Denotational semantics for lazy initialization of letrec

black holes as exceptions rather than divergence

Keiko Nakata
Institute of Cybernetics at Tallinn University of Technology

Abstract

We present a denotational semantics for a simply typed call-by-need letrec calculus, which distinguishes direct cycles, such as *let rec* $x = x$ *in* x and *let rec* $x = y$ *and* $y = x+1$ *in* x, and looping recursion, such as *let rec* $f = \lambda x.f\ x$ *in* $f\ 0$. In this semantics the former denote an exception whereas the latter denotes divergence.

The distinction is motivated by "lazy evaluation" as implemented in OCaml via *lazy/force* and Racket (formerly PLT Scheme) via *delay/force*: when a delayed variable is dereferenced for the first time, it is first pre-initialized to an exception-raising thunk and is updated afterward by the value obtained by evaluating the expression bound to the variable. Any attempt to dereference the variable during the initialization raises an exception rather than diverges. This way, lazy evaluation provides a useful measure to initialize recursive bindings by exploring a successful initialization order of the bindings at runtime and by signaling an exception when there is no such order. It is also used for the initialization semantics of the object system in the F# programming language.

The denotational semantics is proved adequate with respect to a referential operational semantics.

1 Introduction

Lazy evaluation is a well-known technique in practice to initialize recursive bindings. OCaml [6] and Racket (formerly PLT Scheme) [4], provide language constructs, *lazy/force* and *delay/force* operators respectively, to support lazy evaluation atop call-by-value languages with arbitrary side-effects. Their implementations are quite simple: when a delayed variable is dereferenced for the first time, it is first pre-initialized to an exception-raising thunk and is updated afterward by the value obtained by evaluating the expression bound to the variable. Any attempt to dereference the variable during the initialization raises an exception rather than diverges. In other words, lazy evaluation as implemented in OCaml and Racket distinguishes direct cycles [1], which we call "black holes", such as *let rec* $x = x$ *in* x and *let rec* $x = y$ *and* $y = x+1$ *in* x, and looping recursion, such as *let rec* $f = \lambda x.f\ x$ *in* $f\ 0$. The former raise an exception, whereas the latter diverges.

Lazy evaluation provides a useful measure to initialize recursive bindings by exploring a successful initialization order of the bindings at runtime and by signaling an exception when there is no such order. In [12], Syme advocates the use of lazy evaluation for initializing mutually recursive bindings in ML-like languages to permit a wider range of recursive bindings [2]. Flexibility in handling recursive bindings is particularly important for these languages to interface with external abstract libraries such as GUI APIs. Syme's proposal can be implemented using OCaml's *lazy/force* operators and it underlies the initialization semantics of the object system in F# [13].

There is a gap between lazy evaluation, as outlined above, and conventional models for lazy, or call-by-need, computation as found in the literature. Traditionally call-by-need is understood as an economical implementation of call-by-name, which does not distinguish black holes and looping recursion but typically interprets both uniformly as "undefined". The gap becomes evident when a programming language supports exception handling, as both OCaml and Racket do — one can catch exceptions but cannot catch divergence. Indeed catching exceptions due to black holes is perfectly acceptable, or could

[1] Direct cycles are also known as provable divergence.

[2] In ML, the right-hand side of recursive bindings is restricted to be syntactic values.

$$\begin{array}{lllll} \textit{Expressions} & M,N & ::= & n \mid x \mid \lambda x.M \mid M\ N \mid \mathsf{let\ rec}\ x_1\ \mathsf{be}\ M_1, \ldots, x_n\ \mathsf{be}\ M_n\ \mathsf{in}\ M \mid \bullet \\ \textit{Results} & V & ::= & n \mid \lambda x.M \mid \bullet \\ \textit{Types} & \tau & ::= & \mathsf{nat} \mid \tau_1 \to \tau_2 \end{array}$$

Figure 1: Syntax of λ_{letrec}

$$n : \mathsf{nat} \qquad x : \mathsf{type}(x) \qquad \bullet : \tau$$

$$\frac{x : \tau_1 \quad M : \tau_2}{\lambda x.M : \tau_1 \to \tau_2} \qquad \frac{M : \tau_1 \to \tau_2 \quad N : \tau_1}{M\ N : \tau_2} \qquad \frac{x_1 : \tau_1 \quad \ldots \quad x_n : \tau_n \quad M_1 : \tau_1 \quad \ldots \quad M_n : \tau_n \quad N : \tau}{\mathsf{let\ rec}\ x_1\ \mathsf{be}\ M_1, \ldots, x_n\ \mathsf{be}\ M_n\ \mathsf{in}\ N : \tau}$$

Figure 2: Typing rules

be even desired, in practice; it is just like catching null-pointer exceptions due to object initialization failure in object-oriented languages.

In this paper we present a denotational semantics, which matches the lazy evaluation as implemented in OCaml and Racket and used in F#'s object initialization. In this semantics, direct cycles denote exceptions whereas looping recursion denotes divergence. The key observation is to think of lazy evaluation as a most successful initialization strategy of recursive bindings: the initialization succeeds if and only if there is a non-circular order in which the bindings can be initialized. The operational semantics searches such an order by on-demand computation. The denotational semantics searches such one intuitively by initializing recursive bindings in parallel and choosing the most successful result as the denotation.

The denotational semantics, proved adequate with respect to a referential operational semantics, is the main contribution of the paper.

2 Syntax and operational semantics

The syntax of our simply typed letrec calculus, λ_{letrec}, is given in figure 1. An expression is either a natural number $n \in N$, variable x, abstraction $\lambda x.M$, application $M\ N$, letrec $\mathsf{let\ rec}\ x_1\ \mathsf{be}\ M_1, \ldots, x_n\ \mathsf{be}\ M_n\ \mathsf{in}\ M$, or black hole $\bullet$, which represents an exception. Results are natural numbers, abstraction and black holes. A type is either a base type, nat, or a function type of shape $\tau_1 \to \tau_2$. To simplify the calculus, we assume each variable x is associated with a unique type, given, e.g., $\mathsf{type}(x)$. Typing rules are found in figure 2, which are all straightforward.

In figure 3, we present the natural semantics. The natural semantics is identical to that given in our previous work [8], which is very much inspired by Launchbury's [5] and Sestoft's [10]. Heaps, ranged over by metavariables Ψ and Φ, are finite mappings from variables to expressions. We write $x_1 \mapsto M_1, \ldots, x_n \mapsto M_n$ to denote a heap whose domain is $\{x_1, \ldots, x_n\}$, and which maps x_i's to M_i's. The notation $\Psi[x_1 \mapsto M_1, \ldots, x_n \mapsto M_n]$ denotes mapping extension. Precisely, $\Psi[x_1 \mapsto M_1, \ldots x_n \mapsto M_n](x_i) = M_i$ and $\Psi[x_1 \mapsto M_1, \ldots x_n \mapsto M_n](y) = \Psi(y)$ when $y \neq x_i$ for any i in $1, \ldots, n$. We write $\Psi[x \mapsto M]$ to denote a single extension of Ψ with M at x. In rule *Letrec*, M_i''s and N' denote expressions obtained from M_i's and N by substituting x_i''s for x_i's, respectively. We may abbreviate $\langle \Psi \rangle M$ where Ψ is an empty mapping, i.e., the domain of Ψ is empty, to $\langle \rangle M$.

The judgment $\langle \Psi \rangle M \Downarrow \langle \Phi \rangle V$ expresses that an expression M in an initial heap Ψ evaluates to a result V with the heap being Φ. In *Variable* rule, the heap Ψ is updated to map x to $\bullet$ while the expression bound to x is evaluated. For instance, $\langle \rangle\ \mathsf{let\ rec}\ x\ \mathsf{be}\ x\ \mathsf{in}\ x \Downarrow \langle x' \mapsto \bullet \rangle \bullet$ is deduced. This way, an attempt to dereference a variable which is under "initialization" results in a black hole. Error_β rule propagates black holes. Other rules are self-explanatory.

In figure 4 we present the derivation for the expression $\mathsf{let\ rec}\ x\ \mathsf{be}\ f\ x, f\ \mathsf{be}\ \lambda y.y\ \mathsf{in}\ x$. We deliberately

$$\textit{Result}\quad \langle\Psi\rangle V \Downarrow \langle\Psi\rangle V$$

$$\textit{Application}\quad \frac{\langle\Psi\rangle M_1 \Downarrow \langle\Phi\rangle \lambda x.N \quad \langle\Phi[x' \mapsto M_2]\rangle N[x'/x] \Downarrow \langle\Psi'\rangle V \quad x' \text{ fresh}}{\langle\Psi\rangle M_1\ M_2 \Downarrow \langle\Psi'\rangle V}$$

$$\textit{Variable}\quad \frac{\langle\Psi[x \mapsto \bullet]\rangle \Psi(x) \Downarrow \langle\Phi\rangle V}{\langle\Psi\rangle x \Downarrow \langle\Phi[x \mapsto V]\rangle V}$$

$$\textit{Letrec}\quad \frac{\langle\Psi[x'_1 \mapsto M'_1, \ldots, x'_n \mapsto M'_n]\rangle N' \Downarrow \langle\Phi\rangle V \quad x'_1, \ldots, x'_n \text{ fresh}}{\langle\Psi\rangle \mathsf{let\ rec}\ x_1\,\mathsf{be}\,M_1, \ldots, x_n\,\mathsf{be}\,M_n\ \mathsf{in}\ N \Downarrow \langle\Phi\rangle V}$$

$$\textit{Error}_\beta\quad \frac{\langle\Psi\rangle M_1 \Downarrow \langle\Phi\rangle \bullet}{\langle\Psi\rangle M_1\ M_2 \Downarrow \langle\Phi\rangle \bullet}$$

Figure 3: Natural semantics

$$\dfrac{\dfrac{\dfrac{\langle x' \mapsto \bullet, f' \mapsto \bullet\rangle \lambda y.y \Downarrow \langle x' \mapsto \bullet, f' \mapsto \bullet\rangle \lambda y.y}{\langle x' \mapsto \bullet, f' \mapsto \lambda y.y\rangle f' \Downarrow \langle x' \mapsto \bullet, f' \mapsto \lambda y.y\rangle \lambda y.y} \quad \dfrac{\dfrac{\langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle \bullet \Downarrow \langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle \bullet}{\langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle x' \Downarrow \langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle \bullet}}{\langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto x'\rangle y' \Downarrow \langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle \bullet}}{\dfrac{\langle x' \mapsto \bullet, f' \mapsto \lambda y.y\rangle f'\ x' \Downarrow \langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle \bullet}{\dfrac{\langle x' \mapsto f'\ x', f' \mapsto \lambda y.y\rangle x' \Downarrow \langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle \bullet}{\langle\rangle\ \mathsf{let\ rec}\ x\,\mathsf{be}\,f\ x, f\,\mathsf{be}\,\lambda y.y\ \mathsf{in}\ x \Downarrow \langle x' \mapsto \bullet, f' \mapsto \lambda y.y, y' \mapsto \bullet\rangle \bullet}}$$

Figure 4: The derivation for $\mathsf{let\ rec}\ x\,\mathsf{be}\,f\ x, f\,\mathsf{be}\,\lambda y.y\ \mathsf{in}\ x$

chose a black hole producing expression.

3 Denotational semantics

We proceed to the denotational semantics. An expression M of type τ denotes an element of $(V_\tau + \mathsf{Err}_\tau)_\perp$, where $(\cdot)_\perp$ is lifting and Err_τ is a singleton, whose only element is $\bullet_\tau$. V_τ denotes proper values of type τ and is defined by induction on τ:

$$V_{\mathsf{nat}} = N \qquad V_{\tau_0 \to \tau_1} = [(V_{\tau_0} + \mathsf{Err}_{\tau_0})_\perp \to (V_{\tau_1} + \mathsf{Err}_{\tau_1})_\perp]$$

We omit injections for both the lifting and the sum. A metavariable φ ranges over proper function values, i.e., elements of $V_{\tau_0 \to \tau_1}$ for some τ_0 and τ_1. For $d \in (V_{\tau_0 \to \tau_1} + \mathsf{Err}_{\tau_0 \to \tau_1})_\perp$ and $d' \in (V_{\tau_0} + \mathsf{Err}_{\tau_0})_\perp$, application of d to d' is defined by

$$d(d') = \begin{cases} \perp_{\tau_1} & \text{when } d = \perp_{\tau_0 \to \tau_1} \\ \bullet_{\tau_1} & \text{when } d = \bullet_{\tau_0 \to \tau_1} \\ \varphi(d') & \text{when } d = \varphi \end{cases}$$

Moreover we write $(d)^*$ to denote the strict version of d on both $\perp_{\tau_0}$ and $\bullet_{\tau_0}$, i.e.,

$$(d)^*(d') = \begin{cases} \perp_{\tau_1} & \text{when } d = \varphi \text{ and } d' = \perp_{\tau_0} \\ \bullet_{\tau_1} & \text{when } d = \varphi \text{ and } d' = \bullet_{\tau_0} \\ d(d') & \text{otherwise} \end{cases}$$

$$
\begin{array}{rcl}
[\![n:\tau]\!]_\rho & = & n \\
[\![x:\tau]\!]_\rho & = & \rho(x) \\
[\![\bullet:\tau]\!]_\rho & = & \bullet_\tau \\
[\![\lambda x.M:\tau_0\to\tau_1]\!]_\rho & = & \lambda v.[\![M:\tau_1]\!]_{\rho[x\mapsto v]} \\
[\![M^{\tau_0\to\tau_1}\ N^{\tau_0}:\tau_1]\!]_\rho & = & ([\![M:\tau_0\to\tau_1]\!]_\rho)([\![N:\tau_0]\!]_\rho) \\
[\![\mathsf{let\ rec}\ x_1\,\mathsf{be}\,M_1^{\tau_1},\ldots,x_n\,\mathsf{be}\,M_n^{\tau_n}\ \mathsf{in}\ N:\tau]\!]_\rho & = & [\![N:\tau]\!]_{\{\{x_1\mapsto M_1^{\tau_1},\ldots,x_n\mapsto M_n^{\tau_n}\}\}_\rho^{(n)}} \\
\{\{x_1\mapsto M_1^{\tau_1},\ldots,x_n\mapsto M_n^{\tau_n}\}\}_\rho^{(m+1)} & = & \mu\rho'.\rho[x_1\mapsto [\![M_1:\tau_1]\!]_{\rho_m}\cdot[\![M_1:\tau_1]\!]_{\rho'},\ldots,x_n\mapsto [\![M_n:\tau_n]\!]_{\rho_m}\cdot[\![M_n:\tau_n]\!]_{\rho'}] \\
 & & \textit{where } \rho_m=\{\{x_1\mapsto M_1^{\tau_1},\ldots,x_n\mapsto M_n^{\tau_n}\}\}_\rho^{(m)} \\
\{\{x_1\mapsto M_1^{\tau_1},\ldots,x_n\mapsto M_n^{\tau_n}\}\}_\rho^{(0)} & = & \rho[x_1\mapsto\bullet_{\tau_1},\ldots,x_n\mapsto\bullet_{\tau_n}]
\end{array}
$$

Figure 5: Denotational semantics of λ_{letrec}

An environment, ρ, is a function from variables to denotations, which respects types, i.e., $\rho(x)\in(V_\tau+\mathsf{Err}_\tau)_\perp$ where $x:\tau$. The least environment, $\rho_\perp$, maps all variables to bottom elements.

The semantic function $[\![M:\tau]\!]_\rho$ assigns a denotation to a typing derivation $M:\tau$ under an environment ρ and is defined in figure 5 by induction on the derivation [3]. μ stands for the least fixed point operator. $\rho[x_1\mapsto d_1,\ldots,x_n\mapsto d_n]$ stands for extension of ρ with d_i's at x_i's. For $d,d'\in(V_\tau+\mathsf{Err}_\tau)_\perp$, the notation $d\cdot d'$ abbreviates $((\lambda y.\lambda x.x)^*(d))(d')$. The semantic function, being defined using only continuous operations, is continuous and mostly standard (c.f., [14]) except for letrec. The denotation of $\mathsf{let\ rec}\ x_1\,\mathsf{be}\,M_1^{\tau_1},\ldots,x_n\,\mathsf{be}\,M_n^{\tau_n}\ \mathsf{in}\ N:\tau$ is defined with the help of a semantic function for (typed) heaps. The function $\{\{x_1\mapsto M_1^{\tau_1},\ldots,x_n\mapsto M_n^{\tau_n}\}\}_\rho^{(m)}$ takes three parameters, a heap $x_1\mapsto M_1^{\tau_1},\ldots,x_n\mapsto M_n^{\tau_n}$, an environment ρ and a natural number m, and returns an environment. It is defined by induction on m, with semantic functions for $M_i:\tau_i$'s given by the outer induction.

We compute the denotation of a heap $\Psi=x_1\mapsto M_1^{\tau_1},\ldots,x_n\mapsto M_n^{\tau_n}$ under an environment ρ as follows. Let $\rho_m=\{\{\Psi\}\}_\rho^{(m)}$. The variables x_i's are first pre-initialized to black holes, that is, $\rho_0=\rho[x_1\mapsto\bullet_{\tau_1},\ldots,x_n\mapsto\bullet_{\tau_n}]$. Next we compute the denotation $[\![M_i:\tau_i]\!]_{\rho_0}$ of $M_i:\tau_i$ for each i under the initial environment ρ_0, so that we take the fixed-point semantics for the recursive bindings whose initialization was successful. That is, $\rho_1=\mu\rho'.\rho[x_1\mapsto d_1,\ldots,x_n\mapsto d_n]$ where

$$d_i=\begin{cases}\bullet_{\tau_i} & \textit{when } [\![M_i:\tau_i]\!]_{\rho_0}=\bullet_{\tau_i}\\ [\![M_i:\tau_i]\!]_{\rho'} & \textit{otherwise}\end{cases}$$

Indeed it follows from lemmata 3.1 and 3.2 below that this is equivalent to defining $\rho_1=\mu\rho'.\rho[x_1\mapsto[\![M_1:\tau_1]\!]_{\rho_0}\cdot[\![M_1:\tau_1]\!]_{\rho'},\ldots,x_n\mapsto[\![M_n:\tau_n]\!]_{\rho_0}\cdot[\![M_n:\tau_n]\!]_{\rho'}]$. Generally, ρ_{m+1} is given by taking the fixed-point semantics for the recursive bindings whose initialization is successful under the environment ρ_m; i.e., $\rho_{m+1}=\mu\rho'.\rho[x_1\mapsto d_1,\ldots,x_n\mapsto d_n]$ where

$$d_i=\begin{cases}\bullet_{\tau_i} & \textit{when } [\![M_i:\tau_i]\!]_{\rho_m}=\bullet_{\tau_i}\\ [\![M_i:\tau_i]\!]_{\rho'} & \textit{otherwise}\end{cases}$$

This process is iterated for n times, where n is the length of the heap Ψ. The number of the iteration is justified, as it converges by then: $\{\{\Psi\}\}_\rho^{(n)}=\{\{\Psi\}\}_\rho^{(n+m)}$ for any m (lemma 3.3 below). Let us define $\{\{x_1\mapsto M_1^{\tau_1},\ldots,x_n\mapsto M_n^{\tau_n}\}\}_\rho=\{\{x_1\mapsto M_1^{\tau_1},\ldots,x_n\mapsto M_n^{\tau_n}\}\}_\rho^{(n)}$. For instance, we have $\{\{x\mapsto x^{\mathsf{nat}}\}\}_{\rho_\perp}(x)=\bullet_{\mathsf{nat}}$, $\{\{f\mapsto(\lambda x.f\ x)^{\mathsf{nat}\to\mathsf{nat}}\}\}_{\rho_\perp}(f)=\lambda x.\perp_{\mathsf{nat}}$, and $\{\{x\mapsto(f\ x)^{\mathsf{nat}},f\mapsto(\lambda y.y)^{\mathsf{nat}\to\mathsf{nat}}\}\}_{\rho_\perp}=\rho_\perp[x\mapsto\bullet_{\mathsf{nat}},f\mapsto\lambda y.y]$ therefore $[\![\mathsf{let\ rec}\,x\,\mathsf{be}\,(f\ x)^{\mathsf{nat}},f\,\mathsf{be}\,(\lambda y.y)^{\mathsf{nat}\to\mathsf{nat}}\ \mathsf{in}\ x:\mathsf{nat}]\!]_{\rho_\perp}=\bullet_{\mathsf{nat}}$.

[3] We use the lambda notation to express (mathematical) functions on domains.

3.1 Adequacy

The denotational semantics is adequate with respect to the natural semantics. An important fact, to be stated in lemma 3.4, is that, as we iteratively compute denotations of (typed) heaps Ψ under an environment ρ, we reach a fixed point: $\{\{\Psi\}\}_\rho(x) = [\![\Psi(x)]\!]_{\{\{\Psi\}\}_\rho}$.

We define a relation $\ll_\tau \subseteq (V_\tau + \mathsf{Err}_\tau)_\perp \times (V_\tau + \mathsf{Err}_\tau)_\perp$ by induction on τ:

$$\begin{array}{l}
\bullet_\tau \ll_\tau d \quad \text{for any } d \in (V_\tau + \mathsf{Err}_\tau)_\perp \\
\perp_\tau \ll_\tau \perp_\tau \\
n \ll_{\mathsf{nat}} n \\
\varphi \ll_{\tau_1 \to \tau_2} \varphi' \text{ iff } d \ll_{\tau_1} d' \text{ implies } \varphi(d) \ll_{\tau_2} \varphi'(d')
\end{array}$$

Then a relation $\ll$ on environments is defined such that $\rho \ll \rho'$ iff for any x, $\rho(x) \ll_\tau \rho'(x)$ where $x : \tau$. It captures a relationship between (intermediate) denotations of a heap at different iterations (see lemma 3.2 below).

Lemma 3.1. *For any $\rho, \rho', M : \tau$, if $\rho \ll \rho'$, then $[\![M : \tau]\!]_\rho \ll_\tau [\![M : \tau]\!]_{\rho'}$*

Lemma 3.2. *For any $x_1 : \tau_1, \ldots, x_n : \tau_n, M_1 : \tau_1, \ldots, M_n : \tau_n, m$ and ρ, $\{\{x_1 \mapsto M_1^{\tau_1}, \ldots, x_n \mapsto M_n^{\tau_n}\}\}_\rho^{(m)} \ll \{\{x_1 \mapsto M_1^{\tau_1}, \ldots, x_n \mapsto M_n^{\tau_n}\}\}_\rho^{(m+1)}$.*

Lemma 3.3. *For any $x_1 : \tau_1, \ldots, x_n : \tau_n, M_1 : \tau_1, \ldots, M_n : \tau_n, m$ and ρ, $\{\{x_1 \mapsto M_1^{\tau_1}, \ldots, x_n \mapsto M_n^{\tau_n}\}\}_\rho^{(n)} = \{\{x_1 \mapsto M_1^{\tau_1}, \ldots, x_n \mapsto M_n^{\tau_n}\}\}_\rho^{(n+m)}$.*

Lemma 3.4. *For any $x_1 : \tau_1, \ldots, x_n : \tau_n, M_1 : \tau_1, \ldots, M_n : \tau_n, \rho$ and $i \in 1..n$, $\{\{x_1 \mapsto M_1^{\tau_1}, \ldots, x_n \mapsto M_n^{\tau_n}\}\}_\rho^{(n)}(x_i) = [\![M_i : \tau_i]\!]_{\{\{x_1 \mapsto M_1^{\tau_1}, \ldots, x_n \mapsto M_n^{\tau_n}\}\}_\rho^{(n)}}$.*

The natural semantics is correct with respect to the denotational semantics in that evaluations preserve the denotations of expressions.

Proposition 3.1. *For any typed expression $M : \tau$, if $\langle\rangle M \Downarrow \langle\Psi\rangle V$, then $V : \tau$ and $[\![M : \tau]\!]_{\rho_\perp} = [\![V : \tau]\!]_{\{\{\Psi\}\}_{\rho_\perp}}$.*

Moreover an expression evaluates to a result if and only if its denotation is non-bottom.

Proposition 3.2. *For any typed expression $M : \tau$, $[\![M : \tau]\!]_{\rho_\perp} \neq \perp_\tau$ iff there are Φ and V such that $\langle\rangle M \Downarrow \langle\Phi\rangle V$.*

4 Related work

The natural semantics used in the paper is very much inspired by those of Launchbury [5] and Sestoft [10]. Ariola and Felleisen gave a reduction semantics for λ_{letrec} [2], which is proved equivalent to our natural semantics [8]. Our denotational semantics is also influenced by Launchbury's, except that his semantics assigns a bottom element to both black holes and looping recursion; the distinction of the two is at the heart of our work.

Ariola and Klop [3] studied equational theories of cyclic lambda calculi by means of cyclic lambda graphs and observed that having non-restricted substitution leads to non-confluence. Ariola and Blom [1] and Schmidt-Schauß et al. [9] use infinite lambda terms to reason about call-by-need letrec; it is not obvious if their techniques can be adapted so that black holes are distinguished from divergence.

Viewing black holes as exceptions is not new. For instance, Moggi and Sabry's monadic operational semantics for value recursion signals a monadic error when a black hole is encountered [7]. The idea seems to be ascribed to the backpatching semantics of Scheme [11].

5 Conclusion

We have presented a denotational semantics for lazy initialization of letrec. The semantics interprets direct cycles, called black holes, as exceptions, which fits lazy evaluation as implemented in OCaml and Racket and which underlies the initialization semantics of F#'s object system. We think signaling an exception for these "non-sense recursion" is natural and useful in practice; we believe it is also natural in theory.

Acknowledgments I thank Matthias Felleisen and Olivier Danvy for valuable exchanges, Eijiro Sumii, Makoto Tatsuta, Masahito Hasegawa, Yukiyoshi Kameyama and Yasuhiko Minamide for discussions.

This research was supported by the Estonian Centre of Excellence in Computer Science, EXCS, funded by the European Regional Development Fund, and the Estonian Science Foundation grant no. 6940.

References

[1] Z. Ariola and S. Blom. Cyclic lambda calculi. In M. Abadi and T. Ito, editors, *Proc. of 3rd Int. Symp. on Theoretical Aspects of Computer Software (TACS 1997)*, volume 1281 of *Lect. Notes in Comput. Sci.*, pages 77–106. Springer, 1997.

[2] Z. Ariola and M. Felleisen. The call-by-need lambda calculus. *J. Funct. Program.*, 7(3):265–301, 1997.

[3] Z. Ariola and J. Klop. Cyclic lambda graph rewriting. In *Proc. of 9th Symp. on Logic in Computer Science (LICS 1994)*, pages 416–425. IEEE Computer Society, 1994.

[4] Matthew Flatt and PLT. Reference: PLT Scheme. Technical Report PLT-TR2010-reference-v4.2.5, PLT Scheme Inc., April 2010. `http://plt-scheme.org/techreports/` (PLT Scheme is now Racket.).

[5] J. Launchbury. A natural semantics for lazy evaluation. In *Conf. Record of 20th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 1993)*, pages 144–154. ACM Press, 1993.

[6] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, release 3.11, November 2008. `http://caml.inria.fr/`.

[7] E. Moggi and A. Sabry. An abstract monadic semantics for value recursion. *Informatique Théorique et Applications*, 38(4):375–400, 2004.

[8] K. Nakata and M. Hasegawa. Small-step and big-step semantics for call-by-need. *J. Funct. Program.*, 19(6):699–722, 2009.

[9] M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec. In C. Lynch, editor, *Proc. of 21st Int. Conf. on Rewriting Techniques and Applications (RTA 2010)*, volume 6 of *LIPIcs*, pages 295–310. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[10] P. Sestoft. Deriving a lazy abstract machine. *J. Funct. Program.*, 7(3):231–264, 1997.

[11] M. Sperber, K. Dybvig, M. Flatt, A. Straaten, R. Findler, and J. Matthews. Revised^6 report on the algorithmic language scheme. *J. Funct. Program.*, 19(S1):1–301, 2009.

[12] D. Syme. Initializing mutually referential abstract objects: The value recursion challenge. In N. Benton and X. Leory, editors, *Proc. of the ACM-SIGPLAN Workshop on ML (ML 2005)*, volume 148 of *Electr. Notes Theor. Comput. Sci.*, pages 3–25. Elsevier, 2006.

[13] D. Syme and the MSR F# team. The F# programming language, version 2.0.0.0, April 2010. `http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/`.

[14] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, 1993.