A locally nameless representation for a natural semantics for lazy evaluation

Lidia Sánchez-Gil[1], Mercedes Hidalgo-Herrero[2], and Yolanda Ortega-Mallén[1]

[1] Dpto. Sistemas Informáticos y Computación, Facultad de CC. Matemáticas, Universidad Complutense de Madrid, Spain
[2] Dpto. Didáctica de las Matemáticas, Facultad de Educación, Universidad Complutense de Madrid, Spain

Abstract. We propose a locally nameless representation for Launchbury's natural semantics for lazy evaluation. Names are reserved for free variables, while bound variable names are replaced by indices. This avoids the use of α-conversion and Barendregt's variable convention, and facilitates proof formalization. Our definition includes the management of multi-binders to represent simultaneous recursive local declarations. We use cofinite quantification to express the semantic rules that require the introduction of fresh names, but we show that existential rules are admissible too. Moreover, we prove that the choice of names during the evaluation of a term is irrelevant as long as they are fresh enough.

1 Motivation

Call-by-need evaluation, which avoids repeated computations, is the semantic foundation for lazy functional programming languages like Haskell or Clean. Launchbury defines in [7] a natural semantics for lazy evaluation where the set of *bindings*, i.e., (variable, expression) pairs, is explicitly managed to make possible their sharing. In order to prove that this lazy semantics is *correct* and *computationally adequate* with respect to a standard denotational semantics, Launchbury introduces some variations in his natural semantics. On the one hand, functional application is modeled denotationally by extending the environment with a variable bound to a value. This new variable represents the formal parameter of the function, while the value corresponds to the actual argument. For a closer approach of this mechanism, applications are carried out in the alternative semantics by introducing indirections instead of by performing the β-reduction through substitution. On the other hand, the update of bindings with their computed values is an operational notion without counterpart in the standard denotational semantics, so that the alternative natural semantics does no longer update bindings and becomes a *call-by-name* semantics.

Unfortunately, the proof of the equivalence between the lazy natural semantics and its alternative version with indirections and nonupdate is detailed nowhere, and a simple induction turns out to be insufficient. Intuitively, both reduction systems should produce the same results. However, this cannot be directly established since final values may contain free variables which are dependent on the context of evaluation, which is represented by the heap of bindings.

The changes introduced by the alternative semantics do deeply affect the heaps. Although indirections and "duplicated" bindings (a consequence of no updating) do not add relevant information to the context, it is awkward to prove this fact.

In the usual representation of the lambda-calculus, i.e., with variable names for free and bound variables, terms are identifed up to α-conversion. Dealing with α-equated terms usually implies the use of Barendregt's variable convention [3] to avoid the renaming of bound variables. However, the use of the variable convention in rule inductions is sometimes dubious and may lead to *faulty* results (as it is shown by Urban et al. in [15]). Looking for a system of binding more amenable to formalization, we have chosen a *locally nameless* representation (as presented by Charguéraud in [5]). This is a mixed notation where bound variable names are replaced by de Bruijn indices [6], while free variables preserve their names. Hence, α-conversion is no longer needed and variable substitution is easily defined because there is no danger of name capture. Moreover, this representation is suitable for working with proof assistants like Coq [4] or Isabelle [9].

The present work is the first step to prove formally the equivalence between Launchbury's semantics and its alternative version. We start by defining a locally nameless representation of the λ-calculus extended with recursive local declarations. Then we express Launchbury's rules in the new style and present several properties of the reduction system that are useful for the equivalence proof.

Our concern for reproducing and formalizing the proof of this equivalence is not arbitrary. Launchbury's semantics has been cited frequently and has inspired many further works as well as several extensions [2, 8, 13, 17], where the corresponding adequacy proofs have been obtained by just adapting Launchbury's proof scheme. We have extended ourselves the λ-calculus with a new expression that introduces parallelism when performing functional applications [11]. This *parallel application* creates new processes to distribute the computation; these processes exchange values through communication channels. The corresponding adequacy property relies on the adequacy of Launchbury's natural semantics.

The paper is structured as follows: In Section 2 we present the locally nameless representation of the lambda calculus extended with recursive local declarations. In Section 3 we describe a locally nameless translation of Launchbury's natural semantics for lazy evaluation [7], together with the corresponding regularity, introduction and renaming lemmas. The proofs (by hand) of these lemmas and other auxiliary results are detailed in [12]. In Section 4 we comment on some related work. The last two sections are devoted to conclusions and future work.

2 The locally nameless representation

The language described by Launchbury in [7] is a normalized lambda calculus extended with recursive local declarations. We reproduce the restricted syntax in Figure 1. Normalization is achieved in two steps. First an α-conversion is carried out so that all bound variables have distinct names. In a second phase, arguments for applications are enforced to be variables. These static transformations simplify the definition of the reduction rules.

$$
\begin{array}{l}
x \in \mathit{Var} \\
e \in \mathit{Exp} ::= x \mid \lambda x.e \mid (e\ x) \mid \\
\qquad\qquad \texttt{let}\ \{x_i = e_i\}_{i=1}^{n}\ \texttt{in}\ e
\end{array}
$$

Fig. 1. Restricted *named* syntax

$$
\begin{array}{lll}
x \in \mathit{Id} & & i, j \in \mathbb{N} \\
v \in \mathit{Var} & ::= & \texttt{bvar}\ i\ j \mid \texttt{fvar}\ x \\
t \in \mathit{LNExp} & ::= & v \mid \texttt{abs}\ t \mid \texttt{app}\ t\ v \mid \\
 & & \texttt{let}\ \{t_i\}_{i=1}^{n}\ \texttt{in}\ t
\end{array}
$$

Fig. 2. Locally nameless syntax

We give the corresponding locally nameless representation by following the methodology summarized in [5]:

1. Define the syntax of the extended λ-calculus in the locally nameless style.
2. Define the variable opening and variable closing operations.
3. Define the free variables and substitution functions, as well as the local closure predicate.
4. State and prove the properties of the operations on terms that are needed in the development to be carried out.

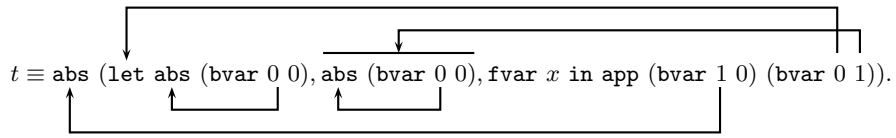
2.1 Locally nameless syntax

The locally nameless (restricted) syntax is shown in Figure 2. *Var* stands now for the set of *variables*, where *bound variables* and *free variables* are distinguished. The calculus includes two binding constructions: λ-abstraction and `let`-declaration. Being the latter a *multi-binder*, we follow Charguéraud [5] and represent bound variables with two natural numbers: The first number is a de Bruijn index that counts how many binders (abstraction or `let`) one needs to cross to the left to reach the corresponding binder for the variable, while the second refers to the position of the variable inside that binder. Abstractions are seen as multi-binders that bind one variable; thus, the second number should be zero. In the following, a list like $\{t_i\}_{i=1}^{n}$ is represented as $\overline{t}$, with length $|\overline{t}| = n$.

Example 1. Let $e \in \mathit{Exp}$ an expression in the named representation:

$$e \equiv \lambda z.\texttt{let}\ x_1 = \lambda y_1.y_1, x_2 = \lambda y_2.y_2, x_3 = x\ \texttt{in}\ (z\ x_2).$$

The corresponding locally nameless term $t \in \mathit{LNExp}$ is:



Notice that x_1 and x_2 denote α-equivalent expressions in e. This is more clearly seen in t, where both expressions are represented with syntactically equal terms. □

As bound variables are nameless, the first phase of Launchbury's normalization is unneeded. However, application arguments are still restricted to variables.

$$
\begin{array}{lll}
\{k \to \overline{x}\}(\texttt{bvar}\ i\ j) & = \begin{cases} \texttt{fvar}\ (\texttt{List.nth}\ j\ \overline{x}) & \text{if } i = k \wedge j < |\overline{x}| \\ \texttt{bvar}\ i\ j & \text{otherwise} \end{cases} \\
\{k \to \overline{x}\}(\texttt{fvar}\ x) & = \texttt{fvar}\ x \\
\{k \to \overline{x}\}(\texttt{abs}\ t) & = \texttt{abs}\ (\{k+1 \to \overline{x}\}\ t) \\
\{k \to \overline{x}\}(\texttt{app}\ t\ v) & = \texttt{app}\ (\{k \to \overline{x}\}\ t)\ (\{k \to \overline{x}\}\ v) \\
\{k \to \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{let}\ (\{k+1 \to \overline{x}\}\ \overline{t})\ \texttt{in}\ (\{k+1 \to \overline{x}\}\ t)
\end{array}
$$

where $\{k \to \overline{x}\}\ \overline{t} = \texttt{List.map}\ (\{k \to \overline{x}\}\,\cdot)\ \overline{t}$.

Fig. 3. Variable opening

2.2 Variable opening and variable closing

Variable opening and *closing* are the main operations to manipulate locally nameless terms. We extend to `let` the definitions given by Charguéraud in [5].[3]

To explore the body of a binder (abstraction or `let`), one needs to replace the corresponding bound variables by fresh names. In the case of an abstraction `abs` t the *variable opening operation* replaces in t with a (fresh) name every bound variable which refers to the outermost abstraction. Analogously, to open `let` $\overline{t}$ `in` t we provide a list of $|\overline{t}|$ distinct fresh names to replace the bound variables that occur in $\overline{t}$ and in the body t which refer to this particular declaration.

Variable opening is defined by means of a more general function $\{k \to \overline{x}\}t$ (Figure 3), where the number k represents the nesting level of the binder to be opened, and $\overline{x}$ is a list of pairwise-distinct identifiers in Id. Since the level of the outermost binder is 0, variable opening is defined as: $t^{\overline{x}} = \{0 \to \overline{x}\}t$. We extend this operation to lists of terms: $\overline{t}^{\overline{x}} = \texttt{List.map}\ (\cdot^{\overline{x}})\ \overline{t}$.

The last definition and those in Figure 3 include some operations on lists. We use an ML-like notation. For instance, `List.nth` $j\ \overline{x}$ represents the $(j+1)^{th}$ element of $\overline{x}$,[4] and `List.map` $f\ \overline{t}$ indicates that the function f is applied to every term in the list $\overline{t}$. In the rest of definitions we will use similar list operations.

Example 2. Let $t \equiv \texttt{abs}\ (\texttt{let bvar 0 1, bvar 1 0 in app (abs bvar 2 0) (bvar 0 1)})$. Hence, the body of the abstraction is:

$$u \equiv \texttt{let bvar 0 1,}\ \boxed{\texttt{bvar 1 0}}\ \texttt{in app (abs}\ \boxed{\texttt{bvar 2 0}}\texttt{) (bvar 0 1)}.$$

But then in u the bound variables referring to the outermost abstraction (shown squared) point to nowhere. Therefore, we consider $u^{[x]}$ instead of u, where

$$
\begin{array}{l}
u^{[x]} = \{0 \to x\}(\texttt{let bvar 0 1, bvar 1 0 in app (abs bvar 2 0) (bvar 0 1)}) \\
\quad = \texttt{let}\{1 \to x\}(\texttt{bvar 0 1, bvar 1 0})\ \texttt{in}\{1 \to x\}(\texttt{app (abs bvar 2 0)(bvar 0 1)}) \\
\quad = \texttt{let bvar 0 1, fvar}\ x\ \texttt{in app (abs}\ \{2 \to x\}\texttt{(bvar 2 0)) (bvar 0 1)} \\
\quad = \texttt{let bvar 0 1, fvar}\ x\ \texttt{in app (abs fvar}\ x\texttt{) (bvar 0 1)}
\end{array}
$$

□

[3] Multiple binders are defined in [5]. Two constructions are given: One for non-recursive local declarations, and another for mutually recursive expressions. Yet both extensions are not completely developed.

[4] Elements in lists are numbered starting with 0 to match bound variables indices.

$$
\begin{array}{ll}
\{k \leftarrow \overline{x}\}(\texttt{bvar}\ i\ j) & = \texttt{bvar}\ i\ j \\
\{k \leftarrow \overline{x}\}(\texttt{fvar}\ x) & = \begin{cases} \texttt{bvar}\ k\ j & \text{if } \exists j : 0 \leq j < |\overline{x}|.x = \texttt{List.nth}\ j\ \overline{x} \\ \texttt{fvar}\ x & \text{otherwise} \end{cases} \\
\{k \leftarrow \overline{x}\}(\texttt{abs}\ t) & = \texttt{abs}\ (\{k+1 \leftarrow \overline{x}\}\ t) \\
\{k \leftarrow \overline{x}\}(\texttt{app}\ t\ v) & = \texttt{app}\ (\{k \leftarrow \overline{x}\}\ t)\ (\{k \leftarrow \overline{x}\}\ v) \\
\{k \leftarrow \overline{x}\}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{let}\ (\{k+1 \leftarrow \overline{x}\}\ \overline{t})\ \texttt{in}\ (\{k+1 \leftarrow \overline{x}\}\ t)
\end{array}
$$

where $\{k \leftarrow \overline{x}\}\ \overline{t} = \texttt{List.map}\ (\{k \leftarrow \overline{x}\}\,\cdot)\ \overline{t}$.

Fig. 4. Variable closing

Inversely to variable opening, there is an operation to transform free names into bound variables. The *variable closing* of a term is represented by ${}^{\backslash\overline{x}}t$, where $\overline{x}$ is the list of names to be bound (recall that the names in $\overline{x}$ are distinct). The definition of variable closing is based on a more general function $\{k \leftarrow \overline{x}\}t$ (Figure 4), where k indicates the level of nesting of binders.Whenever a free variable $\texttt{fvar}\ x$ is encountered, x is looked up in $\overline{x}$. If x occurs in position j, then the free variable is replaced by the bound variable $\texttt{bvar}\ k\ j$, otherwise it is left unchanged.Variable closing is then defined as ${}^{\backslash\overline{x}}t = \{0 \leftarrow \overline{x}\}t$. And its extension to lists is: ${}^{\backslash\overline{x}}\overline{t} = \texttt{List.map}\ ({}^{\backslash\overline{x}}\cdot)\ \overline{t}$.

Example 3. Now we close the term obtained by opening u in Example 2.
Let $t \equiv \texttt{let bvar 0 1}, \texttt{fvar}\ x\ \texttt{in app (abs fvar}\ x) (\texttt{bvar 0 1})$.

$$
\begin{array}{l}
{}^{\backslash x}t = \{0 \leftarrow x\}(\texttt{let}\ \{\texttt{bvar 0 1}, \texttt{fvar}\ x\}\ \texttt{in app (abs (fvar}\ x)) (\texttt{bvar 0 1})) \\
\quad = \texttt{let}\ \{1 \leftarrow x\}(\texttt{bvar 0 1}, \texttt{fvar}\ x) \\
\qquad \texttt{in}\ \{1 \leftarrow x\}(\texttt{app (abs fvar}\ x) (\texttt{bvar 0 1})) \\
\quad = \texttt{let bvar 0 1}, \texttt{bvar 1 0 in app (abs}\ \{2 \leftarrow x\}(\texttt{fvar}\ x)) (\texttt{bvar 0 1}) \\
\quad = \texttt{let bvar 0 1}, \texttt{bvar 1 0 in app (abs bvar 2 0) (bvar 0 1})
\end{array}
$$

Notice that the closed term coincides with u, the body of the abstraction in Example 2, although this is not always the case. □

2.3 Local closure, free variables and substitution

The locally nameless syntax in Figure 2 allows to build terms that have no corresponding expression in *Exp* (Figure 1). For instance, in $\texttt{abs (bvar 1 5})$ index 1 does not refer to a binder in the term. Well-formed terms, i.e., those matching expressions in *Exp*, are called *locally closed*. To determine if a term is locally closed one should check that every bound variable has valid indices, i.e., that they refer to binders in the term. An easier method is to open with fresh names every abstraction and `let`-declaration in the term to be checked, and verify that no bound variable is reached. This checking is implemented with the *local closure* predicate `lc` given in Figure 5.

Observe that we use cofinite quantification (as introduced by Aydemir et al. in [1]) in the rules for the binders, i.e., abstraction and `let`. Cofinite quantification is an elegant alternative to exist-fresh conditions and provides stronger

$$\text{LC\_VAR}\ \frac{}{\texttt{lc}\ (\texttt{fvar}\ x)} \qquad \text{LC\_ABS}\ \frac{\forall x \notin L \subseteq Id \quad \texttt{lc}\ t^{[x]}}{\texttt{lc}\ (\texttt{abs}\ t)}$$

$$\text{LC\_APP}\ \frac{\texttt{lc}\ t \quad \texttt{lc}\ v}{\texttt{lc}\ (\texttt{app}\ t\ v)} \qquad \text{LC\_LET}\ \frac{\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id \quad \texttt{lc}\ [t : \overline{t}]^{\overline{x}}}{\texttt{lc}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t)}$$

$$\text{LC\_LIST}\ \frac{\texttt{List.forall}\ (\texttt{lc}\ \cdot)\ \overline{t}}{\texttt{lc}\ \overline{t}}$$

Fig. 5. Local closure

$$\text{LCK-BVAR}\ \frac{i < k \wedge j < \texttt{List.nth}\ i\ \overline{n}}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{bvar}\ i\ j)} \qquad \text{LCK-APP}\ \frac{\texttt{lc\_at}\ k\ \overline{n}\ t \qquad \texttt{lc\_at}\ k\ \overline{n}\ v}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{app}\ t\ v)}$$

$$\text{LCK-FVAR}\ \frac{}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{fvar}\ x)} \qquad \text{LCK-LET}\ \frac{\texttt{lc\_at}\ (k+1)\ [|\overline{t}| : \overline{n}]\ [t : \overline{t}]}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{let}\ \overline{t}\ \texttt{in}\ t)}$$

$$\text{LCK-ABS}\ \frac{\texttt{lc\_at}\ (k+1)\ [1 : \overline{n}]\ t}{\texttt{lc\_at}\ k\ \overline{n}\ (\texttt{abs}\ t)} \qquad \text{LCK-LIST}\ \frac{\texttt{List.forall}\ (\texttt{lc\_at}\ k\ \overline{n}\ \cdot)\ \overline{t}}{\texttt{lc\_at}\ k\ \overline{n}\ \overline{t}}$$

Fig. 6. Local closure at level k

induction and inversion principles. Proofs are simplified, because it is not required to define exactly the set of fresh names (several examples of this are given in [5]). The rule LC-ABS establishes that an abstraction is locally closed if there exists a finite set of names L such that, for any name x not in L, the term $t^{[x]}$ is locally closed. Similarly, in the rule LC-LET we write $\overline{x}^{|\overline{t}|} \notin L$ to indicate that the list of distinct names $\overline{x}$ of length $|\overline{t}|$ are not in the finite set L. For any list $\overline{x}$ satisfying this condition, the opening of each term in the list of local declarations, $\overline{t}^{\overline{x}}$, and of the term affected by these declarations, $t^{\overline{x}}$, are locally closed. Notice that we have overloaded the predicate `lc` to work both on terms and list of terms. In the following we will overload other predicates and functions similarly. We write $[t : \overline{t}]$ for the list with head t and tail $\overline{t}$. In the following, $[\,]$ represents the empty list, $[t]$ is a unitary list, and $+\!\!+$ is the concatenation of lists.

We define a new predicate that checks if indices in bound variables are valid from a given level: *t is closed at level k*, written $\texttt{lc\_at}\ k\ \overline{n}\ t$ (Figure 6). As usual, k indicates the current depth, that is, how many binders have been passed by. Since binders can be either abstractions or local declarations, we need to keep the size of each binder (1 in the case of an abstraction, n for a `let` with n local declarations). These sizes are collected in the list $\overline{n}$, thus $|\overline{n}|$ should be at least k. A bound variable $\texttt{bvar}\ i\ j$ is closed at level k if i is smaller than k and j is smaller than $\texttt{List.nth}\ i\ \overline{n}$. The list $\overline{n}$ is new with respect to [5] because there the predicate `lc_at` is not defined for multiple binders.

It can be proved that if t is locally closed at level k for a given list of numbers $\overline{n}$, then it is also locally closed at level k for any list of numbers greater than $\overline{n}$.

Lemma 1. LC_AT_M_FROM_N $\quad \texttt{lc\_at}\ k\ \overline{n}\ t \Rightarrow \forall \overline{m} \geq \overline{n}\,.\,\texttt{lc\_at}\ k\ \overline{m}\ t$

$$
\begin{array}{lll}
\texttt{fv}(\texttt{bvar}\ i\ j) & = \emptyset & (\texttt{bvar}\ i\ j)[z/y] = \texttt{bvar}\ i\ j \\
\texttt{fv}(\texttt{fvar}\ x) & = \{x\} & (\texttt{fvar}\ x)[z/y] = \begin{cases} \texttt{fvar}\ z & \text{if } x = y \\ \texttt{fvar}\ x & \text{if } x \neq y \end{cases} \\
\texttt{fv}(\texttt{abs}\ t) & = \texttt{fv}(t) & (\texttt{abs}\ t)[z/y] = \texttt{abs}\ t[z/y] \\
\texttt{fv}(\texttt{app}\ t\ v) & = \texttt{fv}(t) \cup \texttt{fv}(v) & (\texttt{app}\ t\ v)[z/y] = \texttt{app}\ t[z/y]\ v[z/y] \\
\texttt{fv}(\texttt{let}\ \overline{t}\ \texttt{in}\ t) & = \texttt{fv}(\overline{t}) \cup \texttt{fv}(t) & (\texttt{let}\ \overline{t}\ \texttt{in}\ t)[z/y] = \texttt{let}\ \overline{t}[z/y]\ \texttt{in}\ t[z/y]
\end{array}
$$

where $\texttt{fv}(\overline{t}) = \texttt{List.foldright}\ (\cdot \cup \cdot)\ \emptyset\ (\texttt{List.map}\ \texttt{fv}\ \overline{t})$
$\overline{t}[z/y] = \texttt{List.map}\ ([z/y]\cdot)\ \overline{t}$.

Fig. 7. Free variables and substitution

Where $\overline{m} \geq \overline{n}$ is the pointwise lifting to lists of the usual ordering on naturals.

The two approaches for local closure are equivalent, so that it can be proved that a term is locally closed if and only if it is closed at level 0.

Lemma 2. LC_IIF_LC_AT $\quad \texttt{lc}\ t \Leftrightarrow \texttt{lc\_at}\ 0\ [\,]\ t$

If the opening of a term is locally closed then the opening of the term with a different variable is locally closed too.

Lemma 3. LC_OP $\quad \texttt{lc}\ t^{[x]} \Rightarrow \texttt{lc}\ t^{[y]}$

Computing the *free variables* of a term t is very easy in the locally nameless representation, since bound and free variables are syntactically different. The set of free variables of $t \in \mathit{LNExp}$ is denoted as $\texttt{fv}(t)$, and it is defined in Figure 7.

A name x is said to be *fresh for a term* t, written $\texttt{fresh}\ x\ \texttt{in}\ t$, if x does not belong to the set of free variables of t. Similarly for a list of distict names $\overline{x}$:

$$\frac{x \notin \texttt{fv}(t)}{\texttt{fresh}\ x\ \texttt{in}\ t} \qquad \frac{\overline{x} \notin \texttt{fv}(t)}{\texttt{fresh}\ \overline{x}\ \texttt{in}\ t}$$

A term t is *closed* if it has no free variables at all:

$$\frac{\texttt{fv}(t) = \emptyset}{\texttt{closed}\ t}$$

Substitution replaces a variable name by another. For $t \in \mathit{LNExp}$ and $z, y \in \mathit{Id}$, $t[z/y]$ is the term where z substitutes any occurrence of y in t (see Figure 7).

Under some conditions variable closing and variable opening are inverse operations. More precisely, opening a term with fresh names and closing it with the same names, produces the original term. Symmetrically, closing a locally closed term and then opening it with the same names gives back the initial term.

Lemma 4.

CLOSE_OPEN_VAR $\quad \texttt{fresh}\ \overline{x}\ \texttt{in}\ t \Rightarrow {}^{\backslash \overline{x}}(t^{\overline{x}}) = t$

OPEN_CLOSE_VAR $\quad \texttt{lc}\ t \Rightarrow ({}^{\backslash \overline{x}}t)^{\overline{x}} = t$

$$\textsc{Lam}\quad \Gamma : \lambda x.e \Downarrow \Gamma : \lambda x.e \qquad \textsc{App}\quad \frac{\Gamma : e \Downarrow \Theta : \lambda y.e' \qquad \Theta : e'[x/y] \Downarrow \Delta : w}{\Gamma : (e\ x) \Downarrow \Delta : w}$$

$$\textsc{Var}\quad \frac{\Gamma : e \Downarrow \Delta : w}{(\Gamma, x \mapsto e) : x \Downarrow (\Delta, x \mapsto w) : \hat{w}} \qquad \textsc{Let}\quad \frac{(\Gamma, \{x_i \mapsto e_i\}_{i=1}^n) : e \Downarrow \Delta : w}{\Gamma : \texttt{let}\ \{x_i = e_i\}_{i=1}^n\ \texttt{in}\ e \Downarrow \Delta : w}$$

Fig. 8. Natural semantics

3 Natural semantics for lazy λ-calculus

The semantics defined by Launchbury in [7] follows a lazy strategy. Judgements are of the form $\Gamma : e \Downarrow \Delta : w$, that is, the expression $e \in \mathit{Exp}$ in the context of the heap Γ reduces to the value w in the context of the heap Δ. *Values* ($w \in \mathit{Val}$) are expressions in weak-head-normal-form (*whnf*). *Heaps* are partial functions from variables into expressions. Each pair (variable, expression) is called a *binding*, and it is represented by $x \mapsto e$. During evaluation, new bindings may be added to the heap, and bindings may be updated to their corresponding computed values. The rules of this natural semantics are shown in Figure 8. The normalization of the λ-calculus, that has been mentioned in Section 2, simplifies the definition of the operational rules, although a renaming is still needed ($\hat{w}$ in Var) to avoid name clashing. This renaming is justified by Barendregt's variable convention [3].

Example 4. Without the renaming in rule Var heaps may end up binding a same name more than once. Take for instance the evaluation of the expression $e \equiv \texttt{let}\ x_1 = \lambda y.(\texttt{let}\ z = \lambda v.y\ \texttt{in}\ y), x_2 = (x_1\ x_3), x_3 = (x_1\ x_4), x_4 = \lambda s.s\ \texttt{in}\ x_2$ in the context of the empty heap. The evaluation of e implies the evaluation of x_2, and then the evaluation of $(x_1 x_3)$. This application leads to the addition of z to the heap bound to $\lambda v.x_3$. Subsequently, the evaluation of x_3 implies the evaluation of $(x_1 x_4)$. Without a renaming of values, variable z is added again to the heap, now bound to $\lambda v.x_4$. □

Theorem 1 in [7] states that "every heap/term pair occurring in the proof of a reduction is *distinctly named*", but we have found that the renaming fails to ensure this property. At least, it depends on how much fresh is this renaming.

Example 5. Let us evaluate in the context of the empty heap the expression

$$e \equiv \texttt{let}\ x_1 = (x_2\ x_3), x_2 = \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y, x_3 = \lambda s.s\ \texttt{in}\ x_1$$

$$\begin{array}{lll}
 & \{\,\} : e & \\
\textsc{Let} & \{x_1 \mapsto (x_2\ x_3), x_2 \mapsto \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y, x_3 \mapsto \lambda s.s\} : x_1 & \\
\textsc{Var} & \{x_2 \mapsto \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y, x_3 \mapsto \lambda s.s\} : (x_2\ x_3) & \\
\textsc{App} & \{x_2 \mapsto \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y, x_3 \mapsto \lambda s.s\} : x_2 & \\
\textsc{Var} & \{x_3 \mapsto \lambda s.s\} : \lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y & \\
\textsc{Lam} & & \\
 & \{x_3 \mapsto \lambda s.s\} : \boxed{\lambda z.\texttt{let}\ y = \lambda t.t\ \texttt{in}\ y} &
\end{array}$$

At this point the rule VAR requires to rename the value highlighted in the square. Notice that x_1 is fresh in the actual heap/term pair, and hence can be chosen to rename y. This would lead later in the derivation to introduce twice x_1 in the heap. The solution is to consider the condition of freshness in the whole derivation. This notion has not been formally defined by Launchbury. □

3.1 Locally nameless heaps

Before translating the semantic rules in Figure 8 to the locally nameless representation defined in Section 2, we have to establish how *bindings* and *heaps* are represented in this notation.

Recall that bindings associate expressions to free variables, therefore bindings are now pairs $(\texttt{fvar}\ x, t)$ with $x \in Id$ and $t \in LNExp$. To simplify, we will just write $x \mapsto t$. In the following, we will represent a heap $\{x_i \mapsto t_i\}_{i=1}^{n}$ as $(\overline{x} \mapsto \overline{t})$, with $|\overline{x}| = |\overline{t}| = n$. The set of the locally-nameless-heaps is denoted as *LNHeap*.

The *domain* of a heap Γ, written $\texttt{dom}(\Gamma)$, collects the set of names that are bound in the heap.

$$\texttt{dom}(\emptyset) = \emptyset \qquad\qquad \texttt{dom}(\Gamma, x \mapsto t) = \texttt{dom}(\Gamma) \cup \{x\}$$

In a well-formed heap names are defined at most once and terms are locally closed. The predicate `ok` expresses that a heap is well-formed:

$$\text{OK-EMPTY}\ \frac{}{\texttt{ok}\ \emptyset} \qquad\qquad \text{OK-CONS}\ \frac{\texttt{ok}\ \Gamma \qquad x \notin \texttt{dom}(\Gamma) \qquad \texttt{lc}\ t}{\texttt{ok}\ (\Gamma, x \mapsto t)}$$

The function `names` returns the set of names that appear in a heap, i.e., the names occurring in the domain or in the right-hand side terms:

$$\texttt{names}(\emptyset) = \emptyset \qquad\qquad \texttt{names}(\Gamma, x \mapsto t) = \texttt{names}(\Gamma) \cup \{x\} \cup \texttt{fv}(t)$$

This definition can be extended to (heap: term) pairs:

$$\texttt{names}(\Gamma : t) = \texttt{names}(\Gamma) \cup \texttt{fv}(t)$$

Next we define the freshness predicate of a list of names in a (heap:term) pair:

$$\frac{\overline{x} \notin \texttt{names}(\Gamma : t)}{\texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma : t)}$$

Substitution of variable names is extended to heaps as follows:

$$\emptyset[z/y] = \emptyset \qquad\qquad (\Gamma, x \mapsto t)[z/y] = (\Gamma[z/y], x[z/y] \mapsto t[z/y])$$
$$\textit{where}\ x[z/y] = \begin{cases} z & \text{if } x = y \\ x & \text{otherwise} \end{cases}$$

The following property is verified:

Lemma 5. OK_SUBS_OK $\qquad \texttt{ok}\ \Gamma \wedge y \notin \texttt{dom}(\Gamma) \Rightarrow \texttt{ok}\ \Gamma[y/x]$

$$\textsc{LNLam}\quad \frac{\{\texttt{ok}\ \Gamma\} \qquad \{\texttt{lc}\ (\texttt{abs}\ t)\}}{\Gamma : \texttt{abs}\ t \Downarrow \Gamma : \texttt{abs}\ t}$$

$$\textsc{LNVar}\quad \frac{\Gamma : t \Downarrow \Delta : w \qquad \{x \notin \texttt{dom}(\Gamma) \cup \texttt{dom}(\Delta)\}}{(\Gamma, x \mapsto t) : (\texttt{fvar}\ x) \Downarrow (\Delta, x \mapsto w) : w}$$

$$\textsc{LNApp}\quad \frac{\Gamma : t \Downarrow \Theta : \texttt{abs}\ u \qquad \Theta : u^{[x]} \Downarrow \Delta : w \qquad \{x \notin \texttt{dom}(\Gamma) \Rightarrow x \notin \texttt{dom}(\Delta)\}}{\Gamma : \texttt{app}\ t\ (\texttt{fvar}\ x) \Downarrow \Delta : w}$$

$$\textsc{LNLet}\quad \frac{\forall \overline{x}^{|\overline{t}|} \notin L \subseteq Id\ \ (\Gamma, \overline{x} \mapsto \overline{t}^{\overline{x}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{x}}) : w^{\overline{x}} \qquad \{\overline{y}^{|\overline{t}|} \notin L \subseteq Id\}}{\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t \Downarrow (\overline{y} +\!\!+ \overline{z} \mapsto \overline{u}^{\overline{y}}) : w^{\overline{y}}}$$

Fig. 9. Locally nameless natural semantics

3.2 Locally nameless semantics

Once the locally nameless syntax and the corresponding operations, functions and predicates have been defined, three steps are sufficient to translate an inductive definition on λ-terms from the named representation into the locally nameless notation (as it is explained in [5]):

1. Replace the named binders, i.e., abstractions and let-constructions, with nameless binders by opening the bodies.
2. Cofinitely quantify the names introduced for variable opening.
3. Add premises to inductive rules in order to ensure that inductive judgements are restricted to locally closed terms.

We apply these steps to the inductive rules for the lazy natural semantics given in Figure 8. These rules produce judgements involving λ-terms as well as heaps. Hence, we also add premises that ensure that inductive judgements are restricted to well-formed heaps. The rules using the locally nameless representation are shown in Figure 9. For clarity, in the rules we put in braces the side-conditions to distinguish them better from the judgements.

The main difference with the rules in Figure 8 is the rule LNLet. To evaluate $\texttt{let}\ \overline{t}\ \texttt{in}\ t$ the local terms in $\overline{t}$ have to be introduced in the heap, so that the body t is evaluated in this new context. To this purpose fresh names $\overline{x}$ are needed to open the local terms and the body. The evaluation of $t^{\overline{x}}$ produces a final heap and a value. Both are dependent on the names chosen for the local variables. The domain of the final heap consists of the local names $\overline{x}$ and the rest of names, say $\overline{z}$. The rule LNLet is cofinite quantified. As it is explained in [5], the advantage of the cofinite rules over existential and universal ones is that the freshness side-conditions are not explicit. In our case, the freshness condition for $\overline{x}$ is *hidden* in the finite set L, which includes the names that should be avoided during the reduction. The novelty of our cofinite rule, compared with the ones appearing in [1] and [5] (that are similar to the cofinite rules for the predicate `lc` in Figure 5), is that the names introduced in the (infinite) premises do appear

in the conclusion too. Therefore, in the conclusion of the rule LNLet we can replace the names $\overline{x}$ by any list $\overline{y}$ not in L.

The problem with explicit freshness conditions is that they are associated just to rule instances, while they should apply to the whole reduction proof. Take for instance the rule LNVar. In the premise the binding $x \mapsto t$ does no longer belong to the heap. Hence, a valid reduction for this premise may chose x as fresh (this corresponds to the problem shown in Example 5). We avoid this situation by requiring that x remains undefined in the final heap too. By contrast to the rule Var in Figure 8, no renaming of the final value w is needed.

The side-condition of rule LNApp deserves an explanation too. Let us suppose that x is undefined in the initial heap Γ. We must avoid that x is chosen as a fresh name during the evaluation of t. For this reason we require that x is defined in the final heap Δ only if x was already defined in Γ. Notice how the body of the abstraction, that is u, is open with the name x. This is equivalent to the substitution of x for y in the body of the abstraction $\lambda y.e'$ (see rule App in Figure 8).

A *regularity* lemma ensures that the judgements produced by this reduction system involve only well-formed heaps and locally closed terms.

Lemma 6.
REGULARITY $\quad \Gamma : t \Downarrow \Delta : w \Rightarrow \mathtt{ok}\ \Gamma \wedge \mathtt{lc}\ t \wedge \mathtt{ok}\ \Delta \wedge \mathtt{lc}\ w$

Similarly, Theorem 1 in [7] ensures that the property of being *distinctly named* is preserved by the rules in Figure 8. However, as shown in Example 5, the correctness of this result requires that freshness is relative to whole reduction proofs instead to the scope of rules.

The next lemma states that names defined in a context heap remain defined after the evaluation of any term in that context.

Lemma 7.
DEF_NOT_LOST $\quad \Gamma : t \Downarrow \Delta : w \Rightarrow \mathtt{dom}(\Gamma) \subseteq \mathtt{dom}(\Delta)$

Furthermore, fresh names are introduced only by the rule LNLet and, by the previous lemma, they remain bound in the final (heap: value) pair. Hence, any free variable appearing in a final (heap: value) pair is undefined only if the variable already occurs in the initial (heap: term) pair.

Lemma 8.
ADD_VARS $\quad \Gamma : t \Downarrow \Delta : w$
$\Rightarrow (x \in \mathtt{names}(\Delta : w) \Rightarrow (x \in \mathtt{dom}(\Delta) \vee x \in \mathtt{names}(\Gamma : t)))$

A *renaming* lemma ensures that the evaluation of a term is independent of the fresh names chosen in the reduction process. Moreover, any name in the context can be replaced by a fresh one without changing the meaning of the terms evaluated in that context. In fact, reduction proofs for (heap: term) pairs are unique up to renaming of the variables defined in the context heap.

Lemma 9.
RENAMING $\quad \Gamma : t \Downarrow \Delta : w \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Gamma : t) \wedge \mathtt{fresh}\ y\ \mathtt{in}\ (\Delta : w)$
$\Rightarrow \Gamma[y/x] : t[y/x] \Downarrow \Delta[y/x] : w[y/x]$

In addition, the renaming lemma permits to prove an *introduction* lemma for the cofinite rule LNLet which establishes that the corresponding existential rule is admissible too.

Lemma 10.
LET_INTRO $(\Gamma, \overline{x} \mapsto \overline{t^{\overline{x}}}) : t^{\overline{x}} \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{x}}}) : w^{\overline{x}} \wedge \texttt{fresh}\ \overline{x}\ \texttt{in}\ (\Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t)$
$\Rightarrow \Gamma : \texttt{let}\ \overline{t}\ \texttt{in}\ t \Downarrow (\overline{x} +\!\!+ \overline{z} \mapsto \overline{u^{\overline{x}}}) : w^{\overline{x}}$

This result, together with the renaming lemma, justifies that our rule LNLet is equivalent to Launchbury's rule Let used with normalized terms.

4 Related work

In order to avoid α-conversion, we first considered a nameless representation like the de Bruijn notation [6], where variable names are removed and replaced by natural numbers. But this notation has several drawbacks. First of all, the de Bruijn representation is hard to read for humans. Even if we intend to check our results with some proof assistant like Coq [4], human readability helps intuition. At a more technical level, the de Bruijn notation does not have a good way to handle free variables, which are represented by indices, alike to bound variables. This is a serious weakness for our application. Recall that Launchbury's semantics uses contexts heaps that collect the bindings for the free variables that may occur in the term under evaluation. Any change in the domain of a heap, i.e., adding or deleting a binding, would lead to a shifting of the indices, thus complicating the statement and proof of results. Therefore, we prefer the more manageable locally nameless representation, where bound variable names are replaced by indices but free variables keep their names. This mixed notation combines the advantages of both named and nameless representations. On the one hand, α-conversion is avoided all the same. On the other hand, terms stay readable and easy to manipulate.

There exists in the literature different proposals for a locally nameless representation, and many works using these representations. Charguéraud offers in [5] a brief survey on these works, that we recommend to the interested reader.

Launchbury (implicitly) assumes Barendregt's variable convention [3] twice in [7]. First when he defines his operational semantics only for normalized λ-terms (i.e. every binder in a term binds a distinct name, which is also distinct from any free variable); and second, when he requires a (fresh) renaming of the values in the rule Var (see Figure 8). Urban, Berghofer and Norrish propose in [15] a method to strengthen an induction principle (corresponding to some inductive relation), so that Barendregt's variable convention comes already built in the principle. Unfortunately, we cannot apply these ideas to Launchbury's semantics, because the semantic rules (shown in Figure 8) do not satisfy the conditions that guarantee the *variable convention compatibility*, as described in [15]. In fact, as we have already pointed out, Launchbury's Theorem 1 (in [7]) is only correct if the renaming required in each application of the rule Var is fresh in the whole reduction proof. Therefore, we cannot use directly Urban's nominal package for

Isabelle/HOL [14] (including its recent extensions for general bindings described in [16]).

Nevertheless, Urban et al. achieve the "inclusion" of the variable convention in an induction principle by adding to each induction rule a side condition which expresses that the set of *bound* variables (i.e., those that appear in a binding position in the rule) are fresh in some *induction context* ([15]). Furthermore, this context is required to be finitely supported. This is closely related to the cofinite quantification that we have used for the rule LNLet in Figure 9. Besides, one important condition to ensure the variable convention compatibility is the *equivariance* of the functions and predicates occurring in the induction rules. Equivariance is a notion from nominal logic [10]. A relation is equivariant if it is preserved by permutation of names. Although we have not proven that the reduction relation defined by the rules in Figure 9 is equivariant, our *renaming lemma* (Lemma 9) establishes a similar result, that is, the reduction relation is preserved by (fresh) renaming.

5 Conclusions

We have used a more modern approach to binding, i.e., a locally nameless representation for the λ-calculus extended with mutually recursive local declarations. With this representation the reduction rule for local declarations implies the introduction of fresh names. We have used neither an existential nor a universal rule for this case. Instead, we have opted for a cofinite rule as introduced by Aydemir et al. in [1]. Freshness conditions are usually considered in each rule individually. Nevertheless, this technique produces name clashing when considering whole reduction proofs. A solution might be to decorate judgements with the set of forbidden names and indicate how to modify this set during the reduction process (this approach has been taken by Sestoft in [13]). However, this could be too restrictive in many occasions. Besides, existential rules are not easy to deal with because each reduction is obtained just for one specific list of names. If any of the names in this list causes a name clashing with other reduction proofs, then it is cumbersome to demonstrate that an alternative reduction for a fresh list does exist. Cofinite quantification has allowed us to solve this problem because in a single step reductions are guaranteed for an infinite number of lists of names. Nonetheless, our introduction lemma (Lemma 10) guarantees that a more conventional exists-fresh rule is correct in our reduction system too.

The cofinite quantification that we have used in our semantic rules is more complex than those in [1] and [5]. Our cofinite rule LNLet in Figure 9 introduces quantified variables in the conclusion as well, as the latter depends on the chosen names.

Compared to Launchbury's original semantic rules, our locally nameless rules include several extra side-conditions. Some of these conditions require that heaps and terms are well-formed (like in rule LNLam). The rest of side-conditions express restrictions on the choice of fresh names. These restrictions, together

with the cofinite quantification, fix the problem with the renaming in rule VAR that we have shown in Example 5.

For our locally nameless semantics we have shown a *regularity lemma* (Lemma 6) which ensures that every term and heap involved in a reduction proof is well-formed, and with a *renaming lemma* (Lemma 9) which indicates that the choice of names (free variables) is irrelevant as long as they are fresh enough. A heap may be seen as a multiple binder. Actually, the names defined (bound) in a heap can be replaced by other names, provided that terms keep their meaning in the context represented by the heap. Our renaming lemma ensures that whenever a heap is renamed with fresh names, reduction proofs are preserved. This renaming lemma is essential in rule induction proofs for some properties of the reduction system. More concretely, when one combines several reduction proofs coming from two or more premises in a reduction rule (for instance, in rule LNAPP in Figure 9).

In summary, the contributions of this paper are:

1. A locally nameless representation of the λ-calculus extended with recursive local declarations;
2. A locally nameless version of the inductive rules of Launchbury's natural semantics for lazy evaluation;
3. A new version of cofinite rules where the variables quantified in the premises do appear in the conclusion too;
4. A set of interesting properties of our reduction system, including the regularity, the introduction and the renaming lemmas; and
5. A way to guarantee Barendregt's variable convention by redefining Launchbury's semantic rules with cofinite quantification and extra side-conditions.

6 Future work

Our future tasks include the implementation in the proof assistant Coq [4] of the natural semantics redefined in this paper, and the formalization of the proofs for the lemmas given (regularity, renaming, introduction, etc.), which at present are just paper-and-pencil proofs. We will use this implementation to prove formally the equivalence of Launchbury's natural semantics with the alternative version given also in [7]. As we mentioned in Section 1, this alternative version differs from the original one in the introduction of indirections during β-reduction and the elimination of updates. At present we are working on the definition (using the locally nameless representation) of two intermediate semantics, one introducing indirections and the other without updates. Then, we will establish equivalence relations between the heaps obtained by each semantics, which makes able to prove the equivalence of the original natural semantics and the alternative one through the intermediate semantics.

7 Acknowledgments

This work is partially supported by the projects: TIN2009-14599-C03-01 and S2009/TIC-1465.

References

1. B. E. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM Symposium on Principles of Programming Languages, POPL'08*, pages 3–15. ACM Press, 2008.
2. C. Baker-Finch, D. King, and P. W. Trinder. An operational semantics for parallel lazy evaluation. In *ACM-SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 162–173. ACM Press, 2000.
3. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
4. Y. Bertot. Coq in a hurry. *CoRR*, abs/cs/0603118, 2006.
5. A. Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, pages 1–46, 2011.
6. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972.
7. J. Launchbury. A natural semantics for lazy evaluation. In *ACM Symposium on Principles of Programming Languages, POPL'93*, pages 144–154. ACM Press, 1993.
8. K. Nakata and M. Hasegawa. Small-step and big-step semantics for call-by-need. *CoRR*, abs/0907.4640, 2009.
9. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
10. A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.
11. L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. *Trends in Functional Programming*, volume 10, chapter An Operational Semantics for Distributed Lazy Evaluation, pages 65–80. Intellect, 2010.
12. L. Sánchez-Gil, M. Hidalgo-Herrero, and Y. Ortega-Mallén. A locally nameless representation for a natural semantics for lazy evaluation. Technical Report 01/12, Dpt. Sistemas Informáticos y Computación. Universidad Complutense de Madrid, 2012. http://maude.sip.ucm.es/eden-semantics/.
13. P. Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, 1997.
14. C. Urban. Nominal techniques in Isabelle/HOL. *Journal of Automatic Reasoning*, 40(4):327–356, 2008.
15. C. Urban, S. Berghofer, and M. Norrish. Barendregt's variable convention in rule inductions. In *Proceedings of the 21st International Conference on Automated Deduction: Automated Deduction*, pages 35–50. LNCS 4603, Springer-Verlag, 2007.
16. C. Urban and C. Kaliszyk. General bindings and alpha-equivalence in nominal Isabelle. In *Proceedings of the 20th European Symposium on Programming*, pages 480–500. LNCS 6602, Springer-Verlag, 2011.
17. M. van Eekelen and M. de Mol. *Reflections on Type Theory, λ-calculus, and the Mind. Essays dedicated to Henk Barendregt on the Occasion of his 60th Birthday*, chapter Proving Lazy Folklore with Mixed Lazy/Strict Semantics, pages 87–101. Radboud University Nijmegen, 2007.