ is extended with the value pair (v, w'), and $(\mathcal{R} \cup \{(v, w')\}, s, t')$ must belong to X.

Once the compared programs have stopped, the context can make use of elements from its knowledge to make more observations. For example, suppose $(\mathcal{R}, s, s') \in X$ and $(\ell, \ell') \in \mathcal{R}$. This means that location ℓ (resp. ℓ') is known to the context on the left (resp. right) hand side. If $s = t \uplus \{\ell \mapsto v\}$ and $s' = t' \uplus \{\ell' \mapsto v'\}$ (where $\_ \uplus \{\_ \mapsto \_\}$ denotes store extension), then the context can read the contents v (resp. v') of ℓ (resp. ℓ') on the left (resp. right) hand side, and add them to its knowledge, requiring $(\mathcal{R} \cup \{(v, v')\}, s, s') \in X$.

Or, the contents can be updated with any values composed from the knowledge of the context. That is, for any $(w, w') \in \mathcal{R}^\star$, we require $(\mathcal{R}, t \uplus \{\ell \mapsto w\}, t' \uplus \{\ell' \mapsto w'\}) \in X$. Here, $\mathcal{R}^\star$ is the *context closure* of $\mathcal{R}$ and denotes the set of (pairs of) terms that can be composed from values in $\mathcal{R}$. Formally, it is defined as

$$\mathcal{R}^\star = \{([v_1, \ldots, v_n / x_1, \ldots, x_n]e, [v'_1, \ldots, v'_n / x_1, \ldots, x_n]e) \mid (v_1, v'_1), \ldots, (v_n, v'_n) \in \mathcal{R},\ fv(e) \subseteq \{x_1, \ldots, x_n\},\ loc(e) = \emptyset\}$$

where $fv(e)$ is the set of free variables in e and $loc(e)$ is the set of locations that appear in e.

Moreover, the context can also deallocate locations it knows, or allocate fresh ones. For the former case, we require $(\mathcal{R}, t, t') \in X$ for any $(\mathcal{R}, t \uplus \{\ell \mapsto v\}, t' \uplus \{\ell' \mapsto v'\}) \in X$ with $(\ell, \ell') \in \mathcal{R}$. For the latter case, $(\mathcal{R} \cup \{\ell, \ell\}, t \uplus \{\ell \mapsto v\}, t' \uplus \{\ell \mapsto v'\}) \in X$ is required for any $(\mathcal{R}, t, t') \in X$ with fresh ℓ and $(v, v') \in \mathcal{R}^\star$.

Of course, there are also conditions on values other than locations. For instance, if $(\mathcal{R}, s, s') \in X$ and $(\lambda x.\, e, \lambda x.\, e') \in \mathcal{R}$, then $(\mathcal{R}, s \triangleright (\lambda x.\, e)v, s' \triangleright$

[1] Throughout this paper, we often (though not always) follow the notational convention that meta-variables with $'$ are used for objects on the right hand side of binary relations, and ones without for the left hand side (and unary relations).

$(\lambda x.\, e')v') \in X$ is required for any $(v, v') \in R^\star$, because the context can apply any functions it knows ($\lambda x.\, e$ and $\lambda x.\, e'$) to any arguments it can compose (v and v').

Congruence of environmental bisimilarity. As we shall prove, the largest set X satisfying the above conditions—which exists because all of them are monotone on X—is "contextual" in the following sense:

- If a configuration $(\mathcal{R}, s \triangleright e, s' \triangleright e')$ is in X, then its context-closed version $(\mathcal{R}^{\hat{\star}}, s \triangleright E[e], s' \triangleright E[e'])$ is also in X, for any location-free evaluation context E.
- If a configuration $(\mathcal{R}, s, s')$ is in X, then its context-closed version $(\mathcal{R}^{\hat{\star}}, s \triangleright e, s' \triangleright e')$ is also in X, for any $(e, e') \in \mathcal{R}^\star$.

Here, $\mathcal{R}^{\hat{\star}}$ denotes the restriction of $\mathcal{R}^\star$ to values.

The restriction to location-free evaluation contexts in the first item is *not* a limitation of our approach, as already shown in previous work [7, 15]: If one wants to prove the equivalence of e and e' under non-evaluation contexts, it suffices to prove the equivalence of $\lambda x.\, e$ and $\lambda x.\, e'$ (for fresh x) under evaluation contexts only. In addition, if a context needs access to some locations $\ell_1, \ldots, \ell_n$, just requiring $(\ell_1, \ell_1), \ldots, (\ell_n, \ell_n) \in \mathcal{R}$ is sufficient. Programs with free variables are not a problem, either: instead of open e and e', it suffices to consider $\lambda x_1. \ldots \lambda x_n.\, e$ and $\lambda x_1. \ldots \lambda x_n.\, e'$ for $\{x_1, \ldots x_n\} \supseteq fv(e) \cup fv(e')$.

Generalization to contextual relations. The above approach is not limited to the proof of contextual equivalence, but can be generalized for other binary relations as well. For example, if we add a condition "$|dom(s)| \leq |dom(s')|$ for any $(\mathcal{R}, s \triangleright e, s' \triangleright e') \in X$," then one can conclude that e uses fewer locations than e' under arbitrary (evaluation) contexts. In short, any predicate P on configurations can be added to the conditions of X while keeping it contextual, as long as P itself is contextual (i.e., preserved by contexts). It does not have to be a congruence (or even a pre-congruence), hence the term "contextual" rather than "congruent" (or pre-congruent).

Contextual predicates and memory safety. In fact, there is no reason why the proved contextual relations have to be binary. Rather, they can be of arbitrary arity. In particular, the arity can be 1, meaning unary predicates. To obtain conditions for the unary version of X, we just have to remove everything that belongs to the "right hand side." Again, the resulting X is contextual as long as the predicate P itself is contextual.

A prominent example of such unary properties is memory safety. For proving memory safety under arbitrary contexts, let us first classify all locations into "private" and "public" ones. The intent is that private locations are kept secret to the program under consideration, whereas public locations are under the control of the context. Then, let $P(\mathcal{R}, s \triangleright e)$ be false if and only if e is immediately reading from, writing to, or deallocating a private location that is not in $dom(s)$, and $P(\mathcal{R}, s)$ be true if and only if any $\ell \in \mathcal{R}$ is public. Then, just as in the

binary case, we can prove that the largest X satisfying the bisimulation-like conditions is contextual. (Of course, here, we are not considering a congruence or an equivalence relation—or even a binary relation at all!—but the set X is still "bisimulation-like" in the sense that it involves co-induction and is contextual.)

Another example of unary contextual properties is an upper bound on the number of private locations. To be concrete, let $P(\mathcal{R}, s \triangleright e)$ and $P(\mathcal{R}, s)$ be true if and only if the number of private locations in $dom(s)$ is less than a constant c. Then, again, we can use our approach to prove that a term e allocates at most c private locations under arbitrary contexts that do not create private locations.

1.4 Overview of the Paper

The rest of this paper is structured as follows. Section 2 discusses some (not all, because of space constraints) related work. Section 3 defines our target language. Section 4 develops the binary version of our proof technique and Appendix A (available online [13]) gives examples (contextual relations between two multi-set implementations). In addition, Appendix B introduces an auxiliary "up-to" technique to simplify the proofs, with examples in Appendix C. Section 5 defines the unary version of our approach and Section 6 gives an example (directed acyclic graphs with garbage collection with reference counting). Appendix D gives another example (bucket sort). Section 7 concludes with future work.

2 Related Work

As stated above, our technique is rooted in Sumii et al.'s previous work on environmental bisimulations [7, 12, 14, 15]. In particular, our language and the binary version of our proof method (for contextual equivalence) is an extension of their environmental bisimulation for untyped call-by-value λ-calculus with references (`ref`, := and !) in [12, Section 4], enriched with deallocation (`free`). It is also similar to the language and bisimulation of [7], except that we adopt small-step reduction semantics while they used big-step evaluation semantics. However, the fact itself that the extension is possible is striking, especially because deallocation is known to be highly non-trivial in other approaches, including type-based analyses and logical relations. In addition, our generalization of their technique—from contextual equivalence to other properties such as memory safety—is entirely new.

Denotational semantics can be used to prove contextual equivalence of programs (see, for example, [9, pp. 77, 344]). In short, two programs are contextually equivalent if their denotations are the same (provided that the semantics is adequate, of course). However, it is known to be hard to develop fully abstract—i.e., equivalence preserving—denotational semantics for languages with local store [8], let alone full references or deallocation.

Logical relations are relations between (semantics of) programs defined by induction on their types, and can be used for proving properties like contextual equivalence and memory safety. Pitts and Stark [11] defined (binary) syntactic

logical relations—i.e., relations between the syntax of programs itself rather than their semantics—for a simply-typed call-by-value higher-order language with references to integers, and proved that they characterize contextual equivalence in this language. However, it is known to be hard to extend their result to languages with general references [2, 5] (references to arbitrary values, including functions and references themselves) or deallocation. In particular, the latter seems to break monotonicity (of the domain of a store), which is a crucial assumption of Kripke-style logical relations [9, p. 590] like theirs.

Ahmed [1, Chapter 7] defined (unary) step-indexed logical relations—i.e., relations defined by induction on the number of reduction steps instead of types—for a continuation-passing-style higher-order language with regions and their deallocation (like the capability calculus). In her definition, monotonicity is maintained by marking deallocated regions "dead" instead of removing them, thereby forbidding their reuse *at the type level* (still, region *handles* can be reused at the term level). Completeness is not discussed. Ahmed, Fluet, and Morrisett [3, 4] defined (unary) step-indexed logical relations in languages with linear types and deallocation. Their developments depend on the static guarantee by linear types. None of the work above considers contextual equivalence (or other binary properties).

3 The Language

The syntax of our language is given in Figure 2. It is a standard call-by-value λ-calculus extended with references and deallocation, in addition to first-order primitives (such as Boolean values and integer arithmetic) and tuples, which are added solely for the sake of convenience. The operational semantics is also standard and given in Figure 3 in the Appendices [13]. It is parametrized by the semantics of primitives, given as a partial function $[\![\_]\!]$ to constants from operations on constants.

A location ℓ^π is an atomic symbol that models a reference in ML (though it is untyped and deallocatable in our language) or a pointer in C (although our language omits pointer arithmetic for simplicity, it can easily be added by modeling the store as a finite map from locations to *arrays* of values). It has a "security level" $\top$ or $\bot$ to distinguish private and public locations, as outlined in the introduction. In what follows, we omit security levels when they are unimportant. We assume that there exist a countably infinite number of locations, both private and public. A special location $\mathtt{null}^\bot$ is reserved for representing a never allocated location. This treatment is just for the sake of simplicity of examples. We write $\mathit{loc}(e)$ for the set of locations that appear in e (except $\mathtt{null}^\bot$), and $\mathit{fv}(e)$ for the set of free variables in e. Note that there is no binder for locations in the syntax of our language.

Allocation $\nu x^\pi := e_1; e_2$ creates a fresh location ℓ^π of the specified security level π, initializes the contents with the value of e_1, binds the location to x, and executes e_2. (It is easy to separate allocation from initialization like $\nu x^\pi . e$, but the present form is more convenient for examples. In addition, we do not like to fix a single, arbitrary initial value of locations.) As outlined in the introduction,

$\pi, \rho ::=$	security level
$\top$	private
$\bot$	public

$d, e, C, D ::=$	term
x	variable
$\lambda x.\, e$	function
$e_1 e_2$	application
c	constant
$op(e_1, \ldots, e_n)$	primitive
$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	conditional branch
$\langle e_1, \ldots, e_n \rangle$	tupling
$\#_i(e)$	projection
ℓ^π	location
$\nu x^\pi := e_1; e_2$	allocation
$\texttt{free}(e)$	deallocation
$e_1 := e_2$	update
$!e$	dereference
$e_1 \overset{ptr}{=} e_2$	pointer equality

$u, v, w ::=$	value
$\lambda x.\, e$	function
c	constant
$\langle v_1, \ldots, v_n \rangle$	tuple
ℓ^π	location

$E, F ::=$	evaluation context
$[\,]$	hole
Ee	application (left)
vE	application (right)
$op(v_1, \ldots, v_m, E, e_1, \ldots, e_n)$	primitive
$\texttt{if}\ E\ \texttt{then}\ e_1\ \texttt{else}\ e_2$	conditional branch
$\langle v_1, \ldots, v_m, E, e_1, \ldots, e_n \rangle$	tupling
$\#_i(E)$	projection
$\nu x^\pi := E; e$	allocation
$\texttt{free}(E)$	deallocation
$E := e$	update (left)
$v := E$	update (right)
$!E$	dereference
$E \overset{ptr}{=} e$	pointer equality (left)
$v \overset{ptr}{=} E$	pointer equality (right)

Fig. 2. Syntax

our intent is to disallow contexts to allocate private locations. This restriction is a mere matter of a proof technique, and does not limit the computational power of contexts at runtime. In other words, we can always divide locations so that all locations under the control of a context are public.

Deallocation $\texttt{free}(e)$ releases memory and lets it be reused later. Update $e_1 := e_2$ overwrites the contents of a location.

Pointer equality $e_1 \overset{ptr}{=} e_2$ compares locations themselves (not their contents). We do not use it in our examples (except for comparison with $\texttt{null}^\bot$), but it is necessary for contexts to have a realistic observational power. If both locations are live, their equality can be tested by writing to one of the locations and reading from the other. However, this is not possible when either (or both) of them is "dead," i.e., already deallocated.

Throughout this paper, we focus on properties of closed terms and values only. (This is not a limitation, as explained in the introduction.) Thus, we can model a (possibly multi-hole) context C just by a term e with free variables $x_1, \ldots, x_n$, and a context application $C[e_1, \ldots, e_n]$ by a variable substitution $[e_1, \ldots, e_n / x_1, \ldots, x_n]e$. For this reason, we use meta-variables C and D for terms that are used for representing contexts. By convention, we implicitly assume that terms denoted by capital letters are location-free (except for $\texttt{null}^\bot$) and do not include private allocation $\nu x^\top$.

For brevity, we use various syntactic sugar. We write $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ for $(\lambda x.\, e_2)e_1$, and $e_1; e_2$ for $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ where x does not appear free in e_2. Recursive function $\texttt{fix}\ f(x).\, e$ is defined as (the value of) $Y(\lambda f.\, \lambda x.\, e)$ by

using some call-by-value fixed-point operator Y as usual. As in Standard ML, e_1 `before` e_2 denotes `let` $x = e_1$ `in` $e_2; x$, again with x not free in e_2. We also write $e_1 \wedge e_2$ for `if` e_1 `then` e_2 `else false` and $e_1 \vee e_2$ for `if` e_1 `then true` `else` e_2. Note that these conjunction and disjounction operators are not symmetric, as in most programming languages with side effects or divergence. As in Objective Caml, `if` e_1 `then` e_2 abbreviates `if` e_1 `then` e_2 `else` $\langle\rangle$, where $\langle\rangle$ is the nullary tuple. Moreover, `ifnull` e_1 `then` e_2 `else` e_3 abbreviates `if` $e_1 \stackrel{ptr}{=} \mathtt{null}^{\perp}$ `then` e_2 `else` e_3. Finally, $\#^i_j(!e_1) \leftarrow e_2$ stands for `let` $x = e_1$ `in` $x := \langle \#_1(!x), \ldots, \#_{j-1}(!x), e_2, \#_{j+1}(!x), \ldots, \#_i(!x)\rangle$.

We give higher precedence to ; and `before` than λ, `let`, and `if` forms. Thus, for instance, `if` e_1 `then` e_2 `else` $e_3; e_4$ and $\lambda x.\, e_1; e_2$ mean `if` e_1 `then` e_2 `else` $(e_3; e_4)$ and $\lambda x.\, (e_1; e_2)$, respectively, rather than (`if` e_1 `then` e_2 `else` $e_3); e_4$ or $(\lambda x.\, e_1); e_2$. In addition, we take ; as right-associative, which is more convenient when defining a bisimulation (see Appendix D [13]).

Our operational semantics is a standard small-step reduction semantics with evaluation contexts and stores. Here, a store s is a finite map from locations (except $\mathtt{null}^{\perp}$) to values. We write $dom(s)$ for the domain of store s. We also write $s \uplus \{\ell \mapsto v\}$ for the extension of store s with location ℓ mapped to value v, with the assumption that $\ell \notin dom(s)$. It is undefined if $\ell \in dom(s)$. Similarly, $s_1 \uplus s_2$ is defined to be $s_1 \cup s_2$ if $dom(s_1) \cap dom(s_2) = \emptyset$, and undefined otherwise. $s \setminus \tilde{\ell}$ denotes the store obtained from s by removing $\tilde{\ell}$ from its domain. Again, it is undefined if $\tilde{\ell} \notin dom(s)$. We write $\twoheadrightarrow$ for the reflexive and transitive closure of $\rightarrow$.

Note that the reduction is non-deterministic, even up to renaming of locations. For instance, consider $e = \nu x := \langle\rangle; x \stackrel{ptr}{=} \ell$. Then, we have both $\emptyset \triangleright e \rightarrow \{\ell \mapsto \langle\rangle\} \triangleright (\ell \stackrel{ptr}{=} \ell) \rightarrow \{\ell \mapsto \langle\rangle\} \triangleright \mathtt{true}$ and $\emptyset \triangleright e \rightarrow \{m \mapsto \langle\rangle\} \triangleright (m \stackrel{ptr}{=} \ell) \rightarrow \{m \mapsto \langle\rangle\} \triangleright \mathtt{false}$. This is one of the characteristics of our language, where deallocation makes dangling pointers (like ℓ in the above example), which may or may not get reallocated later.

Throughout the paper, we often abbreviate sequences $A_1, \ldots, A_n$ to $\tilde{A}$, for any kind of meta-variables A_i. We also abbreviate sequences of tuples, like $(A_1, B_1), \ldots, (A_n, B_n)$, as $(\tilde{A}, \tilde{B})$. Thus, for example, $[\tilde{v}/\tilde{x}]e$ denotes $[v_1, \ldots, v_n/x_1, \ldots, x_n]e$.

4 Binary Environmental Relations

In this section, we develop our approach for *binary* relations including contextual equivalence, which is closer to (the small-step version [12] of) the original environmental bisimulations [7, 14, 15].

First, we establish the basic terminology for our developments. Intuitions behind the definitions are given in the introduction.

Definition 1 (state and binary configuration). *The pair $s \triangleright e$ of store s and term e is called a* state. *A* binary configuration *is a quintuple of the form $(\mathcal{R}, s \triangleright e, s' \triangleright e')$ or a triple of the form $(\mathcal{R}, s, s')$, where $\mathcal{R}$ is a binary relation on values.*

Note that we do not impose well-formedness conditions such as $loc(e) \subseteq dom(s)$ and $loc(e') \subseteq dom(s')$, because deallocation may (rightfully) make dangling pointers.

Definition 2 (context closure). *The* context closure $\mathcal{R}^\star$ *of a binary relation* $\mathcal{R}$ *on values, is defined by* $\mathcal{R}^\star = \{([\tilde{v}/\tilde{x}]C, [\tilde{v}'/\tilde{x}]C) \mid (\tilde{v}, \tilde{v}') \in \mathcal{R},\ fv(C) \subseteq \{\tilde{x}\}\}$.

We write $R^{\hat{\star}}$ for the restriction of $\mathcal{R}^\star$ to values. Note $\mathcal{R} \subseteq \mathcal{R}^\star = (\mathcal{R}^{\hat{\star}})^\star$.

Then, we give the main definitions in this section. For brevity, we omit some universal and existential quantifications on meta-variables in the conditions below. They should be clear from the context—or, more precisely, from the positions of the first occurrences of the meta-variables. For instance, when we say

> For every $(\mathcal{R}, s \triangleright d, s' \triangleright d') \in X$, if $s \triangleright d \to t \triangleright e$, then $s' \triangleright d' \twoheadrightarrow t' \triangleright e'$ and $(\mathcal{R}, t \triangleright e, t' \triangleright e') \in X$

it means

> For every $(\mathcal{R}, s \triangleright d, s' \triangleright d') \in X$, and for any t and e, if $s \triangleright d \to t \triangleright e$ then for some t' and e' we have $s' \triangleright d' \twoheadrightarrow t' \triangleright e'$ and $(\mathcal{R}, t \triangleright e, t' \triangleright e') \in X$

because t and e first appear in the assumption, whereas t' and e' first appear in the conclusion.

Definition 3 (reduction closure). *A set X of binary configurations is* reduction-closed *if, for every $(\mathcal{R}, s \triangleright d, s' \triangleright d') \in X$,*

i. If $s \triangleright d \to t \triangleright e$, then $s' \triangleright d' \twoheadrightarrow t' \triangleright e'$ and $(\mathcal{R}, t \triangleright e, t' \triangleright e') \in X$.
ii. If $s' \triangleright d' \to t' \triangleright e'$, then $s \triangleright d \twoheadrightarrow t \triangleright e$ and $(\mathcal{R}, t \triangleright e, t' \triangleright e') \in X$.
iii. If $d = v$ and $d' = v'$, then $(\mathcal{R} \cup \{(v, v')\}, s, s') \in X$.

Intuitively, reduction closure means that the property in question is preserved throughout the execution of the programs e and e' (including the returned values v and v', which are then learned by the context). Note that we do not require a condition like "if $d = v$, then $s' \triangleright d' \twoheadrightarrow t' \triangleright v'$" (and vice versa) here. It is a specific property—defined P^{obs} below—of contextual equivalence, while we are interested in other more general properties as well.

In what follows, whenever we say "a predicate P on binary configurations," we silently impose the restriction that if $P(\mathcal{R}, s \triangleright d, s' \triangleright d')$ or $P(\mathcal{R}, s, s')$, and if $(u, u') \in \mathcal{R}$, then the outermost shape of u is the same as that of u'. (This includes equality of constants—that is, $u = c$ if and only if $u' = c$. We assume the existence of equality tests on all constants.) This is for excluding cases where reduction gets stuck on the left hand side and not on the right (or vice versa). For similar reasons, we additionally assume:

- If $(\ell^\pi, \ell'^{\pi'}) \in \mathcal{R}$, then $\pi = \pi' = \bot$ and $\ell^\bot \in dom(s) \iff \ell'^\bot \in dom(s')$.
- If $(\ell_1^\bot, \ell_1'^\bot) \in \mathcal{R}$ and $(\ell_2^\bot, \ell_2'^\bot) \in \mathcal{R}$, then $\ell_1^\bot = \ell_2^\bot \iff \ell_1'^\bot = \ell_2'^\bot$.

Definition 4 (environmental P-simulation). *Let P be a predicate on binary configurations. A reduction-closed subset X of P is called an* environmental P-simulation *if, for every $(\mathcal{R}, s, s') \in X$ and $(u, u') \in \mathcal{R}$,*

1. *If* $u = \lambda x.\, e$ *and* $u' = \lambda x.\, e'$, *then* $(\mathcal{R}, s \triangleright uv, t \triangleright u'v') \in X$ *for any* $(v, v') \in \mathcal{R}^\star$.
2. *If* $u = \langle v_1, \ldots, v_i, \ldots, v_n \rangle$ *and* $u' = \langle v'_1, \ldots, v'_i, \ldots, v'_n \rangle$, *then* $(\mathcal{R} \cup \{(v_i, v'_i)\}, s, s') \in X$.
3. *If* $u = \ell^\perp$, $u' = \ell'^\perp$, $s = t \uplus \{\ell^\perp \mapsto v\}$ *and* $s' = t' \uplus \{\ell'^\perp \mapsto v'\}$, *then*
   (a) $(\mathcal{R}, t, t') \in X$.
   (b) $(\mathcal{R}, t \uplus \{\ell^\perp \mapsto w\}, t' \uplus \{\ell'^\perp \mapsto w'\}) \in X$ *for any* $(w, w') \in \mathcal{R}^\star$.
   (c) $(\mathcal{R} \cup \{(v, v')\}, s, s') \in X$.
4. *For any* $\ell^\perp \notin dom(s)$ *and* $(v, v') \in \mathcal{R}^\star$, *we have* $(\mathcal{R} \cup \{(\ell^\perp, \ell'^\perp)\}, s \uplus \{\ell^\perp \mapsto v\}, s' \uplus \{\ell'^\perp \mapsto v'\}) \in X$ *for some* $\ell'^\perp \notin dom(s')$.

An environmental P-simulation X is called an *environmental P-bisimulation* if its inverse

$$\begin{aligned} X^{-1} = {} & \{(\mathcal{R}^{-1}, s' \triangleright e', s \triangleright e) \mid (\mathcal{R}, s \triangleright e, s' \triangleright e') \in X\} \\ & \cup \{(\mathcal{R}^{-1}, s', s) \mid (\mathcal{R}, s, s') \in X\} \end{aligned}$$

is also an environmental P-simulation (or, if X is an environmental P^{-1}-simulation—this is equivalent because all the other conditions are symmetric). An *environmental bisimulation* X is defined by taking P to be the following P^{obs}.

$$\begin{aligned} P^{obs}(\mathcal{R}, s \triangleright d, s' \triangleright d') &= \text{if } d' = v', \text{ then } s \triangleright d \twoheadrightarrow t \triangleright v \\ P^{obs}(\mathcal{R}, s, s') &= \mathit{true} \end{aligned}$$

Since all the conditions of environmental P-simulations are monotone on X, the union of all environmental P-simulations is also an environmental P-simulation, called the *environmental P-similarity*. In what follows, we often omit the adjective "environmental" and just write "a simulation" to mean an environmental simulation. The same holds for all the combinations of P- and bi- simulations and similarity.

As outlined in the introduction, the conditions of P-simulation reflect observations made by contexts. In Definition 3 (reduction closure), Conditions i and ii mean reduction on the left can be simulated by the right hand side and vice versa. Condition iii adds the returned values of programs to the knowledge of a context.

In Definition 4 (P-simulation), Condition 1 corresponds to function application, and Condition 2 to element projection from tuples. Conditions 3a, 3b, 3c, and 4 represent deallocation of, writing to, reading from, and allocation of locations, respectively.

We are now going to prove the main result of this section: let $P_{\star\rightarrow}$ be the largest contextual, reduction-closed subset of P (which exists because the union of contextual, reduction-closed sets is again contextual and reduction-closed); then the P-similarity coincides with $P_{\star\rightarrow}$, provided that P itself is contextual in the following sense.

Definition 5 (contextuality). *A set P of configurations is* contextual *if its context closure*

$$\begin{aligned} P^\star = \{&(\mathcal{S}, s \triangleright [\tilde{v}/\tilde{x}]E[e], s' \triangleright [\tilde{v}'/\tilde{x}]E[e']) \mid \\ &\quad (\mathcal{R}, s \triangleright e, s' \triangleright e') \in P,\ \mathcal{S} \subseteq \mathcal{R}^{\hat{\star}},\ (\tilde{v}, \tilde{v}') \in \mathcal{R},\ fv(E) \subseteq \{\tilde{x}\}\} \\ \cup \{&(\mathcal{S}, s \triangleright [\tilde{v}/\tilde{x}]C, s' \triangleright [\tilde{v}'/\tilde{x}]C) \mid \\ &\quad (\mathcal{R}, s, s') \in P,\ \mathcal{S} \subseteq \mathcal{R}^{\hat{\star}},\ (\tilde{v}, \tilde{v}') \in \mathcal{R},\ fv(C) \subseteq \{\tilde{x}\}\} \\ \cup \{&(\mathcal{S}, s, s') \mid (\mathcal{R}, s, s') \in P,\ \mathcal{S} \subseteq \mathcal{R}^{\hat{\star}}\} \end{aligned}$$

is included in P.

Note that $P \subseteq P^\star = (P^\star)^\star$. In short, contextuality means that P is preserved under arbitrary contexts. The inclusion $\mathcal{S} \subseteq \mathcal{R}^{\hat{\star}}$ is necessary for the following technical reason: suppose we have a configuration $(\mathcal{R}, s \triangleright d, s' \triangleright d') \in X$ and put it under an evaluation context E, like $(\mathcal{R}, s \triangleright E[d], s' \triangleright E[d']) \in X$. If d and d' reduce to values v and v', respectively, then the context learns these values and adds them to its knowledge, like $(\mathcal{R} \cup \{(v, v')\}, s \triangleright E[v], s' \triangleright E[v']) \in X$. However, according to the conditions of reduction closure, we need $(\mathcal{R}, s \triangleright E[v], s' \triangleright E[v']) \in X$, where the knowledge $\mathcal{R}$ is *smaller* than $\mathcal{R} \cup \{(v, v')\}$. A similar case occurs when the context by itself allocates a fresh location.

This is not a real problem because smaller knowledge means less observations, i.e., more properties. In fact, instead of taking $\mathcal{S} \subseteq \mathcal{R}^{\hat{\star}}$ here, it is also possible to generalize the definition of simulation to allow the increase of knowledge in the middle of an evaluation. This amounts to an up-to environment technique [12].

Theorem 1 (characterization). *For any P, the $P^\star$-similarity coincides with $(P^\star)_{\star\to}$. In particular, if P is contextual, then the P-similarity coincides with $P_{\star\to}$.*

It is easy to check—by simple induction on the evaluation context E—that the previous P^{obs} for contextual equivalence is indeed contextual. (It does not even refer to $\mathcal{R}$ at all!) Thus:

Corollary 1 (bisimilarity equals contextual equivalence). *The bisimilarity coincides with the contextual equivalence $P^{obs}_{\star\to}$.*

Appendix A [13] gives examples of the use of P-bisimulations, including a proof of contextual equivalence and a property of space usage. Appendix B [13] develops an up-to technique that lightens the burden of bisimulation proof.

5 Unary Environmental Predicates

Suppose that we want to prove the memory safety of the directed acyclic graph implementation *dag*. To do so, we can use the "bisimulation" between *dag* and *dag* itself! This idea formalizes to the following definitions.

Definition 6 (memory safety). *State $s \triangleright e$ is* memory unsafe *if e is either $E[\mathtt{free}(\ell^\top)]$, $E[\ell^\top := v]$, or $E[!\ell^\top]$, with $\ell^\top \notin dom(s)$. It is* memory safe *if not memory unsafe. We often omit "memory" and just say "safe" or "unsafe," and write safe for the set of safe $s \triangleright e$.*

Note that the definition above does not imply so-called "type safety," which is a more general property. For instance, *safe* does not preclude stuck states such as $\emptyset \triangleright \ell\ 3$.

Definition 7. *A* unary configuration *is a triple of the form* $(\mathcal{R}, s \triangleright e)$ *or a pair of the form* $(\mathcal{R}, s)$*, where* $\mathcal{R}$ *is a predicate on values.*

Definition 8 (environmental P-predicate). *Let P be a predicate on unary configurations. A set $X \subseteq P$ of unary configurations is called an* environmental P-predicate *if its* duplication $X^2 = \{(\mathcal{R}^2, s \triangleright e, s \triangleright e) \mid (\mathcal{R}, s \triangleright e) \in X\} \cup \{(\mathcal{R}^2, s, s) \mid (\mathcal{R}, s) \in X\}$ *is an environmental P^2-simulation, where $\mathcal{R}^2 = \{(v, v) \mid v \in \mathcal{R}\}$. To spell out all the conditions,*

1. *For every $(\mathcal{R}, s \triangleright d) \in X$,*
   (a) *If $s \triangleright d \to t \triangleright e$, then $(\mathcal{R}, t \triangleright e) \in X$.*
   (b) *If $d = v$, then $(\mathcal{R} \cup \{v\}, s) \in X$.*
2. *For every $(\mathcal{R}, s) \in X$ and $u \in \mathcal{R}$,*
   (a) *If $u = \lambda x.\, e$, then $(\mathcal{R}, s \triangleright uv) \in X$ for any $v \in \mathcal{R}^\star$.*
   (b) *If $u = \langle v_1, \ldots, v_i, \ldots, v_n \rangle$, then $(\mathcal{R} \cup \{v_i\}, s) \in X$.*
   (c) *If $u = \ell^\perp$ and $s = t \uplus \{\ell^\perp \mapsto v\}$, then $(\mathcal{R}, t) \in X$, $(\mathcal{R}, t \uplus \{\ell^\perp \mapsto w\}) \in X$ for any $w \in \mathcal{R}^\star$, and $(\mathcal{R} \cup \{v\}, s) \in X$.*
   (d) *$(\mathcal{R} \cup \{\ell^\perp\}, s \uplus \{\ell^\perp \mapsto v\}) \in X$ for any $\ell^\perp \notin dom(s)$ and $v \in \mathcal{R}^\star$.*

where the unary version of context closure is defined as $\mathcal{R}^\star = \{[\tilde{v}/\tilde{x}]C \mid \tilde{v} \in \mathcal{R},\ fv(C) \subseteq \{\tilde{x}\}\}$.

All the results from binary environmental P-simulations apply to this unary version, because the latter is just a special case of the former (and because equality satisfies all the restrictions on P). This includes soundness and the up-to technique. For pedagogy, we spell out the conditions of environmental P-predicate up-to context and allocation.

Definition 9 (allocation closure). *The (unary)* allocation closure *of X is defined as:*

$$X^\nu = \{(\mathcal{R}, s \triangleright e) \mid (\mathcal{R}, s \triangleright e) \in X\} \cup \{(\mathcal{S}, s \setminus \tilde{m}^\perp \uplus \{\tilde{\ell}^\perp \mapsto \tilde{w}\}) \mid (\mathcal{R}, s) \in X,\ \tilde{m}^\perp \in \mathcal{R},\ \mathcal{S} = \mathcal{R} \cup \{\tilde{\ell}^\perp\},\ \tilde{w} \in \mathcal{S}^\star\}$$

Definition 10 (environmental P-predicate up-to). *A set $X \subseteq P$ of unary configurations is called an* environmental P-predicate up-to context and allocation *(or just a "P-predicate up-to" in short) if:*

1. *For every $(\mathcal{R}, s \triangleright d) \in X$,*
   (a) *If $s \triangleright d \to t \triangleright e$, then $(\mathcal{S}, t \triangleright e) \in (X^\nu)^\star$.*
   (b) *If $d = v$, then $(\mathcal{R} \cup \{v\}, s) \in (X^\nu)^\star$.*
2. *For every $(\mathcal{R}, s) \in X$ and $u \in \mathcal{R}$,*
   (a) *If $u = \lambda x.\, e$, then for any $(\mathcal{S}, t) \in \{(\mathcal{R}, s)\}^\nu$ and $v \in \mathcal{S}^\star$, we have $(\mathcal{S}, t \triangleright uv) \in X$.*
   (b) *If $u = \langle v_1, \ldots, v_i, \ldots, v_n \rangle$, then $(\mathcal{R} \cup \{v_i\}, s) \in (X^\nu)^\star$.*
   (c) *If $u = \ell^\perp$ and $\ell^\perp \in dom(s)$, then $(\mathcal{R} \cup \{s(\ell^\perp)\}, s) \in (X^\nu)^\star$.*

6 An Example

The code in Figure 1 implements directed acyclic graphs (DAGs), with garbage collection by reference counting. For simplicity, we use *immutable* lists in this example (in addition to a mutable data structure for representing the DAGs themselves), and assume their basic operations such as *member*, *append*, and *remove*.

Here, z is bound to the location of the last added node in the DAG. A node is either `null` or a quintuple $\langle i, b, n, p, \ell \rangle$, where i is an integer ID of the node, b a Boolean value meaning whether the node is "in the root set" (i.e., cannot be garbage collected), n the reference count of the node, p the (immutable) list of the integer IDs of child nodes, and ℓ the pointer to the second last added node. This pointer is different from child pointers, for which we use the list of integer IDs.

Function *addn* takes integer x and integer list p, and adds a node with ID x and children p. The code $x + 0$ and $map(\lambda y.\, y + 0)p$ ensures they are indeed an integer and an integer list (assuming that $\_ + 0$ is defined only for integers). An auxiliary function $incr_x$ is used to increment the reference counts of nodes in p, as well as to check if node x already exists (in which case it diverges). Note that the same node may appear more than once in p. Its reference count is increased by the number of appearance.

Function *deln* prepares to delete a node by (un)marking it as non-root. Function *gc* invokes the garbage collector *decr*, which takes a node pointer n and an integer list p. It decreases the reference counts of nodes in p, again according to the number of their appearances. If the reference count becomes 0, and if the root flag is not set, then the node is deleted, and its children are added to p so that their reference counts will be decreased recursively. In the end, *decr* returns the updated node pointer n.

We define the shape predicate for DAGs by induction.

- $DAG_S(\mathtt{null}, \emptyset, \emptyset)$
- $DAG_S(\ell, [(i, b, S_0)] \mathbin{@} L_0, s_0 \uplus \{\ell \mapsto \langle i, b, S(i), S_0, \ell_0 \rangle\})$
  if $\ell \neq \mathtt{null}$, $DAG_{S+S_0}(\ell_0, L_0, s_0)$, and $i \neq i_0$ for any $(i_0, \_, \_) \in L_0$.

Here, the subscript S is a multiset of node IDs, representing the number of references to each node.

We also give a specification of our garbage collector as follows. It is more abstract than the implementation because it looks at only the positiveness of the reference count $S(i)$, not its concrete value (i.e., only whether the node is referred to, not how many times).

$$
\begin{array}{lll}
GC_S([\,]) & = [\,] & \\
GC_S([(i, b, S_0)] \mathbin{@} L_1) & = [(i, b, S_0)] \mathbin{@} GC_{S+S_0}(L_1) & \text{if } b = \mathtt{true} \text{ or } S(i) > 0 \\
GC_S([(i, \mathtt{false}, S_0)] \mathbin{@} L_1) & = GC_S(L_1) & \text{if } S(i) = 0
\end{array}
$$

GC_S takes a list of triples (i, b, S_0) that represent nodes, where i is the node ID, b the root flag, and S_0 the multiset of the IDs of the children. Here, the

subscript S is the multiset of the IDs of nodes pointed to by "external" nodes, i.e., by nodes that are not in the list.

Now, the following lemma can be proved.

Lemma 1. *Suppose* $DAG_S(\ell, L, s)$. *Then, for any* t *and* T, *we have* $s \uplus t \rhd decr(\ell)T \twoheadrightarrow s_0 \uplus t \rhd \ell_0$ *with* $DAG_{S-T}(\ell_0, GC_{S-T}(L), s_0)$. *(Here, we are abusing notation and writing* T *for an integer list representing the integer multiset* T*.)*

Given the lemma above, it is straightforward to give an environmental predicate for *dag* and prove it to be memory safe under arbitrary (public) contexts. In fact, we can prove more properties, e.g., that the number of private locations matches the number of nodes (and therefore the number of *live* nodes after a call to *gc*) plus one (for z). To be specific, take

$$\begin{aligned} X = {} & \{(\emptyset, \emptyset \rhd dag)\} \\ & \cup \{(F^\omega(\{\tilde{\ell}^\perp\}), t \rhd e) \mid DAG_\emptyset(\ell^\top, L, s),\ d \in F^\omega(\{\tilde{\ell}^\perp\}),\ v, \tilde{w} \in (F^\omega(\{\tilde{\ell}^\perp\}))^\star, \\ & \qquad\qquad s \uplus \{m^\top \mapsto \ell^\top\} \uplus \{\tilde{\ell}^\perp \mapsto \tilde{w}\} \rhd d(v) \twoheadrightarrow t \rhd e\} \\ & \cup \{(F^\omega(\{\tilde{\ell}^\perp\}), s \uplus \{m^\top \mapsto \ell^\top\}) \mid DAG_\emptyset(\ell^\top, L, s)\} \end{aligned}$$

where

$$\begin{aligned} F(\mathcal{R}) = {} & \mathcal{R} \cup \{[m^\top/z]addn, [m^\top/z]deln, [m^\top/z]gc\} \\ & \cup \{[v/x][m^\top/z](\lambda p.\, x + 0; \ldots) \mid v \in \mathcal{R}^\star\}. \end{aligned}$$

Then, X is an environmental P-predicate up-to, where

$$\begin{aligned} P = {} & \{(\mathcal{R}, s \uplus \{m^\top \mapsto \ell^\top\} \uplus \{\tilde{\ell}^\perp \mapsto \tilde{w}\} \rhd e) \mid safe(s \uplus \{m^\top \mapsto \ell^\top\} \uplus \{\tilde{\ell}^\perp \mapsto \tilde{w}\} \rhd e)\} \\ & \cup \{(\mathcal{R}, s \uplus \{m^\top \mapsto \ell^\top\} \uplus \{\tilde{\ell}^\perp \mapsto \tilde{w}\}) \mid DAG_\emptyset(\ell^\top, L, s)\}. \end{aligned}$$

Appendix D gives another example of memory safety proof by environmental P-predicates.

7 Conclusion

As is often the case in programming language theories, our theory may seem trivial in hindsight. In particular, all the proofs are arguably straightforward (though sometimes just lengthy because of case analyses) once organized in the way presented here. However, such an organization and the *definitions* were far from trivial, especially because of the non-monotone stores and non-deterministic reduction.

Future work includes deriving such definitions systematically from the operational semantics of a language (cf. [7]), so that the definitions and proofs do not have to be repeated manually for every language. Another direction is mechanization. Although complete automation is clearly impossible, ideas from model checking and type-based analyses may be useful for sound approximation. Weakening the contextuality to restrict the possible contexts—so that more programs can be proved correct—would also be useful in practice.

References

[1] Ahmed, A.: Semantics of Types for Mutable State. PhD thesis, Princeton University (2004)

[2] Ahmed, A., Dreyer, D., Rossberg, A.: State-dependent representation independence. In: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2009) (to appear), http://ttic.uchicago.edu/~amal/papers/sdri.pdf

[3] Ahmed, A., Fluet, M., Morrisett, G.: A step-indexed model of substructural state. In: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming, pp. 78–91 (2005)

[4] Ahmed, A., Fluet, M., Morrisett, G.: L3: A linear language with locations. TLCA 2005 77(4), 397–449 (2007); extended abstract appeared in: Typed Lambda Calculi and Applications. LNCS, vol. 3461, pp. 293–307. Springer (2005)

[5] Bohr, N.: Advances in Reasoning Principles for Contextual Equivalence and Termination. PhD thesis, IT University of Copenhagen (2007)

[6] Crary, K., Walker, D., Morrisett, G.: Typed memory management in a calculus of capabilities. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 262–275 (1999)

[7] Koutavas, V., Wand, M.: Small bisimulations for reasoning about higher-order imperative programs. In: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 141–152 (2006)

[8] Meyer, A.R., Sieber, K.: Towards fully abstract semantics for local variables: Preliminary report. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 191–203 (1988)

[9] Mitchell, J.C.: Foundations for Programming Languages. MIT Press, Cambridge (1996)

[10] Morris Jr., J.H.: Lambda-Calculus Models of Programming Languages. PhD thesis, Massachusetts Institute of Technology (1968)

[11] Pitts, A.M., Stark, I.: Operational reasoning for functions with local state. In: Higher Order Operational Techniques in Semantics, pp. 227–273. Cambridge University Press, Cambridge (1998)

[12] Sangiorgi, D., Kobayashi, N., Sumii, E.: Environmental bisimulations for higher-order languages. In: Twenty-Second Annual IEEE Symposium on Logic in Computer Science, pp. 293–302 (2007)

[13] Sumii, E.: A theory of non-monotone memory (or: Contexts for free), http://www.kb.ecei.tohoku.ac.jp/~sumii/pub/non-mono.pdf

[14] Sumii, E., Pierce, B.C.: A bisimulation for dynamic sealing. Theoretical Computer Science 375, 1–3, 169–192 (2007); extended abstract appeared in: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 161–172 (2004)

[15] Sumii, E., Pierce, B.C.: A bisimulation for type abstraction and recursion. Journal of the ACM 54, 5–26, 1–43 (2007); extended abstract appeared in: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 63–74 (2005)

[16] Tofte, M., Talpin, J.-P.: Implementation of the typed call-by-value λ-calculus using a stack of regions. In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 188–201 (1994)

[17] Wadler, P.: Linear types can change the world! In: Programming Concepts and Methods. North Holland, Amsterdam (1990)