Limitations of Applicative Bisimulation (Preliminary Report)

Vasileios Koutavas*[1], Paul Blain Levy[2] and Eijiro Sumii[3]

[1] Trinity College Dublin
Vasileios.Koutavas@scss.tcd.ie
[2] University of Birmingham
P.B.Levy@cs.bham.ac.uk
[3] Tohoku University
sumii@ecei.tohoku.ac.jp

Abstract. We present a series of examples that illuminate an important aspect of the semantics of higher-order functions with local state. Namely that certain behaviour of such functions can only be observed by providing them with arguments that contain the functions themselves. This provides evidence for the necessity of complex conditions for functions in modern semantics for state, such as logical relations and Kripke-like bisimulations, where related functions are applied to *related* arguments (that may contain the functions). It also suggests that simpler semantics, such as those based on applicative bisimulations where functions are applied to identical arguments, would not scale to higher-order languages with local state.

Keywords. Imperative languages, higher-order functions, local state.

1 Introduction

Some of the latest relational semantics of higher-order languages with local state, based on *bisimulations* [25,26,22,4,10,11,12] and *logical relations* [3,6,2,18,5], examine the behaviour of related functions by applying them to arguments drawn by the relation itself, or by an appropriate compatible closure of it. This imposes a seemingly strong obligation in proofs of equivalence of such functions, compared to the proof obligation of *applicative bisimulation* [1,7,27] where only identical arguments need to be considered. The method of open bisimulation (also known as normal form bisimulation) has also been shown useful for reasoning about state [20,14,15,13,8,24].

A relational semantics of a higher-order lambda-calculus with general store based on Kripke-like bisimulations [11] is the largest set of worlds (s, s', R) that satisfy a set of conditions, each composed by a relation R on terms and stores s and s' under which the relation holds. The relation encodes the related values (most notably locations) known to the context. An example of the conditions

* Vasileios Koutavas was supported by SFI project SFI 06 IN.1 1898.

of this semantics is that (s, s', R) can be in the semantics only if updates to locations in the knowledge R leads to a world also in the semantics.

The Kripke-like bisimulation conditions that are most relevant to applicative bisimulation involves applications of related functions. To prove that a world (s, s', R) which relates two abstractions F and G is in the semantics, we need to show, among other conditions, that for all arguments in the compatible closure of R (i.e. for all $C[\overline{v}]$ and $C[\overline{v'}]$ with $\overline{(v, v')} \in R$), when the application $(F\, C[\overline{v}])$ under s evaluates to a value w and store t, then the application $(G\, C[\overline{v'}])$ under s' evaluates to some value w' and store t', and (t, t', Q) is an appropriate world in the semantics, in which the values w and w' are related. We also need to prove a similar condition for when $(G\, C[\overline{v'}])$ under s' evaluates to w' and t'.[1]

Applicative bisimulations [1], however, only require that functions have a related behaviour when applied to *the same* arguments. They provide an elegant, sound and complete, relational semantics of pure higher-order languages [1,7]. The question, which we answer in the negative in this report, is whether a sound applicative bisimulation semantics can be given as well to higher-order languages with local state.[2]

Mason and Talcott, when they studied a relational theory for a call-by-value lambda calculus with general references [17, Section 7], wrote:

> [Applicative] bisimulation provides an alternative approach to equivalence and deserves consideration in computation systems that permit effects other than non-termination. The definition of bisimulation relation assumes that extensionality is consistent. Since the presence [of] memory effects makes this no longer true, the basic definition would require some modification in order to extend the methods of Abramsky and Howe to the computational language presented in this paper. We plan to investigate this approach.

Although Mason and Talcott have recognised that extensionality does not hold for languages with effects—and therefore applicative bisimulation as defined for pure languages is not sound in the presence of effects—the counterexamples they give [17, Section 3] involve terms that can be distinguished by contexts that update memory accessed by the terms. These contexts, however, are already captured in Kripke-like semantics by conditions that update the stores and are separate to the conditions for the applications of related functions to related arguments. Therefore the question remains if *applicative versions* of Kripke-like bisimulation semantics would be sound for higher-order languages with local store.

Let us now consider such a semantics containing worlds of the form $(s, s', \bar{l}, R)$, where $\bar{l}$ are the locations in both s and s' that can be used in the arguments

[1] Note that in such Kripke-like bisimulation semantics the context C from which arguments are constructed is taken to be location-free, and therefore only the locations related in R are accessible directly in the arguments to functions.

[2] A remaining open question is whether applicative bisimulations can be combined with up-to-context reasoning [21].

to functions. To prove that a world $(s, s', \bar{l}, R)$ which relates F and G is in the semantics, we need to show that for all arguments C using locations from $\bar{l}$, when the application $(F\,C)$ under s evaluates to a value w and store t, then the application $(G\,C)$ under s' evaluates to some value w' and store t', and (t, t', Q) is an appropriate world in the semantics which relates the values w and w'.

It is not immediately clear whether such a simpler version of Kripke-like bisimulations would provide a sound model of higher-order imperative languages. In this report we show that this would in fact be *unsound*, and argue that the stronger conditions are necessary. We do that by presenting several examples, each containing a pair of *contextually inequivalent* higher-order terms that cannot be distinguished by the applicative conditions, but only from the more general conditions.

The intuition in these examples is that in order for the higher-order terms to be distinguished, they will have to be applied to functional values that contain the terms themselves; i.e. values that are *related*, not identical. When these arguments are applied within the related terms they invoke the terms again to make the necessary observations. In other words, the related terms are distinguished only when applied again within the *dynamic extent* of applications of their arguments.

We examine such examples in a number of higher-order languages with local state: the nu-calculus of Pitts and Stark [19,23] (Section 2) and lambda calculi with a first-order store (Section 3.1), a general store (Section 3.2), and existential types (Section 4). The individual features of each of these languages change the distinguishing power of the applicative semantics, and therefore change our examples, but in none of the languages is this semantics sound with respect to contextual equivalence.

2 Simple Names: the Nu-Calculus

We first look at the nu-calculus [19,23], the simplest of the languages we examine in this paper. The nu-calculus is a strongly-normalising, simply-typed, call-by-value lambda calculus over the base types of names ($\mathtt{Nm}$) and booleans ($\mathtt{Bool}$). The only effect is the generation of names ($\nu n.\, e$) that can be passed around and be tested for equality ($n = m$). The big-step operational semantics take the form

$$s \vdash e \Downarrow (t)\, w$$

which means that under the set s of allocated names, expression e evaluates to value w, generating a set of fresh names t in the process. We refer the reader to [23] for details of the syntax and semantics of the calculus.

Example 1. Our example in the nu-calculus involves two higher-order terms M_1 and M_1' of type $(\mathtt{Nm} \to \mathtt{Bool}) \to \mathtt{Bool}$:

$$M_1 \stackrel{\text{def}}{=} \nu n.\, V_1(n)$$
$$M_1' \stackrel{\text{def}}{=} \lambda f{:}\mathtt{Nm} \to \mathtt{Bool}.\, \nu n.\, f\, n$$

where

$$V_1(n) \stackrel{\text{def}}{=} \lambda f{:}\mathtt{Nm} \to \mathtt{Bool}.\, f\, n$$

The term M_1 uses a local name generated once while M_1' generates a fresh local name each time it is applied. The generated name n is passed to the argument of both functions $V_1(n)$ and M_1' but, because there is no possibility of storing the name, it never escapes to the context surrounding the functions. Thus both $V_1(n)$ and M_1' reveal n to the context (i.e. to f) only for the *dynamic extent* of the application $f\, n$. Therefore f can make observations about the related terms that the surrounding context cannot make.

This is important to differentiate M_1 and M_1'. The following context C_1 provides an argument to the terms that internally applies the value bound to g ($V_1(n)$ or M_1') and uses the knowledge of n to differentiate them.

$$\begin{array}{rl} C_1 \stackrel{\text{def}}{=} & \mathtt{let}\, g \;= [\cdot]\, \mathtt{in} \\ & \mathtt{let}\, f \;= \lambda n{:}\mathtt{Nm}.\, g\, (\lambda m{:}\mathtt{Nm}.\, m = n)\, \mathtt{in} \\ & (g\, f) \end{array}$$

Under any nameset $C_1[M_1]$ evaluates to `true` while $C_1[M_1']$ evaluates to `false`.

It is easy to show that M_1 and M_1' are indistinguishable when applied to the *same arguments*.

Proposition 1. *For any namesets s t, any value U of type* $\mathtt{Nm} \to \mathtt{Bool}$ *that uses names in s:*

$$s \vdash M_1\, U \Downarrow (t)\, b \quad \textit{iff} \quad s \vdash M_1'\, U \Downarrow (t)\, b$$

Let us now go back to our applicative adaptation of Kripke-style bisimulation semantics for the nu-calculus. In this semantics we are be able to prove that M_1 is related to M_1' by constructing the set

$$\left\{(s \oplus \{\overline{n}\}, s \oplus \{\overline{n'}\}, \overline{m}, \{(M_1, M_1'), \overline{(V_1(n), M_1')}\}) \mid \overline{m} \in \mathit{dom}(s) \text{ and } \overline{n}, \overline{n'} \notin \mathit{dom}(s)\right\}$$

and showing that all the tuples in the set satisfy the conditions of the semantics using Proposition 1. □

The preceding example shows that our fairly obvious adaptation of Kripke-like bisimulation semantics to applicative conditions for functions is not sound with respect to contextual equivalence in the nu-calculus. This suggests that applicative conditions, although they elegantly express the semantics of pure higher-order functions, are not suitable for higher-order functions with local state. We verify this observation to other higher-order languages with local state in the following sections.

3 Lambda Calculus with Store

3.1 First-Order Store

In this section we consider a simply-typed lambda calculus with locations, ranged over by l, that can store boolean values. The expression $\nu x := b.\, e$ creates a fresh

location in the store containing the boolean constant b and binds x to the location in e. Expression $l := e$ updates the contexts of l with the (boolean) value of e, and $!l$ returns the contents of l. We consider big-step operational semantics for this language of the form

$$\langle s,\, e\rangle \Downarrow \langle t,\, w\rangle$$

meaning that under store s expression e evaluates to the value w and the store becomes t.

In our nu-calculus example of the previous section we provided the distinguishing context C_1 that made use of an equality test for names. Although the language of this section does not have a name-equality construct, we could adapt Example 1 using an encoding of name equality that writes a value to one location and attempts to read it from the other. However, we give a different example inequivalence that is robust to the addition of a general store.

Example 2. Consider the higher-order functions:

$$M_2 \stackrel{\text{def}}{=} \lambda f{:}\mathtt{Unit} \to \mathtt{Unit}.\, f\,()$$
$$M_2' \stackrel{\text{def}}{=} \nu\, \mathit{flag} := \mathtt{false}.\, V_2'(\mathit{flag})$$

where

$$V_2'(\mathit{flag}) \stackrel{\text{def}}{=} \lambda f{:}\mathtt{Unit} \to \mathtt{Unit}.\, \mathtt{if}\ !\mathit{flag}\ \mathtt{then}\ \mathit{div} \\ \mathtt{else}\ \mathit{flag} := \mathtt{true};\, f\,();\, \mathit{flag} := \mathtt{false}$$

and *div* is a diverging term. The term M_2 is a function that always applies its argument to $()$, whereas the term M_2' generates a private location *flag*, initially set to `false`, and becomes $V_2'(\mathit{flag})$. When *flag* is `false`, $V_2'(\mathit{flag})$ sets the flag to `true` during the application $(f\,())$, and when the flag is `true` it diverges. The term therefore diverges only during the extent of the application $(f\,())$.

The following context distinguishes M_2 and M_2':

$$C_2 \stackrel{\text{def}}{=} \begin{array}{l} \mathtt{let}\, g\ = [\cdot]\ \mathtt{in} \\ \mathtt{let}\, f\ = (\lambda x.\, g\,(\lambda y.\, y))\ \mathtt{in} \\ g\, f \end{array}$$

The term $C_2[M_2]$ returns while $C_2[M_2']$ diverges. Applying M_2 and $V_2'(\mathit{flag})$ to the same arguments and stores (modulo the location generated by M_2'), however, does not distinguish them, and therefore does not distinguish M_2 and M_2'.

Proposition 2. *For any store s and t, any value U of type* `Unit` $\to$ `Unit` *that uses names in s, and any flag $\notin s$:*

$$\langle s,\, M_2\, U\rangle \Downarrow \langle t,\, ()\rangle \quad \textit{iff} \\ \langle s[\mathit{flag} = \mathtt{false}],\, V_2'(\mathit{flag})\, U\rangle \Downarrow \langle t[\mathit{flag} = \mathtt{false}],\, ()\rangle$$

Using this proposition we can prove that the following set satisfies the conditions of the applicative bisimulation semantics of the introduction:

$$\left\{(s, s[\overline{\mathit{flag} \mapsto \mathtt{false}}], \overline{m}, \{(M_2, M_2'), \overline{(M_2, V_2'(\mathit{flag}))}\}) \mid \overline{m} \in s \text{ and } \overline{\mathit{flag}} \notin s\right\}$$

□

3.2 General Store

In a language with general store, the observational power of contexts is more significant, compared to that in the languages we examined so far. In such a language, every value revealed by a higher-order function to its argument can be stored and escape the dynamic extent of the function. This might lead one to believe that all observations can be "linearised", and that the applicative Kripke-like bisimulation semantics of the introduction would be sound. However Example 2 is still valid for a lambda calculus with general store and shows that this is not the case.[3]

4 Existential Types

The last language we examine is a lambda calculus with existential types. Expression $\mathtt{pack}\,[\tau', V]\,\mathtt{as}\,\exists\alpha.\,\tau$ creates an existential package of type $\exists\alpha.\,\tau$, where α is a type variable that may be free in τ and value V has type $\tau[\tau'/\alpha]$. A package $U = \mathtt{pack}\,[\tau', V]\,\mathtt{as}\,\exists\alpha.\,\tau$ can be opened by the expression $\mathtt{open}\,U\,\mathtt{as}\,(\alpha, x)\,\mathtt{in}\,e$, which replaces α with τ' and x with V in e.

Although this language has no store, existential packages encode a form of local state. In a package of type $\exists\alpha.\,\tau$ the values of the type α are only locally known to the package, and can selectively be revealed to the context.

It is not immediately clear what the most plausible form of applicative bisimulation would be for this language. The main issue is that simply testing related functions contained in existential packages by applying them to identical arguments is too weak of a condition. Functions within packages of type $\exists\alpha.\,\alpha \times (\alpha \to \mathtt{Bool})$ can only be applied to related arguments (the first component of the existential packages). Thus we would have to allow certain related values within otherwise identical arguments. However, we would certainly not allow the arguments to contain the functions to which they are passed, as this would greatly depart from the "applicative" nature of the conditions. The following inequivalent terms show that this approach would still be unsound for this language.

Example 3. Consider the following existential packages:

$$M_3 \stackrel{\text{def}}{=} \mathtt{pack}\,[\mathtt{Unit}, V_3]\,\mathtt{as}\,T$$
$$M_3' \stackrel{\text{def}}{=} \mathtt{pack}\,[\mathtt{Bool}, V_3']\,\mathtt{as}\,T$$

[3] Note that if we apply M_2 and $V_2(\mathit{flag})$ to identical arguments that refer to locations in related stores containing the values M_2 and $V_2(\mathit{flag})$ then we recover the distinguishing power of related arguments. This is easy to see since any related arguments of the form $C[\overline{v}]$ and $C[\overline{v'}]$ can be translated to identical arguments of the form $C[\overline{l}]$ under the related stores $[\overline{l = v}]$ and $[\overline{l = v'}]$.

where

$$V_3 \stackrel{\text{def}}{=} \lambda f{:}\mathtt{Unit} \to \mathtt{Unit}.\, f\,();\, \mathtt{true}$$
$$V_3' \stackrel{\text{def}}{=} \lambda f{:}\mathtt{Bool} \to \mathtt{Bool}.\, \mathtt{if}\ (f\,\mathtt{false})\ \mathtt{then}\ \mathtt{false}\ \mathtt{else}\ (f\,\mathtt{true})$$
$$T \stackrel{\text{def}}{=} \exists\alpha.\, (\alpha \to \alpha) \to \mathtt{Bool}$$

These are distinguished by the following context:

$$C_3 \stackrel{\text{def}}{=} \mathtt{open}\,[\cdot]\,\mathtt{as}\,(\alpha, g)\,\mathtt{in}$$
$$g\ (\lambda y{:}\alpha.\, \mathtt{if}\ (g\,(\lambda z{:}\alpha.\, y))\ \mathtt{then}\ y\ \mathtt{else}\ \mathit{div})$$

Any form of applicative Kripke-like bisimulation semantics, as we discussed above, would only apply the functions within M_3 and M_3' to arguments of type $\alpha \to \alpha$ that do not contain the functions themselves. These arguments would either behave as the identity function or they would diverge when applied. Hence, it would not be difficult to show that, because V_3 and V_3' are indistinguishable when applied to $\lambda x{:}\alpha.\, x$ or to $\lambda x{:}\alpha.\, \mathit{div}$, the terms M_3 and M_3' would be related in that semantics. □

5 Conclusions

We have studied a number of higher-order languages with local state, for which we gave a number of illuminating contextual inequivalences between terms. These inequivalences can be witnessed by contexts that apply higher-order functions to arguments that contain the functions themselves. We have given an applicative version of Kripke-like bisimulation semantics for these languages and showed that it would relate these inequivalent terms and is therefore unsound with respect to contextual equivalence.

The second author [16] has previously shown that in a typed lambda calculus with McCarthy's amb construct for fair non-determinism at non-ground types, applicative bisimulation is not a congruence.

Jeffrey and Rathke [9] have given a form of applicative bisimulation for the nu-calculus, one of the languages we examined here, based on a labelled transition system and showed that it also is unsound with respect to contextual equivalence. However they showed that the addition of integer references makes their bisimulation sound and complete. A more detailed comparison between their approach and ours is necessary to reveal the source of this incompatibility.

Previous work by Mason and Talcott [17] has showed that function extensionality does not hold in languages with effects, identifying in this way a potential issue with extending applicative bisimulations to imperative higher-order languages. Their counterexamples, however, can be handled by a Kripke form of applicative bisimulations. Nevertheless, the counterexamples we have given in this paper give a more robust argument as to why applicative bisimulations are not appropriate semantics for higher-order languages with local state.

References

1. S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, Boston, MA, USA, 1990.
2. A. Ahmed, D. Dreyer, and A. Rossberg. State-dependent representation independence. *SIGPLAN Not.*, 44(1):340–353, 2009.
3. Amal Ahmed, Andrew W. Appel, Christopher D. Richards, Kedar N. Swadi, Gang Tan, and Daniel C. Wang. Semantic foundations for typed assembly languages. *ACM Trans. Program. Lang. Syst.*, 32(3):1–67, 2010.
4. N. Benton and V. Koutavas. A mechanized bisimulation for the nu-calculus. Technical Report MSR-TR-2008-129, Microsoft Research, September 2008.
5. N. Benton and B. Leperchey. Relational reasoning in a nominal semantics for storage. In *Proc. 7th International Conference on Typed Lambda Calculi and Applications (TLCA 2005)*, volume 3461 of *Lecture Notes in Computer Science*, pages 86–101, Berlin, Heidelberg, and New York, 2005. Springer.
6. D. Dreyer, G. Neis, and L. Birkedal. The impact of higher-order state and control effects on local relational reasoning. In *Proc. 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*, 2010.
7. A. D. Gordon and G. D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proc. 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 1996)*, pages 386–395, New York, NY, USA, 1996. ACM.
8. R. Jagadeesan, C. Pitcher, and J. Riely. Open bisimulation for aspects. In *Proceedings, International Conference on Aspect-Oriented Software Development*. ACM Press, 2007.
9. A. Jeffrey and J. Rathke. Towards a theory of bisimulation for local names. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1999.
10. V. Koutavas and M. Wand. Bisimulations for untyped imperative objects. In P. Sestoft, editor, *Proc. 15th European Symposium on Programming (ESOP 2006), Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 146–161, Berlin, Heidelberg, and New York, 2006. Springer-Verlag.
11. V. Koutavas and M. Wand. Small bisimulations for reasoning about higher-order imperative programs. In *Proc. 33rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL 2006)*, pages 141–152, New York, NY, USA, January 2006. ACM Press.
12. V. Koutavas and M. Wand. Reasoning about class behavior. Appeared in FOOL/WOOD 2007 Workshop, January 2007.
13. J. Laird. A fully abstract trace semantics for general references. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *ICALP*, volume 4596 of *Lecture Notes in Computer Science*, pages 667–679. Springer, 2007.
14. S. B. Lassen and P. B. Levy. Typed normal form bisimulation. In J Duparc and TA Henzinger, editors, *Proc., 23rd Conf. on Comp. Sci. and Logic*, volume 4646 of *LNCS*, 2007.
15. S. B. Lassen and P. B. Levy. Typed normal form bisimulation for parametric polymorphism. In *LICS*, pages 341–352. IEEE Computer Society, 2008.
16. P B Levy. Amb breaks well-pointedness, ground amb doesn't. In *Proc., 23rd Conf. on the Mathematical Foundations of Programming Semantics*, volume 173 of *ENTCS*, 2007.

17. I. A. Mason and C. L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1:287–327, 1991.
18. G. Neis, D. Dreyer, and A. Rossberg. Non-parametric parametricity. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 135–148, New York, NY, USA, 2009. ACM.
19. A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Proc. 18th International Symposium on Mathematical Foundations of Computer Science (MFCS 1993)*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141, Berlin, Heidelberg, and New York, 1993. Springer-Verlag.
20. D. Sangiorgi. The lazy lambda calculus in a concurrency scenario. *Information and Computation*, 111(1):120–153, 1994.
21. D. Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8:447–479, 1998.
22. D. Sangiorgi, N. Kobayashi, and E. Sumii. Environmental bisimulations for higher-order languages. In *Proc. 22th Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 293–302. IEEE Computer Society, 2007.
23. I. D. B. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, Cambridge, UK, December 1994. Also published as Technical Report 363, University of Cambridge Computer Laboratory.
24. K. Støvring and S. B. Lassen. A complete, co-inductive syntactic theory of sequential control and state. In Jens Palsberg, editor, *Semantics and Algebraic Specification*, volume 5700 of *Lecture Notes in Computer Science*, pages 329–375. Springer, 2009.
25. E. Sumii and B. C. Pierce. A bisimulation for dynamic sealing. *Theoretical Computer Science*, 375(1–3):169–192, May 2007.
26. E. Sumii and B. C. Pierce. A bisimulation for type abstraction and recursion. *Journal of the ACM*, 54(5):1–43, 2007.
27. J. Tiuryn and M. Wand. Untyped lambda-calculus with input-output. In H. Kirchner, editor, *Proc. 21st International Colloquium on Trees in Algebra and Programming (CAAP 1996)*, volume 1059 of *Lecture Notes in Computer Science*, pages 317–329, Berlin, Heidelberg, and New York, April 1996. Springer.