A Confluent Calculus of Macro Expansion and Evaluation

Ana Bove* Laura Arbilla†
bove@incouy.edu.uy 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 expressiveness 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.

1 Introduction

The use of syntactic abbreviations in mathematics as well as in programming languages has two advantages [16]. 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 expressiveness of programming languages but do not increase their semantic power [8, 19]. These abbreviations enrich assembly as well as modern high level languages such as C [15] and Scheme [20]. A typical example of a syntactic abbreviation in Scheme is *let*, defined by the following notational definition:

$$(let\ \mathsf{x}\ be\ \mathsf{v}\ in\ \mathsf{B}) \stackrel{df}{\equiv} ((\lambda\,(\mathsf{x})\,\mathsf{B})\ \mathsf{v}) \qquad (1)$$

This equation indicates that the expression to the left of the symbol $\stackrel{df}{\equiv}$ abbreviates the expression to its right. 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 [3].

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,

$$(let\ y\ be\ 3\ in\ (\lambda\,(z)\,(+\ z\ y))) \qquad (2)$$

is an instance of the syntactic abbreviation *let*, where the expressions y, 3, and $(\lambda\ (z)\ (+\ z\ y))$

*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. The Computer Science Department of the University of Texas at Austin generously provided facilities for preparing the final version of this paper.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
1992 ACM LISP & F.P.-6/92/CA
© 1992 ACM 0-89791-483-X/92/0006/0278...$1.50

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:

$$((\lambda\ (y)\ (\lambda\ (z)\ (+\ z\ y)))\ 3) \qquad (3)$$

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, 2]. The resulting semantics respects the binding strategy of the core language and does not perform unwanted capture of identifiers during expansion [17].

In this method, the expansion is done *completely* before starting execution, although some of the expansions may be avoidable. We avoid unnecessary expansion 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. If the evaluation strategy of the core language is lazy [12], 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 discusses related work and future directions of research. Section 6 presents conclusions.

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 $l \stackrel{df}{\equiv} r$, meaning that the meta–expression l abbreviates the meta–expression r. Meta–expression l denotes the set of all possible instances of the macro, while r denotes the set of "one–step expansions" of those instances. Meta–expression 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. Meta–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 (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, 2].

Substitutions are fundamental in the λ–calculus. The classical β–rule [3]:

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

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 [17]. We propose to represent these environments as *explicit substitutions* in a formal semantics of macro expansion, as do 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*:

$$\mathit{freeze}\ \mathsf{a} \stackrel{df}{\equiv} (\lambda x.\mathsf{a}) \qquad (4)$$

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. 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:

$$(\lambda x.e)[s] \rightarrow (\lambda x.e[s\uparrow x])$$

$$\begin{aligned} \{\mathsf{a} \leftarrow e, s\}\uparrow x &\rightarrow \{\mathsf{a} \leftarrow e\uparrow x, s\uparrow x\} \\ \mathsf{a}\{\mathsf{a} \leftarrow e, s\} &\rightarrow e \\ \phi\uparrow x &\rightarrow \phi \end{aligned}$$

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.

3.1 The core language

The syntax of the core language, Λ_n, is in Fig. 1. Language Λ_n is the language of the λ–calculus with numbers [3] and labeled variables. A labeled variable has a name and an exponent. The exponent is a non–negative number that defines the variable's distance (number of λ operators that bind a variable of the same name) from its binding instance. Binding instances are *not* labeled. For example, the expression:

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

would be written in the simple 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's technique [5], but the use of names improves legibility. In this paper, x occasionally denotes x^0.

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 Fig. 1. We use a different font for metavariables, to distinguish them from variables. The symbol ϵ denotes the empty string

[1] We slightly modify the syntax.

Syntactic domains:

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

Syntax of Λ_n:

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

Syntax of $\mathcal{L}$:

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

Figure 1: Languages syntax.

and $\mathrm{MV}(r)$ denotes the set of metavariables appearing in expression r.

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

Notational definitions are as described in Section 2. A notational definition d ($\in \mathit{Ndef}$) is enclosed within square brackets ([ ]). The left-hand-side expression l ($\in \mathit{Lhs}$) is a list of variables and metavariables. The right–hand–side expression r ($\in \mathit{Rhs}$) 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* ($\in \mathit{Kw}$). For instance, in the definition of *freeze* (equation (4)), 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:

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

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,

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

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: $\forall\ i = 1 \ldots m$ $(w_i = u_i$ or $w_i, u_i \in Meta)$

4. No new metavariables are introduced by the rhs: $\mathrm{MV}(r_h) \subseteq \mathrm{MV}(l_h))$, $h = 1 \ldots n$

We refer to these restrictions as ND.

3.3 Instance and instance substitution

Before we can formulate the operational semantics of macro expansion in an expression of language $\mathcal{L}$, we need to formalize 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* $[l \stackrel{df}{\equiv} r]$, *where* $l \equiv w_1 \ldots w_m$, *if the following conditions hold:*

- $(w_i \in Var$ and $t_i \in Var) \Rightarrow t_i = w_i$
- $w_i \in Meta \Rightarrow t_i \in Rhs$

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

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

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

Substitution s relates the metavariables in l to the corresponding expressions in $\langle Lr \rangle$. We also write instance substitutions as $\mathcal{S}(\langle Lr \rangle, l \stackrel{df}{\equiv} r)$. 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 $D\ r \in Pgm$, where D is a list of notational definitions satisfying restrictions ND, and r is an expression where every $\langle Lr \rangle$–subexpression is an instance of a definition in D; that 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:

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

where $\Rightarrow_E^{*D}$ is the reflexive transitive compatible closure of relation $\Rightarrow_E$, defined in Fig. 2. The set of rules $\Rightarrow_E$ is really parameterized over the set of notational definitions D. Occasionally $\Rightarrow_E$ and $\Rightarrow_E^*$ denote $\Rightarrow_E^D$ and $\Rightarrow_E^{*D}$ respectively. The definition of *compatible* closure is in Fig. 3. An expression e is in *E–normal–form* if there does not exist e' such that $e \Rightarrow_E e'$ [3]. For a general introduction to rewriting systems and their properties see [3, 13, 14].

As we show later, the $\Rightarrow_E$ relation is Church–Rosser; that 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 strongly normalizing (SN), i.e. 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 Fig. 2.

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.

Syntax:

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

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

Rules:

Substitution Introduction:

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

Substitution Elimination:

$$num[s] \Rightarrow_E num \qquad (E_{num})$$

$$x^n[s] \Rightarrow_E x^n \qquad (E_{var})$$

$$\mathsf{a}[\{\mathsf{a} \leftarrow r, s\}] \Rightarrow_E r \qquad (E_{mv1})$$

$$\mathsf{b}[\{\mathsf{a} \leftarrow r, s\}] \Rightarrow_E \mathsf{b}[s], \text{ if } \mathsf{a} \neq \mathsf{b} \qquad (E_{mv2})$$

$$(\lambda x.r)[s] \Rightarrow_E (\lambda x.r[s{\uparrow}x^{0,f}]) \qquad (E_{\lambda var})$$

$$(\lambda \mathsf{a}.r)[s] \Rightarrow_E (\lambda x.(r{\uparrow}x^{0,0,f})[\{\mathsf{a} \leftarrow x, s{\uparrow}x^{n,t}\}]), \text{ if } \mathsf{a}[s] \Rightarrow_E^* x^n \qquad (E_{\lambda mv1})$$

$$(\lambda \mathsf{a}.r)[s] \Rightarrow_E (\lambda \mathsf{b}.r[s]), \text{ if } \mathsf{a}[s] \Rightarrow_E^* \mathsf{b} \qquad (E_{\lambda mv2})$$

$$(r_1\ r_2)[s] \Rightarrow_E (r_1[s]\ r_2[s]) \qquad (E_{app})$$

Shift operator:

$$\phi{\uparrow}x^{m,b} \Rightarrow_E \phi \qquad (Ss_{mt})$$

$$\{\mathsf{a} \leftarrow r, s\}{\uparrow}x^{m,b} \Rightarrow_E \{\mathsf{a} \leftarrow r{\uparrow}x^{m,0,b}, s{\uparrow}x^{m,b}\} \qquad (Ss)$$

$$num{\uparrow}x^{m,m',b} \Rightarrow_E num \qquad (Se_{num})$$

$$x^n{\uparrow}x^{m,m',b} \Rightarrow_E x^n, \text{ if } n < m \qquad (Se_{var1})$$

$$x^n{\uparrow}x^{m,m',b} \Rightarrow_E x^{n+1}, \text{ if } n > m \qquad (Se_{var2})$$

$$x^n{\uparrow}x^{n,m',t} \Rightarrow_E x^{m'} \qquad (Se_{var3})$$

$$x^n{\uparrow}x^{n,m',f} \Rightarrow_E x^{n+1} \qquad (Se_{var4})$$

$$y^n{\uparrow}x^{m,m',b} \Rightarrow_E y^n, \text{ if } x \neq y \qquad (Se_{var5})$$

$$\mathsf{a}{\uparrow}x^{m,m',b} \Rightarrow_E \mathsf{a} \qquad (Se_{mv})$$

$$(\lambda x.r){\uparrow}x^{m,m',b} \Rightarrow_E (\lambda x.r{\uparrow}x^{m+1,m'+1,b}) \qquad (Se_{\lambda var1})$$

$$(\lambda y.r){\uparrow}x^{m,m',b} \Rightarrow_E (\lambda y.r{\uparrow}x^{m,m',b}), \text{ if } x \neq y \qquad (Se_{\lambda var2})$$

$$(\lambda \mathsf{a}.r){\uparrow}x^{m,m',b} \Rightarrow_E (\lambda \mathsf{a}.r{\uparrow}x^{m,m',b}) \qquad (Se_{\lambda mv})$$

$$(r_1\ r_2){\uparrow}x^{m,m',b} \Rightarrow_E (r_1{\uparrow}x^{m,m',b}\ r_2{\uparrow}x^{m,m',b}) \qquad (Se_{app})$$

Figure 2: Expansion System

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 ${\uparrow}x^{0,f}$ on substitution s performs this updating. Second,

The compatible closure $\rightarrow$ of $\Rightarrow_E$ is inductively defined as follows:

$$
\begin{array}{rcl}
r, r', z & \in & Exprs \\
s, s' & \in & Subst \\
r \Rightarrow_E r' & \Rightarrow & r \rightarrow r' \\
s \Rightarrow_E s' & \Rightarrow & s \rightarrow s' \\
r \rightarrow r' & \Rightarrow & (\lambda x.r) \rightarrow (\lambda x.r') \\
r \rightarrow r' & \Rightarrow & (\lambda \mathsf{a}.r) \rightarrow (\lambda \mathsf{a}.r') \\
r \rightarrow r' & \Rightarrow & (z\ r) \rightarrow (z\ r') \\
r \rightarrow r' & \Rightarrow & (r\ z) \rightarrow (r'\ z) \\
r \rightarrow r' & \Rightarrow & r[s] \rightarrow r'[s] \\
r \rightarrow r' & \Rightarrow & r{\uparrow}x^{m,m',b} \rightarrow r'x^{m,m',b} \\
r \rightarrow r' & \Rightarrow & \{\mathsf{a} \leftarrow r, s\} \rightarrow \{\mathsf{a} \leftarrow r', s\} \\
s \rightarrow s' & \Rightarrow & r[s] \rightarrow r[s'] \\
s \rightarrow s' & \Rightarrow & \{\mathsf{a} \leftarrow r, s\} \rightarrow \{\mathsf{a} \leftarrow r, s'\} \\
s \rightarrow s' & \Rightarrow & s{\uparrow}x^{m,b} \rightarrow s'{\uparrow}x^{m,b}
\end{array}
$$

Figure 3: Compatible closure.

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 $n > m$ or $n = m$ 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

Only proof sketches are included. We refer the reader to the Technical Report [4] for detailed proofs.

Theorem 1 (Strongly Normalizing) *The relation $\Rightarrow_E$ on correct programs is SN.*

Proof Sketch: We define a positive measure function on terms, f, that is strictly decreasing for every rule. Formally, $\forall\ r_1 \Rightarrow_E r_2, f(r_1) > f(r_2)$ □

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

Proof Sketch: This theorem directly follows from Theorem 1 and from the fact that $\Rightarrow_E$ on correct programs is Weakly Church–Rosser [3, 14], *i.e.*:

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

The proof of WCR follows from the observation that the rewriting system does not have critical pairs [13, 14]. □

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

Proof Sketch: ($\Leftarrow$) Rules of $\Rightarrow_E$ cannot be applied over Λ_n–terms because they do not contain instances of macros or substitutions; thus, Λ_n–terms are E–normal–forms.

($\Rightarrow$) Let p be a correct program and e its E–nf. The correctness of p assures that every instance appearing in p can be eliminated using Rule *Intro* since there always exist a corresponding notational definition in p. Also, the set of notational definitions satisfies restrictions ND;

thus, every metavariable introduced by *Intro* can be replaced by an expression. By case analysis, we prove that e does not contain substitutions, or shift operators. Thus, an E–nf is in Λ_n. □

4 Mixing expansion and reduction

Traditionally macro expansion is completed before evaluation. We explore the possibility of *mixing* expansion and evaluation by adding the β–rule to relation $\Rightarrow_E$. It turns out that the mixture 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: substitutions of variables by expressions. The elimination rules for these substitutions force the definition of a new renaming operation analogous to shift ($\uparrow$). When 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* $n > m \geq 0$. In the $\lambda\sigma$–calculus, composition with the *id* substitution performs this unshift operation using the de Bruijn notation. Specifically, the β–rule in the extended system ($\Rightarrow_{ER}$) is:

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

The set of rules $\Rightarrow_{ER}$ is also parameterized over the set of notational definitions D. The complete set of rules is in the full version of the paper [4]. Note that ϕ_t is the empty substitution of variables by expressions.

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:

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

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:

$$\begin{aligned} & ((\lambda x.y)\ \langle let\ u\ be\ 1\ in\ (y\ u)\rangle) \\ \Rightarrow_{ER}\ & (y[\{x \leftarrow \langle let\ u\ be\ 1\ in\ (y\ u)\rangle, \phi_t\}])\downarrow x^0 \\ \Rightarrow_{ER}\ & (y[\phi_t])\downarrow x^0 \\ \Rightarrow_{ER}\ & y\downarrow x^0 \\ \Rightarrow_{ER}\ & y \end{aligned}$$

Here, the expansion of *let* is avoided.

The set of $\Rightarrow_{ER}$–rules is obviously not SN, since $\rightarrow_\beta$ is not SN in Λ_n. But it is strongly confluent:

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

Notation $\Rightarrow_{ER}^{\epsilon}$ means *zero or one* steps of reduction. The proof of strong confluence is by structural induction on the terms. A detailed proof can be found in [4]. As a corollary of strong confluence, $\Rightarrow_{ER}$ is Church–Rosser.

5 Related and future work

We base our macro definition language on the work by Kohlbecker [16, 17, 18]. It preserves the expression structure of the core language, as does Kohlbecker's, but it does not consider ellipsis or recursive macro definitions. Recursive macros have been omited for simplicity. Their incorporation does not change the essence and properties of the systems as long as notational definitions are restricted so that the expansion of instances always terminates. The formulation of this restriction is difficult and complicates the proofs of Church–Rosserness. The use of ellipsis in defining such recursive macros complicates the definition of instances and instance substitutions. In this case, metavariables are associated to lists of expressions. Therefore, we need new operations on substitutions and rules to manipulate them. Further detailed work needs to be done in this direction.

Work has been done in designing powerful tools for defining macros and efficient algorithms

for their expansion [6, 7, 16, 17, 18]. 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. We have not intended to define a new tool for defining macros, but attempted to provide a clean operational semantics of macro expansion and of the interleaving of macro expansion and program evaluation on the λ–calculus. Further work includes adding more language constructs to our systems, particularly local macros [6].

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 and unshift operations to work with names of identifiers instead of de Bruijn notation and we add rules for eliminating shift and unshift operators. Hardin and Lévy [11] also eliminate them, but they use de Bruijn notation. Explicit substitutions can be used for macro expansion on other languages as well.

Formal semantics of macro expansion has been studied in depth by Kohlbecker [16] and Griffin [9, 10]. Kohlbecker uses a denotational framework to explain macro expansion. 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 and of adding binding information to the macros, as does Griffin [9, 10]. The binding structure of a macro is in the right–hand–side of its definition and the system performs the necessary renamings when eliminating the substitution created by rule *Intro*. However, the lack of explicit binding information in macros causes substitution elimination not to commute with macro expansion. In other words, an expression of the form $\langle Lr\rangle[s]$ forces the expansion of $\langle Lr\rangle$ before substitution s can be eliminated. Thus, macro instances are expanded, and evaluation functions cannot be derived for them. However, their expansion interleaves with the evaluation of a program in our system.

6 Conclusions

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 is intuitive and closely models the practice of macro expansion.

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. The full paper [4] contains the documented source code.

We believe our calculus that expands and evaluates expressions is close to practice and implementation. Although restricted to the λ–calculus, it provides a good basis for a deeper 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 Laurence Puel, who verified the proof of SN of the $\Rightarrow_E$ system. We are grateful to Juan Vicente Echagüe, who read in detail complete drafts of this paper providing helpful insight into some application and formal aspects of this work. We also thank Matthias

Felleisen for providing helpful comments during the preparation of this work.

References

[1] M. Abadi, L. Cardelli, P.L. Curien, and J.J. Lévy. Explicit Substitution. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pages 31–46, 1990.

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

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

[4] A. Bove and L. Arbilla. A Confluent calculus of macro expansion and evaluation. Technical Report INCO-91-01, Instituto de Computación, Universidad de la República, Montevideo, Uruguay, 1991.

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

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

[7] 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.

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

[10] T. Griffin. *Notational Definitions and Top–Down Refinement for Interactive Proof Development Systems.* PhD thesis, Cornell University, August 1988.

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

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

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

[14] 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.

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

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

[17] E. Kohlbecker, 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.

[18] 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.

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

[20] 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.