Available online at [www.sciencedirect.com](http://www.sciencedirect.com/)



[Electronic Notes in Theoretical Computer Science 276 (2011) 215–235](http://dx.doi.org/10.1016/j.entcs.2011.09.023)

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

From Applicative to Environmental Bisimulation [1](#_bookmark0)

Vasileios Koutavas[2](#_bookmark0)*,*[5](#_bookmark0)

*Trinity College Dublin, Ireland*

Paul Blain Levy[3](#_bookmark0)*,*[6](#_bookmark0)

*University of Birmingham, U.K.*

Eijiro Sumii[4](#_bookmark0)

*Tohoku University, Japan*

**Abstract**

We illuminate important aspects of the semantics of higher-order functions that are common in the presence of local state, exceptions, names and type abstraction via a series of examples that add to those given by Stark. Most importantly we show that any of these language features gives rise to the phenomenon that certain behaviour of higher-order functions can only be observed by providing them with arguments which internally call the functions again. Other examples show the need for the observer to accumulate values received from the program and generate new names. This provides evidence for the necessity of complex conditions for functions in the definition of environmental bisimulation, which deviates in each of these ways from that of applicative bisimulation.

*Keywords:* Environmental bisimulation, applicative bisimulation, local state, existential types.

# Introduction

Applicative bisimulation [[1](#_bookmark17)] provides a relational semantics that elegantly encodes extensionality of higher-order functions. It has been shown to be sound and com- plete with respect to contextual equivalence for pure lambda calculi [[1](#_bookmark17),[8](#_bookmark24)], object

1 Preliminary report appeared in the Dagstuhl Seminar Proceedings 10351 [[13](#_bookmark26)].

2 Email: [Vasileios.Koutavas@scss.tcd.ie](mailto:Vasileios.Koutavas@scss.tcd.ie)

3 Email: [P.B.Levy@cs.bham.ac.uk](mailto:P.B.Levy@cs.bham.ac.uk)

4 Email: [sumii@ecei.tohoku.ac.jp](mailto:sumii@ecei.tohoku.ac.jp)

5 Supported by SFI project SFI 06 IN.1 1898.

6 Supported by EPSRC Advanced Research Fellowship EP/E056091/1.

1571-0661 © 2011 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

[doi:10.1016/j.entcs.2011.09.023](http://dx.doi.org/10.1016/j.entcs.2011.09.023)

calculi [[7](#_bookmark23)], and languages with I/O [[33](#_bookmark48)]. However, several recent bisimulation se- mantics of languages (stateful and pure) with higher-order functions are based on *environmental bisimulation* [[31](#_bookmark47),[32](#_bookmark49),[27](#_bookmark43),[15](#_bookmark31),[5](#_bookmark21),[14](#_bookmark27),[16](#_bookmark32)], a definition with significantly more complex conditions. Despite its applicability to numerous higher-order languages, the additional complexity of environmental bisimulation has not been sufficiently justified. The question we ask in this paper (which we answer in the affirmative) is whether this complexity is necessary.

Both applicative and environmental bisimulation test related functions by ap- plying them to arguments, and require that the two applications equi-terminate and give related results. However, there are two ways in which environmental bisimula- tion deviates from applicative bisimulation, each by giving a degree of freedom to the context in the tests that it can make.

* *Accumulation of values:* In applicative bisimulation, the context may call only related functions it has just received from the related programs. In environmental bisimulation, it accumulates the functions it has received, so it can call them at any time, possibly multiple times.
* *Resourceful arguments:* In applicative bisimulation, the context supplies a single *closed* value to which related functions get applied. But in environmental bisimu- lation it supplies a single *open* value, closed by corresponding functions from the inventory, and therefore the related functions get applied to different arguments.

In a stateful language (or a language with names), the necessity of the first devi- ation is quite plausible. Accumulation is needed because a function may return a different value the second time it is applied. But the necessity of resourceful argu- ments appears more questionable. Even more so in a language with exceptions and polymorphism.

A related point was made by Mason and Talcott [[22](#_bookmark38)] in their study of a rela- tional theory for a language with general store. They gave an example inequivalence showing the need for the opponent to assign to global locations before each func- tion application [[22](#_bookmark38), Sec. 3]. Therefore they identified the need for extending the definition of applicative bisimulation to apply to effectful languages:

“[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.”

In this paper we investigate two versions of environmental bisimulations, one with no accumulation and one without resourceful arguments. We emphasize the bisimulation without resourceful arguments since it is the more subtle. We show by examples that these bisimulations are unsound in an array of languages that use state, exceptions, names, and polymorphism. In this way we identify common aspects in the behaviour of higher-order functions in all of these languages, and

justify the complexity in the definition of environmental bisimulation.

Our examples for exceptions and polymorphism are, we believe, the first in the literature that invalidate the defective bisimulations in these languages. However, our examples for state and names are not the first of their kind; Stark gave examples that embody the same principles [[28](#_bookmark44), pp. 24–25, examples 11 and 14, and p. 104, examples 12 and 14].

Additionally, in this paper we investigate the provision in the definition of en- vironmental bisimulation for the context to create new names at every step of the bisimulation, instead of only generating an arbitrary amount of names at the be- ginning. For deterministic languages we believe this is unnecessary. In the presence of nondeterminism, however, there is no single answer. We give novel examples that show the necessity of this provision for environmental bisimulation to be sound with respect to must-testing, in the presence of countable nondeterminism, and with respect to lower bisimilarity, in the presence of even finite nondeterminism. We believe that name generation at every step is unnecessary for may testing and, in the presence of finite nondeterminism, for must testing.

We start by examining a pure language (Sec. [2](#_bookmark1)), for which we define applicative and environmental bisimulation. We then study a language with general state (Sec. [3](#_bookmark4)). In this setting, we define the two defective versions of environmental bisimulation, and present simple examples that show their unsoundness with respect to contextual equivalence. We then present similar examples for a language with exceptions (Sec. [4.1](#_bookmark10)), names (Sec. [4.2](#_bookmark11)), and existential types (Sec. [4.3](#_bookmark15)). Finally, we present examples that show the unsoundness of fixed name-set bisimulation in the cases mentioned above (Sec. [5](#_bookmark16)).

# Bisimulations in a Pure Language

* 1. *The Language λ*

To understand the two deviations from applicative bisimulation, discussed in the introduction, we first review applicative and environmental bisimulation in a pure setting. We choose a call-by-value *λ*-calculus with recursion, which we call *λ* and will serve as the basis for the languages we study later on in this paper. The types of *λ* are given by

*A, B, C* ::= 0 *| A* + *A |* 1 *| A × A | A → A |* X *|* rec X*. A*

We use the syntax of “fine-grain call-by-value”, in which values and computations are distinguished, and returning and sequencing (to) are made explicit:

*V* ::= x *|* inl *V |* inr *V | ⟨⟩ | ⟨V, V ⟩ | λ*x*.M |* rec f *λ*x*.M |* fold *V M* ::= return *V | M* to x*. M | V V |* match *V* as *{}*

*|* match *V* as *{*inl x*. M,* inr x*. M} |* match *V* as *⟨⟩. M*

*|* match *V* as *⟨*x*,* y*⟩. M |* match *V* as fold x*. M*

We have typing judgements Γ *▶ M* : *A* and for values Γ *▶*v *V* : *A*, where Γ is a list of distinct closed-typed identifiers and *A* is a closed type, defined in the usual

inductive way. We write Γ *▶*v *−→ −→* v *−→*

def

*V* : *A* to mean Γ *▶*

*Vi* : *Ai*, for all *i < |A|*.

We abbreviate bool = 1 + 1 with true and false defined accordingly, abbreviate

diverge = (rec f *λ*x*.* fx)*⟨⟩* and write *M* to x*. N* as *M* ; *N* when x does not occur in *N* . We then define as usual *M ⇓B V* , for *▶ M* : *B* and *▶*v *V* : *B*.

def

* 1. *Ultimate Pattern Matching*

It is useful to note that any closed value consists of tags and functions. The tags constitute an *ultimate pattern* [[18](#_bookmark34)], and the functions constitute the *ﬁlling* of the pattern. For example, we divide the value

*⟨*inl *λ*x*.M,* inr *⟨*inl *λ*y*.N, ⟨⟩⟩⟩*

into the ultimate pattern *⟨*inl *−,* inr *⟨*inl *−, ⟨⟩⟩⟩* and the filling *λ*x*.M, λ*y*.N* .

To make this precise, we define for each type *A* a set ulpatt(*A*) of ultimate patterns *p*, each equipped with a list *H*(*p*) of function types. These sets are defined by induction:

* + 1. *− ∈* ulpatt(*A → B*) and *H*(*−* ) d=ef *A → B*

*A→B A→B*

* + 1. *⟨⟩ ∈* ulpatt(*1*) and *H*(*⟨⟩*) d=ef *ε*
    2. if *p ∈* ulpatt(*A*) and *pj ∈* ulpatt(*Aj*) then *⟨p, pj⟩∈* ulpatt(*A × Aj*) and *H*(*⟨p, pj⟩*) d=ef *H*(*p*) *· H*(*pj*)
    3. if *p ∈* ulpatt(*A*) then inl *p ∈* ulpatt(*A* + *Aj*) and *H*(inl *p*) d=ef *H*(*p*)
    4. if *p ∈* ulpatt(*Aj*) then inr *p ∈* ulpatt(*A* + *Aj*) and *H*(inr *p*) d=ef *H*(*p*)
    5. if *p ∈* ulpatt(*A*[rec X*. A/*X]) then fold *p ∈* ulpatt(rec X*. A*) and *H*(fold *p*) d=ef *H*(*p*).

Given *p ∈* ulpatt

v *−→*

v *−→*

(*A*), and list of values Γ *▶ V* : *H*(*p*), we define a value Γ *▶*

*p*[ *V* ]:

*A* in the obvious way. Unique decomposition is immediate:

**Theorem 2.1** *For any closed value ▶*v *V* : *A, there is unique p ∈* ulpatt(*A*) *and list*

v *−→ −→*

*of closed values ▶ W* : *H*(*p*) *such that V* = *p*[*W* ]*.*

* 1. *Applicative Bisimulation*

We now define our applicative bisimulation. It will be convenient for later extensions of this definition to give it as a set of tuples which we call *relatees*. A relatee is a

*−−−−→ −→ −→j*

tuple (*A → B*; *V* ; *V* ) consisting of

*−−−−→*

* + 1. a list of function types *A → B*
    2. v *−→ −−−−→*

a list of functions *▶ V* : *A → B*

* + 1. v *−→j −−−−→*

a list of functions *▶ V* : *A → B*.

These three zones of the relatee represent the public information known to the

*−→*

context (for this language this is only a list of types) and the two situations, *V* and

*−→j*

*V* , that we want to relate.

The conditions of applicative bisimulation say that when we apply corresponding

functions to the same closed value, the applications equi-terminate and the resulting values have the same ultimate pattern with bisimilar fillings.

**Definition 2.2** A set *R* of relatees is an *applicative bisimulation* when

*−−−−→ −→ −→j*

(*A → B*; *V* ; *V* ) *∈R* implies that for any

*−−−−→*

* + - * index *i < |A → B|*
      * and closed value *▶*v *U* : *Ai*,

*−→*

if *V U ⇓*

v *−→j*

*i Bi p*[*W* ] then there exists a filling *▶*

*W* : *H*(*p*) such that

*V j U ⇓*

*−→j*

*−→ −→j*

*i Bi p*[*W* ] and (*H*(*p*); *W* ; *W* ) *∈R*

and the converse of the above condition holds when *V j U ⇓*

*−→j*

*i Bi p*[*W* ].

**Definition 2.3** Closed terms *▶ M, Mj* : *A* are *applicatively bisimilar* when there exists applicative bisimulation R such that

*−→*

* + - * if *M ⇓*

v *−→j*

*j −→j*

*A p*[*W* ] then there exists *▶*

*−→ −→*

*j*

(*H*(*p*); *W* ; *W* ) is contained in R,

*W* : *H*(*p*) such that *M*

*⇓A p*[*W* ] and

* + - * the converse of the above condition holds when *Mj*

*−→j*

*⇓A p*[*W* ].

As we discussed in the introduction, Def. [2.2](#_bookmark2) is *non-accumulating* : the fi-

*−→ −→*

*j*

nal relatee (*H*(*p*); *W* ; *W* ) does not contain the functions in the starting relatee

*−−−−→ −→ −→j*

(*A → B*; *V* ; *V* ). Moreover, the definition is *non-resourceful* because it applies re-

lated functions to the same closed arguments.

* 1. *Environmental Bisimulation*

We define environmental bisimulation for the pure language *λ* to illustrate its dif- ferences from applicative bisimulation. In the following section we will adapt this definition for a language with state.

**Definition 2***−→***.4** A set *R* of relatees is an *environmental bisimulation* when

*−−−−→ −→*

(*A → B*; *V* ; *V*

*j*) *∈R* implies that for any

*−−−−→*

* index *i < |A → B|*

*−−−−−−−→*

v

and open value f : *A → B ▶ U* : *Ai*,

*−−→ −→*

v *−→j*

if *Vi U* [*V /*f] *⇓Bi p*[*W* ] then there exists a filling *▶*

*W* : *H*(*p*) such that

*j −−j →*

*−→j*

*−−−−→*

*−→ −→ −→j*

*−→j*

*Vi U* [*V*

*/*f] *⇓Bi p*[*W* ] and (*A → B · H*(*p*); *V · W* ; *V*

*· W* ) *∈R*

*j −−j → −→j*

and the converse of the above condition holds when *Vi U* [*V /*f] *⇓Bi p*[*W* ].

**Definition 2.5** Closed terms *▶ M, Mj* : *A* are *environmentally bisimilar* when there exists environmental bisimulation R such that

*−→*

* if *M ⇓*

v *−→j*

*j −→j*

*A p*[*W* ] then there exists *▶*

*−→ −→*

*j*

(*H*(*p*); *W* ; *W* ) is contained in R,

*W* : *H*(*p*) such that *M*

*⇓A p*[*W* ] and

* the converse of the above condition holds when *Mj*

*−→j*

*⇓A p*[*W* ].

As with applicative bisimulation, the conditions of environmental bisimulation require the applications of functions *Vi* and *V j*, related in the originating relatee, to equi-terminate and their resulting values to have the same ultimate pattern with bisimilar fillings. However, in environmental bisimulation the argument provided to

*i*

*j −−−−−−−→* v

each of *Vi* and *Vi* is constructed by closing the same open value f : *A → B ▶*

*U* : *Ai*

*−→ −→j*

with the functions

*V* and *V*

, respectively, from the inventory of the originating

relatee. This encodes the principle of *resourceful arguments*. Moreover, the con-

*−−−−→*

*−→ −→*

*−→j*

*−→j*

catenation of the types *A → B · H*(*p*) and the values *V · W* and *V · W*

relatee encode the *accumulation of values*.

# Environmental Bisimulation for State

in the final

We give a stateful language and the definition of environmental bisimulation for this language. We then give two simpler versions of bisimulation, one without resourceful arguments and one without accumulation, and demonstrate their unsoundness with respect to contextual equivalence.

* 1. *The language λs*

We add to the language *λ* the facility to generate fresh locations that may be assigned to and read from. Rather than treating locations as values (as for names in Sec. [4.2](#_bookmark11)), we use the syntax

*−−−−→*

*M* ::= *··· |* l := *V. M |* read l as x*. M |* new l := *V. M*

Thus locations can neither be stored nor returned. The typing judgements are now Δ; Γ *▶ M* : *A* and Δ; Γ *▶*v *V* : *A*, where Δ is a list of distinct closed-typed locations. For a state *s* we write Δ; Γ *▶*v *s* : Δ*j* to mean *dom*(*s*) = *dom*(Δ*j*) and Δ; Γ *▶*v *s*(*i*) : Δ*j*(*i*). Evaluation takes the form Δ*, s,M ⇓A* Θ*, t,V* , for term Δ; *▶ M* : *A*, state Δ; *▶ s* : Δ, location list Θ extending Δ, value Θ; *▶*v *V* : *A* and

state Θ; *▶ t* : Θ.

*−−−−→*

The operational semantics of the term new l := *V. M* uses a gensym operation

to generate fresh locations, which we call *private* since the context of the term has no direct access to them. The particular selection of the gensym operation leaves the semantics unaffected. Thus, without loss of generality, we assume that there is a countably infinite set of locations that is disjoint from the range of gensym. We call these locations *public* and use them in the following section as the domain

of the public state in our relations; i.e. the state to which the context has direct access.

* 1. *Environmental Bisimulation for State*

We shall write a location context as Δpub Δpriv, where Δpub is a public location context and Δpriv is a private location context, with domains the set of public and private locations, respectively. Likewise, we write a state as *s*pub *s*priv, and we call *s*pub a *public state* and *s*priv a *private state*. Note that every state is uniquely decomposable to a public and private state.

When the context calls a function, the public state provides additional commu- nication between a function and its context, besides the argument and the result. It is essentially another argument to the function and, when the function returns, it is another result which needs to be ultimately pattern-matched. Note that this is not the case for private state since the context has no access to it.

Thus, for a public location context Δpub, we write

ulpatt(*Δ*pub) =

Y

(l:*C*)*∈*Δpub

def

l *'→* ulpatt(*C*)

For *p ∈* ulpatt(*Δ*pub), we define *H*(*p*) to be the concatenation over (l : *C*) *∈* Δpub

v *−→* v *−→*

of *H*(*p*(l)). Then for any Δ; Γ *▶*

*W* : *H*(*p*), we define Δ; Γ *▶ p*[*W* ]: Δpub in the

obvious way. As an example, let l1 and l2 be public locations and

Δpub d=ef l1 : bool *×* bool*,* l2 :1 + (bool *→* bool) then the public state

Δpub; *▶*v l1 *'→ ⟨*true*,* false*⟩,* l2 *'→* inr *λ*x*.* read l1 as *⟨*y*,* z*⟩.* return y : Δpub

has ultimate pattern *p* d=ef l1 *'→ ⟨*true*,* false*⟩,* l1 *'→* inr *−*.

Again, we have unique decomposition:

**Theorem 3.1**

* + 1. *For any value* Δ; *▶*v *V* : *A, there is a unique p ∈* ulpatt

*−→*

*such that V* = *p*[*W* ]*.*

1. *and* Δ; *▶*

*−→*

*W* : *H*(*p*)

v

* + 1. *For any public state* Δ; *▶*v *s*pub : Δpub*, there is a unique p ∈* ulpatt(*Δ*pub) *and*

v *−→ −→*

Δ; *▶ W* : *H*(*p*) *such that s*pub = *p*[*W* ]*.*

Environmental bisimulation for *λs* is defined over *relatees* that are tuples of the

form (Δ

*−−−−→*

*−→ j*

*j −→j*

pub*, A → B*; Δpriv*, s*priv*, V* ; Δpriv*, s*priv*, V*

* a public location context Δpub

*−−−−→*

* a list of function types *A → B*
* a private location context Δpriv

), consisting of:

* + a private state Δpub Δpriv; *▶*v *s*priv : Δpriv
  + a list of functions Δ Δ

v *−→*

*−−−−→*

pub

priv; *▶*

*V* : *A → B*

* + a private location context Δ*j*

priv

* + a private state Δpub Δ*j*

priv

v *j*

priv

; *▶ s*

*j*

priv

: Δ

* + a list of functions Δ

Δ*j*

v *−→j*

*−−−−→*

pub

priv; *▶ V*

: *A → B*.

As in the applicative setting, a relatee is organized in three zones (separated by semicolons), representing public information and the two situations that we want to relate. Here the public information contains the type of the public state, besides the types of the functions. The other two zones contain the private state (and its type) that may be used in the functions.

The definition of environmental bisimulation for *λs* follows the same structure as that for *λ* (Def. [2.4](#_bookmark3)), with the addition of the creation of a public state before the applications, which is ultimately pattern-matched at the end.

**Definition 3.2** A set *R* of relatees is an *environmental bisimulation* when

*−−−−→*

(Δ

*−→ j*

*j −→j*

pub*, A → B*; Δpriv*, s*priv*, V* ; Δpriv*, s*priv*, V*

) *∈R*

implies that for any

*−−−−→*

* + index *i < |A → B|*
  + public location context Θpub extending Δpub
  + public state Θ

*−−−−−−−→* v :Θ

pub; f : *A → B ▶*

*s*pub

pub

* + and value Θ if

*−−−−−−−→*

pub; f : *A → B ▶*v

*−−→*

*U* : *Ai*,

*−−→*

*−→ −→*

Θpub Δpriv*, s*pub[*V /*f] *s*priv*, Vi U* [*V /*f] *⇓Bi* Θpub Θpriv*, q*[ *T* ] *t*priv*, p*[*W* ]

then there exists

* + a private location context Θ*j*

priv

Θ ; *▶*

* + a filling Θ

pub

*j*

priv

v *−→j*

: *H*(*q*)

* + a private state Θpub Θ*j*

*T*

priv

Θ ; *▶*

*W*

v *j*

priv

; *▶ t*

*j*

priv

: Θ

* + a filling Θ such that

pub

*j*

priv

v *−→j*

: *H*(*p*)

Θ Δ*j ,s*

*−−j → j*

*j −−j →*

Θ Θ*j*

*−→j j*

*−→j*

pub

priv

pub[*V*

*/*f] *s*priv*, Vi U* [*V*

*/*f] *⇓Bi*

pub

priv*, q*[*T* ] *t*priv*, p*[*W* ]

and

(Θ

*−−−−→*

*−→ −→ −→ j*

*j −→j*

*−→j*

*−→j*

pub*, A → B · H*(*q*) *· H*(*p*); Θpriv*, t*priv*, V · T · W* ; Θpriv*, t*priv*, V*

*· T · W* ) *∈R* (1)

Moreover, the converse condition holds when

Θ Δ*j ,s*

*−−j → j*

*j −−j →*

Θ Θ*j*

*−→j j*

*−→j*

pub

priv

pub[*V*

*/*f] *s*priv*, Vi U* [*V*

*/*f] *⇓Bi*

pub

priv*, q*[*T* ] *t*priv*, p*[*W* ]

**Definition 3.3** We say that closed terms ; *▶ M, Mj* : *A* are *environmentally bisim- ilar* when there exists an environmental bisimulation R such that

* + - if

*−→*

*ε, ε, M ⇓A* Θpriv*, t*priv*, p*[*W* ]

then there exists

* a private location context Θ*j*

priv

* a private state Θ*j*

priv

*W*

v *j*

priv

; *▶ t*

*j*

priv

: Θ

* a filling Θ*j* ; *▶*

priv

such that

v *−→j*

: *H*(*p*)

*ε, ε, Mj ⇓A* Θ*j*

priv

*j*

*, t*

priv

*−→j*

*, p*[*W* ]

*−→*

and (*ε, H*(*p*); Θ *,t*

*j , tj*

*−→j*

priv

priv*, W* ; Θpriv

priv*, W* ) is contained in R,

* + - the converse of the above condition holds when

*ε, ε, Mj ⇓* Θ*j*

*, tj*

*−→j*

*A* priv

priv*, p*[*W* ]

Def. [3.2](#_bookmark7) encompasses both deviations from applicative bisimulation discussed in the introduction; specifically:

* + - 1. The use of *open* argument Θ

*−−−−−−−→*

v *U* : *A*

and state *s*

*−−−−−−−→*

Θ v

pub; f : *A → B ▶* *i*

pub

pub; f : *A → B ▶ s*pub : Θpub encodes the *resourceful arguments* principle.

*−−−−→ −→ −→ −→*

* + - 1. The concatenation of the types *A → B · H*(*q*) *· H*(*p*) and the values *V · T · W*

*−→ −→ −→*

*j*

*j*

*j*

and *V · T · W*

in the final relatee encode the *accumulation of values*.

Dropping each of these principles leads to two alternative versions of Def. [3.2](#_bookmark7):

1. If we require Θpub; *▶*v *U* : *Ai* and Θpub; *▶*v *s*pub : Θpub, we say that *R* is a

*closed-argument bisimulation*.

1. If we replace ([1](#_bookmark8)) by (Θ

*,H*(*q*)*·H*(*p*); Θ *,t*

*−→ −→*

*j , tj*

*−→j −→j*

pub

priv

priv*, T ·W* ; Θpriv

priv*, T ·W* ) *∈ R*

we say that *R* is a *no-accumulation bisimulation*.

This also leads to two corresponding versions of relations for closed expressions. Clearly, if ; *▶ M, Mj* : *A* are environmentally bisimilar, then they are also closed- argument and no-accumulation bisimilar.

* 1. *Resourceful Arguments and Accumulation*

To show the necessity of resourceful arguments, we need to show closed-argument bisimulation unsound. The following example accomplishes that.

**Example 3.4** *(Resourceful Arguments)* Consider *M*1 and *Mj*

1

of type

(1 *→* 1) *→* bool:

*M*1 d=ef return *V*1

*Mj* d=ef new flag := true*.* return *V j*

1 1

where

*V*1 d=ef *λ*f:1 *→* 1*.* f *⟨⟩*; return true

*V j* d=ef *λ*f:1 *→* 1*.* read flag as

1

*{*true*.* flag := false; f *⟨⟩*; flag := true; return true false*.* return false*}*

The function *V j* returns a different value than *V*1 when flag = false, which is only the case within the extent of the application f *⟨⟩*. The following context distinguishes *M*1 and *Mj* :

1

1

*C*1 d=ef new record := true*.* [*·*] to g*.*

g (*λ*x*.* g (*λ*y*.* return y) to z*.* record := z; return *⟨⟩*);

read record as x*.* return x

The term *C*1[*M*1] returns true, while *C*1[*Mj* ] returns false. This context is clearly resourceful: the outer argument provided to g contains g itself.

1

However, the following set of relatees is a closed-argument bisimulation that

relates *V*1 to *V j*

1

with flag *'→* true. To prove that, it suffices to show that the

applications f *⟨⟩* in *V*1 and *V j* equi-terminate in stores with related private parts. This follows from the fact that f and the public state are constructed by closed values and thus the applications do not depend on, and do not change, the value

1

of the flag. Thus *M*1 and *Mj*

1

are closed-argument bisimilar, and closed-argument

bisimulation is unsound for *λs*.

Δ

pub*,* ((1 *→* 1) *→* bool *· A →B*); Δpriv

*−−−−→ −→*

*,s*

priv*,* (*V*1 *· T* );

(flag : bool *·* Δ*j*

)*,* (flag *'→* true *· s*

*j −→*

priv

*θ* : Δpriv *→* Δ*j*

priv; *▶*

priv*θ*)*,* (*V*1 *· T θ*) *|*

is a bijective renaming*,*

priv

*T* : *A →B,* Δpub

Δ Δ

pub

v *−→*

*−−−−→*

Δ ; *▶*v *s*

:Δ } *2*

We now provide an example that shows that accumulation is necessary in a bisimulation for state. Therefore no-accumulation bisimulation is unsound.

priv

priv

priv

**Example 3.5** *(Accumulation)* Consider *M*2 and *Mj* of type 1 *→* bool:

2

*M*2 d=ef return *V*2

*Mj* d=ef new flag := true*.* return *V j*

2 2

where

*V*2 d=ef *λ⟨⟩.* return true

*V j* d=ef *λ⟨⟩.* read flag as *{*true*.* flag := false; return true

2

false*.* return false*}*

The function *V j* is a function that returns true the first time it is applied and false

2

all subsequent times, whereas *V*2 always returns true.

The following context distinguishes *M*2 and *Mj* :

2

*C*2 d=ef [*·*] to f*.* f *⟨⟩*; f *⟨⟩*

The term *C*2[*M*2] returns true, while *C*2[*Mj* ] returns false.

2

We can show, however, that the following set of relatees is a no-accumulation

bisimulation, and therefore *M*2 and *Mj*

2

are no-accumulation bisimilar. Hence no-

accumulation bisimulation is unsound for *λs*.

Δpub*,* 1 *→* bool; *ε, ε, V*2; flag : bool*,* flag *'→* true*,V j* } *∪*

2

Δ

pub*, A →B*; Δpriv

*−−−−→ −→*

*,s*

priv*, T* ;

priv*θ*)*, T θ*

*|*

(flag : bool *·* Δ*j*

)*,* (flag *'→* false *· sj −→*

*θ* : Δpriv *→* Δ*j*

priv

priv

priv; *▶*

*T* : *A →B,*

pub

priv

priv

priv

is a bijective renaming*,*

Δ Δ

pub

v *−→*

*−−−−→*

Δ

Δ ; *▶*v *s*

:Δ } *2*

# Other Language Extensions

* 1. *Exceptions*

We add to the *λ* language the facility to generate fresh exceptions that may be raised and caught. Following [[4](#_bookmark19)] our syntax takes the form

*−−−−−−−−−→*

*M* ::= *··· |* new e*. M |* raise e *| M {*to x*. M,* catch e*. M*e*}*

*−−−−−−−−−→*

To evaluate *M {*to x*. P,* catch e*. M*e*}*, we first evaluate *M* , and if it returns *V*

we evaluate *P* [*V /*x]. If instead it raises an exception e, then if e appears in *−→*e

we proceed to evaluate *M*e, otherwise e remains raised. We define two big-step relations, for returning and raising respectively:

* Δ*,M ⇓A,* Θ*,V* for Δ; *▶ M* : *A*, exceptions Θ extending Δ and Θ; *▶*v *V* : *A*.
* Δ*,M* *A* Θ*,* e for a term Δ; *▶ M* : *A*, exceptions Θ extending Δ and exception e

appearing in Θ.

Ultimate pattern matching and environmental bisimulation—and its two defective variants—are defined as in Sec. [3.2](#_bookmark5).

**Example 4.1** *(Resourceful arguments)* Consider *M*3*,Mj* of type (1 *→* 1) *→* 1:

3

*M*3 d=ef new e*.* return *V*3 *Mj* d=ef new e*.* return *V j*

3 3

where

*V*3 d=ef *λ*f*.* f *⟨⟩*; raise e

*V j* d=ef *λ*f*.* f *⟨⟩ {*to x*.* raise e*,* catch e*.* return *⟨⟩}*

3

These terms are distinguished by the context

*C*3 d=ef [*·*] to g*.* g (*λ*x*.* g (*λ*y*.* return y))

The term *C*3[*M*3] raises an exception, while *C*3[*Mj* ] returns *⟨⟩*. However, the terms *M*3 and *Mj* are closed-argument bisimilar because the applications of *V*3 and *V j* to the same closed argument will have the same behaviour; such arguments will not be able to throw an exception e. *2*

3

3

3

**Example 4.2** *(Accumulation)* Let *M*4 and *Mj*

4

of type (1 *→* 1) *×*

(1 *→* (1 *→* 1) *→* bool) be the terms:

*M*4 d=ef new e*.* return *⟨λ⟨⟩.* raise e*, λ⟨⟩.* return *V*4*⟩ Mj* d=ef new e*.* return *⟨λ⟨⟩.* raise e*, λ⟨⟩.* return *V j⟩*

4 4

where

*V*4 d=ef *λ*f*.* f *⟨⟩ {*to x*.* return true*,* catch e*.* return false*}*

*V j* d=ef *λ*f*.* f *⟨⟩ {*to x*.* return true*,* catch e*.* return true*}*

4

A distinguishing context is

*C*4 d=ef [*·*] to *⟨*f*,* g*⟩.* g *⟨⟩* to h*.* hf

so that *C*4[*M*4] returns false and *C*4[*Mj* ] returns true. Note that the context *C*4 employs accumulation: it uses the function f, obtained before the application of g, as an argument to h, obtained as a result of the application of g. The terms *M*4 and *Mj* are no-accumulation bisimilar, because by the time a non-accumulating context has obtained h by applying g, it has discarded the function f.

4

4

Because a no-accumulation bisimulation uses resourceful arguments, to prove the above terms no-accumulation bisimilar, we need the following fact: for any Δ; y : (1 *→* 1) *→* bool *▶ M* : *B*, where Δ does not contain e, if Δ *·* e*,M* [*V*4*/*y] evaluates to Θ *·* e*,W* then there exists Θ; y : (1 *→* 1) *→* bool *▶ V* : *B* such that

*W* = *V* [*V*4*/*y] and Δ *·* e*,M* [*V j/*y] evaluates to Θ *·* e*,V* [*V j/*y], and likewise if it raises

4 4

an exception. This is proved by induction on the big-step relation. *2*

* 1. *Names*

We add to *λ* the facility to generate fresh names that are values and may be com- pared for equality, similar to the nu-calculus [[25](#_bookmark41),[28](#_bookmark44)]. Our syntax becomes

*A* ::= *··· |* name

*V* ::= *··· |* m

*M* ::= *··· |* new x*. M | V* = *V*

Unlike previous work on environmental bisimulation for languages with first- class names (e.g. [[31](#_bookmark47),[15](#_bookmark31),[5](#_bookmark21)]) here we do not relate names in the relatees. Instead, private names that are revealed by related functions to their context are renamed into identical public names by means of ultimate pattern-matching. For example, the value

m0*,* m1 n0*,* n1*,* n2; *▶*v

*⟨⟨*n2*,* m0*⟩, ⟨*n1*, ⟨*n2*,* inl *λ*x*.*return *⟨*n0*,* n1*⟩⟩⟩⟩* (2)

: (name *×* name) *×* (name *×* (name *×* ((1 *→* (name *×* name)) + 1)))

has ultimate pattern

*p* = *⟨⟨*m2*,* m0*⟩, ⟨*m3*, ⟨*m2*,* inl *−⟩⟩⟩* (3)

def

To obtain ([3](#_bookmark12)) algorithmically, we scan ([2](#_bookmark13)) from left to right, converting a private name encountered for the first time into a public one, replacing functions by *−* and retaining public names and tags. We encounter n2 before n1, therefore they are converted to m2 and m3 respectively.

We recover ([2](#_bookmark13)) from ([3](#_bookmark12)) by providing

* the filling of the hole under the new name scheme, viz. *λ*x*.*return *⟨*n0*,* m3*⟩*
* the list of converted private names in order of appearance, viz. n2*,* n1.

Once we have reformulated Thm. [3.1](#_bookmark6) along these lines, the notion of environmental bisimulation—and its two defective variants—is defined as in Sect. [3.2](#_bookmark5).

**Example 4.3** *(Resourceful arguments)* Let *M*5 and *Mj*

5

of type

(name *→* bool) *→* bool be

*M*5 d=ef new n*.* return *V*5 *V*5 d=ef *λ*f*.* fn

*Mj* d=ef return *V j V j* d=ef *λ*f*.* new n*.* fn

5 5 5

These are distinguished by

*C*5 d=ef [*·*] to g*.* g (*λ*n*.* g (*λ*m*.* m = n))

Evidently *C*5[*M*5] evaluates to true while *C*5[*Mj* ] evaluates to false. On the other hand, *M*5 and *Mj* are closed-argument bisimilar, because a closed argument cannot know about n. In particular, it cannot store n when applied to it, for future use, because the language does not allow storage of names. *2*

5

5

**Example 4.4** *(Accumulation)* The following are terms of type 1 *→* name:

*M*6 d=ef new n*.* return *V*6 *V*6 d=ef *λ⟨⟩.* return n

*Mj* d=ef return *V j V j* d=ef *λ⟨⟩.* new n*.* return n

6 6 6

These terms are only distinguished by a context that accumulates the function in its hole and applies it twice, such as the following:

*C*6 d=ef [*·*] to g*.* g *⟨⟩* to x*.* g *⟨⟩* to y*.* x = y

*C*6[*M*6] returns true but *C*6[*Mj* ] returns false. However *M*6 and *Mj*

are no-

6 6

accumulation bisimilar since when applied just once (even to resourceful arguments)

they behave the same. *2*

* 1. *Polymorphism*

We add polymorphism to *λ*, so our syntax of types and terms becomes

*A* ::= *··· |* X*.A |* ΣX*.A*

*V* ::= *··· |* ΛX*.M |* rec f ΛX*.M | ⟨A, V ⟩*

*M* ::= *··· | VA |* match *V* as *⟨*X*,* x*⟩. M*

Using the ultimate pattern-matching theorem developed in [[19](#_bookmark35)], we again formulate environmental bisimulation incorporating the principles of resourceful arguments and accumulation, and the two defective versions. The following examples show the necessity of these principles.

**Example 4.5** *(Resourceful arguments)* Consider the following existential packages of type ΣX*.* (X *→* (1 + X)) *→* bool:

*M*7 d=ef return *⟨*1*, V*7*⟩*

*Mj* d=ef return *⟨*bool*,V j⟩*

7 7

where

*V*7 d=ef *λ*f*.* f *⟨⟩* to *{*inl *⟨⟩.* return false*,*

inr *⟨⟩.* f *⟨⟩*; return true*}*

*V j* d=ef *λ*f*.* f false to *{*inl *⟨⟩.* return false

7

inr true*.* return false

inr false*.* f true as *{*inl *⟨⟩.* return true

inr true*.* return true inr false*.* return false*}}*

These are distinguished by the context

*C*7 d=ef [*·*] to *⟨*X*,* g*⟩.*

g (*λ*y:X*.* g (*λ*z:X*.* return inr y) to *{*true*.* return inr y*,*

false*.* return inl *⟨⟩}*)

The term *C*7[*M*7] *⇓* true, while *C*7[*Mj* ] *⇓* false. But *M*7 and *Mj*

are closed-

7 7

argument bisimilar, because a closed argument for *V*7 and *V j* has polymorphic type

7

X *→* (1 + X), and there are only three such: the constant functions *λ*x*.* return inl *⟨⟩* and *λ*x*.* diverge, and the function *λ*x*.* return inr x. Evidently *V*7 and *V j* behave the same way when applied to these arguments. *2*

7

Note that functions with existentially quantified argument type X (such as those in the following example) will necessarily need to be provided with resourceful argu- ments, because the context cannot construct closed values of type X. Thus one may think that, for this language, a weaker notion of resourcefulness might be sound. However, the preceding example demonstrates the need not just for a resourceful argument but for one in which a resource (value from the inventory) is used under *λ*, just as in the rest of the languages we studied so far.

**Example 4.6** *(Accumulation)* Accumulation is necessary since the context may receive new arguments to an old function. For example consider the terms *M*8 and

*M* of type ΣX*.* (1 *→* X) *×* (X *→* bool):

*j*

8

*M*8 d=ef return *⟨*1*, V*8*⟩ V*8 d=ef *⟨λ⟨⟩.* return *⟨⟩, λ*x*.* return true*⟩*

*Mj* d=ef return *⟨*1*,V j⟩ V*8 d=ef *⟨λ⟨⟩.* return *⟨⟩, λ*x*.* return false*⟩*

8 8

These are distinguished by the context

*C*8 d=ef [*·*] to *⟨*X*, ⟨*f*,* g*⟩⟩.* f *⟨⟩* to x*.* gx

Evidently *C*8[*M*8] returns true while *C*8[*Mj* ] returns false. However *M*8 and *Mj*

8 8

are no-accumulation bisimilar because when the context receives the two functions

it cannot distinguish the right-hand function since it does not yet have a value to apply them to. *2*

# Repeated Generation of Fresh Names

Another questionable point in the definition of environmental bisimulation is the ability of the context to add public names at each step of the bisimulation. Per- haps we could fix Δpub in Def. [3.2](#_bookmark7), giving a notion of Δpub*-bisimulation*, and then require terms ; *▶ M, Mj* : *A* to be Δpub-bisimilar for all Δpub? For the deterministic languages we considered so far, we believe this to be a sound modification.

In the presence of nondeterminism, however, there is no single answer; it depends on the kind of nondeterminism and on the contextual equivalence we consider. Here we study the extensions of the language with names from Sec. [4.2](#_bookmark11) with finite and countable nondeterminism. We believe the above restriction is sound for may-testing and, in the finitely non-deterministic setting, also for must-testing. Here we give two examples that show this to be unsound for must-testing, in the presence of countable nondeterminism, and for lower bisimilarity, in the presence of even finite nondeterminism.

In this section we abbreviate nat = rec X*.* 1+ X and namelist = rec x*.* 1+ name *×* X. We use the usual constructors zero and succ for nat, and empty and cons for namelist. We use the following functions:

* ; *▶*v member : (name*×*namelist) *→* bool tells us whether its first argument appears in its second
* ; *▶*v distnames : namelist *→* nat returns the number of distinct names in its argument
* ; *▶*v mkfreshlist : nat *→* namelist creates a list of fresh names of length equal to its argument
* ; *▶*v min : nat *×* nat *→* nat is the minimum function.
  1. *Countable Nondeterminism and Must-Testing*

We extend the language from Sec. [4.2](#_bookmark11) with both binary and countable (erratic) nondeterministic choice—of course the former is redundant. Our syntax is now

*M* ::= *··· | M* or *M |* choose x*. M*

where x has type nat. The bigstep semantics is given in the standard manner [[23](#_bookmark39)]. First, we inductively define a relation Δ*,M ⇓A* Θ*,V* , meaning that Δ*,M* may evaluate to Θ*,V* . Then, we coinductively define a predicate Δ*,M ⇑A*, meaning that Δ*,M* may diverge.

To soundly reason about both may-testing (possibility of convergence) and must- testing (impossibility of divergence) a set of relatees *R* needs to be a *convex* en- vironmental bisimulation, i.e. if *C* and *Cj* are *R*-related configurations, then any convergence step or divergence that one may perform can be imitated by the other.

If we do not require divergence to be imitated, then *R* is merely a *lower* environ- mental bisimulation [[34](#_bookmark50),[20](#_bookmark36)], which is sound for may-testing only.

**Example 5.1** *(Generation)* We consider the terms *M*9 and *Mj*

9

of type

*T* d=ef rec X*.* name *→* 1+ X:

*M*9 d=ef choose w*. V*9 *⟨*w*,* empty*⟩*

*Mj* d=ef *M*9 or *V j* empty

9 9

where

*V*9 d=ef rec (f : (nat *×* namelist) *→T* ) *λ*(c*,* lst)*.* return fold *λ*n*.*

member *⟨*n*,* lst*⟩* to *{* true*.* return inl *⟨⟩,* false*.* match c as *{*

zero*.* return inl *⟨⟩,*

succ y*.* f *⟨*y*,* cons *⟨*n*,* lst*⟩⟩* to g*.* return inr g *} }*

*V j* d=ef rec (f : namelist *→T* ) *λ*lst*.* return *λ*n*.*

9

member *⟨*n*,* lst*⟩* to *{*

true*.* return inl *⟨⟩,*

false*.* f (cons *⟨*n*,* lst*⟩*) to g*.* return inr g *}*

Here *M*9 and *Mj* return a “hungry function” that after accepting a name it returns another function that can accept more names, if the name is undoubtedly new to the function, and inl *⟨⟩*, otherwise. The term *M*9 grants the context an arbitrary finite number w of distinct names it can remember; *Mj* has the additional possibility

9

9

of granting an unbounded memory.

The two terms are distinguished by the context

*C*9 d=ef [*·*] to f*.*

(rec (h : *T →* 1) *λ*(fold y)*.* new n*.* yn to *{*

inl *⟨⟩.* return *⟨⟩,*

inr g*.* hg

*}*) f

Evidently *C*9[*Mj* ] is able to diverge, whereas *C*9[*M*9] is not. But for any fixed list of names Δpub, it is clear that *M*9 and *Mj* are convex Δpub-bisimilar. This example shows that, in order for environmental bisimulation to be sound for must-testing in the presence of countable nondeterminism, we cannot dispense with the provision for the context to generate fresh names at each step. *2*

9

9

* 1. *Lower Bisimulation as a Congruence*

**Example 5.2** *(Generation)* Let *M*10 and *Mj*

10

be the following terms of type

namelist *→* nat:

*M*10 d=ef choose w*.* return *V*10

*j* d=ef return *V j*

*M*

10

10

or *M*10

where

*V*10 d=ef *λ*lst*.* min*⟨*distnames(lst)*,* w*⟩*

*j* d=ef *λ*lst*.* distnames(lst)

*V*

10

The term *M*10 picks a number w (or diverges) and returns a function that can

count up to w distinct names in a list; *Mj*

10

has the additional possibility of returning

a function that can count an unbounded number of distinct names in a list.

Let *C*10 be the following context of type 1 *→* nat:

*C*10 d=ef [*·*] to f*.*

return *λ⟨⟩.* choose k*.* mkfreshlist(k) to lst*.* f lst

Now *C*10[*Mj*

10

] may return a function that, when applied to *⟨⟩*, may return any

natural number, whereas *C*10[*M*10] cannot do this. Therefore any conceivable notion

of lower bisimulation (or even lower simulation) must distinguish *C*10[*Mj*

10

] from

*C*10[*M*10], and hence, if it is a congruence, *Mj*

10

from *M*10. Yet for every fixed list

of names Δpub, *Mj*

10

and *M*10 are lower (in fact, even convex) Δpub bisimilar.

This example shows that, in order for lower environmental bisimulation to be a congruence (or simulation to be a precongruence) in the presence of nondeterminism, we cannot dispense with the provision for the context to generate fresh names at each step.

Note that for this example, we may replace countable nondeterministic choice by a variant that may also diverge, which is expressible using only binary nondeter- ministic choice (or) and recursion. The change is immaterial for lower bisimulation. *2*

# Conclusions

In early developments of Environmental Bisimulation [[31](#_bookmark47),[32](#_bookmark49)] it was realized that when we attempt to prove a contextual equivalence by brute force, we repeatedly find that the proof divides into two parts. One part is “boilerplate”; it does not change from example to example. The other part requires understanding of the particular example. Environmental bisimulation is simply a convenient way of pack- aging the boilerplate, so that only the example-specific part of the proof remains to be done. Once environmental relations are defined, the bisimulation conditions can be discovered by the soundness proof of these relations [[12](#_bookmark28)].

This paper motivates in retrospect the necessity of the complexity in the condi- tions of environmental bisimulation by giving a collection of examples in a variety of higher-order languages. The examples show, for these languages, the need to deviate in two significant ways from the standard notion of applicative bisimulation.

There are also other sound operational techniques for these languages which resemble applicative bisimulation even less. Notably open bisimulation [[19](#_bookmark35),[18](#_bookmark34),[26](#_bookmark42),[9](#_bookmark25)] and complete traces [[17](#_bookmark33)]; and logical relations in conjunction with step-indexing [[3](#_bookmark20),[2](#_bookmark18),[6](#_bookmark22)] and biorthogonal closure [[24](#_bookmark40)]. A logical relation is typically a single relation relating all contextually equivalent terms—thus accumulation is a given. It also supplies related arguments to related functions which, because the logical relation is a congruence, follows the principle of resourceful arguments. Open bisimulation and complete traces for languages similar to those studied here accumulate values [[17](#_bookmark33),[9](#_bookmark25)]. However, they follow a quite different approach for functions: they provide them with fresh identifiers, and make the applications of identifiers observable.

Jeffrey and Rathke [[10](#_bookmark29)] showed the unsoundness of an accumulating but closed- argument form of bisimulation for the nu-calculus. The addition of infinitely many global name references makes their bisimulation sound and complete. That language is unaffected by Ex. [4.3](#_bookmark14) because names may be stored, and unaffected by Ex. [3.4](#_bookmark9) because there are no local references. On the other hand, their notion of bisimulation for a fragment of Concurrent ML [[11](#_bookmark30)] both accumulates values and uses resourceful arguments.

The notion of environmental bisimulation in the polymorphic setting introduced in [[32](#_bookmark49)] allows the context to supply resourceful type arguments to polymorphic functions. It would be interesting to know whether this is necessary, in the light of genericity results such as those in [[21](#_bookmark37)].

# References

1. Abramsky, S., *The lazy lambda calculus*, in: D. A. Turner, editor, *Research Topics in Functional Programming*, Addison-Wesley, Boston, MA, USA, 1990 pp. 65–116.
2. Ahmed, A., A. W. Appel, C. D. Richards, K. N. Swadi, G. Tan and D. C. Wang, *Semantic foundations for typed assembly languages*, ACM Trans. Prog. Lang. Syst. **32** (2010), pp. 1–67.
3. Appel, A. W. and D. McAllester, *An indexed model of recursive types for foundational proof-carrying* *code*, ACM Trans. Prog. Lang. Sys. **23** (2001), pp. 657–683.
4. Benton, N. and A. Kennedy, *Exceptional syntax*, J. Funct. Program. **11** (2001), pp. 395–410.
5. Benton, N. and V. Koutavas, *A mechanized bisimulation for the nu-calculus*, Technical Report MSR- TR-2008-129, Microsoft Research (2008).
6. Dreyer, D., G. Neis and L. Birkedal, *The impact of higher-order state and control effects on local relational reasoning*, in: *ACM SIGPLAN International Conference on Functional Programming* (2010),

pp. 143–156.

1. Gordon, A. D. and G. D. Rees, *Bisimilarity for a first-order calculus of objects with subtyping*, in: *ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1996), pp. 386–395.
2. Howe, D. J., *Proving congruence of bisimulation in functional programming languages*, Information and Computation **124** (1996), pp. 103–112.
3. Jagadeesan, R., C. Pitcher and J. Riely, *Open bisimulation for aspects*, in: *International Conference on Aspect-Oriented Software Development* (2007).
4. Jeffrey, A. and J. Rathke, *Towards a theory of bisimulation for local names*, in: *IEEE Symposium on* *Logic in Computer Science* (1999).
5. Jeffrey, A. and J. Rathke, *A theory of bisimulation for a fragment of Concurrent ML with local names*, in: *IEEE Symposium on Logic in Computer Science, 2000*, 2000, pp. 311 –321.
6. Koutavas, V., “Reasoning about Imperative and Higher-Order Programs,” Ph.D. thesis, Northeastern University (2008).
7. Koutavas, V., P. B. Levy and E. Sumii, *Limitations of applicative bisimulation (preliminary report)*, in: A. Ahmed, N. Benton, L. Birkedal and M. Hofmann, editors, *Modelling, Controlling and Reasoning About State*, number 10351 in Dagstuhl Seminar Proceedings (2010).

URL <http://drops.dagstuhl.de/opus/volltexte/2010/2807>

1. Koutavas, V. and M. Wand, *Bisimulations for untyped imperative objects*, in: P. Sestoft, editor,

*European Symposium on Programming*, LNCS **3924** (2006), pp. 146–161.

1. Koutavas, V. and M. Wand, *Small bisimulations for reasoning about higher-order imperative programs*, in: *ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages* (2006), pp. 141– 152.
2. Koutavas, V. and M. Wand, *Reasoning about class behavior*, FOOL/WOOD Workshop (2007).
3. Laird, J., *A fully abstract trace semantics for general references*, in: L. Arge, C. Cachin, T. Jurdzinski and A. Tarlecki, editors, *International Colloquium on Automata, Languages and Programming*, LNCS **4596** (2007), pp. 667–679.
4. Lassen, S. B. and P. B. Levy, *Typed normal form bisimulation*, in: J. Duparc and T. Henzinger, editors,

*Computer Science and Logic*, LNCS **4646**, 2007.

1. Lassen, S. B. and P. B. Levy, *Typed normal form bisimulation for parametric polymorphism*, in: *LICS*

(2008), pp. 341–352.

1. Lassen, S. B. and C. S. Pitcher, *Similarity and bisimilarity for countable non-determinism and higher- order functions (extended abstract)*, Electronic Notes in Theoretical Computer Science **10** (1998),

pp. 246–266.

1. Longo, G., K. Milsted and S. Soloviev, *The genericity theorem and the notion of parametricity in the polymorphic λ-calculus*, in: *IEEE Logic in Computer Science, 1993. LICS ’93., Proceedings of Eighth* *Annual IEEE Symposium on*, 1993, pp. 6 –14.
2. Mason, I. A. and C. L. Talcott, *Equivalence in functional languages with effects*, Journal of Functional Programming **1** (1991), pp. 287–327.
3. Moran, A. K., *Natural semantics for non-determinism*, Licentiate Thesis, Chalmers University of Technology and University of G¨oteborg, Sweden (1994).
4. Pitts, A. and I. Stark, *Operational reasoning for functions with local state*, in: A. Gordon and A. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, Cambridge University Press, 1998 pp. 227–273.
5. Pitts, A. M. and I. D. B. Stark, *Observable properties of higher order functions that dynamically create local names, or: What’s new?*, in: *Mathematical Foundations of Computer Science*, LNCS **711** (1993),

pp. 122–141.

1. Sangiorgi, D., *The lazy lambda calculus in a concurrency scenario*, Information and Computation **111**

(1994), pp. 120–153.

1. Sangiorgi, D., N. Kobayashi and E. Sumii, *Environmental bisimulations for higher-order languages*, in:

*IEEE Symposium on Logic in Computer Science* (2007), pp. 293–302.

1. Stark, I. D. B., “Names and Higher-Order Functions,” Ph.D. thesis, University of Cambridge, Cambridge, UK (1994), Technical Report 363.
2. Sumii, E. and B. C. Pierce, *A bisimulation for dynamic sealing*, in: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2004), pp. 161–172.
3. Sumii, E. and B. C. Pierce, *A bisimulation for type abstraction and recursion*, in: *ACM SIGPLAN- SIGACT Symposium on Principles of Programming Languages* (2005), pp. 63–74.
4. Sumii, E. and B. C. Pierce, *A bisimulation for dynamic sealing*, Theoretical Computer Science **375**

(2007), pp. 169–192, (extended abstract appeared in [[29](#_bookmark45)]).

1. Sumii, E. and B. C. Pierce, *A bisimulation for type abstraction and recursion*, Journal of the ACM **54**

(2007), pp. 1–43, (extended abstract appeared in [[30](#_bookmark46)]).

1. Tiuryn, J. and M. Wand, *Untyped lambda-calculus with input-output*, in: H. Kirchner, editor, *Colloquium on Trees in Algebra and Programming*, LNCS **1059** (1996), pp. 317–329.
2. Ulidowski, I., *Equivalences on observable processes*, in: A. Scedrov, editor, *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science* (1992), pp. 148–161.