A confluent calculus of macro expansion and evaluation

Ana Bove Laura Arbilla

Technical Report INCO-91-01
December 1991

Instituto de Computación
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay

A confluent calculus of macro expansion and evaluation

Ana Bove[*]
bove@incouy.edu.uy

Laura Arbilla[†]
arbilla@incouy.edu.uy

Instituto de Computación
Facultad de Ingeniería, Universidad de la República
Montevideo, Uruguay

Abstract

Syntactic abbreviations or *macros* provide a powerful tool to increase the syntactic expressivity of programming languages. The expansion of these abbreviations can be modeled with substitutions. This paper presents an operational semantics of macro expansion and evaluation where substitutions are handled explicitly. The semantics is defined in terms of a confluent, simple, and intuitive set of rewriting rules. The resulting semantics is also a basis for developing correct implementations.

Keywords: Macros, explicit substitutions, rewriting semantics, programming languages.

[*]Supported by a scholarship of Escuela Superior Latinoamericana de Informática, La Plata, Argentina.
[†]Partially supported by a PEDECIBA (Programa de Desarrollo de las Ciencias Básicas) grant.

1 Introduction

In computer science as well as in mathematics and logic, the use of abbreviations is common practice. A typical example used in logic is [10]:

$$a \leftrightarrow b \overset{df}{\equiv} (a \Rightarrow b) \wedge (b \Rightarrow a)$$

This means that the expression to the left of the symbol $\overset{df}{\equiv}$, which we read "a if and only if b," is an abbreviation of the expression to its right, which we read "a implies b, and b implies a". In computer science this practice is not less common, though it realizes in different forms. Most programming languages, such as C [14], Scheme [18], and many assembly languages, provide macro definition tools.

The use of syntactic abbreviations in mathematics as well as in programming languages has two advantages [15]. First, it allows the abstraction over repeated syntactic components, and therefore improves the readability and clarity of programs. Second, it eases the design of new language constructs, which can be defined by language designers and programmers. The semantics of these constructs is straightforward because they are defined in terms of previously defined constructs that have well understood semantics.

Syntactic abbreviations, or *macros*, improve the syntactic expressivity of programming languages but do not increase their semantic power [8, 17]. These abbreviations enrich assembly as well as modern high level languages. A typical example of a syntactic abbreviation in Scheme is *let*, defined by the following notational definition:

$$(let\ \ \mathsf{x}\ \ be\ \ \mathsf{v}\ \ in\ \ \mathsf{B}) \overset{df}{\equiv} ((\lambda\ (\mathsf{x})\ \mathsf{B})\ \ \mathsf{v}) \tag{1}$$

In this definition, *let*, *be*, and *in* are keywords while x, v, and B are metavariables.

Syntactic abbreviations are defined over a *core* language of well–known semantics. Our core language is the λ–calculus with numerical constants [2].

Associated with the definition of a macro are the notions of *instance* and *expansion*. An instance of a macro is a term having the same form as the expression appearing in the left of the definition, where arbitrary expressions appear in the places of the metavariables. For example,

$$(let\ \ y\ \ be\ \ 3\ \ in\ \ (\lambda\ (z)\ (+\ z\ y))) \tag{2}$$

is an instance of the syntactic abbreviation *let*, where the expressions y, 3, and $(\lambda\ (z)\ (+\ z\ y))$ take the places of the metavariables x, v, and B, respectively.

The expansion of an instance is the core expression that results from one or more steps of expansion. In each step, an instance is replaced by the right–hand–side of its notational definition, where metavariables are replaced by the expressions that take their places in the instance. For example, the expansion of (2) is:

$$((\lambda\ (y)\ (\lambda\ (z)\ (+\ z\ y)))\ \ 3) \tag{3}$$

Traditionally, an interpreter or compiler expands all the instances appearing in a program before evaluation (at parsing time or even earlier when pre–processors are available). Observing that the replacement of metavariables by expressions when expanding an instance can be modeled with substitutions, we develop a simple and intuitive operational semantics of macro expansion using the explicit substitution calculus of Abadi et al [1]. The resulting semantics respects the binding strategy of the core language and does not perform unwanted capture of identifiers during expansion.

In this method, the expansion is done *completely* before starting execution, although some of the expansions may be avoidable. We avoid unnecessary expansions by defining a straightforward extension of the semantics to one that does expansion and evaluation at the same time. This extension suggests an interpreter implementation where expansion is delayed as much as possible. Given a lazy [11] reduction strategy for our core language, the expansion of an instance appearing as argument in an application may never be performed if the argument is not used.

The rest of the paper is organized as follows. Section 2 informally presents notational definitions and introduces the main ideas behind manipulating substitutions explicitly. Section 3 presents a macro declaration language and gives a semantics of macro expansion using explicit substitutions. In Section 4, we present a variation of this semantics that also evaluates expressions. Section 5 presents conclusions, related work, and future directions of research.

2 Notational definitions and explicit substitutions

Syntactic abbreviations are introduced by notational definitions, briefly described here. In addition, this section motivates the use of explicit substitutions to model the expansion of these abbreviations.

2.1 Notational definitions

Notational definitions are equations of the form $l \overset{df}{\equiv} r$, meaning that the expression l abbreviates the expression r. For this reason, l is also called a *syntactic abbreviation*. An example of a notational definition is equation (1).

Syntactic abbreviations are not terms of the core and contain only keywords and metavariables. Expression r is formed with elements of the core language as well as previously defined syntactic abbreviations. Syntactic abbreviations *syntactically extend* the core language.

A *macro processor* is a virtual machine that, given a set D of notational definitions and an expression e of the core language extended with the syntactic abbreviations defined in D, transforms every instance of an abbreviation appearing in e into its expansion, and obtains a core expression. In Section 3, we present a formal specification of a macro processor.

2.2 Explicit substitutions

In this subsection, we show how the practice of macro expansion can be modeled through the manipulation of *explicit substitutions* [1].

Substitutions are fundamental in the λ–calculus. The classical β–rule [2]:

$$((\lambda x.e_1) \;\; e_2) \rightarrow \;\; e_1\{e_2/x\}$$

manipulates substitutions *implicitly*. If e_1 and e_2 are λ–terms, expression $e_1\{e_2/x\}$ is not a λ–term but a notation that represents the term e_1 where all the free occurrences of variable x are replaced by e_2.

Abadi et al introduce a variation of the λ–calculus, the $\lambda\sigma$–calculus, where substitutions generated by β–reductions are manipulated *explicitly.* Substitutions and substitution application have a syntactic representation in the $\lambda\sigma$–calculus. The β–rule is:[1]

$$((\lambda x.e_1)\ \ e_2) \rightarrow\ e_1[\{x \leftarrow e_2, \phi\}]$$

where $e[s]$ is a $\lambda\sigma$–term denoting the term e to which substitutions s is applied, and substitution $\{x \leftarrow e, \phi\}$ denotes a substitution where x is replaced by e. The empty substitution is denoted by ϕ.

Most implementations of macro expansion use an environment where each metavariable in the left–hand–side of a definition is associated to the subexpression that takes its place in the instance [7]. We propose to represent these environments as *explicit substitutions* in a formal semantics of macro expansion, as does Abadi et al in the $\lambda\sigma$–calculus.

The use of substitutions in macro expansion is best illustrated with an example. Consider the following definition of $freeze$:

$$freeze\ \ \mathsf{a} \overset{df}{\equiv} (\lambda x.\mathsf{a}) \tag{4}$$

To obtain the expansion of the instance $freeze\ \ e$, for expression e, we substitute in the right–hand–side of definition (4) the metavariable a for the expression e. This can be represented by the expression $(\lambda x.\mathsf{a})[\{\mathsf{a} \leftarrow e, \phi\}]$, where $\{\mathsf{a} \leftarrow e, \phi\}$ is an explicit substitution. Observe that these are substitutions of metavariables by expressions and not of variables by expressions as in the $\lambda\sigma$–calculus.

This substitution on λ–expressions cannot be naïve, however, because if the variable x appears free in e, the λ operator captures it. Abadi et al introduce in the $\lambda\sigma$–calculus a renaming operator ($\uparrow$) for avoiding this capturing. We adapt this operator so that it takes an argument x indicating the conflicting name. The following rules are needed for expansion:

$$\begin{array}{rcl}
(\lambda x.e)[s] & \rightarrow & (\lambda x.e[s\uparrow x]) \\
\{\mathsf{a} \leftarrow e, s\}\uparrow x & \rightarrow & \{\mathsf{a} \leftarrow e\uparrow x, s\uparrow x\} \\
\mathsf{a}\{\mathsf{a} \leftarrow e, s\} & \rightarrow & e \\
\phi\uparrow x & \rightarrow & \phi
\end{array}$$

Using these rules, the expansion of $freeze\ \ x$, resulting from $(\lambda x.\mathsf{a})[\{\mathsf{a} \leftarrow x, \phi\}]$ is $(\lambda x.x\uparrow x)$. The expression $x\uparrow x$ is similar to $x[\uparrow]$ in the $\lambda\sigma$–calculus. Instead of composition of operators $\uparrow$, we use exponents on variable names. Thus, we represent $x\uparrow x$ with x^1, $(x\uparrow x)\uparrow x$ with x^2 , and so on. With our notation, the above expansion takes the form of $(\lambda x.x^1)$.

3 Macro expansion semantics

In this section, we formalize the ideas of Section 2, and present an operational semantics of macro expansion over the core language.

[1]We slightly modify the syntax.

Syntactic domains:

$$\begin{array}{rcll}
\mathsf{a}, \mathsf{b} & \in & \mathit{Meta} & \text{(Metavariables)} \\
u, w, x & \in & \mathit{Var} & \text{(Variables)} \\
num, n, m, m' & \in & \mathit{Nat} & \text{(Natural numbers)} \\
b & \in & \mathit{Bool} = \{t, f\} & \text{(Booleans)} \\
y & \in & \mathit{Meta} \cup \mathit{Var} & \\
\mathit{Meta} \cap \mathit{Var} & = & \mathit{Var} \cap \mathit{Nat} = \mathit{Meta} \cap \mathit{Nat} = \emptyset &
\end{array}$$

Syntax of Λ_n:

$$e \ ::= \ num \mid x^n \mid (\lambda x.e) \mid (e \ \ e)$$

Syntax of $\mathcal{L}$:

$$\begin{array}{rcll}
p & ::= & D \ r \quad \text{s.t. } \mathrm{MV}(r) = \emptyset & \mathit{Pgm} \\
d & ::= & [l \overset{df}{\equiv} r] & \mathit{Ndef} \\
l & ::= & y \mid y \ l & \mathit{Lhs} \\
r & ::= & num \mid x^n \mid \mathsf{a} \mid (\lambda x.r) \mid (\lambda \mathsf{a}.r) \mid (r \ \ r) \mid \langle Lr \rangle & \mathit{Rhs} \\
D & ::= & \epsilon \mid d \ D & \mathit{LNdef} \\
Lr & ::= & r \mid r \ Lr & \mathit{LRhs}
\end{array}$$

Figure 1: Languages syntax.

3.1 The core language

The syntax of the core language, Λ_n, is in Figure 1. Language Λ_n is the language of the λ–calculus with numbers [2] where variables that are not binding instances are labeled. A variable has a name and an exponent. The exponent defines its distance (number of λ operators that bind a same named variable) from its binding instance. For example, the expression:

$$(\lambda x.(\lambda w.(\lambda x.((x^0 \ \ x^1) \ \ w^0))))$$

would be written in the λ–calculus as:

$$(\lambda u.(\lambda w.(\lambda x.((x \ \ u) \ \ w))))$$

The use of exponents in variable names efficiently implements the α–conversion, as does de Bruijn technique [4], but the use of names improves legibility. In this paper, sometimes x^0 will be denoted as x.

3.2 The macro definition language

We define language $\mathcal{L}$ as Λ_n enriched with notational definitions and instances of macros. The syntax of $\mathcal{L}$ is in Figure 1. We use a different font for metavariables, to distinguish them from variables. The symbol ϵ denotes the empty string and $\mathrm{MV}(r)$ denotes the set of metavariables appearing in expression r.

A program p ($\in$ *Pgm*) is a list D ($\in$ *LNdef*), possibly empty, of notational definitions followed by an expression that does not contain metavariables.

Notational definitions are as described in Section 2. A notational definition d ($\in$ *Ndef*) is enclosed within square brackets ([ ]). The left-hand-side expression l ($\in$ *Lhs*) is a list of variables and metavariables. The right-hand-side expression r ($\in$ *Rhs*) is a natural number, a variable, a metavariable, a functional abstraction, an application, or a list of expressions of *Rhs* between angle brackets ($\langle\ \rangle$). Users may introduce fresh identifiers in the right hand side of macro definitions. Names appearing in the left hand side that are not metavariables are *keywords*. For instance, in the definition of $freeze$, x is a fresh identifier and $freeze$ a keyword. We treat fresh identifiers and keywords as variables for convenience.

Functional abstractions bind variables and metavariables. Angle brackets enclose an instance of a syntactic abbreviation.

An example of a valid program is:

$$[let\ \ \mathsf{x}\ \ be\ \ \mathsf{v}\ \ in\ \ \mathsf{B} \overset{df}{\equiv}\ ((\lambda \mathsf{x}.\mathsf{B})\ \ \mathsf{v})]\ \ (\lambda x.(\lambda w.\langle let\ \ u\ \ be\ \ x\ \ in\ \ (u\ \ w)\rangle))$$

containing notational definition (1). The body of the program contains an instance of *let*.

To simplify our study, we restrict the set of notational definitions as follows. Let $d_1 \cdots\ d_n$ be the list of definitions of a program, and $l_1 \cdots\ l_n$ and be $r_1 \cdots\ r_n$ their respective left and right–hand–sides. The following conditions must hold:

1. (non–recursive)
   - r_1 is an expression of the core language Λ_n.
   - r_j, $j = 2 \ldots n$, is formed with expressions of Λ_n and instances of definitions d_i where $i < j$.
2. (linear) A metavariable appears only once in a left–hand–side. Formally,

   $$\text{for any } l_h = w_1 \ldots w_m,\ h = 1 \ldots n. \quad (i \neq j \wedge w_i, w_j \in Meta \ \ \Rightarrow \ \ w_i \neq w_j)$$

3. (unambiguous) The set of notational definitions does *not* contain two left–hand–sides that are equal modulo a renaming of the metavariables.

   Two left–hand–sides $l_h = w_1 \ldots w_m$ and $l_k = u_1 \ldots u_m$, where $h \neq k$ are *equal modulo a renaming of metavariables* iff: $\forall\ i = 1 \ldots m\ \ (w_i = u_i \vee w_i, u_i \in Meta)$
4. No new metavariables are introduced by the rhs: $\mathrm{MV}(r_h) \subseteq \mathrm{MV}(l_h))$, $h = 1 \ldots n$

3.3 Instance and instance substitution

Before we can formally state the operational semantics of macro expansion in an expression of language $\mathcal{L}$, we need to formally define the notions of macro instance and instance substitution.

Definition 1 (Instance) *An expression $\langle t_1 \ldots t_m \rangle$ is an* instance *of a macro definition $[l \overset{df}{\equiv} r]$, where $l \equiv w_1 \ldots w_m$, if the following conditions hold:*

- $(w_i \in \mathit{Var} \wedge t_i \in \mathit{Var}) \quad \Rightarrow \quad t_i = w_i$
- $w_i \in \mathit{Meta} \quad \Rightarrow \quad t_i \in \mathit{Rhs}$

We define a function $\mathcal{I}? : \mathit{Rhs} \times \mathit{Ndef} \to \mathit{Boolean}$ such that $\mathcal{I}?(\langle Lr \rangle, l \overset{df}{\equiv} r)$ is *true* if $\langle Lr \rangle$ is an instance of $l \overset{df}{\equiv} r$, and *false* otherwise.

Definition 2 (Instance Substitution) *Given* $[l \overset{df}{\equiv} r]$, *where* $l \equiv w_1 \ldots w_n$ *and* $\langle Lr \rangle \equiv \langle t_1 \ldots t_m \rangle$ *an instance of* $[l \overset{df}{\equiv} r]$, *we define the* instance substitution s *as follows:*

$$s = \{w_i \leftarrow t_i \;\; s.t. \;\; w_i \in \mathit{Meta}\}$$

The instance substitution s relates the metavariables in l to the corresponding expressions in $\langle Lr \rangle$. We also write instance substitutions as $\mathcal{S}(\langle Lr \rangle, l \overset{df}{\equiv} r)$. Given an instance substitution s, the expression $\{\mathsf{a} \leftarrow r, s\}$ denotes the substitution $\{\mathsf{a} \leftarrow r\} \cup s$.

3.4 Operational semantics

The operational semantics of macro expansion is given by a function *expan* that maps correct programs to Λ_n–terms. A correct program is an expression $D\ r \in \mathit{Pgm}$, where D is a list of notational definitions, with the restrictions mentioned in subsection 3.2, and r is an expression where every $\langle Lr \rangle$–subexpression is an instance of a definition in D; this is:

$$\forall \, \langle Lr \rangle \text{ appearing in } r \;\; (\exists \, d \in D \text{ s.t. } \mathcal{I}?(\langle Lr \rangle, d))$$

The *expan* function is defined through a rewriting system $\Rightarrow_E$ as follows:

$$\mathit{expan}(D\ r) = e \qquad \text{iff} \qquad D\ r \Rightarrow_E^* D\ e \quad \text{and} \quad e \text{ is an } E\text{–normal form}$$

where $\Rightarrow_E^*$ is the reflexive transitive closure of relation $\Rightarrow_E$, defined in Figure 2. An expression e is an *E–normal–form* if there does not exist e' such that $e \Rightarrow_E e'$ [2]. For a general introduction to rewriting systems and their properties see [12, 2, 13].

As we show later, the $\Rightarrow_E$ relation is Church–Rosser; this is:

$$\forall \; xy.(\exists z. \; z \Rightarrow_E^* x \;\wedge\; z \Rightarrow_E^* y) \Rightarrow (\exists u. \; x \Rightarrow_E^* u \;\wedge\; y \Rightarrow_E^* u)$$

and noetherian (there is no infinite sequence $e_1 \Rightarrow_E e_2 \Rightarrow_E \ldots \Rightarrow_E e_n \Rightarrow_E \ldots$). Thus, *expan* is a total function. We also prove that the set of E–normal forms is the language Λ_n; therefore, *expan* expands all macro instances in the program.

For defining the $\Rightarrow_E$ relation, we extend the syntax of language $\mathcal{L}$ to manipulate substitutions explicitly. The extended syntax is in Figure 2.

In the rules, the set D of notational definitions of a program is implicit; so, we write $r \Rightarrow_E r'$ for the rule $D\ r \Rightarrow_E D\ r'$. The set of rewriting rules consists of three groups: the substitution introduction rule, substitution elimination rules, and the rules for the shift operation on substitutions and expressions.

Rule *Intro* formalizes the expansion of macro instances using Definitions 1 and 2 and explicit substitutions.

Among the substitution elimination rules, rules $E_{\lambda var}$ to $E_{\lambda mv2}$ explain how substitutions operate on λ–abstractions and deserve detailed explanation. In general, the substitution *enters* the scope of the identifier (binding instance) bound by the λ operator. There are three different cases. First, when the binding instance is a variable x, the exponents of all free occurrences of x in substitution s must be updated since their distance from their corresponding binding instance is incremented by 1; operation $\uparrow x^{0,0,f}$ on substitution s performs this updating. Second, when the binding instance is a metavariable a that appears in s associated to a variable x^n, variable x must take the place of a and become a binding instance. All occurrences of x^n in the substitution must be captured because they appear replacing other metavariables of the same macro. In this case, the use of the same names is *not* a coincidence but indicates the intention of capturing according to the macro definition. On the other hand, all occurrences of x^n in the body of the abstraction are updated as in the first case because they appear in the definition of the macro and do not relate to the expressions that instantiate the metavariables. Other occurrences of x (with a different exponent) in the substitution are also updated as in the first case. Operation $\uparrow x^{0,0,f}$ performs this updating on expressions. Afterwards, every occurrence of a in r must be replaced by x. Third, when the binding instance is a metavariable that rewrites to another metavariable, no updating is needed. This latter case occurs when instances of previously defined macros appear in right-hand-sides of macro definitions.

The shift operation on expressions, $r\uparrow x^{m,m',b}$, means: *rename all free occurrences of x^n in r by x^{n+1}, when $n \geq m$ and $b = f$, and by $x^{m'}$ when $m = n$ and $b = t$.* Rules Ss_{mt} to Se_{app} give the semantics of operator $\uparrow$ on substitutions and expressions.

3.5 Properties of the semantics

We only state the properties; detailed proofs are in Appendix A.

Theorem 1 (Noetherian) *The relation $\Rightarrow_E$ on correct programs is noetherian.*

Theorem 2 (Church–Rosser) *The relation $\Rightarrow_E$ on correct programs is Church–Rosser.*

Theorem 3 (Normal forms = Λ_n) *The set of E–normal–forms of correct programs is language Λ_n.*

3.6 Examples

We present two examples that illustrate the hygienic [7] manipulation of identifiers of our semantics. More examples can be found in [3].

Example 1. Use of locally defined identifiers in a macro definition.

We define a macro *begin* as follows:

$$[\mathit{begin}\ \ \mathsf{a}\ \ \mathsf{b}\ \overset{df}{\equiv}\ ((\lambda x.\mathsf{b})\ \ \mathsf{a})]$$

The instance $\langle \mathit{begin}\ \ y^0\ \ x^0\rangle$ expands to $((\lambda x.x^1)\ \ y^0)$ as follows:

$$
\begin{array}{lll}
\langle begin\ \ y^0\ \ x^0\rangle & \overset{(Intro)}{\Rightarrow_E} & ((\lambda x.\mathsf{b})\ \ \mathsf{a})[\{\mathsf{a} \leftarrow y^0, \{\mathsf{b} \leftarrow x^0, \phi\}\}] \\
& \overset{(E_{app})}{\Rightarrow_E} & ((\lambda x.\mathsf{b})[\{\mathsf{a} \leftarrow y^0, \{\mathsf{b} \leftarrow x^0, \phi\}\}]\ \ \mathsf{a}[\{\mathsf{a} \leftarrow y^0, \{\mathsf{b} \leftarrow x^0, \phi\}\}]) \\
& \overset{(E_{\lambda var})}{\Rightarrow_E} & ((\lambda x.\mathsf{b}[\{\mathsf{a} \leftarrow y^0, \{\mathsf{b} \leftarrow x^0, \phi\}\}\uparrow x^{0,0,f}])\ \ \mathsf{a}[\{\mathsf{a} \leftarrow y^0, \{\mathsf{b} \leftarrow x^0, \phi\}\}]) \\
& \overset{(E_{mv1})}{\Rightarrow_E} & ((\lambda x.\mathsf{b}[\{\mathsf{a} \leftarrow y^0, \{\mathsf{b} \leftarrow x^0, \phi\}\}\uparrow x^{0,0,f}])\ \ y^0) \\
& \overset{(Ss)}{\Rightarrow_E} & ((\lambda x.\mathsf{b}[\{\mathsf{a} \leftarrow y^0\uparrow x^{0,0,f}, \{\mathsf{b} \leftarrow x^0, \phi\}\uparrow x^{0,0,f}\}])\ \ y^0) \\
& \overset{(Ss)}{\Rightarrow_E} & ((\lambda x.\mathsf{b}[\{\mathsf{a} \leftarrow y^0\uparrow x^{0,0,f}, \{\mathsf{b} \leftarrow x^0\uparrow x^{0,0,f}, \phi\uparrow x^{0,0,f}\}\}])\ \ y^0) \\
& \overset{(E_{mv2})}{\Rightarrow_E} & ((\lambda x.\mathsf{b}[\{\mathsf{b} \leftarrow x^0\uparrow x^{0,0,f}, \phi\uparrow x^{0,0,f}\}])\ \ y^0) \\
& \overset{(E_{mv1})}{\Rightarrow_E} & ((\lambda x.x^0\uparrow x^{0,0,f})\ \ y^0) \\
& \overset{(Se_{var2})}{\Rightarrow_E} & ((\lambda x.x^1)\ \ y^0)
\end{array}
$$

Example 2. Use of global identifiers in a macro definition.
We define a macro *curryf* as follows:

$$[curryf\ \ \mathsf{a}\ \ \mathsf{b}\ \ \overset{df}{\equiv}\ \ ((f^0\ \ \mathsf{a})\ \ \mathsf{b})]$$

The instance $\langle curryf\ \ f^0\ \ x^0\rangle$ expands to $((f^0\ \ f^0)\ \ x^0)$ as follows:

$$
\begin{array}{lll}
\langle curryf\ \ f^0\ \ x^0\rangle & \overset{(Intro)}{\Rightarrow_E} & ((f^0\ \ \mathsf{a})\ \ \mathsf{b})[\{\mathsf{a} \leftarrow f^0, \{\mathsf{b} \leftarrow x^0, \phi\}\}] \\
& \overset{(E_{app})}{\Rightarrow_E} & ((f^0\ \ \mathsf{a})[\{\mathsf{a} \leftarrow f^0, \{\mathsf{b} \leftarrow x^0, \phi\}\}]\ \ \mathsf{b}[\{\mathsf{a} \leftarrow f^0, \{\mathsf{b} \leftarrow x^0, \phi\}\}]) \\
& \overset{(E_{mv2})}{\Rightarrow_E} & ((f^0\ \ \mathsf{a})[\{\mathsf{a} \leftarrow f^0, \{\mathsf{b} \leftarrow x^0, \phi\}\}]\ \ \mathsf{b}[\{\mathsf{b} \leftarrow x^0, \phi\}]) \\
& \overset{(E_{mv1})}{\Rightarrow_E} & ((f^0\ \ \mathsf{a})[\{\mathsf{a} \leftarrow f^0, \{\mathsf{b} \leftarrow x^0, \phi\}\}]\ \ x^0) \\
& \overset{(E_{app})}{\Rightarrow_E} & ((f^0[\{\mathsf{a} \leftarrow f^0, \{\mathsf{b} \leftarrow x^0, \phi\}\}]\ \ \mathsf{a}[\{\mathsf{a} \leftarrow f^0, \{\mathsf{b} \leftarrow x^0, \phi\}\}])\ \ x^0) \\
& \overset{(E_{var})}{\Rightarrow_E} & ((f^0\ \ \mathsf{a}[\{\mathsf{a} \leftarrow f^0, \{\mathsf{b} \leftarrow x^0, \phi\}\}])\ \ x^0) \\
& \overset{(E_{mv1})}{\Rightarrow_E} & ((f^0\ \ f^0)\ \ x^0)
\end{array}
$$

4 Mixing expansion and reduction

Traditionally macro expansion is completely done before evaluation. We explore the possibility of *mixing* expansion and evaluation by adding the β–rule to relation $\Rightarrow_E$.

Syntax:

$$r \;::=\; num \mid x^n \mid \mathsf{a} \mid (\lambda x.r) \mid (\lambda \mathsf{a}.r) \mid (r\;\;r) \mid \langle Lr\rangle \mid r[s] \mid r{\uparrow}x^{m,m',b} \qquad \mathit{Exprs}$$

$$s \;::=\; \phi \mid \{\mathsf{a} \leftarrow r, s\} \mid s{\uparrow}x^{m,b} \qquad \mathit{Subst}$$

Rules:

Substitution Introduction:

$$\langle Lr\rangle \Rightarrow_E r[s], \text{ if } \exists\; d = \; l \overset{df}{\equiv} r \;\in D \;\text{ such that } \mathcal{I}?(\langle Lr\rangle, d) \;\text{ and }\; s = \mathcal{S}(\langle Lr\rangle, d) \qquad (\mathit{Intro})$$

Substitution Elimination:

$$\begin{array}{rll}
num[s] & \Rightarrow_E \; num & (E_{num}) \\
x^n[s] & \Rightarrow_E \; x^n & (E_{var}) \\
\mathsf{a}[\{\mathsf{a} \leftarrow r, s\}] & \Rightarrow_E \; r & (E_{mv1}) \\
\mathsf{b}[\{\mathsf{a} \leftarrow r, s\}] & \Rightarrow_E \; \mathsf{b}[s], \text{ if } \mathsf{a} \neq \mathsf{b} & (E_{mv2}) \\
(\lambda x.r)[s] & \Rightarrow_E \; (\lambda x.r[s{\uparrow}x^{0,f}]) & (E_{\lambda var}) \\
(\lambda \mathsf{a}.r)[s] & \Rightarrow_E \; (\lambda x.(r{\uparrow}x^{0,0,f})[\{\mathsf{a} \leftarrow x, s{\uparrow}x^{n,t}\}]), & \\
 & \qquad\qquad\qquad \text{if } \mathsf{a}[s] \Rightarrow_E^* x^n & (E_{\lambda mv1}) \\
(\lambda \mathsf{a}.r)[s] & \Rightarrow_E \; (\lambda \mathsf{b}.r[s]), \qquad \text{if } \mathsf{a}[s] \Rightarrow_E^* \mathsf{b} & (E_{\lambda mv2}) \\
(r_1\;\;r_2)[s] & \Rightarrow_E \; (r_1[s]\;\;r_2[s]) & (E_{app})
\end{array}$$

Shift operator:

$$\begin{array}{rll}
\phi{\uparrow}x^{m,b} & \Rightarrow_E \; \phi & (Ss_{mt}) \\
\{\mathsf{a} \leftarrow r, s\}{\uparrow}x^{m,b} & \Rightarrow_E \; \{\mathsf{a} \leftarrow r{\uparrow}x^{m,0,b}, s{\uparrow}x^{m,b}\} & (Ss) \\
num{\uparrow}x^{m,m',b} & \Rightarrow_E \; num & (Se_{num}) \\
x^n{\uparrow}x^{m,m',b} & \Rightarrow_E \; x^n, \text{ if } n < m & (Se_{var1}) \\
x^n{\uparrow}x^{m,m',b} & \Rightarrow_E \; x^{n+1}, \text{ if } n > m & (Se_{var2}) \\
x^n{\uparrow}x^{n,m',t} & \Rightarrow_E \; x^{m'} & (Se_{var3}) \\
x^n{\uparrow}x^{n,m',f} & \Rightarrow_E \; x^{n+1} & (Se_{var4}) \\
y^n{\uparrow}x^{m,m',b} & \Rightarrow_E \; y^n, \text{ if } x \neq y & (Se_{var5}) \\
\mathsf{a}{\uparrow}x^{m,m',b} & \Rightarrow_E \; \mathsf{a} & (Se_{mv}) \\
(\lambda x.r){\uparrow}x^{m,m',b} & \Rightarrow_E \; (\lambda x.r{\uparrow}x^{m+1,m'+1,b}) & (Se_{\lambda var1}) \\
(\lambda y.r){\uparrow}x^{m,m',b} & \Rightarrow_E \; (\lambda y.r{\uparrow}x^{m,m',b}), \text{ if } x \neq y & (Se_{\lambda var2}) \\
(\lambda \mathsf{a}.r){\uparrow}x^{m,m',b} & \Rightarrow_E \; (\lambda \mathsf{a}.r{\uparrow}x^{m,m',b}) & (Se_{\lambda mv}) \\
(r_1\;\;r_2){\uparrow}x^{m,m',b} & \Rightarrow_E \; (r_1{\uparrow}x^{m,m',b}\;\;r_2{\uparrow}x^{m,m',b}) & (Se_{app})
\end{array}$$

Figure 2: Expansion System

Syntax:

$$\begin{array}{rcll} r & ::= & num \mid x^n \mid \mathsf{a} \mid (\lambda x.r) \mid (\lambda \mathsf{a}.r) \mid (r\ \ r) \mid \langle Lr \rangle \mid & \\ & & r[s] \mid r[t] \mid r{\uparrow}x^{m,m',b} \mid r{\downarrow}x^n & VExprs \\ s & ::= & \phi \mid \{\mathsf{a} \leftarrow r, s\} \mid s{\uparrow}x^{m,b} & Subst \\ t & ::= & \phi_t \mid \{x^n \leftarrow r, t\} \mid t{\uparrow}x & VarSubst \end{array}$$

$$\Rightarrow_{ER}\ = \Rightarrow_E \cup \Rightarrow_R$$

Rules:

VarSubstitution Introduction:

$$((\lambda x.r_1)\ \ r_2) \Rightarrow_R (r_1[\{x \leftarrow r_2, \phi_t\}]){\downarrow}x^0 \qquad (\beta)$$

Unshift Eliminiation:

$$\begin{array}{rcll} num{\downarrow}x^m & \Rightarrow_R & num & (Ue_{num}) \\ x^n{\downarrow}x^m & \Rightarrow_R & x^n, \text{ if } n < m & (Ue_{var1}) \\ x^n{\downarrow}x^m & \Rightarrow_R & x^{n-1}, \text{ if } n \geq m & (Ue_{var2}) \\ y^n{\downarrow}x^m & \Rightarrow_R & y^n, \text{ if } x \neq y & (Ue_{var3}) \\ \mathsf{a}{\downarrow}x^m & \Rightarrow_R & \mathsf{a} & (Ue_{mv}) \\ (\lambda x.r){\downarrow}x^m & \Rightarrow_R & (\lambda x.r{\downarrow}x^{m+1}) & (Ue_{\lambda var1}) \\ (\lambda y.r){\downarrow}x^m & \Rightarrow_R & (\lambda y.r{\downarrow}x^m), \text{ if } x \neq y & (Ue_{\lambda var2}) \\ (\lambda \mathsf{a}.r){\downarrow}x^m & \Rightarrow_R & (\lambda \mathsf{a}.r{\downarrow}x^m) & (Ue_{\lambda mv}) \\ (r_1\ \ r_2){\downarrow}x^m & \Rightarrow_R & (r_1{\downarrow}x^m\ \ r_2{\downarrow}x^m) & (Ue_{app}) \end{array}$$

VarSubstitution Elimination:

$$\begin{array}{rcll} num[t] & \Rightarrow_R & num & (V_{num}) \\ x^n[\phi_t] & \Rightarrow_R & x^n & (V_{var1}) \\ x^n[\{x^n \leftarrow r, t\}] & \Rightarrow_R & r & (V_{var2}) \\ x^n[\{x^m \leftarrow r, t\}] & \Rightarrow_R & x^n[t], \text{ if } n \neq m & (V_{var3}) \\ x^n[\{y^n \leftarrow r, t\}] & \Rightarrow_R & x^n[t], \text{ if } x \neq y & (V_{var4}) \\ (\lambda x.r)[t] & \Rightarrow_R & (\lambda x.r[t{\uparrow}x]) & (V_{\lambda var}) \\ (r_1\ \ r_2)[t] & \Rightarrow_R & (r_1[t]\ \ r_2[t]) & (V_{app}) \end{array}$$

Shift operator on VarSubstitutions:

$$\begin{array}{rcll} \phi_t{\uparrow}x & \Rightarrow_R & \phi_t & (Sv_{mt}) \\ \{x^m \leftarrow r, t\}{\uparrow}x & \Rightarrow_R & \{x^{m+1} \leftarrow r{\uparrow}x^{0,0,f}, t{\uparrow}x\} & (Sv_1) \\ \{y^m \leftarrow r, t\}{\uparrow}x & \Rightarrow_R & \{y^m \leftarrow r{\uparrow}x^{0,0,f}, t{\uparrow}x\} & (Sv_2) \end{array}$$

Figure 3: Expansion and Reduction System

$$
\begin{array}{ll}
 & [\mathit{let}\ \mathsf{x}\ \mathit{be}\ \mathsf{v}\ \mathit{in}\ \mathsf{B} \overset{df}{\equiv} ((\lambda\mathsf{x}.\mathsf{B})\ \ \mathsf{v})]\ \langle \mathit{let}\ x\ \mathit{be}\ 1\ \mathit{in}\ ((\lambda y.y)\ \ x)\rangle \qquad \in \mathit{Pgm} \\
\overset{(Intro)}{\Rightarrow_{ER}} & ((\lambda\mathsf{x}.\mathsf{B})\ \ \mathsf{v})[\{\mathsf{x} \leftarrow x, \{\mathsf{v} \leftarrow 1, \{\mathsf{B} \leftarrow (\lambda y.y), \phi\}\}\}] \qquad \text{Not. def. omited} \\
\overset{(E_{app})}{\Rightarrow_{ER}} & ((\lambda\mathsf{x}.\mathsf{B})[s]\ \ \mathsf{v}[s]) \\
 & \text{where } s = \{\mathsf{x} \leftarrow x, \{\mathsf{v} \leftarrow 1, \{\mathsf{B} \leftarrow (\lambda y.y), \phi\}\}\} \\
\overset{(E_{mv2})}{\Rightarrow_{ER}} & ((\lambda\mathsf{x}.\mathsf{B})[s]\ \ \mathsf{v}[\{\mathsf{v} \leftarrow 1, \{\mathsf{B} \leftarrow ((\lambda y.y)\ \ x), \phi\}\}]) \\
\overset{(E_{mv1})}{\Rightarrow_{ER}} & ((\lambda\mathsf{x}.\mathsf{B})[s]\ \ 1) \\
\overset{(E_{\lambda mv1})}{\Rightarrow_{ER}} & ((\lambda x.(\mathsf{B}{\uparrow}x^{0,0,f})[\{\mathsf{x} \leftarrow x, s{\uparrow}x^{0,0,t}\}])\ \ 1) \\
\overset{(Se_{mv})}{\Rightarrow_{ER}} & ((\lambda x.\mathsf{B}[\{\mathsf{x} \leftarrow x, s{\uparrow}x^{0,0,t}\}])\ \ 1) \\
\overset{(E_{mv2})}{\Rightarrow_{ER}} & ((\lambda x.\mathsf{B}[s{\uparrow}x^{0,0,t}])\ \ 1) \\
\overset{(Ss)}{\Rightarrow_{ER}} & ((\lambda x.\mathsf{B}[\{\mathsf{x} \leftarrow x{\uparrow}x^{0,0,t}, \{\mathsf{v} \leftarrow 1, \{\mathsf{B} \leftarrow ((\lambda y.y)\ \ x), \phi\}\}{\uparrow}x^{0,0,t}\}])\ \ 1) \\
\overset{(E_{mv2})}{\Rightarrow_{ER}} & ((\lambda x.\mathsf{B}[\{\mathsf{v} \leftarrow 1, \{\mathsf{B} \leftarrow ((\lambda y.y)\ \ x), \phi\}\}{\uparrow}x^{0,0,t}])\ \ 1) \\
\overset{(SSsu)}{\Rightarrow_{ER}} & ((\lambda x.\mathsf{B}[\{\mathsf{v} \leftarrow 1{\uparrow}x^{0,0,t}, \{\mathsf{B} \leftarrow ((\lambda y.y)\ \ x), \phi\}{\uparrow}x^{0,0,t}\}])\ \ 1) \\
\overset{(E_{mv2})}{\Rightarrow_{ER}} & ((\lambda x.\mathsf{B}[\{\mathsf{B} \leftarrow ((\lambda y.y)\ \ x), \phi\}{\uparrow}x^{0,0,t}])\ \ 1) \\
\overset{(Ss)}{\Rightarrow_{ER}} & ((\lambda x.\mathsf{B}[\{\mathsf{B} \leftarrow ((\lambda y.y)\ \ x){\uparrow}x^{0,0,t}, \phi{\uparrow}x^{0,0,t}\}])\ \ 1) \\
\overset{(E_{mv1})}{\Rightarrow_{ER}} & ((\lambda x.((\lambda y.y)\ \ x){\uparrow}x^{0,0,t})\ \ 1) \\
\overset{(Se_{app})}{\Rightarrow_{ER}} & ((\lambda x.((\lambda y.y){\uparrow}x^{0,0,t}\ \ x{\uparrow}x^{0,0,t}))\ \ 1) \\
\overset{(Se_{\lambda var2})}{\Rightarrow_{ER}} & ((\lambda x.((\lambda y.y{\uparrow}x^{0,0,t})\ \ x{\uparrow}x^{0,0,t}))\ \ 1) \\
\overset{(Se_{var5})}{\Rightarrow_{ER}} & ((\lambda x.((\lambda y.y)\ \ x{\uparrow}x^{0,0,t}))\ \ 1) \\
\overset{(Se_{var4})}{\Rightarrow_{ER}} & ((\lambda x.((\lambda y.y)\ \ x))\ \ 1) \\
\overset{(\beta)}{\Rightarrow_{ER}} & (((\lambda y.y)\ \ x)[\{x \leftarrow 1, \phi_t\}]){\downarrow}x^0 \\
\Rightarrow_{ER} & \ldots \quad \text{continues}
\end{array}
$$

Figure 4: Evaluation of a program

4.1 Adding the β–rule

Adding the β–rule to relation $\Rightarrow_E$ is not trivial because rules for manipulating new operations are needed.

The inclusion of the β–rule adds a new type of substitutions to the system (also existing in the $\lambda\sigma$–calculus): substitutions of variables by expressions. The elimination rules for these substitutions force the definition of a new renaming operation analogous to shift ($\uparrow$). When

$$
\begin{array}{ll}
\ldots & (((\lambda y.y)\ \ x)[\{x \leftarrow 1, \phi_t\}])\downarrow x^0 \qquad\qquad \text{(cont.)} \\
\overset{(V_{app})}{\Rightarrow_{ER}} & ((\lambda y.y)[\{x \leftarrow 1, \phi_t\}]\ \ x[\{x \leftarrow 1, \phi_t\}])\downarrow x^0 \\
\overset{(V_{var2})}{\Rightarrow_{ER}} & ((\lambda y.y)[\{x \leftarrow 1, \phi_t\}]\ \ 1)\downarrow x^0 \\
\overset{(V_{\lambda var})}{\Rightarrow_{ER}} & ((\lambda y.y[\{x \leftarrow 1, \phi_t\}\uparrow y])\ \ 1)\downarrow x^0 \\
\overset{(Sv_2)}{\Rightarrow_{ER}} & ((\lambda y.y[\{x \leftarrow 1\uparrow y^{0,0,f}, \phi_t\uparrow y\}])\ \ 1)\downarrow x^0 \\
\overset{(V_{var4})}{\Rightarrow_{ER}} & ((\lambda y.y[\phi_t\uparrow y])\ \ 1)\downarrow x^0 \\
\overset{(Sv_{mt})}{\Rightarrow_{ER}} & ((\lambda y.y[\phi_t])\ \ 1)\downarrow x^0 \\
\overset{(V_{var1})}{\Rightarrow_{ER}} & ((\lambda y.y)\ \ 1)\downarrow x^0 \\
\overset{(Ue_{app})}{\Rightarrow_{ER}} & ((\lambda y.y)\downarrow x^0\ \ 1\downarrow x^0) \\
\overset{(Ue_{\lambda var2})}{\Rightarrow_{ER}} & ((\lambda y.y\downarrow x^0)\ \ 1\downarrow x^0) \\
\overset{(Ue_{num})}{\Rightarrow_{ER}} & ((\lambda y.y\downarrow x^0)\ \ 1) \\
\overset{(Ue_{var3})}{\Rightarrow_{ER}} & ((\lambda y.y)\ \ 1) \\
\overset{(\beta)}{\Rightarrow_{ER}} & (y[\{y \leftarrow 1, \phi_t\}])\downarrow y^0 \\
\overset{(V_{var2})}{\Rightarrow_{ER}} & 1\downarrow y^0 \\
\overset{(Ue_{num})}{\Rightarrow_{ER}} & 1
\end{array}
$$

Figure 5: Evaluation of a program, continuation

a β–reduction is done, a λ operator disappears; therefore, the exponent of identifiers must be updated accordingly. The renaming unshift operator ($\downarrow$) performs this updating, decreasing the exponent of variables. The expression $r\downarrow x^m$ means *rename every free occurrence of x^n in r by x^{n-1} when $n \geq m$.* Specifically, the β–rule in the extended system ($\Rightarrow_{ER}$) is:

$$((\lambda x.r_1)\ \ r_2) \Rightarrow_{ER} (r_1[\{x \leftarrow r_2, \phi\}])\downarrow x^0$$

The complete set of rules is in Figure 3. An example is in Figure 4.

The new system, which naturally mixes evaluation and expansion, has two advantages over the traditional systems that expand macros and evaluate the expanded code in two different passes. First, our system evaluates the program in one pass obtaining the same result. This equivalence can be stated:

$$D\ p \Rightarrow_E^* e \ \text{ and }\ e \overset{*}{\rightarrow}_\beta v \quad \Leftrightarrow \quad D\ p \Rightarrow_{ER}^* v$$

where $\rightarrow_\beta$ is the relation that defines the semantics of Λ_n. Second, the expansion of some instances may be avoided by defining a strategy of rewriting where β–rules are applied first.

The following example illustrates this optimization [3]:

$$\begin{array}{ll}
 & (\lambda y.((\lambda x.y)\ \langle let\ \ u\ \ be\ \ (\lambda w.w)\ \ in\ \ (u\ \ y)\rangle)) \\
\overset{\beta}{\Rightarrow}_{ER} & (\lambda y.(y[\{x \leftarrow \langle let\ \ u\ \ be\ \ (\lambda w.w)\ \ in\ \ (u\ \ y)\rangle, \phi\}])\downarrow x^0) \\
\overset{V_{var4}}{\Rightarrow}_{ER} & (\lambda y.(y[\phi])\downarrow x^0) \\
\overset{V_{var1}}{\Rightarrow}_{ER} & (\lambda y.y\downarrow x^0) \\
\overset{V_{var3}}{\Rightarrow}_{ER} & (\lambda y.y)
\end{array}$$

Here, the expansion of *let* is avoided.

4.2 Properties of the extended system

The set of $\Rightarrow_{ER}$–rules is not noetherian, since $\rightarrow_\beta$ is not noetherian in Λ_n. But it is Church–Rosser. Church–Rosserness follows as a corollary of strong confluence. The detailed proof is in Appendix A.

Lemma 1 (Strong Confluence) *The relation $\Rightarrow_{ER}$ on correct programs is strongly confluent:*

$$\forall\ xyz.(x \Rightarrow_{ER} y\ \wedge\ x \Rightarrow_{ER} z) \Rightarrow \exists u.(y \overset{*}{\Rightarrow}_{ER} u\ \wedge\ z \overset{\epsilon}{\Rightarrow}_{ER} u)$$

where the notation $\overset{\epsilon}{\Rightarrow}_{ER}$ means *zero or one* steps of reduction.

Theorem 4 (Church–Rosser) *The relation $\Rightarrow_{ER}$ on correct programs is Church–Rosser.*

5 Conclusions and related work

We present a language $\mathcal{L}$ for defining macros over the λ–calculus with numerical constants and labeled variables (Λ_n). A formal semantics of a macro processor, which maps $\mathcal{L}$ into Λ_n, is defined using explicit substitutions. A rewriting semantics of $\mathcal{L}$ results from mixing expansion of macros and evaluation. This semantics has three advantages. First, it is an *uniform* framework for β–reduction and expansion of macro instances. Second, it is based on a well–known theory: the λ–calculus. Third, it intuitive and closely models the practice of macro expansion.

We base our macro definition language on the work by Kohlbecker [7, 15, 16]. It preserves the expression structure of the core language, as does Kohlbecker's, but it does not include ellipsis or recursive macro definitions.

Work has been done in designing powerful tools for defining macros and efficient algorithms for their expansion [5, 6, 7, 15, 16]. These works provide a good diagnosis on the problems that must be solved at macro expansion. Kohlbecker defines hygiene in macro expansion; that is, locally introduced identifiers should not been captured by expansion, and identifiers global to the macro definition should not be renamed. Our systems are hygienic according to these criteria. Clinger et al [5] developed an efficient and hygienic expansion algorithm for macro expansion. Their algorithm also solves local macros, a case that we do not consider. However, whereas Clinger's algorithm renames all variables bound by procedure abstractions, our semantics keeps the same variable names in all cases, because our renaming involves only the exponents. This fact corresponds to one of the principles of notational definitions defined by Griffin [9, 10] that says that *names should be preserved.*

Our formal semantics of $\mathcal{L}$ is based on the $\lambda\sigma$–calculus of Abadi et al. Two types of substitutions interact in the calculus: reduction substitutions (introduced by the application of the β–rule), and expansion substitutions (introduced by the application of the *Intro*–rule). We adapt Abadi's shift operator to work with names of identifiers instead of de Bruijn notation and we add rules for eliminating shift operators. Hardin et al [19] also eliminate them, although they use de Bruijn notation.

Formal semantics of macro expansion has been studied in depth by Kohlbecker [15] and Griffin [9, 10]. Kohlbecker uses a denotational framework. Our work is closer to Griffin's. He uses an extended typed λ-calculus to encode a language comparable to our language $\mathcal{L}$. Notational definitions are encoded as new functional constants, that when applied produce the expansion of an instance. Thus, the resulting encoding also mixes expansion and evaluation. We explicitly manipulate substitutions to model macro expansion. Our semantics directly and dynamically generates such substitutions from notational definitions, avoiding the need of encoding the language. We believe our calculus that expands and evaluates expressions is closer to practice and implementation. In addition, explicit substitutions can be used for macro expansion on other languages.

We implemented a prototype of the $\Rightarrow_{ER}$–system in Lazy ML where we experimented different evaluation strategies. Our "lazy" prototype delays macro expansion as long as possible. Further work is to be done in sophisticating the macro definition language to include ellipsis, recursive notational definitions (we still need a restriction that guarantee finite expansion), and local macros. Our work provides a good basis for deepen the study of macros in programming languages.

Acknowledgements

Thanks to Erik Crank for helping with the details of formal proofs and reading earlier drafts of this paper, and to Matthias Felleisen for providing helpful comments during the preparation of this work. We also thank Laurence Puel, who verified the proof of noetherianity of the $\Rightarrow_E$ system. We are grateful to Juan Vicente Echagüe, who read in detail complete drafts of this report providing helpful insight into some application and formal aspects of this work.

References

[1] M. Abadi, L. Cardelli, P.L. Curien, and J.J. Lévy. Explicit Substitutions. Technical Report SRC 54, Digital Equipment Corporation, Palo Alto, California, 1990.

[2] H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, Amsterdam, 1981.

[3] A. Bove. Una Semántica de Abreviaciones Sintácticas usando Sustituciones Explícitas. *Proyecto de Grado, Escuela Superior Latinoamericana de Informática*, 1990.

[4] N. De Bruijn. Lambda–calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation. *Indag. Mat.*, 34:381–392, 1972.

[5] W. Clinger and J. Rees. Macros That Work. In *Proc. 18th ACM Symposium on Principles of Programming Languages*, pages 155–162, Orlando, 1991.

[6] K. Dybvig, D. Friedman, and C. Haynes. Expansion–Passing Style: Beyond Conventional Macros. In *ACM Conference on Lisp and Functional Programming*, pages 143–150, 1986.

[7] Kohlbecker E., D.P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161, 1986.

[8] M. Felleisen. On the expressive power of programming languages. In *Proc. 1990 European Symposium on Programming. Neil Jones, Ed. Lecture Notes in Computer Science, 432*, pages 134–151, 1990.

[9] T. Griffin. An Environment for Formal Systems. Technical Report 87-846, Department of Computer Sciene, Cornell University, 1987.

[10] T. Griffin. Notational definition – A formal account. In *Proc. Symp. Logic in Computer Science*, pages 372–383, 1988.

[11] J. R. Hindley and J. P. Seldin. *An Introduction to Combinators and Lambda Calculus.* London Mathematics Society, 1987.

[12] G. Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *CACM*, 27(4):797–821, October 1980.

[13] G. Huet and D. C. Oppen. Equations and Rewrite Rules: A Survey. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems.* Academic Press, 1980.

[14] B. W. Kernigham and D. M. Ritchie. *The C Programming Language.* Prentice–Hall, New Jersey, 1978.

[15] E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, August 1986.

[16] E. Kohlbecker and M. Wand. Macro-by-example: Deriving Syntactic Transformations from their Specifications. In *Proc. 14th ACM Symposium on Principles of Programming Languages*, pages 77–85, 1987.

[17] P. J. Landin. The next 700 programming languages. *CACM*, 9(3):157–166, 1966.

[18] J. Rees and W. Clinger (Eds.). The revised^3 report on the algorithmic language Scheme. In *SIGPLAN Notices*, volume 21 (12), pages 37–79, 1986.

[19] T. Hardin and J.J. Lévy. A Confluent Calculus of Substitutions. In *Japan Artificial Intelligence and Computer Science Symposium*, Izu, December 1989.

A Proofs

Theorem 1 (Noetherian) *The relation $\Rightarrow_E$ on correct programs is noetherian.*

Proof.

Program correctness assures that the *Intro* rule is always applicable in the presence of a macro instance, and that the set of notational definitions follows the restrictions defined in 3.2. We define a positive measure function f on the expressions $r \in \mathit{Exprs}$ such that for every expansion rule $r {\Rightarrow_E}\, r'$, the following condition holds:

$$f(r) \sqsupset f(r')$$

The function f is a pair defined as follows: $f(r) = (f_E(r), f_S(r))$ where the $\sqsupset$ relation is defined as:

$$\begin{aligned}(f_E(r), f_S(r)) \sqsupset (f_E(r'), f_S(r')) \quad \Leftrightarrow \quad & f_E(r) > f_E \\ & \text{or } f_E(r) = f_E(r') \text{ and } f_S(r) > f_S(r')\end{aligned}$$

The definition of the two auxiliary functions is in the following table:

r	$f_E(r)$	$f_S(r)$
$\langle e_1 \ldots e_m \rangle$		
where $\mathcal{I}?(\langle e_1 \ldots e_m \rangle, l \overset{df}{\equiv} r)$	$(\sum_{i=1}^{m} f_E(e_i) + 3) * f_E(r)$	$(\sum_{i=1}^{m} f_S(e_i) + 3) * f_S(r)$
$r[s]$	$f_E(r) * f_E(s)$	$f_S(r) * f_S(r)$
num	2	2
x^n	2	2
a	2	2
$(\lambda x.r)$	$f_E(r) + 2$	$f_S(r) + 2$
$(\lambda \mathsf{a}.r)$	$f_E(r) + 2$	$f_S(r) + 2$
$(r_1 \;\; r_2)$	$f_E(r_1) + f_E(r_2) + 1$	$f_S(r_1) + f_S(r_2) + 1$
$r{\uparrow}x^{m,m',b}$	$f_E(r)$	$2 * f_S(r)$
ϕ	2	2
$\{\mathsf{a} \leftarrow r, s\}$	$f_E(r) + f_E(s)$	$f_S(r) + f_S(s) + 2$
$s{\uparrow}x^{m,m',b}$	$f_E(s)$	$2 * f_S(s)$

Observations:

1. for all r, $f(r) > (0, 0)$.
2. $f(s) \sqsupseteq (2, 2)$, $s \in \mathit{Subst}$.

Rules *Intro*, $E_{\lambda mv1}$, and $E_{\lambda mv2}$ in Figure 2 are in an abbreviated form. For this proof, we write these rules in their extended form:

(Intro)
$$[l_1 \overset{df}{\equiv} r_1] \ldots [l_i \overset{df}{\equiv} r_i] \ldots [l_n \overset{df}{\equiv} r_n] \langle Lr \rangle {\Rightarrow_E}$$
$$r_i[\mathcal{S}(\langle Lr \rangle, l_i \overset{df}{\equiv} r_i)], \qquad \text{if } \mathcal{I}?(\langle Lr \rangle, l_i \overset{df}{\equiv} r_i)$$

$(E_{\lambda mv1})$ $(\lambda \mathsf{a}.r)[\{\mathsf{b}_1 \leftarrow r_1, \ldots \{\mathsf{a} \leftarrow r_a, \ldots \{\mathsf{b}_m \leftarrow r_m, \phi\} \ldots\} \ldots\}] \Rightarrow_E$

$$(\lambda x.r{\uparrow}x^0[\{\mathsf{b}_1 \leftarrow r_1{\uparrow}x^0, \ldots \{\mathsf{a} \leftarrow x, \ldots \{\mathsf{b}_m \leftarrow r_m{\uparrow}x^0, \phi\} \ldots\} \ldots\}]) \text{ where } r_a \overset{(*)}{\Rightarrow}_{ER} x^n$$

$(E_{\lambda mv2})$ $(\lambda \mathsf{a}.r)[\{\mathsf{b}_1 \leftarrow r_1, \ldots \{\mathsf{a} \leftarrow r_a, \ldots \{\mathsf{b}_m \leftarrow r_m, \phi\} \ldots\} \ldots\}] \Rightarrow_E$

$$(\lambda \mathsf{b}.r[\{\mathsf{b}_1 \leftarrow r_1, \ldots \{\mathsf{a} \leftarrow \mathsf{b}, \ldots \{\mathsf{b}_m \leftarrow r_m, \phi\} \ldots\} \ldots\}]) \text{ where } r_a \overset{(*)}{\Rightarrow}_{ER} \mathsf{b}$$

Let an arbitrary expansion rule be of the form $r \Rightarrow_E r'$. It is easy to verify that:

- For the *Intro* and the elimination (E) rules: $f_E(r) > f_E(r')$. For the rest of the rules (S): $f_E(r) = f_E(r')$.
- For the shift (S) rules on expressions and substitutions: $f_S(r) > f_S(r')$.

We present a detailed proof only for the case of *Intro*; the other cases are straight forward. The *Intro* rule is $\langle e_1 \ldots e_m\rangle \Rightarrow_E r[s]$, where $\mathcal{I}?(\langle e_1 \ldots e_m\rangle, l \overset{df}{\equiv} r)$, and $s = \mathcal{S}(\langle e_1 \ldots e_m\rangle, l \overset{df}{\equiv} r)$.

$$f_E(\langle e_1 \ldots e_n\rangle) = (\textstyle\sum_{i=1}^{m} f_E(e_i) + 3) * f_E(r)$$

$$\begin{aligned}
f_E(r[s]) &= f_E(s) * f_E(r) \\
&= f_E(\mathcal{S}(\langle e_1 \ldots e_m\rangle, l \overset{df}{\equiv} r)) * f_E(r) \\
&= f_E([w_{j1} \leftarrow e_{j1}, \ldots, w_{jh} \leftarrow e_{jh}, \phi]) * f_E(r) \quad, h \le n \\
&= (f_E(e_{j1}) + \ldots + f_E(e_{jh}) + f_E(\phi)) * f_E(r) \\
&= (f_E(e_{j1}) + \ldots + f_E(e_{jh}) + 2) * f_E(r) \\
&\le (f_E(e_1) + \ldots + f_E(e_n) + 2) * f_E(r) \\
&< (f_E(e_1) + \ldots + f_E(e_n) + 3) * f_E(r) \\
&= f_E(\langle e_1 \ldots e_n\rangle)
\end{aligned}$$

Thus, for every rule $r \Rightarrow_E r'$, $f(r) \sqsupset f(r')$. Since f is a positive function, the relation defined by the $\Rightarrow_E$–rules is strongly normalizing.
□

Definition 3 (Critical pair) *Let $L_1 \Rightarrow_E R_1$, and $L_2 \Rightarrow_E R_2$ be two rules of $\Rightarrow_E$, and let M be a subterm of L_1 at position u such that M is not a variable[2]. Then, $\langle P, Q\rangle$ is a* critical pair *in $\Rightarrow_E$ iff:*

1. *M and L_2 are unifiable.*
2. *The substitutions used for unification are σ_1, and σ_2 such that the term $N = \sigma_1(M) = \sigma_2(L_2)$ does not have variables in common with L_1.*
3. *$P = \sigma_1(L_1)[u \leftarrow \sigma_2(R_2)]$*

[2]A rule is applied when its left hand side unifies with an expression. In the unification, rule variables match subexpressions of the given expression.

$$\begin{tikzcd} & \sigma_1(L_1) \arrow[dl, "L_1 \Rightarrow_E R_1"'] \arrow[dr, "L_2 \Rightarrow_E R_2"] & \\ Q = \sigma_1(R_1) & & P = \sigma_1(L_1)[u \leftarrow \sigma_2(R_2)] \end{tikzcd}$$

Figure 6: Critical Pairs.

4. $Q = \sigma_1(R_1)$

This concept is best illustrated in the diagram of Figure 6. For a detailed explanation of critical pairs, we refer the reader to [12, 13].

Lemma 1 (Weakly Church Rosser) *The relation $\Rightarrow_E$ on correct programs is weakly Church–Rosser, this is:*

$$\forall\ xyz.(x \Rightarrow_E y \wedge x \Rightarrow_E z) \Rightarrow\ (\exists u.x \overset{*}{\Rightarrow}_E u \wedge y \overset{*}{\Rightarrow}_E u)$$

Proof.

The proof is based on the lemma that says that a term rewriting system with no critical pairs is WCR [12]. Two steps are needed.

First, we must prove that $\Rightarrow_E$ is a term rewriting system. A term rewriting system is a set of rewriting rules such that the application of a rule does not introduce new variables. The rules that may offer doubts are *Intro*, $E_{\lambda_{mv1}}$, and $E_{\lambda_{mv2}}$. Observe their extended forms:

- The *Intro* rule does not introduce new variables since the expression r appears in the set of notational definitions, and the substitution s is built from metavariables appearing in r and expressions appearing in $\langle Lr \rangle$.
- For rule $E_{\lambda_{mv1}}$, expression r_a reduces to x^n. Thus x appears in r_a and n may result from renaming operations.
- For rule $E_{\lambda_{mv2}}$, the metavariable b necessarily appears in r_a.

Second, considering every possible pair of rules of $\Rightarrow_E$, we easily verify that there are no critical pairs in $\Rightarrow_E$.

□

Theorem 2 (Church–Rosser) *The relation $\Rightarrow_E$ on correct programs is Church–Rosser.*

Proof.

This theorem directly follows from Theorem 1 and Lemma 1.

□

Theorem 3 (E–normal forms = Λ_n) *The set of E–normal–forms of correct programs is language Λ_n.*

Proof.

Program correctness assures that every macro instance in the program corresponds to a notational definition appearing in the list of macro declarations.

- ($\Rightarrow$) An E–normal–form of a correct program is in Λ_n.

  Let r be an E–nf of a correct program.

  1. r does not contain macro instances because if it had them, they would be eliminated by the application of the rule *Intro*. If the *Intro* rule were applicable, r would not be an E–nf.
  2. r does not contain substitutions. Substitutions are generated by the *Intro*–rule. Suppose by contradiction that r contains a subexpression of the form $r'[s]$. If a metavariable appears in r', it has a corresponding expression in s because r derives from a correct program and the set of notational definitions is well–defined. It is easy to show, by induction on the structure of r', that s can be eliminated whether or not metavariables appear in r'.
  3. r does not contain $\uparrow$ operators. By induction of the structure of r, we can prove that any $\uparrow$ operator can be eliminated.
  4. r does not contain metavariables because it does not contain substitutions and derives from a correct program.

  Thus, r does not contain metavariables, macro instances, substitutions, and $\uparrow$ operators; r is in Λ_n.

- ($\Leftarrow$) Every expression $e \in \Lambda_n$ is an E–normal–form.

  An expression of Λ_n (see figure 2), never contains a macro instance, a substitution, or a $\uparrow$–operator. Thus, no $\Rightarrow_E$ rule can reduce it.

□

Lemma 2 (Strong confluence) *The relation $\Rightarrow_{ER}$ is strongly confluent.*

Proof.

The proof goes by structural induction on *VExprs*, *Subst*, and *VarSubst*.

We want to prove that:

$$\forall\ rr'r''.(r \Rightarrow_{ER} r' \ \wedge\ r \Rightarrow_{ER} r'') \Rightarrow \exists u_r.(r' \Rightarrow^{*}_{ER} u_r \ \wedge\ r'' \Rightarrow^{\epsilon}_{ER} u_r)$$

Notation: we use r' and r'' for expressions that derive in one step from r, and u_r for the term to which r' and r'' converge (see Figure 7). Rule names may appear in the arrows as well; (A) and (B) range over rule names.

Base Cases.

When an expression is a number, an identifier, a metavariable, or an empty substitution no rule $\Rightarrow_{ER}$ is applicable. Thus, the theorem follows.

For terms and subtitutions listed in the following table, only one rule is applicable. Thus, the theorem follows.

Notation:

$$\begin{tikzcd}
 & r \arrow[dl] \arrow[dr] & \\
r' \arrow[dr, "*"'] & & r'' \arrow[dl, "\epsilon"] \\
 & u_r &
\end{tikzcd}$$

Figure 7: Strong confluence.

Expression	Rule
$\phi\uparrow x^{m,m',b}$	Ss_{mt}
num $\uparrow x^{m,m',b}$	Se_{num}
$x^n \uparrow x^{m,m',b}$	Se_{var1}, Se_{var2}, Se_{var4}, Se_{var3}
$y^n \uparrow x^{m,m',b}$	Se_{var5}
a $\uparrow x^{m,m',b}$	Se_{mv}
num$\downarrow x^m$	Ue_{num}
$x^n\downarrow x^m$	Ue_{var1}, Ue_{var2}
$y^n\downarrow x^m$	Ue_{var3}
a $\downarrow x^m$	Ue_{mv}
$x^n[\phi_t]$	V_{var1}
$\phi_t\uparrow$x	Sv_{mt}

Inductive Cases.

We present only the diagrams for application and macro instances. For the rest of the cases, we reason in an analogous manner. We assume the theorem follows for expressions $r_1 \dots r_n$, and prove the theorem for expressions built up from these.

Application. Case (1):

$$\begin{tikzcd}
 & (r_1 \;\; r_2) \arrow[dl, "(A)"'] \arrow[dr, "(B)"] & \\
(r'_1 \;\; r_2) \arrow[dr, "*"'] & & (r''_1 \;\; r_2) \arrow[dl, "\epsilon"] \\
 & (u_{r_1} \;\; r_2) &
\end{tikzcd}$$

Application. Case (2):

$$\begin{tikzcd}
 & (r_1 \;\; r_2) \arrow[dl, "(A)"'] \arrow[dr, "(B)"] & \\
(r_1 \;\; r'_2) \arrow[dr, "*"'] & & (r_1 \;\; r''_2) \arrow[dl, "\epsilon"] \\
 & (r_1 \;\; u_{r_2}) &
\end{tikzcd}$$

Application. Case (3):

$$\begin{tikzcd}
 & (r_1 \;\; r_2) \arrow[dl, "(A)"'] \arrow[dr, "(B)"] & \\
(r'_1 \;\; r_2) \arrow[dr, "(B)"'] & & (r_1 \;\; r'_2) \arrow[dl, "(A)"] \\
 & (r'_1 \;\; r'_2) &
\end{tikzcd}$$

Application. Case (4), rule β:

$$\begin{tikzcd}
 & ((\lambda x.r_1) \;\; r_2) \arrow[dl, "\beta"'] \arrow[dr, "(A)"] & \\
(r_1[\{x \leftarrow r_2, \phi_t\}]){\downarrow}x^0 \arrow[dr, "(A)"'] & & ((\lambda x.r'_1) \;\; r_2) \arrow[dl, "\beta"] \\
 & (r'_1[\{x \leftarrow r_2, \phi_t\}]){\downarrow}x^0 &
\end{tikzcd}$$

Application. Case (5), rule β:

$$\begin{tikzcd}
 & ((\lambda x.r_1) \;\; r_2) \arrow[dl, "\beta"'] \arrow[dr, "(A)"] & \\
(r_1[\{x \leftarrow r_2, \phi_t\}]){\downarrow}x^0 \arrow[dr, "(A)"'] & & ((\lambda x.r_1) \;\; r'_2) \arrow[dl, "\beta"] \\
 & (r_1[\{x \leftarrow r'_2, \phi_t\}]){\downarrow}x^0 &
\end{tikzcd}$$

Macro instance. Case (1):

$$\begin{tikzcd}
 & \langle r_1 \ldots r_i \ldots r_j \ldots r_n \rangle \arrow[dl, "(A)"'] \arrow[dr, "(B)"] & \\
\langle r_1 \ldots r'_i \ldots r_j \ldots r_n \rangle \arrow[dr, "(B)"'] & & \langle r_1 \ldots r_i \ldots r'_j \ldots r_n \rangle \arrow[dl, "(A)"] \\
 & \langle r_1 \ldots r'_i \ldots r'_j \ldots r_n \rangle &
\end{tikzcd}$$

Macro instance. Case (2):

$$\begin{tikzcd}
 & \langle r_1 \ldots r_i \ldots r_n \rangle \arrow[dl, "(A)"'] \arrow[dr, "(B)"] & \\
\langle r_1 \ldots r'_i \ldots r_n \rangle \arrow[dr, "*"'] & & \langle r_1 \ldots r''_i \ldots r_n \rangle \arrow[dl, "\epsilon"] \\
 & \langle r_1 \ldots u_{r_i} \ldots r_n \rangle &
\end{tikzcd}$$

Macro instance. Case (3), rule *Intro*:

$$\begin{tikzcd}
 & \langle r_1 \ldots r_i \ldots r_n \rangle \arrow[dl, "Intro"'] \arrow[dr, "(A)"] & \\
r[\{\mathsf{x}_1 \leftarrow r_1, \ldots \{\mathsf{x}_i \leftarrow r_i, \ldots \{\mathsf{x}_n \leftarrow r_n, \phi\}\}\}] \arrow[dr, "(A)"] & & \langle r_1 \ldots r'_i \ldots r_n \rangle \arrow[dl, "Intro"] \\
 & r[\{\mathsf{x}_1 \leftarrow r_1, \ldots \{\mathsf{x}_i \leftarrow r'_i, \ldots \{\mathsf{x}_n \leftarrow r_n, \phi\}\}\}] &
\end{tikzcd}$$

□

Theorem 4 (Church–Rosser) *The relation $\Rightarrow_{ER}$ is Church–Rosser.*

Proof.

Directly follows from Lemma 2, since strong confluence implies confluence.

□

B A prototype in lazy ML

A prototype has been implemented in lazy ML. The source code is listed below.

Module "expand.m"

```
module

-- Expansion and evaluation of a program

#include "lenguaje.t"
#include "eliminst.t"

export Expand;

rec

-- Expands and evaluates a program
Expand(program(D,e)) = Expand1(D)(e)(false)

and

-- Expands and evaluates an expression. If the third parameter is true,
-- the function stops at an abstraction of a variable
Expand1(D)(exp)(bool) =
   case exp in
        num(n)            : num(n) ||
        var(x,n)          : var(x,n) ||
        mvar(a)           : mvar(a) ||
        vabs(x,e)         : if bool
                                then vabs(x,e)
                                else vabs(x,Expand1(D)(e)(false)) ||
        mvabs(a,e)        : mvabs(a,e) ||
        apl(e1,e2)        :
            case Expand1(D)(e1)(true) in
                 vabs(x,e) :
                     Expand1(D)(eunshift(tsust(e,ct(tpar(x,0,e2),vt)), x,0))
                                  (bool) ||
                 otherwise : apl(Expand1(D)(e1)(true),Expand1(D)(e2)(false))
            end ||
        sust(e,s)         : Expand1(D)(ElimSust(D)(e)(s))(bool) ||
        tsust(e,s)        : Expand1(D)(ElimTSust(D)(e)(s))(bool) ||
        eshift(e,x,n,m,b) : Expand1(D)(ExpShift(D)(e)(x)(n)(m)(b))(bool) ||
        eunshift(e,x,n)   : Expand1(D)(ExpUnShift(D)(e)(x)(n))(bool) ||
        eerror            : eerror ||
        ins(le)           : Expand1(D)(ElimInst(D)(ins(le)))(bool)
   end

and

-- Elimination of an aplication of a substitution
ElimSust(D)(exp)(st) =
   case exp in
        num(n)            : num(n) ||
        var(x,n)          : var(x,n) ||
        mvar(a)           :
             case st in
                  vs               : eerror ||
                  cs(par(b,e),s)   : if a = b
                                        then e
                                        else sust(mvar(a),s) ||
                  sshift(s,x,n,b) : ElimSust(D)(mvar(a))(SustShift(s)(x)(n)(b))
             end ||
        vabs(x,e)         : vabs(x,sust(e,sshift(st,x,0,false))) ||
        mvabs(a,e)        :
              case Expand1(D)(sust(mvar(a),st))(false) in
                   var(x,n)   : vabs(x,sust(eshift(e,x,0,0,false),
                                   cs(par(a,var(x,0)),sshift(st,x,n,true)))) ||
                   mvar(b)    : mvabs(b,sust(e,st)) ||
                   otherwise : eerror
              end ||
        apl(e1,e2)        : apl(sust(e1,st),sust(e2,st)) ||
        sust(e,s)         : ElimSust(D)(ElimSust(D)(e)(s))(st) ||
        tsust(e,t)        : ElimSust(D)(ElimTSust(D)(e)(t))(st) ||
        eshift(e,x,n,m,b) : ElimSust(D)(ExpShift(D)(e)(x)(n)(m)(b))(st) ||
        eunshift(e,x,n)   : ElimSust(D)(ExpUnShift(D)(e)(x)(n))(st) ||
        eerror            : eerror ||
        ins(le)           : ElimSust(D)(ElimInst(D)(ins(le)))(st)
   end

and

-- Elimination of the shift operator over substitutions
SustShift(sust)(x)(n)(bool) =
   case sust in
        vs          : vs ||
        cs(p,s)     : cs(ParShift(p)(x)(n)(bool),sshift(s,x,n,bool)) ||
        sshift(s,y,m,b) : SustShift(SustShift(s)(y)(m)(b))(x)(n)(bool)
   end

and
```

```
-- Elimination of the shift operator over pairs of metavariables and
-- expressions
ParShift(pair)(x)(n)(bool) =
   case pair in
        par(a,r) : par(a,eshift(r,x,n,0,bool))
   end

and

-- Elimination of the shift operator over expressions
ExpShift(D)(exp)(x)(m)(m1)(bool) =
   case exp in
        num(n)              : num(n) ||
        var(y,n)            : if x = y
                                 then if n < m
                                         then var(x,n)
                                         else if n > m
                                                 then var(x,n+1)
                                                 else if bool
                                                         then var(x,m1)
                                                         else var(x,n+1)
                                 else var(y,n) ||
         mvar(a)            : mvar(a) ||
         vabs(y,e)          : if x = y
                                 then vabs(x,eshift(e,x,m+1,m1+1,bool))
                                 else vabs(y,eshift(e,x,m,m1,bool)) ||
         mvabs(a,e)         : mvabs(a,eshift(e,x,m,m1,bool)) ||
         apl(e1,e2)         : apl(eshift(e1,x,m,m1,bool),eshift(e2,x,m,m1,bool)) ||
         sust(e,s)          : ExpShift(D)(ElimSust(D)(e)(s))(x)(m)(m1)(bool) ||
         tsust(e,t)         : ExpShift(D)(ElimTSust(D)(e)(t))(x)(m)(m1)(bool) ||
         eshift(e,y,n,n1,b) : ExpShift(D)(ExpShift(D)(e)(y)(n)(n1)(b))(x)(m)(m1)(bool) ||
         eunshift(e,y,n)    : ExpShift(D)(ExpUnShift(D)(e)(y)(n))(x)(m)(m1)(bool) ||
         eerror             : eerror ||
         ins(le)            : ExpShift(D)(ElimInst(D)(ins(le)))(x)(m)(m1)(bool)
   end

and

-- Elimination of var-substitutions
ElimTSust(D)(exp)(tst) =
   case exp in
        num(n)            : num(n) ||
        var(x,n)          :
            case tst in
                 vt                 : var(x,n) ||
                 ct(tpar(y,m,e),t) : if x = y
```

```
                                 then if n = m
                                         then e
                                         else tsust(var(x,n),t)
                                 else tsust(var(x,n),t) ||
                 tsshift(t,y)      : ElimTSust(D)(var(x,n))(TSustShift(t)(y))
              end ||
          mvar(a)            : eerror ||
          vabs(x,e)          : vabs(x,tsust(e,tsshift(tst,x))) ||
          mvabs(a,e)         : eerror ||
          apl(e1,e2)         : apl(tsust(e1,tst),tsust(e2,tst)) ||
          sust(e,s)          : ElimTSust(D)(ElimSust(D)(e)(s))(tst) ||
          tsust(e,t)         : ElimTSust(D)(ElimTSust(D)(e)(t))(tst) ||
          eshift(e,x,n,m,b)  : ElimTSust(D)(ExpShift(D)(e)(x)(n)(m)(b))(tst) ||
          eunshift(e,x,n)    : ElimTSust(D)(ExpUnShift(D)(e)(x)(n))(tst) ||
          eerror             : eerror ||
          ins(le)            : ElimTSust(D)(ElimInst(D)(ins(le)))(tst)
   end

and

-- Elimination of shift operator over var-substitutions
TSustShift(tsust)(x) =
   case tsust in
        vt            : vt ||
        ct(p,t)       : ct(ParTShift(p)(x),tsshift(t,x)) ||
        tsshift(t,y)  : TSustShift(TSustShift(t)(y))(x)
   end

and

-- Elimination of the shift operator over pairs of variables and expressions
ParTShift(pair)(x) =
   case pair in
        tpar(y,n,e) : if y = x
                      then tpar(x,n+1,eshift(e,x,0,0,false))
                      else tpar(y,n,eshift(e,x,0,0,false))
   end

and

-- Elimination of the unShift operator over expressions
ExpUnShift(D)(exp)(x)(m) =
   case exp in
        num(n)             : num(n) ||
        var(y,n)           : if x = y
                                then if n < m
```

```
                                        then var(x,n)
                                        else var(x,n-1)
                                  else var(y,n) ||
         mvar(a)             : mvar(a) ||
         vabs(y,e)           : if x = y
                                  then vabs(x,eunshift(e,x,m+1))
                                  else vabs(y,eunshift(e,x,m)) ||
         mvabs(a,e)          : mvabs(a,eunshift(e,x,m)) ||
         apl(e1,e2)          : apl(eunshift(e1,x,m),eunshift(e2,x,m)) ||
         sust(e,s)           : ExpUnShift(D)(ElimSust(D)(e)(s))(x)(m) ||
         tsust(e,t)          : ExpUnShift(D)(ElimTSust(D)(e)(t))(x)(m) ||
         eshift(e,y,n,m,b) : ExpUnShift(D)(ExpShift(D)(e)(y)(n)(m)(b))(x)(m) ||
         eunshift(e,y,n)     : ExpUnShift(D)(ExpUnShift(D)(e)(y)(n))(x)(m) ||
         eerror              : eerror ||
         ins(le)             : ExpUnShift(D)(ElimInst(D)(ins(le)))(x)(m)
   end

end
```

Module “eliminst.m”

```
module

-- Elimination of an instance

#include "lenguaje.t"

export ElimInst;

rec

-- Introduction Rule
ElimInst(ld)(ins(le)) =
   case ld in
        vd : eerror ||
        cd(d,l) : if Inst(ins(le))(d)
                  then CreoSust(ins(le))(d)
                  else ElimInst(l)(ins(le))
   end

and

-- Returns true if the expression is an instance of the declaration,
false otherwise
Inst(ins(le))(dec(lh,e1)) = Inst1(le)(lh)

and

-- Returns true if the list of expressions matches a left hand side,
false otherwise
Inst1(le)(lh) =
   case le in
        vl      : case lh in
                       vlh : true ||
                       otherwise : false
                  end ||
        cl(e,l) : case lh in
                       clh(mv,lh1) : if Match(e)(mv)
                                     then Inst1(l)(lh1)
                                     else false ||
                       otherwise : false
                  end
   end
```

```
and

-- Returns true if the expression and the first parameter are both the
same identifier
-- or if the second parameter is a metavariable, false otherwise
Match(exp)(mv) =
   case mv in
        v(x) : case exp in
                    var(y,0) : if x = y
                                  then true
                                  else false ||
                    otherwise : false
               end ||
        m(x) : true
   end

and

-- Creates the expression that results from the application of the
introduction rule
CreoSust(ins(le))(dec(l,r)) = sust(r,CreoSust1(le)(l))

and

-- Creates the substitution that result from the instance and the lhs
CreoSust1(le)(ld) =
   case le in
        vl      : vs ||
        cl(e,l) : case ld in
                       clh(v(x),ld1) : CreoSust1(l)(ld1) ||
                       clh(m(x),ld1) : cs(par(x,e),CreoSust1(l)(ld1))
                  end
   end

end
```

Module “print.m”

```
module

-- Pretty printer functions

#include "lenguaje.t"

export ProgPrint, EPrint;

rec

-- Programs
ProgPrint(program(ldec,exp)) = LDPrint(ldec) @ EPrint(exp)

and

-- List of declarations
LDPrint(ldec) =
   case ldec in
        vd       : "" ||
        cd(d,ld) : "{" @ DPrint(d) @ "}\n" @ LDPrint(ld)
   end

and

-- Declarations
DPrint(dec(l,r)) = LhsPrint(l) @ " = " @ EPrint(r)

and

-- Left hand sides
LhsPrint(lh) =
   case lh in
        vlh         : "" ||
        clh(v(x),l) : "v(" @ x @ ")," @ LhsPrint(l) ||
        clh(m(x),l) : "m(" @ x @ ")," @ LhsPrint(l)
   end

and

-- Expressions
EPrint(exp) =
   case exp in
        eerror            : "Error" ||
        num(n)             : "n(" @ itos(n) @ ")" ||
        var(x,n)           : "v(" @ x @ "," @ itos(n) @ ")" ||
        mvar(x)            : "mv(" @ x @ ")" ||
        apl(e1,e2)         : "(" @ EPrint(e1) @ " " @ EPrint(e2) @ ")" ||
        vabs(x,e)          : "v(" @ x @ ")." @ EPrint(e) ||
        mvabs(x,e)         : "mv(" @ x @ ")." @ EPrint(e) ||
        sust(e,s)          : "(" @ EPrint(e) @ ")" @ "[" @ SPrint(s) @ "]" ||
        tsust(e,t)         : "(" @ EPrint(e) @ ")" @ "[" @ TSPrint(t) @ "]" ||
        eshift(e,x,n,m,b) : "(" @ EPrint(e) @ "^" @ x @ "," @ itos(n) @ "," @
                                BoolPrint(b) @ ")" ||
        eunshift(e,x,n)    : "(" @ EPrint(e) @ "v" @ x @ "," @ itos(n) @ ")" ||
        ins(le)            : "<" @ LEPrint(le) @ ">"
   end

and

-- List of expressions
LEPrint(listexp) =
   case listexp in
        vl       : "" ||
        cl(e,le) : EPrint(e) @ "," @ LEPrint(le)
   end

and

-- Pairs of metavariables and expressions
PPrint(par(x,e)) = "{mv(" @ x @ ")" @ "," @ EPrint(e) @ "}"

and

-- Substitutions
SPrint(sust) =
   case sust in
        vs               : "" ||
        cs(p,s)          : PPrint(p) @ "," @ SPrint(s) ||
        sshift(s,x,n,b) : "((" @ SPrint(s) @ ")^" @ x @ itos(n) @
BoolPrint(b) @ ")"
   end

and

-- Pairs of variables and expressions
TPPrint(tpar(x,n,e)) = "{v(" @ x @ "," @ itos(n) @ ")" @ "," @ EPrint(e) @ "}"

and
```

```
-- Var-substitutions
TSPrint(tsust) =
   case tsust in
        vt            : "" ||
        ct(tp,ts)     : TPPrint(tp) @ "," @ TSPrint(ts) ||
        tsshift(ts,x) : "((" @ TSPrint(ts) @ ")^" @ x @ ")"
   end

and

-- Booleans
BoolPrint(b) =
   case b in
        true  : "TT" ||
        false : "FF"
   end

end
```

Module “lenguaje.m”

```
module

-- Definition of the language

#define id (List(Char))

export Prog, ListDec, Dec, lhs, MV, Exp, ListExp, Sust, Par, TPar, TSust;

rec

-- Programs
type Prog = program(ListDec # Exp)

and

-- List of declarations
type ListDec = vd +
               cd(Dec # ListDec)

and

-- Declaration or Notational Definition
type Dec = dec(lhs # Exp)

and

-- Left hand side (lhs) of a declaration
type lhs = vlh +
           clh(MV # lhs)

and

-- Auxiliary definition used in lhs
type MV = v(id) +
          m(id)

and

-- Expressions
type Exp = eerror +
           num(Int) +
           var (id # Int) +
           mvar (id) +
           apl (Exp # Exp) +
           vabs (id # Exp) +
           mvabs(id # Exp) +
           sust (Exp # Sust) +
           tsust(Exp # TSust) +
           eshift (Exp # id # Int # Int # Bool) +
           eunshift(Exp # id # Int) +
           ins(ListExp)

and

-- List of expressions used in instances
type ListExp = vl +
               cl(Exp # ListExp)

and

-- Pairs of metavariables and expressions used in substitutions
type Par = par(id # Exp)

and

-- Substitutions used in expansion
type Sust = vs +
            cs(Par # Sust) +
            sshift(Sust # id # Int # Bool)

and

-- Pairs of variables and expressions used in var-substitutions
type TPar = tpar(id # Int # Exp)

and

-- Var-substitutions used in beta reduction
type TSust = vt +
            ct(TPar # TSust) +
            tsshift(TSust # id)

end
```

Contents

1 Introduction **2**

2 Notational definitions and explicit substitutions **3**

2.1 Notational definitions 3

2.2 Explicit substitutions 3

3 Macro expansion semantics **4**

3.1 The core language 5

3.2 The macro definition language 5

3.3 Instance and instance substitution 6

3.4 Operational semantics 7

3.5 Properties of the semantics 8

3.6 Examples 8

4 Mixing expansion and reduction **9**

4.1 Adding the β–rule 12

4.2 Properties of the extended system 14

5 Conclusions and related work **14**

References **15**

A Proofs **17**

B A prototype in lazy ML **24**