Counterexamples to Simulation in Non-Deterministic Call-by-Need Lambda-Calculi with letrec

Manfred Schmidt-Schauss[1] and Elena Machkasova[2] and David Sabel[1]

[1] Dept. Informatik und Mathematik, Inst. Informatik, J.W. Goethe-University,
PoBox 11 19 32, D-60054 Frankfurt, Germany,
{schauss,sabel}@ki.informatik.uni-frankfurt.de
[2] Division of Science and Mathematics,
University of Minnesota, Morris, MN 56267-2134, U.S.A
elenam@morris.umn.edu

Technical Report Frank-38

Research group for Artificial Intelligence and Software Technology
Institut für Informatik,
Fachbereich Informatik und Mathematik,
Johann Wolfgang Goethe-Universität,
Postfach 11 19 32, D-60054 Frankfurt, Germany

August 10, 2009

Abstract. This note shows that in non-deterministic extended lambda-calculi with letrec, the tool of applicative (bi)simulation is in general not usable for contextual equivalence, by giving a counterexample adapted from data flow analysis. It also shown that there is a flaw in a lemma and a theorem concerning finite simulation in a conference paper by the first two authors.

1 Introduction and Related Work

In this note we discuss the problem of whether applicative (bi)simulation can be applied to non-deterministic lambda calculi with letrec and show that there are limitations.
In particular, we will adapt a counterexample to a problem for non-deterministic data flow programs in [Pan95,Rus89]. This will also show that the claims on comparing abstractions in the approach of finite simulation for letrec-calculi is wrong in [SSM08a]. For more proofs of the properties of this calculus see [SSM08b].
A paper that discusses similar problems and counterexamples in a call-by-value calculus with `amb` is [Lev07]: it shows that there is no well-pointed denotational semantics for a call-by-value calculus with `amb`, which also means that applicative (bi)simulation fails.

$$
\begin{array}{ll}
(s\ t)^{S\vee T} & \to (s^S\ t)^V \\
(\texttt{letrec}\ Env\ \texttt{in}\ t)^T & \to (\texttt{letrec}\ Env\ \texttt{in}\ t^S)^V \\
(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C[x^S]) & \to (\texttt{letrec}\ x = s^S, Env\ \texttt{in}\ C[x^V]) \\
(\texttt{letrec}\ x = s, y = C[x^S], Env\ \texttt{in}\ r) & \to (\texttt{letrec}\ x = s^S, y = C[x^V], Env\ \texttt{in}\ r) \\
 & \qquad \text{if } C \neq [.] \\
(\texttt{seq}\ s\ t)^{S\vee T} & \to (\texttt{seq}\ s^S\ t)^V \\
(\texttt{case}\ s\ alts)^{S\vee T} & \to (\texttt{case}\ s^S\ alts)^V
\end{array}
$$

Fig. 1. The labeling to find the normal-order redex

2 The Counterexample

First we present the example using the syntax of the calculus L_S in [SSM08a]. Later we show how the example can be adapted to different calculi.

2.1 The Syntax

The syntax for expressions E in the call-by-need calculi L and L_S in [SSM08b,SSM08a] is as follows:

$$
\begin{array}{l}
E ::= V \mid (c\ E_1 \ldots E_{\mathrm{ar}(c)}) \mid (\texttt{seq}\ E_1\ E_2) \mid (\texttt{case}_T\ E\ Alt_1 \ldots Alt_{\#(T)}) \mid (E_1\ E_2) \\
\qquad (\texttt{choice}\ E_1\ E_2) \mid (\lambda\ V.E) \mid (\texttt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \texttt{in}\ E) \\
Alt ::= (Pat\ \to\ E) \qquad Pat ::= (c\ V_1 \ldots V_{\mathrm{ar}(c)})
\end{array}
$$

where E, E_i are expressions, V, V_i are variables, and c denotes a constructor. Expressions $(\texttt{case}_T \ldots)$ have exactly one alternative for every constructor of type T. We assume that types consist of pairwise disjoint sets of constructors with a given arity.
The normal-order reduction is defined in [SSM08a] and also weak head normal forms (WHNFs), see also Figures 1 and 2, where an L_S-*WHNF* is defined as an abstraction or a cv-expression (an expressions of the form $(c\ x_1 \ldots x_n)$, where c is a constructor and x_i are variables), or an expression $(\texttt{letrec}\ Env\ \texttt{in}\ v)$, where v is an abstraction or a cv-expression. We will use the calculus L_S in the following. Note that it is not essential which calculus (of those defined in [SSM08a]) we choose, since they are shown to be equivalent w.r.t. contextual equivalence.
An example for a normal-order reduction in L_S is $(\lambda x.x)\ ((\lambda y.y)\ (\lambda z.z)) \to (\texttt{letrec}\ x = ((\lambda y.y)\ (\lambda z.z))\ \texttt{in}\ x) \to (\texttt{letrec}\ x = (\texttt{letrec}\ y = (\lambda z.z)\ \texttt{in}\ y)\ \texttt{in}\ x) \to (\texttt{letrec}\ x = y, y = \lambda z.z\ \texttt{in}\ x) \to (\texttt{letrec}\ x = \lambda z'.z', y = \lambda z.z\ \texttt{in}\ x) \to (\texttt{letrec}\ x = \lambda z'.z', y = \lambda z.z\ \texttt{in}\ \lambda z''.z'')$, where the final term is a weak head normal form (WHNF) for the calculus L_S. An expression s is may-convergent ($s{\downarrow}$) iff there is a normal-order reduction sequence starting with s and ending in a WHNF. Two expressions s, t are contextually equivalent, $s \sim t$, if $s \leq_c t$ and $t \leq_c s$ where $s \leq_c t$ iff for all contexts $C[\cdot] : C[s]{\downarrow} \implies C[t]{\downarrow}$. In

(lbeta)	$((\lambda x.s)^S\ r) \to (\texttt{letrec}\ x = r\ \texttt{in}\ s)$
(cp-in)	$(\texttt{letrec}\ x = v^S, Env\ \ \texttt{in}\ C[x^V])$
	$\to (\texttt{letrec}\ x = v, Env\ \ \texttt{in}\ C[v])$
	where v is an abstraction or a cv-expression
(cp-e)	$(\texttt{letrec}\ x = v^S, Env, y = C[x^V]\ \texttt{in}\ r)$
	$\to (\texttt{letrec}\ x = v, Env, y = C[v]\ \texttt{in}\ r)$
	where v is an abstraction or a cv-expression
(abs)	$(c\ t_1 \dots t_n)^{S \vee T} \to (\texttt{letrec}\ x_1 = t_1, \dots, x_n = t_n\ \texttt{in}\ (c\ x_1 \dots x_n))$
	if $(c\ t_1 \dots t_n)$ is not a cv-expression
(llet-in)	$(\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ r)^S)$
	$\to (\texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ r)$
(llet-e)	$(\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ s_x)^S\ \texttt{in}\ r)$
	$\to (\texttt{letrec}\ Env_1, Env_2, x = s_x\ \texttt{in}\ r)$
(lapp)	$((\texttt{letrec}\ Env\ \texttt{in}\ t)^S\ s) \to (\texttt{letrec}\ Env\ \texttt{in}\ (t\ s))$
(lcase)	$(\texttt{case}_T\ (\texttt{letrec}\ Env\ \texttt{in}\ t)^S\ alts) \to (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{case}_T\ t\ alts))$
(seq-c)	$(\texttt{seq}\ v^S\ t) \to t$ if v is a value
(lseq)	$(\texttt{seq}\ (\texttt{letrec}\ Env\ \texttt{in}\ s)^S\ t) \to (\texttt{letrec}\ Env\ \texttt{in}\ (\texttt{seq}\ s\ t))$
(case)	$(\texttt{case}\ (c\ t_1 \dots t_n)^S \dots ((c\ y_1 \dots y_n) \to s) \dots)$
	$\to\ \ (\texttt{letrec}\ y_1 = t_1, \dots, y_n = t_n\ \texttt{in}\ s)$
(choice-l)	$(\texttt{choice}\ s\ t)^{S \vee T} \to s$
(choice-r)	$(\texttt{choice}\ s\ t)^{S \vee T} \to t$

Fig. 2. Reduction rules of L_S

[SSM08a] it is also proved that all reductions with the exception of the `choice`-reduction are correct w.r.t. $\sim$, i.e. if $s \to t$ where the used reduction rule is not a `choice`-reduction, then $s \sim t$.

2.2 The Counterexample

One of the properties that a finite simulation is based on is the ability to identify contextually equivalent expressions based on their behavior on all substitutions that substitute values for free variables. For instance, if two expressions with a free variable x behave the same for all substitutions for x; or alternatively, if they behave the same for all contexts of the form $(\texttt{letrec}\ x_1 = v_1, \dots, x_n = v_n\ \texttt{in}\ [\cdot])$, where v_i are closed values then these two expressions are said to be in a finite simulation relation. Intuitively, this should imply that the expressions are also contextually equivalent (see Proposition 7.2 in [SSM08a] for a precise statement; but note that the proof has a flaw).
Now we describe a counterexample to this property inspired by Panangaden [Pan95,Rus89] for data-flow analysis, adapted to our calculus L_S. Specifically, we consider all substitutions for a free variable that come from the set of so-called pseudo-values (defined below) and show that although the two expressions that we constructed behave the same on all pseudo-values, there are contexts that distinguish them.

The expressions s_1, s_2 are defined as follows, where `isList` is defined as $\lambda xs.\texttt{case}_{\text{list}}\ xs\ \texttt{of}\ (\texttt{Nil} \to \texttt{True})\ ((\texttt{Cons}\ x\ xs) \to \texttt{True})$, and $\texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ c$ is an abbreviation for $\texttt{case}_{\text{bool}}\ a\ (\texttt{True} \to b)\ (\texttt{False} \to c)$. Let Ω stand for a closed non-converging expression (all such expressions are equivalent in our calculus).

$$
\begin{array}{l}
s_1 := (\texttt{choice}\ (\texttt{Cons}\ 0\ (\texttt{if}\ (\texttt{isList}\ xs)\ \texttt{then}\ (\texttt{Cons}\ 0\ \texttt{Nil})\ \texttt{else}\ \Omega)) \\
\qquad\qquad (\texttt{if}\ (\texttt{isList}\ xs)\ \texttt{then}\ (\texttt{Cons}\ 0\ (\texttt{Cons}\ 1\ \texttt{Nil}))\ \texttt{else}\ \Omega)) \\
s_2 := (\texttt{choice}\ (\texttt{Cons}\ 0\ (\texttt{if}\ \ (\texttt{isList}\ xs)\ \texttt{then}\ (\texttt{Cons}\ 0\ \texttt{Nil})\ \texttt{else}\ \Omega)) \\
\qquad\qquad (\texttt{choice}\ (\texttt{if}\ (\texttt{isList}\ xs)\ \texttt{then}\ (\texttt{Cons}\ 0\ (\texttt{Cons}\ 1\ \texttt{Nil}))\ \texttt{else}\ \Omega) \\
\qquad\qquad\qquad (\texttt{Cons}\ 0\ (\texttt{if}\ (\texttt{isList}\ xs)\ \texttt{then}\ (\texttt{Cons}\ 1\ \texttt{Nil})\ \texttt{else}\ \Omega))))
\end{array}
$$

These two expressions are indistinguishable, if compared for all substitutions $\sigma := [v/x]$, where v ranges over all closed pseudo-values. Closed pseudo-values are expressions built from constructors, Ω, and abstractions. The test to distinguish two expressions is to ask for convergence; decomposing data structures is permitted.
It is sufficient to check the following expressions and lists:

- $xs = \Omega$ or a data-object that is a non-list, i.e. where $(\texttt{isList}\ xs)\Uparrow$. Then s_1 may reduce either to Ω or to an expression that is contextually equivalent to $(\texttt{Cons}\ 0\ \Omega)$. Here we used the correctness of garbage collection as a program transformation. The possibility of non-convergence is irrelevant for may-convergence. The same for s_2.
- $xs = \texttt{Nil}$ or $= \texttt{Cons}\ a\ b$ for any a, b. Then s_1 may reduce to expressions $(\texttt{Cons}\ 0\ (\texttt{Cons}\ 0\ \texttt{Nil}))$ or to $(\texttt{Cons}\ 0\ (\texttt{Cons}\ 1\ \texttt{Nil}))$ (modulo contextual equivalence). The expression s_2 has the possibility to reduce to expressions that are contextually equivalent either to $(\texttt{Cons}\ 0\ (\texttt{Cons}\ 0\ \texttt{Nil}))$ or to $(\texttt{Cons}\ 0\ (\texttt{Cons}\ 1\ \texttt{Nil}))$.

This means that for forms of applicative (bi)simulation, the expressions s_1, s_2 cannot be distinguished.
However, the two expressions are not contextually equivalent, as seen using the following context: $C := (\texttt{letrec}\ xs = [\cdot]\ \texttt{in}\ xs)$. The expression $C[s_1]$ can only be reduced to a diverging expression or to an expression contextually equivalent to $\texttt{Cons}\ 0\ (\texttt{Cons}\ 0\ \texttt{Nil}))$. The expression $C[s_2]$ can evaluate to expressions that are contextually equivalent to Ω, $(\texttt{Cons}\ 0\ (\texttt{Cons}\ 0\ \texttt{Nil}))$ or to $(\texttt{Cons}\ 0\ (\texttt{Cons}\ 1\ \texttt{Nil}))$. We add a further context D that tests for the second element of a list and terminates if this element is equal to 1, otherwise diverges. Then $D[C[s_1]]\Uparrow$, but $D[C[s_2]]\downarrow$, hence the expressions s_1, s_2 are not contextually equivalent.

2.3 Consequences of the Counterexample

The counterexample shows that Proposition 7.2 in [SSM08b,SSM08a] is not correct as claimed. The proof has a gap, since reduction contexts of the form

(`letrec` $Env, x = [\cdot]$ `in` r) and similar cycle-creating contexts were not considered; and also Theorem 7.3 is not correct. Hence the simulation method as described there cannot be used for abstractions.

2.4 Variations of the Counterexample and Tests

2.4.1 Restricting to Closed Expressions The counterexample is also valid if the two expressions are closed: $s_1' := \lambda xs.s_1$, $s_2' := \lambda xs.s_2$. Then s_1', s_2' cannot be distinguished in an applicative (bi)simulation style, i.e. if applied to any pseudo-value. The proof is the same as for s_1, s_2. But s_1', s_2' are not contextually equivalent, which can be seen using the context $C' := (\texttt{letrec}\ y = [\cdot]\ y\ \texttt{in}\ y)$. Note that $C'[s_1'] = (\texttt{letrec}\ y = s_1'\ y\ \texttt{in}\ y)$, which reduces to $(\texttt{letrec}\ y = s_1, xs = y\ \texttt{in}\ y) \sim (\texttt{letrec}\ xs = s_1\ \texttt{in}\ xs)$ using the correct program transformations in [SSM08b]. We see that $C'[s_1'] \sim_c C[s_1]$, and $C'[s_2'] \sim_c C[s_2]$, which means that s_1', s_2' are not contextually equivalent. This example also shows that it is not sufficient to take contexts of the form $(\texttt{letrec}\ Env, x = [\cdot]\ \texttt{in}\ x)$ into account for the simulation test since such contexts cannot distinguish the elements s_1' and s_2'.

2.4.2 Applicative Bisimulation Testing all Expressions If the condition for equivalence of expressions under applicative (bi)simulation is that they must not be distinguishable by arbitrary substitution or by using arbitrary closing environments of the form $(\texttt{letrec}\ Env\ \texttt{in}\ [\cdot])$, then the counterexample remains valid since only one additional case has to be considered: when $xs = \texttt{choice}\ \Omega\ \texttt{Nil}$. The expression $\sigma(s_1)$ may evaluate to $\Omega, \texttt{Cons}\ 0\ \Omega, \texttt{Cons}\ 0\ (\texttt{Cons}\ 0\ \texttt{Nil}))$ or $\texttt{Cons}\ 0\ (\texttt{Cons}\ 1\ \texttt{Nil}))$. The same holds for s_2. Thus they remain indistinguishable by applicative bisimulation, but are not contextually equivalent, using the same argument as above.

3 The Counterexample in Other Calculi

3.1 Calculi with only a Boolean Choice

Note that the counter-example does not rely on unrestrained nondeterminism (in the sense of [SS92]) provided by choice. "Unrestrained" means that choice can be applied to any expressions, whereas "restrained" limits arguments of choice to atomic values. In our case using a simpler (atomic) choice on Booleans is sufficient to encode the unrestrained nondeterminism. This follows, since the following law is easy to prove using the diagram techniques and the context lemma for may- and must-convergence, (see e.g. [SSS08] and [SSM08b] for the context-lemma for may-convergence):

$$\texttt{choice}\ s\ t \sim \texttt{if choice True False then}\ s\ \texttt{else}\ t$$

This translation does not work for (bottom-avoiding) amb, and it appears to be impossible to encode amb-expressions using restricted amb-expressions with

certain small arguments. Hence call-by-need calculi with letrec and choice do not exhibit an improved behavior as it is claimed in [Lev07] for certain forms of amb-calculi when amb is restricted to arguments of ground type.

3.2 Typed Calculi

The counterexample in L_S is polymorphically typable, since all the constructors and functions have a consistent polymorphic type. The same for the cyclic context and the testing contexts. Hence, the counterexample remains valid in a typed variant of the L_S-calculus.

3.3 Calculi With a Nonrecursive let

Note that this counterexample does not work in the non-deterministic calculus of [MSS06], which is rather similar to the calculus considered here with the only difference that a non-recursive let is used. Plugging s_1, s_2 in a fixpointing context results in expressions like $Y\ (\lambda xs.s_1)$ and $Y\ (\lambda xs.s_2)$, which on evaluation are permitted to copy the abstractions $(\lambda xs.s_i)$, and hence the effect of `letrec` to provide an immediate combination of recursion and sharing is not possible in that calculus.

3.4 Calculi with `amb`

A further consequence is that applicative (bi)simulation methods cannot be applied to the calculus in [Sab08], which is a call-by-need lambda-calculus with `amb`, `letrec`, `case`, and constructors. This is easy to check for may-convergence, since the same reasoning as above is valid. The arguments in subsection 2.2 also show that the must-convergence behavior of the terms s_1, s_2 is identical, if checked for all replacements of values for xs. But note that with contexts it is possible to distinguish s_1 and s_2 also by their must-convergence behaviour only: let D be a context that takes the second element of a list and let $D' = \texttt{if}\ (\texttt{amb}\ D\ 0) = 0\ \texttt{then}\ 0\ \texttt{else}\ \bot$. Then we have $D'[C[s_1]]\Downarrow$ while $D'[C[s_2]]\uparrow$ where C is defined as $\texttt{letrec}\ xs = [\cdot]\ \texttt{in}\ xs$.

3.5 On a Conjecture on Behavioral Simulation

The Conjecture 14.5. in [SSSS04] which claims that applicative simulation implies contextual equivalence in a non-deterministic letrec-calculus, is wrong. Note that the suspicion that it may be too hard to prove or may even be wrong lead to another successful approach manifested in [SSSS05] and [SSSS08].

3.6 The Fudget-Calculus with letrec, Choice, Case and Constructors

The call-by-need non-deterministic calculus in [MSC99,MSC03] comprises expressions with letrec, case and constructors, and uses a contextual semantics with may and (total) must-convergence. Our counterexample also shows that applicative bisimulation is not applicable for this calculus even if only the may-convergence is used.

3.7 Counterexample in Typed Calculus without Case and Constructors but with seq

In this section we investigate whether the counterexample can be adapted for an untyped non-deterministic call-by-need calculus without case and constructors, The syntax is:

$$E ::= V \mid (E_1\ E_2) \mid (\texttt{choice}\ E_1\ E_2) \mid$$
$$(\texttt{seq}\ E_1\ E_2) \mid (\lambda\ V.E) \mid (\texttt{letrec}\ V_1 = E_1, \ldots, V_n = E_n\ \texttt{in}\ E)$$

where E, E_i are expressions, and V, V_i are variables
The normal-order reduction and the notion of WHNF is as before.
A Church-like encoding of numbers and lists (in Haskell-style) is as follows :

```
cnil        = \c n -> n
cisnil      = \l -> l (\h t -> cfalse) ctrue
cislist    = \l -> l (\h t -> ctrue) ctrue
ccons       = \h t c n -> c h (t c n)
chead       = \l -> l (\h t -> h) cfalse
ctail       = \l -> cfst (l (\x p -> cpair (csnd p) (ccons x (csnd p)))
                            (cpair cnil cnil))
cpair       = \x y z -> z x y
cfst        = \p -> p (\x y -> x)
csnd        = \p -> p (\x y -> y)
ctrue       = \a b -> a
cfalse      = \a b -> b
cifthenelse = \test thenclause elseclause ->   test thenclause elseclause
ciscons     = \l -> cifthenelse (cisnil l) cfalse ctrue
one         = ctrue
zero        = cfalse

s1          = choice (ccons zero (seq xs (ccons zero cnil)))
                     (seq xs (ccons zero (ccons one cnil)))
s2          = choice (ccons zero (seq xs (ccons zero cnil)))
                     (choice (seq xs (ccons zero (ccons one cnil)))
                             (ccons zero (seq xs (ccons one cnil))))

s1test      =  let xs = [s1] in chead (ctail xs)
s2test      =  let xs = [s2] in chead (ctail xs)
```

$[s1]$ and $[s2]$ means the textual replacement

Fig. 3. Church-like encoding of the Counterexample

Note that the code in Figure 3 can be made executable in Haskell (except for `choice` which would require an extension) The same arguments as above show

that the counterexample is also valid in this calculus using the encoding in Figure 3.

3.8 Counterexample in an Untyped Calculus with `letrec` but without Case and Constructors and seq

The Church-like encoding above, of course, also works if the calculus is untyped by simply dropping the types. Replacing the `seq`-expression by an `isList`-application as before can be done, where the encoding of `islist` as above is used.

3.9 Typed Calculus without Case and Constructors

We could not decide the question whether there is an adaptation of the counterexample in a polymorphically typed call-by-need calculus without case, constructors and seq. Trying the Church-like encoding results in an error-message generated by the type checker complaining about infinite types. Some experimentation and analysis shows that it is also not possible to encode variations of the counterexample. So this calculus variant appears to be exceptional.
This leads to the conjecture that applicative bisimulation (or finite simulation) may be valid as a tool for recognizing contextual equivalence in a polymorphically typed call-by-need calculus with choice and letrec, but without case, constructors and seq.

3.10 Call-By-Value Calculi

An investigation in a concurrent call-by-value calculus with futures is in [NSS06,NSSSS07,SSNSS09]. The so-called futures are like letrec-bound top variables in the calculus and all its variants. In the latter reference, the calculus $\lambda^{\tau}(\mathsf{fc})$ is a typed version of the calculus with lists, where the counterexample can be encoded and where all arguments are valid. Hence applicative simulation cannot be used as a tool for contextual equivalence in $\lambda^{\tau}(\mathsf{fc})$. In all variants of the calculus in [NSSSS07,SSNSS09], the counterexample can be encoded. The counterexample can also be encoded in [NSS06], which is a typed calculus without data structures, since call-by-value can enforce evaluation, so no `seq` is needed. Thus the exceptional case for call-by-need calculi in 3.9 does not show up for call-by-value calculi with mutually recursive futures.

3.11 Conclusion and Further Work

We are investigating further restrictions that enable the finite simulation method. For example, deterministic letrec calculi appear to permit simulation methods, however, as far as we know there is no proof yet. We are also studying further the generality of the counterexample.

Acknowledgments

We thank Paul B. Levy for insightful discussions on the subject of this paper.

References

Lev07. P. B. Levy. Amb breaks well-pointedness, ground amb doesn't. *Electron. Notes Theor. Comput. Sci.*, 173(1):221–239, 2007.

MSC99. Andrew K. D. Moran, David Sands, and Magnus Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. In *Coordination '99*, volume 1594 of *Lecture Notes in Comput. Sci.*, pages 85–102. Springer-Verlag, 1999.

MSC03. Andrew K. D. Moran, David Sands, and Magnus Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. *Sci. Comput. Program.*, 46(1-2):99–135, 2003.

MSS06. Matthias Mann and Manfred Schmidt-Schauß. How to prove similarity a precongruence in non-deterministic call-by-need lambda calculi. Frank report 22, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, January 2006.

NSS06. Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. *Theoret. Comput. Sci.*, 364(3):338–356, November 2006.

NSSSS07. Joachim Niehren, David Sabel, Manfred Schmidt-Schauß, and Jan Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. *Electron. Notes Theor. Comput. Sci.*, 173:313–337, 2007.

Pan95. Prakash Panangaden. The expressive power of indeterminate primitives in asynchronous computation. In *Proceedings of FSTTCS*, pages 124–150, 1995.

Rus89. J.R. Russell. Full abstraction for nondeterministic dataflow networks. In *Proceedings of FOCS*, pages 170–175, 1989.

Sab08. David Sabel. *Semantics of a Call-by-Need Lambda Calculus with McCarthy's amb for Program Equivalence*. Dissertation, Goethe-Universität Frankfurt, Institut für Informatik. Fachbereich Informatik und Mathematik, 2008.

SS92. H. Søndergard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.

SSM08a. Manfred Schmidt-Schauß and Elena Machkasova. A finite simulation method in a non-deterministic call-by-need calculus with letrec, constructors and case. In *Proc. of RTA 2008*, number 5117 in LNCS, pages 321–335. Springer-Verlag, 2008.

SSM08b. Manfred Schmidt-Schauß and Elena Machkasova. A finite simulation method in a non-deterministic call-by-need calculus with letrec, constructors and case. Frank report 32, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, 2008.

SSNSS09. Jan Schwinghammer, David Sabel, Joachim Niehren, and Manfred Schmidt-Schauß. On correctness of buffer implementations in a concurrent lambda calculus with futures. Frank report 37, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, 2009.

SSS08. David Sabel and Manfred Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 18(03):501–553, 2008.

SSSS04. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. On the safety of Nöcker's strictness analysis. Frank report 19, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, 2004.

SSSS05. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. A complete proof of the safety of Nöcker's strictness analysis. Frank report 20, Inst. f. Informatik, J.W.Goethe-University, Frankfurt, 2005. submitted for publication.

SSSS08. Manfred Schmidt-Schauß, Marko Schütz, and David Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.