A Finite Simulation Method in a Non-Deterministic Call-by-Need Lambda-Calculus with letrec, constructors, and case

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

[1] Inst. Informatik, J.W. Goethe-University

[2] University of Minnesota, Morris

Rewriting Techniques and Applications (RTA) 2008

Motivation for finite simulation approach

Proving correctness of compiler transformations is especially challenging for higher-order languages with:

- concurrency
- memory manipulation
- user interactions

Our calculus models a Haskell-like language:
letrec, non-determinism, call-by-need, constructors.
We aim to prove *contextual equivalence* of expressions:
two expressions are the same if they behave the same in every context (maximal equivalence).

Motivation for finite simulation approach

Diagram methods for proving correctness of transformations (Plotkin75, Ariola&Klop96 and later modifications) require closing commutative diagrams - some do not hold, some difficult to prove.

Howe's (Howe89,96) *simulation* method has been applied to similar calculi (Mann04), but fails on cyclic dependencies.

We propose a *finite simulation method*. It provides a way to prove contextual equivalence of some expressions based on *answer-sets* (pre-evaluated expressions).

If computation of answer-sets succeeds at a finite depth, it shows contextual equivalence of expressions.

Overview: small-step operational semantics

We consider a *non-deterministic call-by-need* calculus.
`choice` represents non-determinism.
Semantics via normal order reduction:
rewrite (small-step) operational semantics.

$$\texttt{letrec}\ x = \texttt{True}, y = \texttt{False}, z = \lambda u.u\ \texttt{in}\ z(\texttt{choice}\ x\ y) \rightarrow$$
$$\texttt{letrec}\ Env\ \texttt{in}\ (\lambda u.u)\ (\texttt{choice}\ x\ y) \rightarrow$$
$$\texttt{letrec}\ Env\ \texttt{in}\ (\lambda u.u)\ y \rightarrow$$
$$\texttt{letrec}\ Env\ \texttt{in}\ (\lambda u.u)\ \texttt{False} \rightarrow$$
$$\texttt{letrec}\ Env\ \texttt{in}\ \texttt{letrec}\ u = \texttt{False}\ \texttt{in}\ u \rightarrow \ldots$$

where Env stands for $x = \texttt{True}, y = \texttt{False}, z = \lambda u.u$.

Overview: may-convergence

An expression t *may-converges* if there is a sequence of normal order steps $t \xrightarrow{*} t'$, where t' is a normal form: $t\downarrow$.
If there is no such sequence then $t\uparrow$ (t diverges).

Examples:

$(\texttt{choice}\ \Omega\ \texttt{True})\downarrow$

$(\texttt{choice}\ (\lambda xy.x)\ (\lambda xy.y))\Omega\Omega\uparrow$

Notation: $\Omega = (\lambda x.xx)(\lambda x.xx)$

Overview: pre-order, contextual equivalence

Let $C[\cdot]$ denote a one-hole context, $C[t]$ - context C filled with an expression t (free variables may be captured).
Contextual *pre-order*:

$$s \leq_c t \quad \text{iff} \quad \forall C[\cdot]: \quad C[s]\downarrow \Rightarrow C[t]\downarrow$$

Example: $(\texttt{choice}\ s\ \Omega) \leq_c s$.
Contextual *equivalence*:

$$s \sim_c t \quad \text{iff} \quad s \leq_c t \wedge t \leq_c s$$

Example: $(\texttt{choice True False}) \sim_c (\texttt{choice False True})$.

Overview: the idea of answer-set approach

For an expression t we construct an *answer-set* $ans(t)$: a set of all "values" v s.t. $v \leq_c t$, where v is built from abstractions and constructors (e.g. lists) and may contain Ω in place of some letrec-bound variables.
For example, $\texttt{letrec}\ x = (cons\ 1\ x)\ \texttt{in}\ x$ has answers

$$\{(cons\ 1\ \Omega), (cons\ 1\ (cons\ 1\ \Omega)), \ldots\}$$

where *cons* is a list constructor.
Main contribution: we can compare expressions based on $\leq_c$ relation of their sets of answers.

Calculus syntax

The syntax of the calculus is as follows (note: L_S in the paper):

$$\begin{array}{rcl} E & ::= & V \mid (c\ E_1 \dots E_m) \mid E_1\ E_2 \mid \lambda\, V.E \mid (\texttt{choice}\ E_1\ E_2) \\ & & \mid (\texttt{letrec}\ V_1 = E_1, \dots, V_n = E_n\ \texttt{in}\ E) \\ & & \mid (\texttt{case}\ E\ (Pat_1\ \rightarrow\ E_1) \dots (Pat_n\ \rightarrow\ E_n)) \\ Pat & ::= & (c\ V_1 \dots V_{\mathrm{ar}(c)}) \end{array}$$

where E are expressions, V are variables, c is a constructor (each of a fixed arity), *Pat* denotes a pattern.
`case` represents *pattern-matching* - taking apart a constructor expression; exactly one alternative matches.

Marking algorithm

Marking (*unwind*): find a needed subexpression. Notations:

- T - top-level expression
- V - visited subexpression
- S - current (needed) expression

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

Marking specifies a normal order reduction strategy.

Marking algorithm

Marking (*unwind*): find a needed subexpression. Notations:

- T - top-level expression
- V - visited subexpression
- S - current (needed) expression

$(s\ t)^{S \vee T} \to (s^S\ t)^V$

$(\texttt{letrec}\ Env\ \texttt{in}\ t)^T \to (\texttt{letrec}\ Env\ \texttt{in}\ t^S)^V$

$(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C[x^S]) \to (\texttt{letrec}\ x = s^S, Env\ \texttt{in}\ C[x^V])$

$(\texttt{letrec}\ x = s, y = C[x^S], Env\ \texttt{in}\ r) \to$
$\quad (\texttt{letrec}\ x = s^S, y = C[x^V], Env\ \texttt{in}\ r)$ if $C \neq [.]$

$(\texttt{case}\ s\ alts)^{S \vee T} \to (\texttt{case}\ s^S\ alts)^V$

$(\texttt{letrec}\ x = (\lambda y.y)(\lambda z.z)\ \texttt{in}\ x\ \texttt{True})^T \to$
$\quad (\texttt{letrec}\ x = (\lambda y.y)(\lambda z.z)\ \texttt{in}\ (x\ \texttt{True})^S)^V$

Marking algorithm

Marking (*unwind*): find a needed subexpression. Notations:

- T - top-level expression
- V - visited subexpression
- S - current (needed) expression

$(s\ t)^{S \vee T} \rightarrow (s^S\ t)^V$

$(\mathtt{letrec}\ Env\ \mathtt{in}\ t)^T \rightarrow (\mathtt{letrec}\ Env\ \mathtt{in}\ t^S)^V$

$(\mathtt{letrec}\ x = s, Env\ \mathtt{in}\ C[x^S]) \rightarrow (\mathtt{letrec}\ x = s^S, Env\ \mathtt{in}\ C[x^V])$

$(\mathtt{letrec}\ x = s, y = C[x^S], Env\ \mathtt{in}\ r) \rightarrow$
$\quad (\mathtt{letrec}\ x = s^S, y = C[x^V], Env\ \mathtt{in}\ r)$ if $C \neq [.]$

$(\mathtt{case}\ s\ alts)^{S \vee T} \rightarrow (\mathtt{case}\ s^S\ alts)^V$

$(\mathtt{letrec}\ x = (\lambda y.y)(\lambda z.z)\ \mathtt{in}\ (x\ \mathtt{True})^S)^V \rightarrow$
$\quad (\mathtt{letrec}\ x = (\lambda y.y)(\lambda z.z)\ \mathtt{in}\ (x^S\ \mathtt{True})^V)^V$

Marking algorithm

Marking (*unwind*): find a needed subexpression. Notations:

- T - top-level expression
- V - visited subexpression
- S - current (needed) expression

$(s\ t)^{S \vee T} \rightarrow (s^S\ t)^V$

$(\texttt{letrec}\ Env\ \texttt{in}\ t)^T \rightarrow (\texttt{letrec}\ Env\ \texttt{in}\ t^S)^V$

$(\texttt{letrec}\ x = s, Env\ \texttt{in}\ C[x^S]) \rightarrow (\texttt{letrec}\ x = s^S, Env\ \texttt{in}\ C[x^V])$

$(\texttt{letrec}\ x = s, y = C[x^S], Env\ \texttt{in}\ r) \rightarrow$
$\quad (\texttt{letrec}\ x = s^S, y = C[x^V], Env\ \texttt{in}\ r)$ if $C \neq [.]$

$(\texttt{case}\ s\ alts)^{S \vee T} \rightarrow (\texttt{case}\ s^S\ alts)^V$

$(\texttt{letrec}\ x = (\lambda y.y)(\lambda z.z)\ \texttt{in}\ (x^S\ \texttt{True})^V)^V \rightarrow$
$\quad (\texttt{letrec}\ x = ((\lambda y.y)(\lambda z.z))^S\ \texttt{in}\ (x^V\ \texttt{True})^V)^V \ldots$

Operational semantics rules

(lbeta) $((\lambda x.s)^S\ r) \to (\mathtt{letrec}\ x = r\ \mathtt{in}\ s)$

(cp-in) $(\mathtt{letrec}\ x = w^S, Env\ \ \mathtt{in}\ C[x^V])$
$\to (\mathtt{letrec}\ x = w, Env\ \ \mathtt{in}\ C[w])$
where w is $\lambda y.t$ or $(c\ x_1 \dots x_n)$

(cp-e) $(\mathtt{letrec}\ x = w^S, Env, y = C[x^V]\ \mathtt{in}\ r)$
$\to (\mathtt{letrec}\ x = w, Env, y = C[w]\ \mathtt{in}\ r)$
where w is $\lambda y.t$ or $(c\ x_1 \dots x_n)$

(llet-in) $(\mathtt{letrec}\ Env_1\ \mathtt{in}\ (\mathtt{letrec}\ Env_2\ \mathtt{in}\ r)^S)$
$\to (\mathtt{letrec}\ Env_1, Env_2\ \mathtt{in}\ r)$ (skip more let rules)

(case) $(\mathtt{case}\ (c\ t_1 \dots t_n)^S \dots ((c\ y_1 \dots y_n) \to s) \dots)$
$\to\ \ (\mathtt{letrec}\ y_1 = t_1, \dots, y_n = t_n\ \mathtt{in}\ s)$

(choice-l) $(\mathtt{choice}\ s\ t)^{S \vee T} \to s$

(choice-r) $(\mathtt{choice}\ s\ t)^{S \vee T} \to t$

Normal order reductions, WHNF

Normal order reduction $s \xrightarrow{no} t$:

- run the marking algorithm on s
- if success, apply the rules so that labels are matched

$$\texttt{letrec}\ x = ((\lambda y.y)^S(\lambda z.z))^V\ \texttt{in}\ (x^V\ \texttt{True})^V)^V \xrightarrow{no}$$
$$\texttt{letrec}\ x = (\texttt{letrec}\ y = \lambda z.z\ \texttt{in}\ y)\ \texttt{in}\ (x\ \texttt{True})$$

Weak Head Normal Form (WHNF) - normal form of normal order reduction. Let v (*value*) be $\lambda x.s$ or $(c\ x_1 \ldots x_n)$. WHNF is:

- a value v, or
- $\texttt{letrec}\ Env\ \texttt{in}\ v$

Evaluation: $s \xrightarrow{no,*} s'$ where s' is WHNF. Denote: $s\downarrow$.

Contextual preorder

Contextual preorder: $s \leq_c t$ iff $\forall C[\cdot]: \quad C[s]\downarrow \Rightarrow C[t]\downarrow$.
Ω is the least element: $\forall s: \; \Omega \leq_c s$.

Contextual equivalence: $s \sim_c t$ iff $s \leq_c t \wedge t \leq_c s$.
$(\texttt{choice True}\ \Omega) \sim_c \texttt{True}$.

Extra transformations

Transformations (for proofs and compiler optimizations)

- reduction rules applied in contexts other than those labeled
- additional rules

A sample additional rule:

$$\text{(gc)} \quad (\texttt{letrec}\ x = s, \mathit{Env}\ \texttt{in}\ t) \rightarrow (\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ t)$$
$$\text{if } x \text{ does not occur in } \mathit{Env} \text{ nor in } t$$

Proved: transformations preserve may-convergence in all contexts (i.e. are *correct*).

Standardization

Non-deterministic calculus: any sequence of correct reductions preserves may-convergence.

Theorem (Standardization)

If $t \xrightarrow{*} t'$ where t' is a WHNF and the sequence $\xrightarrow{*}$ consists of any sequence of reduction or transformation steps then $t{\downarrow}$.

($*$ denotes reflexive transitive closure)

$$\begin{tikzcd} t \arrow[r, "*"] \arrow[d, dashed, "{no,*}"'] & t'(WHNF) \\ t''(WHNF) & \end{tikzcd}$$

“Stop” reduction $\circledcirc$ and pseudovalues

We approximate evaluation results by *finite simulation*.
Components with a possibility of infinite recursion are replaced by a symbol $\circledcirc$ (read: *Stop*). Denotes potential divergence, i.e. synonym to Ω.

$$\texttt{letrec}\; x = \lambda y.x \;\texttt{in}\; x \;\texttt{True} \rightarrow \texttt{letrec}\; x = \lambda y.x \;\texttt{in}\; \circledcirc \;\texttt{True}$$

A *pseudo-value* is an expression built from $\circledcirc$, constructors and abstractions: ($cons\ \circledcirc\ \lambda x.x$).
An *answer* is a pseudo-value that is not the constant $\circledcirc$.

Approximation calculus

Extend the calculus with $\circledcirc$ and *approximation reduction* to compute *sets of answers*. s is a closed expression, for instance $\texttt{letrec}\ y = \lambda z.y\ \texttt{in}\ (cons\ y\ y)$.

Pre-evaluation of expressions (approximation reduction):

- Start with $s' = \texttt{letrec}\ x = s\ \texttt{in}\ x$:
  $\texttt{letrec}\ x = (\texttt{letrec}\ y = \lambda z.y\ \texttt{in}\ (cons\ y\ y))\ \texttt{in}\ x$
- Evaluate s' to WHNF:
  $\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ y\ y)$
- perform (non-deterministically) any number of *copy* steps:
  $\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ (\lambda z.y)\ y)$,
  $\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ (\lambda z.(\lambda z.y))\ y)$,
  etc.

Approximation calculus

Extend the calculus with $\circledcirc$ and *approximation reduction* to compute *sets of answers*. s is a closed expression, for instance $\texttt{letrec}\ y = (\lambda z.y)\ \texttt{in}\ (cons\ y\ y)$.

Pre-evaluation of expressions (approximation reduction):

- Start with $s' = \texttt{letrec}\ x = s\ \texttt{in}\ x$:
  $\texttt{letrec}\ x = (\texttt{letrec}\ y = \lambda z.y\ \texttt{in}\ (cons\ y\ y))\ \texttt{in}\ x$
- Evaluate s' to WHNF:
  $\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ y\ y)$
- perform (non-deterministically) any number of *copy* steps:
  $\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ (\lambda z.y)\ y)$,
  $\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ (\lambda z.(\lambda z.y)\ y)$,
  etc.

Approximation calculus

Extend the calculus with $\circledcirc$ and *approximation reduction* to compute *sets of answers*. s is a closed expression, for instance $\mathtt{letrec}\ y = (\lambda z.y)\ \mathtt{in}\ (cons\ y\ y)$.

Pre-evaluation of expressions (approximation reduction):

- Start with $s' = \mathtt{letrec}\ x = s\ \mathtt{in}\ x$:
  $\mathtt{letrec}\ x = (\mathtt{letrec}\ y = \lambda z.y\ \mathtt{in}\ (cons\ y\ y))\ \mathtt{in}\ x$
- Evaluate s' to WHNF:
  $\mathtt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \mathtt{in}\ (cons\ y\ y)$
- perform (non-deterministically) any number of *copy* steps:
  $\mathtt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \mathtt{in}\ (cons\ (\lambda z.y)\ y)$,
  $\mathtt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \mathtt{in}\ (cons\ (\lambda z.\ (\lambda z.y))\ y)$,
  etc.

Approximation calculus (cont.)

Some results of the previous step:

$\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ (\lambda z.y)\ y)$,
$\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ (\lambda z.\lambda z.y)\ y)$,
$\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ y\ (\lambda z.y))$

Approximation reduction (cont.):

- remove the top letrec-environment, replace remaining let-bound variables by $\circledcirc$:
  $(cons\ (\lambda z.\circledcirc)\ \circledcirc)$, $(cons\ (\lambda z.\lambda z.\circledcirc)\ \circledcirc)$, $(cons\ \circledcirc\ (\lambda z.\circledcirc))$, etc.

These are answers $ans(s)$ for
$s = \texttt{letrec}\ y = \lambda z.y\ \texttt{in}\ (cons\ y\ y)$.

Approximation calculus (cont.)

Some results of the previous step:

$\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ (\lambda z.y)\ y)$,
$\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ (\lambda z.\lambda z.y)\ y)$,
$\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ y\ (\lambda z.y))$

Approximation reduction (cont.):

- remove the top-letrec-environment, replace remaining let-bound variables by $\circledcirc$:
  $(cons\ (\lambda z.\circledcirc)\ \circledcirc)$, $(cons\ (\lambda z.\lambda z.\circledcirc)\ \circledcirc)$, $(cons\ \circledcirc\ (\lambda z.\circledcirc))$, etc.

These are answers $ans(s)$ for
$s = \texttt{letrec}\ y = \lambda z.y\ \texttt{in}\ (cons\ y\ y)$.

Approximation calculus (cont.)

Some results of the previous step:
$\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ (\lambda z.y)\ y)$,
$\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ (\lambda z.\lambda z.y)\ y)$,
$\texttt{letrec}\ x = (cons\ y\ y), y = \lambda z.y\ \texttt{in}\ (cons\ y\ (\lambda z.y))$
Approximation reduction (cont.):

- remove the top-letrec-environment, replace remaining let-bound variables by $\circledcirc$:
  $(cons\ (\lambda z.\circledcirc)\ \circledcirc)$, $(cons\ (\lambda z.\lambda z.\circledcirc)\ \circledcirc)$, $(cons\ \circledcirc\ (\lambda z.\circledcirc))$, etc.

These are answers $ans(s)$ for
$s = \texttt{letrec}\ y = \lambda z.y\ \texttt{in}\ (cons\ y\ y)$.

Answers as a finite model of expressions

$R[]$ *reduction contexts* denote a position in an expression where a normal order reduction takes place.

Theorem

Let R be a reduction context, s a closed expression, $R[s]\downarrow$. Then there is $v \in ans(s)$ such that $R[v]\downarrow$.

Main ideas of the proof:

$$R[\texttt{letrec}\ x = s\ \texttt{in}\ x] \xrightarrow{\quad\quad\quad n \quad\quad\quad} \mathit{WHNF}$$

$$R[\texttt{letrec}\ \mathit{Env}\ \text{in}\ x] \xrightarrow{cp,n+1} R[w] \xrightarrow{\circledcirc,*} R[v] \xrightarrow{\leq n} \mathit{WHNF}$$

- *Env* has labeled bindings derived from $x = s$
- in $R[\texttt{letrec}\ \mathit{Env}\ \texttt{in}\ x]$, copy all bindings of *Env* into the bound variables $n + 1$ times
- replace the remaining letrec-bound variables by $\circledcirc$.
- $R[v]\downarrow$ since all the positions affected by reductions in $R[]$ have values. $\circledcirc$ appears only in unreachable positions.

Answer sets and $\leq_c$

U - a set of expressions, t - an expression. t is a *lub (least upper bound)* of U iff $\forall u \in U : u \leq_c t$, and for any s s.t. $\forall u \in U : u \leq_c s$ it holds that $t \leq_c s$.
The expression t is called a *contextual lub (club)* of U, iff for $C[]$: $C[t]$ is a lub of $\{C[r] \mid r \in U\}$.
$W(t) = ans(t) \cup \{u \mid u \text{ is a } club \text{ of } A \subseteq ans(t)\}$ (some extra conditions given in the paper)

Theorem

Let s, t be closed expressions. If for all $v \in ans(s)$ there is some $w \in W(t)$ with $v \leq_c w$, then $s \leq_c t$.

Procedure for comparing answer sets

$s \leq_c t$ if $\forall v \in ans(s)\ \exists w \in ans(t)$ s.t. $v \leq_c w$.
For instance, $t = (\texttt{choice}\ \Omega\ s) \sim_c s$ since $ans(t) = ans(s)$.

How to compare complex pseudovalues?

- constructors: $(c\ s_1 \ldots s_n) \leq_c (c\ t_1 \ldots t_n)$ iff $s_i \leq_c t_i$ for all i.
- abstractions: $\lambda x.s \leq_c \lambda x.t$ iff for all closed pseudo-values v: $(\lambda x.s)\ v \leq_c (\lambda x.t)\ v$.

Effectiveness of the method

The method provides an effective (finite) procedure for deciding $s \leq_c t$ if the following takes place:

- bounded reductions to WHNF
- comparable answer sets (may be infinite)
- the ability to test equivalence of answers

Conclusions and future work

Conclusions:
We developed and proved correct a finite simulation method for a non-deterministic call-by-need calculus with cyclic bindings. The method provides a procedure for deciding $\leq_c$ and $\sim_c$ relations in a may-convergence framework which is effective if certain conditions hold.
Future work: to extend the method to must-convergence and to work towards general simulation.

Selected Bibliography

- Z. Ariola & W. Klop *Equational Term Graph Rewriting*, 1996.
- D. Howe *Equality in lazy computation systems*, 1989
- D. Howe *Proving congruence of bisimulation in functional programming languages*, 1996
- M. Mann *Congruence of Bisimulation in a Non-Deterministic Call-By-Need Lambda Calculus*, 2004
- M. Schmidt-Schauss & E. Machkasova *A Finite Simulation Method in a Non-Deterministic Call-by-Need Calculus with letrec, constructors and case (Technical Report)*, 2008