, , 1–34 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Incremental Construction of Declarative Programs

JEAN-LOUIS GIAVITTO giavitto@lami.univ-evry.fr
LaMI - UMR du CNRS, Université d'Évry Val d'Essonne, Boulevard des coquibus
91025 Évry Cedex, France

OLIVIER MICHEL michel@lami.univ-evry.fr
LaMI - UMR du CNRS, Université d'Évry Val d'Essonne, Boulevard des coquibus
91025 Évry Cedex, France

Abstract. Binding and substitution of bound variables are major issues in the design of languages. The mechanism of name capture lies at the heart of modular, incremental and object-oriented programming.

In this paper, we present a new reflexive type-free formalism, the *amalgams* based upon three operators that focus on the notion of names and name capture. The formalism is targeted towards the modeling of incremental program construction. Because of its versatility, it allows a natural emulation of several programming styles.

The *core* formalism is first presented and then examples of the definition of first-class environments, modeling a declarative object-oriented programming are given.

Editor:

Keywords: name, name capture, object-oriented and declarative programming, mixins

1. Introduction

In this paper we investigate an approach for the incremental construction of declarative programs based on the concept of name. Standard approaches rely on functionnal composition (e.g. in dataflow). However, the concept of *naming* is a widespread and heavily used notion in computer science in general and in programming languages in particular. The concept of name can be found, among others, in the following areas:

- Imperative languages are built on the notion of state which is a partial function from names to values.
- Names have recently been introduced in the λ-calculus for the following purposes:
  - allowing an out of order binding of the terms in a λ-abstraction [18],
  - allowing the access to (possibly redefined) terms at various different abstract levels [12, 10, 9, 11],
- Names are used in dynamic applications where they represent entry-points for the sharing of information. Examples are:

  - dynamic linking that occurs at run-time with shared libraries [21] used in a program,
  - "Applets" of WWW browsers in Java [43] or Caml-Light [41] correspond to code dynamically loaded through the access of specific parts of a WWW page.

- In [38], Milner emphasizes naming as the key idea of the π-calculus [37], a model of distributed computing.
- Names are central issues in many data and program structuring mechanisms:
  - the object-as-record point of view [8] corresponds to a cartesian product where names are associated to expressions,
  - the use of name as the key to the construction of incremental programs is a view widely shared [20, 27, 28],
  - in the context of modular construction of programs, the notion of *mixins* [5], where names are used as deffered references in another mixins, generalizes inheritance [6, 16], module composition [2, 15] and separate compilation [1].

The previous examples show that the concept of name is a central notion in the incremental construction of programs and this view has been subsequently stressed by many authors [20, 27, 28, 15, 6].

In this work, we develop a core language used for defining components called *systems*: a collection of definitions of components, where the definition of some of them can be deferred to another system (eventually in a mutually recursive way). Thus, the typical operator for composing systems is a binary merge operator "#". The combination mechanism relies on free names (the deferred components) and name capture (the instantiation method).

Our approach in system composition is to retain the explicit composition operator of the functional style and the naming scheme of the declarative style. The motivation is to capture some structure induced by the functional combinators (e.g. to formalize the linking process, the scoping rules, etc.) while relying on the concept of name which is central in many coarse-grained composition mechanisms (like class inheritance, module composition, link editing, message passing, remote procedure call, applet downloading, etc.).

We provide a formal foundation for the system notion. More precisely, we define a semantics of systems in the natural semantics style for three basic operators: the *amalgamation* operator "{}" which creates *systems*, the *merge* operator "#" and the *selection* operator "**.**". A notion of name is defined, called a *reference*, which can either be bound or free. Two syntactically equal [40] references refer to the same object. An element of a *system* is a pair (*identifier*, *expression*) where the expression involves references and the three operators. References are explicitly annotated so that they may refer to redefined definitions. Finally, a mechanism of *propagation* of definitions to bound references is defined, allowing the dynamic completion of open expressions. This mechanism, together with the operators are called *amalgams*.

We give in the next two sections an intuitive definition of amalgams and how the entities it defines are handled. We define in section 4 a formal semantics of the amalgams. Section 5 is dealing with examples of amalgams. We first show the emulation of arithmetical functions in the pure amalgams and then the use of amalgams in a declarative language to allow an object-oriented programming style. We discuss the relation between amalgams and other formalisms and languages that do address the same problems in section 6. We conclude in section 7 with the current status of this work and its integration into a declarative language.

2. An Intuitive Presentation of the Amalgams

We first describe amalgams through an intuitive presentation to give a flavor of the formalism. Amalgams try to capture the three following features:

1. *specify* a set of *definitions*,
2. *build* a new set of definitions through the *merge* of two existing sets,
3. *evaluate* an expression *using* a set of definitions.

We remark that:

- a definition associates a name with an expression,
- the evaluation of an expression using a set of definitions means, from the amalgam point of view, that names involved in the expression have to be substituted by their definition.

We are going to focus on those three mechanisms without introducing any additional object or control structure. We get the "*pure calculus of the amalgams*", which consists of three operators: the n-ary *amalgamation* operator "$\{\ldots\}$" (point 1), the binary *merge* operator "$\#$" (point 2) and the binary *selection* operator "**.**" (point 3).

2.1. *Introduction : Systems, Equations and References*

2.1.1. Definition of a System. The result of an amalgamation is a *system.* A system is a set of *definitions* where a definition is a pair:

$$identifier = expression$$

For example, the expression $\{a = 1, b = 2 + 3\}$ denotes a system gathering two definitions: $a = 1$ and $b = 2 + 3$.We also call these definitions *equations.* When an identifier appears immediately at the right of an equal sign, it is called a *reference.* We suppose that all left hand-sides (l.h.s.) of a system are different. The right hand-sides (r.h.s.) of a system are expressions. We may define nested systems. In this example:

$$\{a = 1, B = \{c = d, e = a\}\}$$

we find: four definitions $a = 1$, $B = \{c = d, e = a\}$, $c = d$ and $e = a$; four identifiers a (the outermost one), B, c and e; two references d and (the innermost) a. The references can be bound or free, whether they correspond or not to the identifier of a r.h.s. of an equation (here, reference d is free while reference a is bound).

2.1.2. Free and Bound References. The binding mechanism associates the expression e in the r.h.s. of equation $id = e$ to a reference id. For example, in the following expression, the reference b in the r.h.s. of the first equation refers to the second equation and is therefore a bound reference (we indicate with an arrow which definition is referred to):

$$\begin{tikzcd}[column sep=tiny] \{a = & b \arrow[rr, bend left] & , b & = & 2\} \end{tikzcd} \tag{1}$$

The order of the equations is not significant. The expression $\{b = 2, a = b\}$ defines the same system. Circular references are allowed:

$$\begin{tikzcd}[column sep=tiny] \{x & = & y \arrow[rr, bend left] & ,\ y & = & x\} \arrow[llll, bend left] \end{tikzcd} \tag{2}$$

The scope of definitions does not extend outside their system. For example, in the expression:

$$\{a = x,\ B = \{x = 1,\ y = 2\}\}$$

the reference x in the r.h.s. of the first equation cannot be bound to the definition of x in the enclosed system defined by B. A system defines a notion of scope. The scoping rules follow the usual rules defined for block structures (like in the `C` language for example).

The nesting of systems allows redefinitions. Therefore, the problem of accessing redefinitions arises. A simple rule is to shadow all previously defined expressions with the same identifier (as does `C`). But allowing access to redefined equations leads to interesting features: for instance, in an object-oriented programming style, allowing the access to redefined methods gives access to methods of a super-class. Consequently, we choose to allow the access to redefined equations by introducing an explicit scope escaping operator[1] : id^n is a reference that is looking for the definition of id in the m^{th} enclosing scope, such that $m \geq n$. For example, in the expression:

$$\begin{tikzcd}[column sep=tiny] \{a & = & 1,\ B = \{a = 2, x = & a^1\}\} \arrow[ll, bend left] \end{tikzcd}$$

the reference to a in the r.h.s. of the definition of x refers to the equation $a = 1$ through the escaping operator "$\square^1$". Such a reference is said to be *bound*.

A reference that is not bound to a definition is a *free* reference, as for example for x in the system $\{a = x\}$. An expression involving a free reference is an *open* expression and an expression with no free references is *closed*. Following the "escaping of scope" operator, it should be useful to be able to "jump over definitions". We therefore use the same operator for free references. For example, in the system:

$$\{A = \{x = 1, y = x^1\}\} \tag{3}$$

the reference x, in the r.h.s. of the equation defined by y, is not bound to the equation $x = 1$ because the “$\square^1$” operators specifies a binding one scope away from the current scope where the reference appears.

Since the reference id^0 leads to the same behavior as id, we define by convention that a reference with no explicit escaping operator corresponds to id^0, thus all references are of the form id^n where $n \in \mathbb{N}$ and id is an identifier.

2.1.3. Evaluation of an amalgam expression. The *evaluation* process roughly corresponds to the substitution of bound references by their corresponding definition and to the simplification of the three operators, whenever possible (see below). We formalize that in section 4.

2.2. A Data flow Representation.

There are many ways to look upon amalgams. We emphasize here on a data flow interpretation because of its intuitive graphical representation.

There is a simple data flow representation of a system as an incomplete graph. Every operator in an expression is a node. Nodes are linked together by edges. A definition $id = op(..., ...)$ is a node op with output edges named id. The input edges correspond to identifiers appearing as arguments of the operator. A *pending input edge* corresponds to a free reference. Output edges are simply identifiers defined by the system (Cf. Fig. 1).

There are several ways in which data flow graphs can be composed. System composition corresponds graphically to connect some output edges with some pending input edges.

In the applicative [23] or functional [4] style, the pending input edge and the output edges of a graph are linearly ordered and connected on this basis, without considering their identifiers. One drawback is that the management of links (like forking, forgiving, etc.) must be explicitly done. The connection itself can be of several kind: parallel composition, serial composition, feedback, etc., Cf. Fig. 2.

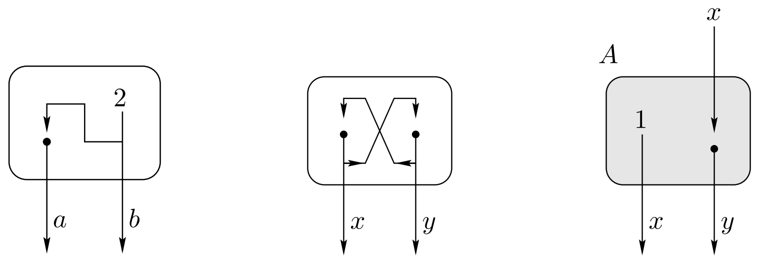

Figure 1. System (1) is pictured at the left as a data flow graph. The graph in the middle represents system (2). The graph at the right corresponds to the definition of A in the system (3).

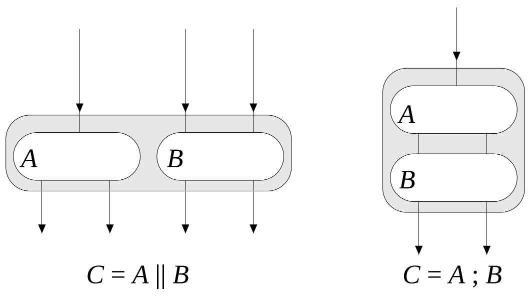

Figure 2. Parallel functional composition (at left) and serial functional composition of two data flow graphs A and B.

In some calculi for concurrent systems (CCS [36] for example), there is another way of describing a composition. It is based upon the names of the edges. So, if we want to use the same kind of system in two different places, we have to rename one of the instances. However, one advantage of the approach is the explicit identification of the system parameters and outputs.

2.3. System Composition

2.3.1. Merging Systems. We have seen that an expression may involve free references. The merging of two systems combines the equations and binds the free references whenever possible. For example, the following expression:

$$\{a = 1, b = c^0\} \# \{c = 2, d = a^0\} \tag{4}$$

is evaluated to the expression $\{a = 1, b = c^0, c = 2, d = a^0\}$ and then to $\{a = 1, b = 2, c = 2, d = 1\}$. To be merged together, both operands of a merge operator have to be systems. As we can see, the merge of two systems is more complicated than just *packing* together two sets of expressions. The binding of free references allow the completion of open expressions with definitions coming from other expressions.

The data flow representation of the merge operator is very simple (Fig. 3): just connect the pending input edges of one graph to the output edges of the other graph, and vice-versa. This process is based on the name of the edges and is symmetric (we insist in the assumption that expressions leading to the definition of systems with two equations for the same identifier are rejected).

2.3.2. Extracting a Definition from a System. If a system is a set of definitions, there must be an operator to "extract" the value of some definition. This operator is called a *selection.* We generalize this operator to handle the evaluation of any expression in the *environment* defined by the system. For example, the expression:

$$\{e = a^0, \quad r = \{a = 1, b = 2\} \,\textbf{.}\, (e^1 + b^0)\}$$

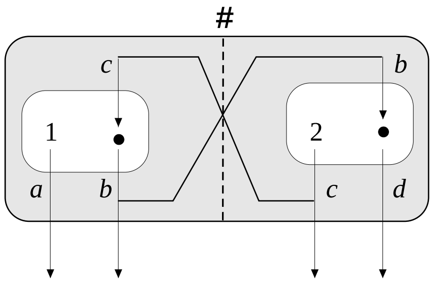

Figure 3. Merging of two systems. The picture illustrates expression (4).

evaluates the r.h.s. of the selection using the definitions provided first by the l.h.s. and then by the including systems. The expression is first be evaluated to $\{e = a^0,\ r = \{a = 1, b = 2\} \centerdot (a^0 + 2)\}$, and then to $\{e = a^0,\ r = \{a = 1, b = 2\} \centerdot (1 + 2)\}$ and finally to $\{e = a^0,\ r = 3\}$. To allow the evaluation of the r.h.s. of a selection, the l.h.s. has to be a system. If it is not the case, the l.h.s. is evaluated, until it becomes a system; then, the r.h.s. can be evaluated using the l.h.s. definitions. As we can see, the system as first operand of a selection plays the role of an environment providing definitions to the expressions that have to be evaluated. Note that the l.h.s. of a selection constitutes a scope for the r.h.s.

As a first approximation, the selection operates like an *extensible* `let rec ... in ...`: the r.h.s. expression is evaluated according to the definitions of the l.h.s. Unlike the `let rec` construction and because of the reflexive nature of the systems, the definitions are denotable, that is: the set of the definitions is computable (in `let rec` only the value of the definitions are computed but the set of the definitions is statically known).

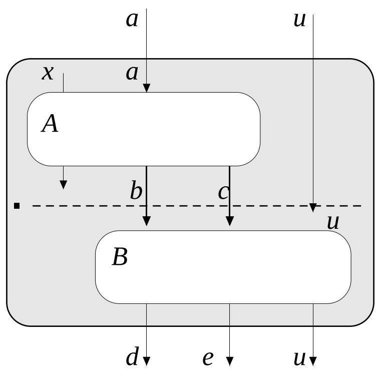

Figure 4. The selection $A \centerdot B$. The outputs of the selection are the outputs of B. The input of the selection are the inputs of B that are not fed by A (eventually augmented by the inputs of A that are needed for the evaluation of the outputs of A used by the inputs of B).

The data flow representation of the selection operator is very simple too (Cf. Fig. 4): just connect the outputs of the l.h.s. of the operator with the inputs of the r.h.s., and retain in the result only the outputs of the r.h.s. This is reminiscent of the serial composition.

A more complete example of amalgams shown as a high-order data-flow graph is given in Fig. 5.

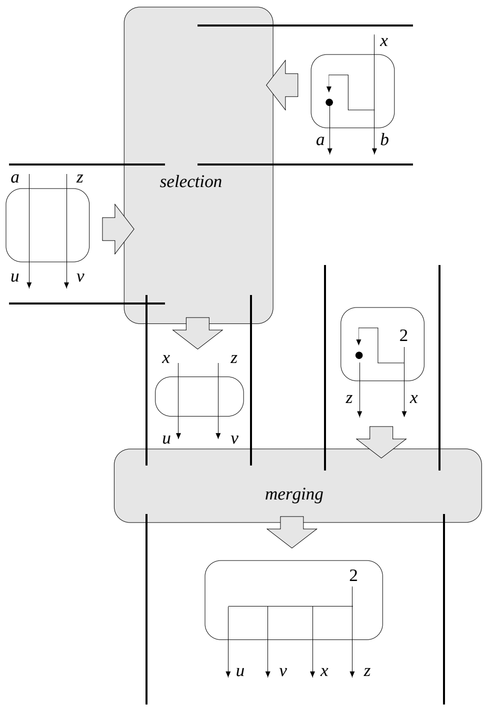

Figure 5. Merge and select can be seen as high-order data flow operator. They are then pictured as "macro-nodes" in gray. The operational semantics of a data flow graph is based on the circulation of tokens labeled with a value. Thus, tokens representing entire data flow graphs are flowing through the edges linking the macro-nodes.
The high-order data flow graph represents the expression $(\{b = x, a = b\} \centerdot \{u = a, v = z\})$ # $\{x = 2, z = x\}$ and some of the intermediate data flow graph produced during the evaluation process.

3. Difficulties in the Evaluation of an Expression

Before going into the details of the formalization of the amalgams, we want to show first some examples of the subtleties involved by the evaluation of an expression to its normal form (the normal form of an expression is defined in section 4.6).

The evaluation process roughly corresponds to a propagation of definitions to bound references and to the simplification of operators, whenever possible. However, this propagation cannot be done in any order. For example, in the following expression

$$\{y = 2, x = y^0, a = \{b = x^1, y = 1\}\}$$

if the definition of x is propagated in a before being totally evaluated, b in a evaluates to 1. On the contrary, if the definition of x is evaluated before propagation, the value of b in a is 2. Remember that the order of the definitions in a system is irrelevant. We cannot therefore rely on it for the propagation order.

Our strategy is to fully evaluate a definition before substitution. However, the meaning of "fully evaluated" is not obvious. In the following expression

$$\{b = \{x = 1\} \ \mathbf{.}\ (y^0 \ \mathbf{.}\ x^0)\}$$

the reference x cannot be substituted by the definition $x = 1$ because the free reference y^0 may be substituted in some other context by a system providing another definition for x.

In an expression $l \ \mathbf{.}\ r$, l is the immediate enclosing scope for r. This is not the case for an expression $l \# r$ where l and r are bound in the enclosing scope before being merged together. For instance

$$\{a = 1, c = \{b = a^0\} \# \{a = 2\}\}$$

is evaluated to $\{a = 1, c = \{b = 1, a = 2\}\}$. This strategy is coherent with the fact that the enclosing definitions have to be used to evaluate l and r in $l \# r$ as long as l and r are not both systems.

The interested reader may refer to [32, ch. 8] for more details. We describe in the following sections a semantics for the amalgams along these lines.

4. Formal Semantics

In the next sections, we define an *evaluation process* for *core* amalgams expressions (no constants or arithematical operators), that is, *reducing* a term to its *normal form*.

We use in the next sections the following notations: $\texttt{List}(X)$ represents the set of lists of elements of X; $[\,]$ is the empty list; $h{::}\,l$ is the concatenation of an element h and a list l; $l@l'$ is the concatenation of l and l'; $\texttt{nth}(n, l)$ gives access to the n^{th} element of a list; $\texttt{tl}(l)$ is the list without its first element. By convention, $\texttt{tl}([\,]) = [\,]$, $\texttt{tl}(h :: t) = t$ and the iterates of $\texttt{tl}$ are defined by: $\texttt{tl}^0(l) = l$, $\texttt{tl}^1(l) = \texttt{tl}(l)$ and $\texttt{tl}^n(l) = \texttt{tl}^{(n-1)}(\texttt{tl}(l))$. We manipulate pairs in the ML-style, with projection functions `fst` and `snd`. $\mathcal{P}(E)$ is the *powerset* of the set E. The function `map` applies its first functional argument to the elements of its second argument which is a list *or* a set; `flatten` flattens its argument, a list of lists (*or* a set of sets), by concatenating the first level of lists (*or* sets).

4.1. Set of Amalgam Terms

Definition 1 (Id, Ref*,* Sys *and* Bool*).* The set Id is the set of identifiers x. The set Ref is the set of identifiers with references x^n. The set Sys is the set of terms with $\{...\}$ as head-operator. Bool is the set of boolean values $\texttt{true}, \texttt{false}$.

Definition 2 (Σ*).* The set of amalgam terms Σ is the smallest set such that:

1. a *reference* x^n is an element of Σ,
2. if $e_1, ..., e_n$ are elements of Σ and $x_1, ..., x_n$ are n distinct identifiers, then a *system* $\{x_1 = e_1, ..., x_n = e_n\}$ is an element of Σ,
3. if e and e' are elements of Σ, then $e \Diamond e'$ with $\Diamond \in \{\#, \textbf{.}\}$ is an element of Σ.

We take the following conventions: i, j, n range over $\mathbb{N}$; $x, y, z, x', \ldots, x_1, \ldots$ range over Id; $e, u, v, e', \ldots, e_1, \ldots, M$ and N range over Σ; p ranges over Ref; s ranges over Sys. We use $\equiv$ for the (syntactical) equality of elements of Id and Σ.

For convenience, we abbreviate a system $\{x_1 = e_1, \ldots, x_n = e_n\}$ by $\{\overrightarrow{x = e}\}$. The notation $\{\overrightarrow{x}\}$ stands for the set $\{x_1, \ldots, x_n\}$. In these notations, the number n of elements is left implicit and we always assume that $x_i \not\equiv x_j$ if $i \neq j$. Beware that two different abbreviations $\{\overrightarrow{x = e}\}$ and $\{\overrightarrow{x' = e'}\}$ stand for different systems, respectively $\{x_1 = e_1, \ldots, x_n = e_n\}$ and $\{x'_1 = e'_1, \ldots, x'_m = e'_m\}$. The notation $\{\overrightarrow{x = e}, \overrightarrow{x' = e'}\}$ abbreviates $\{x_1 = e_1, \ldots, x_n = e_n, x'_1 = e'_1, \ldots, x'_m = e'_m\}$ and we always assumes that $\{\overrightarrow{x}\} \cap \{\overrightarrow{x'}\} = \emptyset$.

As stated in section 2.1.2, we identify (if required) an element of Id (say x) with the "same" element of Ref with 0 as escaping value (x^0), and conversely.

Definition 3 (Defined Identifiers). The total function $\texttt{id} : \Sigma \mapsto \text{Id} \cup \{\star\}$ (where $\star \notin \text{Id}$) giving the set of *defined identifiers* in a system is specified by induction on the structure of a term u of Σ:

$$\begin{array}{lcl} \texttt{id}(\{\overrightarrow{x = e}\}) & = & \{\overrightarrow{x}\} \\ \texttt{id}(u) & = & \star \quad \text{for } u \text{ not in Sys.} \end{array}$$

4.2. Bound References

We define the set of *bound occurences* of references in a term.

4.2.1. Paths and Occurences of Terms. We first proceed by the definition of an occurence of a term at a given path.

Definition 4 (Path and Occurence of a Term).

A path is an element of $\Pi = \texttt{List}(\{1,2\} \cup \textsc{Id})$ which locates a specific subterm of a term (see [13] for a standard definition of what they call a *position*).

The partial function $| : \Sigma \times \Pi \to \Sigma$ giving the term at path π within a term u, noted $u|_\pi$ is defined as:

1. $u|_{[\,]} = u$,
2. $\{\overrightarrow{x = e}\}|_{x_i::\pi} = e_i|_\pi$ if $x_i \in \{\overrightarrow{x}\}$,
3. $(e_1 \Diamond e_2)|_{i::\pi} = e_i|_\pi$ for $i \in \{1,2\}$ and $\Diamond \in \{\#, \centerdot\}$.

We speak of path π being *above* path π' in some term u if π is a *prefix* of π', that is, $u|_{\pi'}$ is within $u|_\pi$. We say that an *occurence* of a term v is at path π in u if $u|_\pi = v$.

For example, if $M \equiv \{a = b^0, d = e^0, c = \{a = d^0\} \centerdot \{c = a^0; e = a^1\}\}$ then the term b^0 is found at $M|_{[a]}$; term $\{a = d^0\}$ is found at $M|_{[c,1]}$ and term $\{c = a^0; e = a^1\}$ is found at $M|_{[c,2]}$.

4.2.2. Bound Occurences. At first sight, the occurence of x^n at path $\pi = [\pi_1, ..., \pi_m]$ in u is bound if a definition of x can be found amongst the terms $u|_{[\pi_1,...,\pi_p]}$ where $m - p \geq n$.

However, such a definition at path π' may be hidden to the occurence at π because there may exist a term below π' and above π that may introduce in *further* computations some additional bindings. This is the case for the selection operator where its l.h.s. is not a system. For instance, in a term $u \centerdot x^0$ (with $u \notin \textsc{Sys}$), x^0 cannot be bound because u might provide, in the course of its reduction, a definition of x closer to x^0 than any other definition provided by an enclosing term. This prevents the immediate reduction of x^0 until u becomes a system. This motivates the use of a flag to freeze the binding process in the stack of environments.

An *binding environment* is an element of $\mathcal{E}_{\texttt{def}} = \texttt{List}(\mathcal{P}(\textsc{Id}) \cup \{\star\})$. It is used to store the identifiers introduced by the enclosing terms at a given path. The constant $\star$ is the flag used to "stop" the process of looking for a definition in an upper term. ϵ ranges over $\mathcal{E}_{\texttt{def}}$.

Definition 5 (Set of Bound Occurences). The total predicate lookup : $\textsc{Ref} \times \mathcal{E}_{\texttt{def}} \mapsto \textsc{Bool}$ holds if a definition for a reference is found in an environment specified by the second argument of the function. The recursive search stops if a constant $\star$ is encountered and the predicate does not hold. The definition of lookup is:

$$\begin{array}{lcl}
\texttt{lookup}(x^n, [\,]) & = & \texttt{false} \\
\texttt{lookup}(x^n, \star :: t) & = & \texttt{false} \\
\texttt{lookup}(x^0, h :: t) & = & (x \in h) \vee \texttt{lookup}(x^0, t) \\
\texttt{lookup}(x^n, h :: t) & = & \texttt{lookup}(x^{n-1}, t)
\end{array}$$

The total function $\texttt{boundocc} : \Sigma \mapsto \texttt{List}(\Pi)$ computes the set of bound occurences in a term:

$$\texttt{boundocc}(u) = \texttt{boundocc}'(u, [\,], [\,])$$

where $\texttt{boundocc}'$ is the total function from $\Sigma \times \Pi \times \mathcal{E}_{\texttt{def}}$ to $\mathcal{P}(\Pi)$ specified as:

$$\begin{array}{lcl}
\texttt{boundocc}'(x^n,\ \pi,\ \epsilon) & = & \texttt{if}\,\texttt{lookup}(x, \epsilon)\,\texttt{then}\,\{\pi\}\ \texttt{else}\ \emptyset \\
\texttt{boundocc}'(\{\overrightarrow{x = e}\},\ \pi,\ \epsilon) & = & \bigcup_i \texttt{boundocc}'(e_i,\ \pi@[x_i],\ \{\overrightarrow{x}\} :: \epsilon) \\
\texttt{boundocc}'(e_1 \# e_2,\ \pi,\ \epsilon) & = & \texttt{boundocc}'(e_1,\ \pi@[1],\ \epsilon) \cup \texttt{boundocc}'(e_2,\ \pi@[2],\ \epsilon) \\
\texttt{boundocc}'(e_1 \,.\, e_2,\ \pi,\ \epsilon) & = & \texttt{boundocc}'(e_1,\ \pi@[1],\ \epsilon) \cup \\
 & & \qquad \texttt{boundocc}'(e_2,\ \pi@[2],\ \texttt{id}(e_1) :: \epsilon)
\end{array}$$

For example, the set of bound occurences of the term M given in page 11 is $\{[c, 1, a], [c, 2, c], [c, 2, e]\}$.

4.3. Clear Form and Stable Form

A term u is in a *clear form* relative to an environment ϵ, if there is no bound occurence in u relative to ϵ. Being free from any bound occurence, does not means that some "computation" cannot occur : an additional property is needed, being in *stable form.* Section 4.6 states the relationships between clear form, stable form and the evaluation process described in section 4.5.

Definition 6 (ϵ-Clear Form). A term u is said to be in ϵ-clear form ($\epsilon \in \mathcal{E}_{\texttt{def}}$) iff its set of bound occurences with respect to ϵ is empty, that is iff $\texttt{boundocc}'(u, [\,], \epsilon) = \emptyset$. We say that a term u is in *clear form* if it is in $[\,]$-clear form, i.e.: $\texttt{boundocc}(u) = \emptyset$.

Definition 7 (Stable Form). A term u is in *stable* form if $\texttt{stable}(u)$ is $\texttt{true}$, where the syntactic predicate $\texttt{stable}$ is defined by :

$$\begin{array}{lcl}
\texttt{stable}(x^n) & = & \texttt{true} \\
\texttt{stable}(\{\overrightarrow{x = e}\}) & = & \bigwedge_i \texttt{stable}(e_i) \\
\texttt{stable}(e_1 \# e_2) & = & \big((e_1 \notin \textsc{Sys}) \vee (e_2 \notin \textsc{Sys})\big) \wedge \texttt{stable}(e_1) \wedge \texttt{stable}(e_2) \\
\texttt{stable}(e_1 \,.\, e_2) & = & (e_1 \notin \textsc{Sys}) \wedge \texttt{stable}(e_1) \wedge \texttt{stable}(e_2)
\end{array}$$

4.4. Evaluating a Term

We now define the *evaluation process* of *reducing* a Σ-term. When a term cannot be further reduced, we say that it is in its *normal form.* The reduction process consists in the substitution of bound occurences with their definitions and the *simplification* (whenever possible) of subterms involving merge and selection operators. One of the remarkable properties of the amalgams is that the set of bound occurences *evolves* once substitutions of definitions to their bound occurences have been performed.

Indeed, the substitution of a definition to its bound occurence *creates* some new bound occurences, and requires consequently further substitutions (as consequences of the new bound occurences) and simplification.

4.4.1. Reduction Environment. The reduction of a subterm has to take into account the definitions appearing in an enclosing term. We define a *reduction environment* to keep track of this information. Since terms might be nested, a list structure is used for the environment, where each element is a *simple environment*, that is, a partial function which associates a name to its definition in the term.

Definition 8 (Simple Environment $\mathcal{E}$). A *simple environment* is a partial function $\sigma \in \mathcal{E} = \textsc{Id} \to \Sigma$. The formula $[x_1 \mapsto e_1, \ldots, x_n \mapsto e_n]$ is the function which associates the term e_i to the identifier x_i ($i \in [1, n]$), and which is undefined for any other identifier. For convenience, we use the notation $[\overrightarrow{x \mapsto e}]$. The domain of the environment σ is noted $\texttt{Def}(\sigma)$. By convention, σ ranges over $\mathcal{E}$.

Definition 9 (Reduction Environment $\mathcal{E}_{\mathrm{E}}$). A *reduction environment* ρ is a list of $\mathcal{E}$ environments: $\mathcal{E}_{\mathrm{E}} = \texttt{List}(\mathcal{E})$.

We define the *augmentation* $\{\overrightarrow{x = e}\} \uplus \rho$ of an environment ρ by the definitions of a system $\{\overrightarrow{x = e}\}$, as the concatenation $[\overrightarrow{x \mapsto e}] :: \rho$. By convention, ρ ranges over $\mathcal{E}_{\mathrm{E}}$.

A non empty reduction environment $\sigma :: \rho$ can be applied to an identifier $x \in \textsc{Id}$:

$$(\sigma :: \rho)(x) = \texttt{if}\ (x \in \texttt{Def}(\sigma))\ \texttt{then}\ \sigma(x)\ \texttt{else}\ \rho(x)$$

and $[\,](x)$ is undefined. We extend the application of an environment to an identifier to references:

$$\rho(x^n) = \big(\texttt{tl}^n(\rho)\big)(x)$$

We remark that the constant $\star$ is not used any more. The operational semantics defined in the next section handles the problem raised by a selection operator directly.

Note also that given a reduction environment ρ the expression $\texttt{tl}^n(\rho)$ retrieves the reduction environment of the definition of x which binds x^n. The expression $\texttt{tl}^n(\rho)(x)$ denotes the r.h.s. of the definition $x = e$ that binds x^n in ρ. The reduction environment $\texttt{tl}^n(\rho)$ is the reduction environment that applies for the reduction of e at its point of definition.

Definition 10 (Bound Predicate). The total predicate $\texttt{bound} : \mathcal{E}_{\mathrm{E}} \times \textsc{Ref} \mapsto \textsc{Bool}$ specifies when a reference is bound or not with respect to a reduction environment:

$$\begin{array}{lcl}
\texttt{bound}(\rho, x^n) & = & \texttt{rbound}(\texttt{tl}^n(\rho), x) \\
\texttt{rbound}([\,], x) & = & \texttt{false} \\
\texttt{rbound}(\sigma :: t, x) & = & \texttt{if}\ (x \in \texttt{Def}(\sigma))\ \texttt{then true else rbound}(t, x)
\end{array}$$

4.5. Operational Semantics

An operational semantics for the amalgams, following Kahn's natural semantics [24] style, is given through the specification of a reduction relation $\twoheadrightarrow \subseteq \mathcal{E}_{\mathrm{E}} \times \Sigma \times \Sigma$. The relation $\twoheadrightarrow$ is the least relation satisfying the rules given in Fig. 6. The next section gives some properties of the semantics and the section 4.7 comments all the reduction rules.

The formula $\rho \vdash u \twoheadrightarrow u'$ means that expression u is reduced in u' in a single step in the ρ environment (big-steps semantics). We say that a term u *evaluates* to u' if $[\,] \vdash u \twoheadrightarrow u'$. For simplicity, we write $u \twoheadrightarrow u'$ for $[\,] \vdash u \twoheadrightarrow u'$) and equivalently we say that the value of u is u'.

x_1^n
$$\frac{\rho' = \mathtt{tl}^n(\rho) \quad \rho' \vdash \rho'(x^n) \twoheadrightarrow e \quad \rho \vdash e \twoheadrightarrow e'}{\rho \vdash x^n \twoheadrightarrow e'} \quad \mathtt{bound}(\rho, x^n)$$

x_2^n
$$\frac{}{\rho \vdash x^n \twoheadrightarrow x^n} \quad \neg\, \mathtt{bound}(\rho, x^n)$$

$\{\}$
$$\frac{\{\overrightarrow{x = e}\} \uplus \rho \vdash e_1 \twoheadrightarrow e_1' \quad \dots \quad \{\overrightarrow{x = e}\} \uplus \rho \vdash e_n \twoheadrightarrow e_n'}{\rho \vdash \{\overrightarrow{x = e}\} \twoheadrightarrow \{\overrightarrow{x = e'}\}}$$

$\#_1$
$$\frac{\rho \vdash u \twoheadrightarrow u' \quad \rho \vdash v \twoheadrightarrow v'}{\rho \vdash u\#v \twoheadrightarrow u'\#v'} \quad (u' \notin \textsc{Sys}) \vee (v' \notin \textsc{Sys})$$

$\#_2$
$$\frac{\rho \vdash u \twoheadrightarrow \{\overrightarrow{x = e}\} \quad \rho \vdash v \twoheadrightarrow \{\overrightarrow{x' = e'}\} \quad \rho \vdash \{\overrightarrow{x = e}, \overrightarrow{x' = e'}\} \twoheadrightarrow w}{\rho \vdash u \# v \twoheadrightarrow w}$$
when $\{\overrightarrow{x}\} \cap \{\overrightarrow{x'}\} = \emptyset$

$\centerdot_1$
$$\frac{\rho \vdash u \twoheadrightarrow u'}{\rho \vdash u \,\centerdot\, v \twoheadrightarrow u' \,\centerdot\, v} \quad u' \notin \textsc{Sys}$$

$\centerdot_2$
$$\frac{\rho \vdash u \twoheadrightarrow u' \quad u' \uplus \rho \vdash v \twoheadrightarrow v'}{\rho \vdash u \,\centerdot\, v \twoheadrightarrow v'} \quad u' \in \textsc{Sys}$$

Figure 6. A big-step semantics of the amalgams through the specification of a reduction $\twoheadrightarrow$.

4.6. Properties of the Semantics

Definition 11 (Normal Form and Stuck Term). We say that a term u is in ρ-normal form iff $\rho \vdash u \twoheadrightarrow u'$ implies that $u \equiv u'$, i.e. when a term only reduces to itself. A term is in normal form if it is in $[\,]$-normal form. A term is *ρ-stuck* there is no term u' such that $\rho \vdash u \twoheadrightarrow u'$; it is stuck iff it is $[\,]$-stuck.

Definition 12 (Term equivalence modulo a system permutation). Let α be a permutation of $\{1, \ldots, n\}$. We define the permutation of a system $\{x_1 = e_1, \ldots, x_n = e_n\}$ by:

$$\alpha\big(\{x_1 = e_1, \ldots, x_n = e_n\}\big) \equiv \{x_{\alpha 1} = e_{\alpha 1}, \ldots, x_{\alpha n} = e_{\alpha n}\}$$

We note $s \equiv_p s'$ if there exists a permutation α such that $s \equiv \alpha(s')$.

The semantics described in Fig. 6 ensures the following properties (see the appendix for the proofs):

THEOREM 1 (DETERMINISM OF THE SEMANTICS) *The reduction relation is deterministic, that is, if $u \twoheadrightarrow u'$ and $u \twoheadrightarrow u''$ then $u' \equiv u''$.*

THEOREM 2 (SEMANTICS REDUCES TO NORMAL FORM) *The reduction relation reduces a term to normal form, that is, if $u \twoheadrightarrow u'$ then u' is in normal form.*

THEOREM 3 (CLEAR, STABLE FORM AND NORMAL FORM) *Term u is in normal form iff u is clear and stable*

THEOREM 4 (COMMUTATIVITY IN A SYSTEM) *If $u \in$ SYS and $\rho \vdash u \twoheadrightarrow u'$ then $\rho \vdash \alpha(u) \twoheadrightarrow \alpha(u')$.*

THEOREM 5 (COMMUTATIVITY OF THE MERGE) *Let terms u and v such that $\rho \vdash u \twoheadrightarrow u'$ and $\rho \vdash v \twoheadrightarrow v'$ and $u', v' \in$ SYS, and $\rho \vdash u \,\#\, v \twoheadrightarrow w$ and $\rho \vdash v \,\#\, u \twoheadrightarrow w'$. Then $w \equiv_p w'$.*

Together, these results tell us that:

1. $\twoheadrightarrow$ is really a partial function,
2. normal forms are fixed points of this function,
3. reducing a term results in a normal form (if one exists),
4. the merge is symmetric and the ordering of equations in a system does not matter.

4.7. Reduction Rules

The reduction rules of Fig. 6 are of four different kinds: two rules for the reduction of a reference, a merge or a selection operator and one rule for the reduction of a system.

4.7.1. Substitution of a Bound Reference. Rules "x^n" are the *substitution* rules of the semantics. The first rule states that a reference $\bar{x}^n$ in an environment ρ that is bound in to a definition in an environment $\rho' = \mathtt{tl}^n(\rho)$ has its definition:

1. first reduced to a term e in ρ'-normal form and then,
2. term e is reduced in ρ-normal form.

In other words: a definition is first totally reduced in the environment at the point of definition before substitution and further reduction at the point of reference.

The second rule states that a free reference remains unchanged.

4.7.2. Reduction of a System. The reduction of a system is defined by the rule "{}". It consists in the reduction of the r.h.s. of every equation, in the environment augmented with the definitions of the current system. The semantics that is proposed here is *strict.*

We remark that the rule, by definition, does not allow the same identifier to appear more than once in the l.h.s. of a definition in a system. Indeed, given a term $u \in \Sigma$ all subterms v of u which belong to SYS satisfy this property (because the definition of SYS). In addition, the only reduction rule that creates a new system during the reduction process is rule "$\#_2$" which can be trigerred only if the the new system assumes this property.

4.7.3. Reduction of a Merge. The reduction of a term having a merge operator is handled by rules "$\#_1$" and "$\#_2$". The reduction of $e \equiv u \# v$ in an environment ρ depends whether the ρ-normal form of u and v are both systems or not. Let u' (respectively v') be the ρ-normal form of u (respectively v):

- If u' and v' are both elements of SYS, then rule "$\#_2$" applies and the result of e in ρ is the system s consisting of all the equations coming from u' and v', provided that no identifer appears at the same time in u' and v'.
- If u' or v' is not a system, then rule "$\#_1$" applies and the result of e in ρ is $u' \# v'$.

We remark that rule "$\#_2$" may only be applied to a term e if no identical identifiers appear at the same time in u and v. If this condition does not hold, the reduction process is *stuck.*

4.7.4. Reduction of a Selection. The reduction of a term having a selection operator is handled by rules "$\centerdot_1$" and "$\centerdot_2$". The reduction of $e \equiv u \centerdot v$ in an environment ρ depends whether the ρ-normal form of u is a system or not. Let u' be the ρ-normal form of u:

- If u' is an elements of SYS, then rule "$\centerdot_2$" applies and the result of e in ρ is the result of the $(u' \uplus \rho)$-normal form of v, that is, the result of the reduction of v in the definitions brought by the system u'.

- If u' is not a system, then rule “$\bullet_1$” applies and the result of e in ρ is $u' \bullet v$.

4.8. Examples of Stuck Terms and of Reductions

4.8.1. Examples of Stuck Terms. We gives here three examples of stuck terms:

1. $\{a = x^0\} \# \{a = y^0\}$ is a stuck term because rule “$\#_2$” cannot be applied because the sets of defined identifiers of each sub-term of $\#$ are not disjoint. And no other rule applies.
2. $\{x = \{a = x^0\}\}$ is stuck because it cannot be reduced to a finite term.
3. $\{x = x^0\}$ is stuck because the relation $\twoheadrightarrow$ is the least relation satisfying the rules in Fig. 6. As a matter of fact, $\texttt{bound}(\big[[x \mapsto x^0]\big], x^0)$ holds and then we can only try to apply rule “x_1^n”. The application of this rule has conclusion $\big[[x \mapsto x^0]\big] \vdash x^0 \twoheadrightarrow x^0$ but can be applied only if we prove hypothesis $\big[[x \mapsto x^0]\big] \vdash x^0 \twoheadrightarrow x^0$. Because the minimality of $\twoheadrightarrow$, we cannot derive $\big[[x \mapsto x^0]\big] \vdash x^0 \twoheadrightarrow x^0$ and then x^0 is $\big[[x \mapsto x^0]\big]$-stuck. As a consequence $\{x = x^0\}$ is stuck.

4.8.2. Examples of Reductions. We now give some examples of succesful reductions.

The reduction of $\{a = b, b = c, c = d\}$ gives $\{a = d, b = d, c = d\}$. Indeed, let ρ denotes the environment $\big[[a \mapsto b, b \mapsto c, c \mapsto d]\big]$. Then $\rho \vdash d^0 \twoheadrightarrow d^0$ by rules x_2^n. We have $\rho(c^0) = d^0$ and then $\rho \vdash c^0 \twoheadrightarrow d^0$ by rule x_1^n. In addition, we have $\rho(a^0) = b^0$ and $\rho \vdash b^0 \twoheadrightarrow d^0$ because $\rho(b^0) = c^0$ and we have seen that $\rho \vdash c^0 \twoheadrightarrow d^0$. In consequence, application of rule $\{\}$ ensures this result. The proof tree is:

$$\dfrac{\mathcal{A} \qquad \mathcal{B} \qquad \mathcal{C}}{[\,] \vdash \{a = b^0, b = c^0, c = d^0\} \twoheadrightarrow \{a = d^0, b = d^0, c = d^0\}}\,\{\}$$

Let σ denotes the function $[a \mapsto b^0, b \mapsto c^0, c \mapsto d^0]$, then $\mathcal{A}$ is the following proof-tree (remark that we always have $[\sigma]$ as environment because $\texttt{tl}^0([\sigma]) = [\sigma]$):

$$\dfrac{\dfrac{\dfrac{}{[\sigma] \vdash \sigma(c^0) \twoheadrightarrow d^0}\,x_2^n \quad \dfrac{}{[\sigma] \vdash d^0 \twoheadrightarrow d^0}\,x_2^n}{[\sigma] \vdash \sigma(b^0) \twoheadrightarrow d^0}\,x_1^n \qquad \dfrac{}{[\sigma] \vdash d^0 \twoheadrightarrow d^0}\,x_2^n}{[\sigma] \vdash b^0 \twoheadrightarrow d^0}\,x_1^n$$

We remark that $\sigma(b^0) = c^0$ and $\sigma(c^0) = d^0$. We have $\mathcal{B}$:

$$\dfrac{\dfrac{}{[\sigma] \vdash \sigma(c^0) \twoheadrightarrow d^0}\,x_2^n \quad \dfrac{}{[\sigma] \vdash d^0 \twoheadrightarrow d^0}\,x_2^n}{[\sigma] \vdash c^0 \twoheadrightarrow d^0}\,x_1^n$$

and $\mathcal{C}$ is:

$$\dfrac{}{[\sigma] \vdash d^0 \twoheadrightarrow d^0}\,x_2^n$$

Here is a more complex example, involving the nesting of systems. For the sake of brievety, in the following explanations the statement "u is in $[[x_1 \mapsto e_1, x_2 \mapsto e_2, x_3 \mapsto e_3, \dots], [x'_1 \mapsto e'_1, x'_2 \mapsto e'_2, \dots], \dots]$-normal form" is shortened as "u is in $[[x_1, x_2, x_3, \dots], [x'_1, x'_2, \dots], \dots]$-normal form".

We start from system:

$$\{a = b^0, b = c^0, r = \{c = z^0, b = y^0, v = a^1\}\}$$

the reduction of the bound reference a^1 at occurence $[r, v]$ to its $[[c, b, v], [a, b, r]]$-normal form requires the reduction of b^0 (*the definition* of the bound reference) in $[[a, b, r]]$-normal form, which itself requires the reduction of c^0 in $[[a, b, r]]$-normal form. Since c^0 is already in $[[a, b, r]]$-normal form, the value of c^0 is c^0; the value of b^0 in $[[a, b, r]]$-normal form is also c^0, that cannot be further reduced. Finally, the value of a^1 in $[a, b, r]$-normal form is c^0. *But c^0 is not in $[[c, b, v], [a, b, r]]$-normal form.* In this environment, it is a bound reference, with z^0 as definition. Consequently, the reduction of c^0 requires the reduction of z^0 in $[[c, b, v], [a, b, r]]$-normal form, which is z^0 which is already in $[[c, b, v], [a, b, r]]$-normal form.

Then, *the final result* of the reduction of a^1 at occurence $[r, v]$ to its $[[c, b, v], [a, b, r]]$-normal form is z^0

5. Expressive Power of the Amalgams

We illustrate the expressive power of the amalgams through expressions of the amalgams. In the next section, we describe a naïve coding of boolean functions; in section 5.3, we add to the core formalism a conditional operator and describe a coding of recursive functions; then, in section 5.5 we describe how the amalgams can be integrated into a declarative framework and show how an object-oriented programming style can be achieved.

5.1. Coding Boolean Functions in Amalgams

As a first example, we show how a boolean function can be coded using amalgams (inspired by [10, pp 66]). We consider the example of the negation of a boolean value. Boolean values are coded as the references true^0 and false^0.

The computation of the negation is done in two steps. First, The boolean value is translated, using a selection, into a free reference: true^0 becomes f^0 and false^0 becomes t^0. The translated value is accessible through $first$. Then the previously computed value is translated back, using again a selection: t^0 becomes true^0 and f^0 becomes false^0. The translated value is accessible through *second.*

The program corresponding to the coding of the boolean values in amalgams, where $\ulcorner bool \urcorner$ corresponds to the free reference coding the boolean value, is:

$$\begin{array}{l}\{value = \ulcorner bool \urcorner, \\ \quad not = \{first = \{true = f^0, false = t^0\} \mathbin{.} value^2, \\ \qquad\qquad second = \{t = true^0, f = false^0\} \mathbin{.} first^1 \\ \qquad\quad \} \mathbin{.} second^0 \\ \}\end{array}$$

The result of the negated value, defined through *value*, is accessible through the definition of *not*. The reader interested in the translation scheme should refer to [35] for other connectors. The result of the evaluation of $\ulcorner not(true) \urcorner$ is, as expected:

$$\{value = true^0, not = false^0\}$$

5.2. *Adding a Conditional Operator*

We introduce a conditional operator $\texttt{if}(c, t, f)$ to the set of amalgam terms Σ. The condition is a boolean in the style just introduced.

Definition 13 ($\Sigma_{\texttt{if}}$). The set $\Sigma_{\texttt{if}}$ of amalgam terms extended with a conditional operator, is the smallest set satisfying the three rules of definition 2 (where Σ is replaced by $\Sigma_{\texttt{if}}$) with the additional rule:

4. if e_1, e_2 and e_3 are elements of $\Sigma_{\texttt{if}}$, then $\texttt{if}(e_1, e_2, e_3)$ is an element of $\Sigma_{\texttt{if}}$.

All the functions previously defined on Σ are naturally lifted to $\Sigma_{\texttt{if}}$.

The semantics of the new operator is given in Fig. 7. The treatment of the conditional is lazy (we only require that the branch selected by the conditional to have a ρ-normal form).

$$\texttt{if}_{\texttt{true}} \quad \frac{\rho \vdash c \twoheadrightarrow true^0 \quad \rho \vdash t \twoheadrightarrow t'}{\rho \vdash \texttt{if}(c, t, f) \twoheadrightarrow t'}$$

$$\texttt{if}_{\texttt{false}} \quad \frac{\rho \vdash c \twoheadrightarrow false^0 \quad \rho \vdash f \twoheadrightarrow f'}{\rho \vdash \texttt{if}(c, t, f) \twoheadrightarrow f'}$$

$$\texttt{if} \quad \frac{\rho \vdash c \twoheadrightarrow c'}{\rho \vdash \texttt{if}(c, t, f) \twoheadrightarrow \texttt{if}(c', t, f)} \quad c \notin \{true^0, false^0\}$$

Figure 7. The semantic rules of the reduction of the conditional operator.

One may wonder whether such a conditional form could be implemented using only the core formalism of the amalgams or not. The definition of the semantics given here does not allow such a form because it requires a term to be fully reduced to get its value (big-step semantics). One of the expression of the conditional may not have a normal form and therefore the computation would not terminate.

However, we have given in [35] a small-step semantics which accommodates partial terms. For instance, in this semantics, the following expression

$$\{a = \{b = a^1, c = d^0\} \centerdot c^0\}$$

has the value $\{a = d^0\}$. In the semantics presented in this paper this term has no normal form. In the small-steps semantics of [35], a conditional form can be defined in the amalgams but the semantics is intricate and complicates unnecessarily the presentation of the formalism.

5.3. Coding the Arithmetic in Amalgams

We describe how numeric recursive functions can be translated in $\Sigma_{\mathtt{if}}$. This example shows the formal expressive power of the amalgams. We restrict ourselves to the class of *total functions*. Indeed, the translation scheme that we propose doesn't ensure the non-termination of the computation of the amalgam associated with the application of a function on arguments that do not belong to the definition domain. The definition of the primitive recursion that we use is slightly different from the one usually used, but is equivalent. The definition adopted here is easy to translate. The function $\boldsymbol{U_i^p}$ represents the i^{th} projection, $\boldsymbol{S}$ the successor function and $\boldsymbol{Z}$ the function that returns zero.

5.3.1. Definability using Amalgams. *Definition 14 (Representation of integers in $\Sigma_{\mathtt{if}}$).* For each $n \in \mathbb{N}$, a term $\ulcorner n \urcorner \in \Sigma_{\mathtt{if}}$ is defined in the following way:

$$\begin{aligned} \ulcorner 0 \urcorner &\equiv \{b = \mathit{true}^0\} \\ \ulcorner n+1 \urcorner &\equiv \{p = \ulcorner n \urcorner, b = \mathit{false}^0\} \end{aligned}$$

Definition 15 (Definability using amalgams). Let φ be a numerical function of arity p. We say that φ is *definable using amalgams*, or a-definable, if there exists a system s such that:

$$\forall \vec{n}, \{a1 = \ulcorner n_1 \urcorner, \ldots, \mathrm{ap} = \ulcorner n_p \urcorner\} \centerdot s \twoheadrightarrow \{\mathit{value} = \ulcorner \varphi(\vec{n}) \urcorner, \ldots\}$$

In this expression, we have abbreviated $n_1, \ldots, n_p$ by $\vec{n}$; the " ... " in the result means that s is a system that must have at least a definition for *value* and that can have additional definitions if necessary. The names $a1, a2, \ldots$ are, by convention, names given to the arguments for φ and that are not used elsewhere (s may depend on those names).

5.3.2. Coding of Numeric Functions. The basic numeric functions are a-definable, using the following terms:

$$\boldsymbol{U_i^p} \equiv \{\mathit{value} = ai\}$$

$$\begin{array}{rcl}\boldsymbol{S} & \equiv & \{value = \{p = a1, b = false^0\}\} \\ \boldsymbol{Z} & \equiv & \{value = \ulcorner 0 \urcorner\}\end{array}$$

The numerical function P such that $P(n+1) = n$ is a-definable:

$$\boldsymbol{P} \equiv \{value = a1 \,.\, p\}$$

The functions a-definable are closed under composition. Indeed, let $\phi, \psi_1, \dots, \psi_m$ be a-defined respectively by the terms $G, H1, \dots, H\mathrm{m}$, then:

$$\varphi(\vec{n}) = \phi(\psi_1(\vec{n}), \dots, \psi_m(\vec{n}))$$

is defined by:

$$\begin{array}{rcl}\{value & = & args \,.\, G, \\ args & = & \{a1 = H1 \,.\, value, \dots, a\mathrm{m} = H\mathrm{m} \,.\, value\}\}\end{array}$$

We are now able to prove that the a-definable functions are closed under primitive recursion. The intuition behind the definition of a recursive calling scheme is to create an expression corresponding to the term resulting from the translation of the function, but with no bound reference. Afterwards, definitions are given to the free references, at the moment of the function-call.

We detail the scheme for a function φ of two variables, using the conditional and the recursive scheme. Let φ be the function defined by $\varphi(0, y) = \phi(y)$ and $\varphi(x+1, y) = \psi(x, \varphi(x, \psi_1(y)))$ with ϕ a-defined by F, ψ a-defined by G and ψ_1 a-defined by H. Then φ is a-defined by (x and y are the names of the arguments used for φ):

$$\begin{array}{rcl}\{parameter & = & \{x = px^0, y = py^0\}, \\ px & = & x^1 \,.\, p^0, \\ py & = & \{x = y^2\} \,.\, H, \\ fct & = & \mathtt{if}(x^0 \,.\, b^0, \\ & & \quad \{x = a0^1\} \,.\, F, \\ & & \quad \{x = x^1, y = arg^0 \,.\, fct^0\} \,.\, G), \\ value & = & (\{arg = parameter^0\} \,.\, (\{x = x^3, y = y^3\} \,.\, fct^0))\}\end{array}$$

The way that we have defined the translation of the primitive recursion, it should be obvious to the reader that it does not matter if we increment or decrement x in the scheme of the primitive recursion: this leads naturally to an implementation of the minimization.

5.3.3. Example of the Addition. The definition of the addition:

$$\begin{array}{lcl}add(0, n) & = & n \\ add(n+1, m) & = & add(n, S(m))\end{array}$$

takes the form of a primitive recursion where we have $\phi = \boldsymbol{U}^1_1, \psi = \boldsymbol{U}^2_2, \psi_1 = \boldsymbol{S}$. The translation of $add(2, 1)$ is the term defined in figure 8. The identifiers A and F are meta-variables used to describe the terms in pieces; they are not elements of Id.

$$\begin{aligned}
\mathrm{A} &\equiv \{x = \{b = \mathit{false}^0, p = \{b = \mathit{false}^0, p = \{b = \mathit{true}^0\}\}\},\\
&\qquad y = \{b = \mathit{false}^0, p = \{b = \mathit{true}^0\}\}\}\\
\mathrm{F} &\equiv \{\mathit{parameter} = \{a5 = px^0, a4 = py^0\},\\
&\qquad px = (a5^1 \,.\, p^0),\\
&\qquad py = (\{a6 = a4^2\} \,.\, \{b = \mathit{false}^0, p = a6^1\}),\\
&\qquad \mathit{fct} = \texttt{if}((a5^0 \,.\, b^0), a4^0, (\mathit{arg}^0 \,.\, \mathit{fct}^0)),\\
&\qquad \mathit{value} = (\{\mathit{arg} = \mathit{parameter}^0\} \,.\, (\{a5 = x^0, a4 = y^0\} \,.\, \mathit{fct}^0))\}\\
\mathit{add}(2,1) &\equiv \mathrm{A} \,.\, \mathrm{F}
\end{aligned}$$

Figure 8. The term corresponding to the addition of 2 to 1 in the arithmetic coded using amalgams. These terms have been slightly modified to improve readability.

5.4. Adding Constants and Operations on Constants

We extend the language $\Sigma_{\texttt{if}}$ of the amalgams to allow the manipulation of integers and operations on integers.

Definition 16 ($\Sigma_{\texttt{if},\mathbb{N}}$). The set $\Sigma_{\texttt{if},\mathbb{N}}$ of amalgam terms extended with conditional, integers and operations on integers is the smallest set satisfying the four rules of definition 13 (where $\Sigma_{\texttt{if}}$ is replaced by $\Sigma_{\texttt{if},\mathbb{N}}$) with the additional rules:

5. a *constant* n (element of $\mathbb{N}$) is an element of $\Sigma_{\texttt{if},\mathbb{N}}$,
6. if $e_1, ..., e_n$ are elements of $\Sigma_{\texttt{if},\mathbb{N}}$ and f is an n-ary functional symbol, then the application of f to the $e_1, ..., e_n$, noted $f(e_1, ..., e_n)$ is an element of $\Sigma_{\texttt{if},\mathbb{N}}$.

and nothing else is an element of $\Sigma_{\texttt{if},\mathbb{N}}$. All the functions previously defined on $\Sigma_{\texttt{if}}$ are naturally lifted to $\Sigma_{\texttt{if},\mathbb{N}}$.

By convention, c ranges over the set of constants and f ranges over the set of functional operators. We abbreviates $f(e_1, ..., e_n)$ by $f(\overrightarrow{e})$.

We add to the set of rules defined in Fig. 6 and Fig. 7 three rules:

- a rule for the reduction of a constant,
- two rules, similar to the δ-rule of the λ-calculus, for the reduction of any n-ary functional symbol.

These two new rules allow us to manipulate constants and arithmetical operators with domain and codomain in $\mathbb{N}$ rather than having to code them explicitly in the amalgams. The reduction rules are given in Fig. 9.

5.4.1. Reduction of a Constant.

The rule defining the value of a constant is straightforward: the value of a constant is the constant itself.

$$\mathbb{N} \qquad \frac{}{\rho \vdash c \twoheadrightarrow c} \quad c \in \mathbb{N}$$

$$\delta_{\mathbb{N}} \qquad \frac{\rho \vdash e_1 \twoheadrightarrow c_1 \quad \dots \quad \rho \vdash e_n \twoheadrightarrow c_n \quad f(c_1, ..., c_n) = c'}{\rho \vdash f(\overrightarrow{e}) \twoheadrightarrow c'} \quad \forall i, c_i \in \mathbb{N}$$

$$\delta_{\neg\mathbb{N}} \qquad \frac{\rho \vdash e_1 \twoheadrightarrow c_1 \quad \dots \quad \rho \vdash e_n \twoheadrightarrow c_n}{\rho \vdash f(\overrightarrow{e}) \twoheadrightarrow c'} \quad \exists i \in [1, n], e'_i \notin \mathbb{N}$$

Figure 9. The semantic rules of the reduction of constants and any n-ary functional symbol.

5.4.2. reduction of an n-ary Functional Symbol. The reduction of an expression $u \equiv f(e_1, ..., e_n)$ involving an n-ary operator f requires first to reduce each operand e_i to its ρ-normal form e'_i. If each e_i is reduced to a constant, then rule "$\delta_{\mathbb{N}}$" is applied and the result of reducing u in ρ is the result of the application of the function $f_{\mathbb{N}}$ (with signature $\mathbb{N}^n \mapsto \mathbb{N}$) to the constants. If one of the terms e'_i is not a constant, rule "$\delta_{\neg\mathbb{N}}$" applies and the result of reducing u in ρ is the term $f(e'_1, ..., e'_n)$.

5.5. *Object-Oriented Programming Style*

Amalgams have been initially developed to structure 81/2 programs [19, 30]. 81/2 is a declarative language defining streams by equations (each variable, defined by an equation, represents a succession of values in time). With the features of the amalgams, it is possible to adopt an object-oriented programming style by considering fragments of programs and their composition. We detail in this section an example of such a programming style.

The notion of system allows the definition of *environments*. The composition of systems by merging enables the definition of *extensible environments*. Moreover, open expressions and the ability to complete these expressions with definitions, allow the design of a programming style similar to the one found in object-oriented languages. It is possible to design and compose fragments of programs following a *class* structure and using a mechanism similar to the *class instantiation* mechanism found in those languages. We describe, through an example, how to "emulate" a programming style close to that of object-oriented languages.

A system represents both the notions of *class* and *class constructor* that are used to create an instance of a class. The arguments required by the constructor are the free variables of the system. The *instantiation* of a class corresponds to the merge of the system with the arguments required by the constructor. Additional definitions may be added to an object, through the use of the merge operator, and corresponds to the *inheritance* mechanism.

A closed system (with no free references) corresponds to an object, as in object-oriented languages. The object model that we are defining is the *embedding based* model, where all the information about an object is in the object itself. It is obvious that our “model” lacks all the high-level mechanism of protection and encapsulation proposed by classical object-oriented languages.

To illustrate this programming style in 81/2, we define, following an object-oriented programming style, a model of the trajectory of a planet in a circular uniform movement around a star. The star itself is following a rectilinear uniform movement. First, we define a class *Mobile* of moving objects. The *Mobile* class is represented by a system with two free references: *initial* which represents the initial position of the object, and *dp* which represents the elementary movements of the object. With these free references, which are vectors of two elements corresponding to the *Ox* and *Oy* axis, the system *Mobile* defines a position:

$$Mobile = \{position = initial^0\ \texttt{fby}\ \$Mobile\ .\ position + dp^0\},$$

The *position* field of a *Mobile* is a stream of values representing the trajectory of the mobile along time. The $ operator gives access to the previous value in a stream; `fby` is the analog for infinite streams of the `cons` operator on lists.

Once *Mobile* is defined, we can define a new class of objects: mobile objects with a uniform speed. The class *U–Trajectory* awaits an initial position (required by the *Mobile* class from which it inherits) and a vector *speed* to instance itself:

$$U\!-\!Trajectory \ = \ Mobile \,\#\, \{dp = speed^0\}$$

The system *U–Trajectory* is a system with all the definitions of the *Mobile* system because it is a system extended by the definitions of *dp* used to compute the elementary movements with a uniform trajectory (we suppose that *speed* will be a constant equal to the difference between two successive values of the stream). The merge operation combines these two systems and binds the free reference *dp* of the anonymous system with the definition of *U–Trajectory*.

We follow with this example by using *Mobile* to represent the circular trajectory of a planet around a star which follows a uniform trajectory. The class *C–Trajectory* awaits a radius, a center and an angular speed:

$$\begin{aligned}
&C\!-\!Trajectory \ = \\
&\quad Mobile \,\#\, \{initial = \{center^0\ .\ 0, angle^0 + center^0\ .\ 1\},\\
&\qquad\qquad dp = \{dx, dy\},\\
&\qquad\qquad t = \$t + angular\_speed,\\
&\qquad\qquad dx = \ldots \text{ formula involving } sin \text{ and } cos \text{ of } t \ldots\\
&\qquad\qquad dy = \ldots\}
\end{aligned}$$

Now, we just have to instantiate the classes to describe the movement of a planet around a star in a uniform translation:

$$\begin{aligned}
Star \ &= \ U\!-\!Trajectory \,\#\, \{speed = \{1.0, 1.0\}, initial = \{0.0, 0.0\}\},\\
Planet \ &= \ C\!-\!Trajectory \,\#\, \{angle = 1.0, center = Star\ .\ position\}
\end{aligned}$$

6. Related Works

6.1. Environments as First-class Values.

The notion of binding is essential in the approach taken by the Pebble language [7] to design modules and interfaces. A binding is an association $(name, value)$, the binding itself being a value. The scope of the bindings is limited by the classical `LET`, `IN` and `WHERE` operators. An environment is defined as a set of bindings. Sets of bindings may be combined using the ";" construction such that $B_1; B_2$ defines the set of bindings appearing in B_1 and B_2.

Pebble bindings do not allow the definition of "recursive" sets of binding, like the expression: $\{a = 1, b = c^0\} \# \{c = a^0, d = b^0\}$ where each free reference in an environment is solved by the definitions of another environment. Furthermore, redefinitions of bindings overlap previous definitions, whereas they are still reachable in the amalgams. Pebble bindings are similar to the *data parameters* of [26] but suffer from the same restrictions (see below).

Symmetric Lisp [22] is a concurrent language allowing the definition of environments through the explicit operator `ALPHA`. It is possible to extend these environments but the extension can only take place between an open environment (defined using the `OPEN-ALPHA` form). After this first step towards the gathering of definitions into environments, Jagannathan defines the two explicit operators `reflect` and `reify` to translate a data structure into an environment and an environment into a data structure.

These operators are reminiscent of the reflexive languages. In these languages, it is possible to access to the *interpreter* of a program, using `reflect` and `reify`, to modify the interpreter's structures of the running program. In this approach, the environments are not denotable [14, 45]. They are now first-class values (an environment is denotated by a closure and reified into a *record*) but, unlike the approach followed by Pebble and the amalgams, they are distinguished from other data structures. Operators defined on data structures cannot operate on environments. Therefore they require two explicit operators that we keep implicit. Furthermore, redefinitions of bindings in an environment cannot be accessed.

6.2. Formalization of Incremental Computation.

Lamping initiated the work on *parameterization* [26]. A system (in its common definition) is parameterized when the value of the outputs depends from one or several of its inputs. Lamping proposes, in addition to the classical lexical binding, an environment based binding, using a special form of variables: *data parameters.* A data parameter is declared with the explicit operator `data: x` and the value of `x` is given by a `supply` operation. The composition of environments is possible through the $\circ$ operator.

No difference between lexical and dynamic binding is made by the amalgams. Their late lexical binding strategy allows the binding of lexical references and the dynamic resolution of free references. Furthermore, redefinitions are accessible in the amalgams whereas the composition of environments in Lamping's system overlaps redefinitions.

In [27] a new kind of variable is introduced: a *quasi-static* variable. The special form `qs-lambda` is used to define a quasi-static procedure that represents a piece of parameterized code. The special form `resolve1` is used to bind a quasi-static variable of a quasi-static procedure to a definition. Actually, a quasi-static variable is a pair $(name, variable)$, the variable being subject to α-conversion whereas the name is not.

Our approach is simpler: a system is implicitly parameterized by its free references. No distinction is made between two different types of variables, only references are manipulated and resolution of free references is implicitly done by a capture mechanism.

The λ-calculus is a well known formalism and is heavily used to model features of todays programming languages. The $\lambda\mathbf{C}$-calculus [28] is a tentative step towards the formalization of the incremental construction of programs. To reach this goal, the notion of name (a *context*) is introduced in the λ-calculus. Nevertheless, this introduction is not trivial: the interaction between β-substitution and *hole filling* (the name capture mechanism defined to substitute a name with an expression) is not straightforward. A solution to the problems encountered is found in the separation of the domains of β-substitution and hole-filling. Therefore, contexts and λ-terms do not share the same name-space. Another solution to the same problem can be found in [20] using an explicit typing system.

The approach followed by the amalgams is different. Since we rely on a uniform system (we only have a single kind of reference, and a single kind of substitution policy), we do not have to solve the problem of interactions between β-substitution and hole-filling. Besides technical matters, our resolution of the problem is also different: we rely on an implicit approach where free references are implicitly abstracted when appearing into the scope of a definition whereas for the $\lambda\mathbf{C}$-calculus, contexts need to be explicitly abstracted and solved.

6.3. Mixins and Modularity.

After its first introduction in the LISP community [25, 39] to represent an abstract subclass, the notion of *mixin* has been widespread in the object-oriented community to denote a class where some components of different nature (types, exceptions, methods, slots, ...) are not defined. The definition of such component is *deferred* and can effectively be used for instantiation only when combined with some other class which provides the missing definitions [6, 29, 15]. This general definition can be seen as independent of the object-oriented framework and can be formulated in the more general context of module composition [5, 15].

The mixin approach put the emphasis on the composition of mixins (rather than on the instantiation of deferred components). In the field of module construction, the main operator is the binary *merge* operator: if M_1 and M_2 are two mixins, then $M_1{+}M_2$ is a mixin where some definitions of M_1 are associated with the corresponding declarations in M_2 and conversely. This operator is commutative and is defined whenever no components are defined on both sides [2]. Note that this approach enables the recursive definition of components split over several modules [15], which is not possible with regular modules (like in Standard ML for example). Additional operators (*restrict*, *hide*, *freeze*, *rename*, *functional composition*, ...) are defined to manage name clashes, redefinitions, access to a component, etc.

A *mixin module* is very close to a system: deferred components are free references in the amalgams; the *merge* operator corresponds to the # operator; functional composition $M_1 \circ M_2$, where definitions in M_1 are used in M_2 and not conversely (this is a one-way merge) is similar to the selection operator. Moreover, the *freeze* operator (that allows the building of a module independently of the redefinition of some components) is not required in the amalgams since binding cannot be redefined: once a reference is bound, it is substituted by its definition. Operators like *hide* and *restrict* that are used to manage name clashes are not considered in the core definition of the amalgams.

To our knowledge, the approach followed by Ancona and Zucca [3] is the only one that defines a formal semantics of mixins independently of the semantics of the embedding language. Thus, this approach, like ours, concentrates on the pure notion of system composition, independently of the nature of the system elements. However, the semantic developed by Ancona and Zucca relies on the concept of function to represent a system with deferred components (deferred components are argument of the function). Since our approach does not rely on the concept of function, we believe that our proposition provides a more primitive formalization of system composition.

7. Conclusion

We believe that amalgams are orthogonal to the notion of function in declarative languages. Indeed, open expressions are allowed, which serves as incomplete pieces of code that can be completed later in several places. Our proposition is not to replace the use of functions by amalgams, but rather to use amalgams to structure and parameterize coarse pieces of code and to compute new programs from already existing ones. As far as distributed incremental program construction is concerned, a major advantage of the amalgams over the classical λ-calculus relies on the intrinsic incremental property of the amalgams: the free references together with the merge operation naturally allow dynamic extensions of programs, whereas the λ-calculus needs to be deeply improved to allow the same behavior (cf. section 6 and the works of [12, 17, 20, 28]).

However, amalgams lack a typing system: actually they are an untyped formalism. The evaluation of an expression, using the semantics defined in this paper, may not terminate (consider $\{a = \{x = a, b = 1\} \,\textbf{.}\, b\}$ for example that exhibits a cycle through the definition of x even if only b is necessary for the resolution of the selection).

From the semantics described in Fig. 6, an environment (called Mercure) has been developped in ML allowing the evaluation of expressions of the amalgams[2]. With this first evaluator, amalgams are currently being embedded into the declarative data-parallel language 81/2 [19, 30]. The 81/2 language manipulates intensionally [34] the notions of stream [44] and collection [42]. Since the notions of stream and collection are orthogonal to the definition of amalgams, they are naturally added as a ground type in the amalgams formalism. Amalgams are the key to the definition of parameterized expressions allowing programs to be incrementally constructed at run-time through the free references of the expressions.

The integration of amalgams in 81/2 consists in the definition of an evaluator of streams of amalgams enabling the definition of incremental computations, symbolic computation and an object oriented programming style (see examples in [31, 33, 32, 35]).

We are working on an extension of the core formalism presented here with a notion of *location* and *distributed operators* for the modeling of *distributed incremental compulations*.

Acknowledgments

The authors are grateful to the members of the LaMI and Parallel Architectures team in LRI for many fruitful discussions. We would like to thank M. Tofte and the anonymous reviewers of a previous version of this paper for their many helpful comments and suggestions. L. Maranget at INRIA has kindly accepted to read a latter version of this work and made useful remarks.

Appendix

We give in the following sections a sketch the proofs of the properties given in section 4.6 concerning the relation $\twoheadrightarrow$.

A.1. Determinism of the Semantics

Proof: We want to prove that if $u \twoheadrightarrow u'$ and $u \twoheadrightarrow u''$ then $u' \equiv u''$. To establish this result, it is sufficient to check that at most one rule can apply to a given term (because then there is at most only one possible proof tree for $\rho \vdash u \twoheadrightarrow u'$).

The four set of rules $\{x_1^n, x_2^n\}$, $\{\#_1, \#_2\}$, $\{\cdot_1, \cdot_2\}$ and $\{\{\}\}$ are mutually exclusive because they do not apply to the same terms. The rules in the three first sets are mutually exclusive because of their side condition (the side condition of rule x_2^n is the negation of the side condition of rule x_1^n, etc.). So at most one rule can apply to a term. ■

A.2. Semantics Reduces to Normal Form

Proof:
We want to prove that if $u \twoheadrightarrow u'$ then u' is in normal form, that is: $u' \twoheadrightarrow u'$. The proof is by establishing the following property (P): for any u and ρ, if $\rho \vdash u \twoheadrightarrow u'$ then $\rho \vdash u' \twoheadrightarrow u'$, by induction on the height of the reduction tree of u.

If the reduction tree of u is of height one, then only rule x_2^n can be used and we have $u \equiv x^n$ and $\rho \vdash x^n \twoheadrightarrow x^n$ which satisfies (P).

Suppose by induction hypothesis that (P) is true for any term u' when the height of the reduction tree of u' is less than d and let a term u such that the height of the reduction tree of u is $d+1$. We prove that (P) holds for u by inspection of the term u ; we present here only two cases, the other ones are similar:

1. if $u \equiv x^n$, then, because the height of the reduction tree of u is $d+1$ only rule x_1^n can apply. Let $u' \equiv \rho(x^n)$ and e, e' such that $\mathtt{tl}^n(\rho) \vdash u' \twoheadrightarrow e$ and $\rho \vdash e \twoheadrightarrow e'$. The height of the reduction tree of e' is necessarily less or equal to d because it is a sub-tree of the reduction tree of u. Then property (P) is true for e' and we have $\rho \vdash e' \twoheadrightarrow e'$. That is, property (P) holds also for u.

2. if $u \equiv \{\overrightarrow{x = e}\}$ and $\rho \vdash u \twoheadrightarrow \{\overrightarrow{x = e'}\}$, then the reduction tree of e_i is less or equal to d and then $u \uplus \rho \vdash e'_i \twoheadrightarrow e'_i$ by rule $\{\}$. Therefor, $\rho \vdash \{\overrightarrow{x = e'}\} \twoheadrightarrow \{\overrightarrow{x = e'}\}$.

■

A.3. Clear, stable form and normal form

Proof:

We begin by generalizing the definition of clear form. Let $\rho \in \mathcal{E}_{\mathrm{E}}$, we say that a term u is in ρ-clear form if it is in $\mathit{def}(\rho)$-clear form, where the function $\mathit{def} : \mathcal{E}_{\mathrm{E}} \to \mathcal{E}_{\mathtt{def}}$ is defined by:

$$\begin{array}{lcl} \mathit{def}(\emptyset) & = & \emptyset \\ \mathit{def}(\sigma :: \rho) & = & \mathtt{Def}(\sigma) :: (\mathit{def}\,\rho) \end{array}$$

We want to prove the more general statement:

$$u \text{ is in } \rho\text{-normal form} \Leftrightarrow u \text{ is in a } \rho\text{-clear and stable form}$$

We first establish that if u is in a ρ-clear and stable form, then u is in ρ-normal form. The proof is by induction on the structure of u:

1. Base case. We have $u \equiv x^n$. Because u is in ρ-clear form, only rule x_2^n can apply and we have $\rho \vdash u \twoheadrightarrow u$.

2. Induction step.

   (A) Let $u \equiv \{\overrightarrow{x = e}\}$ and $\rho' = \{\overrightarrow{x = e}\} \uplus \rho$. It is easy to see that each e_i is in ρ'-clear and stable form because u is in ρ-clear and stable form. The induction hypothesis can be applied to each e_i to conclude that each e_i is in ρ'-clear and stable form, that is : $\rho' \vdash e_i \twoheadrightarrow e_i$. Then $\rho \vdash u \twoheadrightarrow u$, that is u is in ρ-normal form.

(B) Let $u \equiv v_1 \# v_2$. v_1 and v_2 are in ρ-clear and stable form because u is. Then we habe by the induction hypothesis: $\rho \vdash v_1 \twoheadrightarrow v_1$ and $\rho \vdash v_2 \twoheadrightarrow v_2$. Because u is in stable form, only rule $\#_1$ may apply, and thus $\rho \vdash u \twoheadrightarrow u$, that is, u is in ρ-normal form

(C) Let $u \equiv v_1 \centerdot v_2$. We have v_1 in ρ-clear and stable form and then, by induction hypothesis, v_1 is in ρ-normal form. Then because u is in stable form, $v_1 \notin$ SYS and only rule $\centerdot_1$ may apply and thus $\rho \vdash v_1 \centerdot v_2 \twoheadrightarrow v_1 \centerdot v_2$. In order word, u is in ρ-normal form.

In the opposite direction, we first establish that if u is not in ρ-clear form, then u cannot be in ρ-normal form by induction on the structure of u.

1. Base case. We have $u \equiv x^n$. Because u is not in ρ-clear form, rule x_1^n applies and we have $\rho \vdash e'$ with $\texttt{tl}^n(\rho) \vdash \rho(x^n) \twoheadrightarrow e$ and $\rho \vdash e \twoheadrightarrow e'$. If $e' \not\equiv x^n$, u is not in ρ-normal form and we have the result. If $e' \equiv x^n$, then the height of the reduction tree cannot be finite (that is, x^n is ρ-stuck: this is a variant of the example 3 in section 4.8.1).

   We argue by contradiction, suppose that the height of the reduction tree of x^n is finite, say d. By hypothesis we have $\rho \vdash e \twoheadrightarrow x^n$. This can be the case only if e is of form $y_1^{n_1}$ or $v \centerdot y_1^{n_1}$ with $y_1 \in$ ID, $v \in \Sigma$ and $n_1 \in \mathbb{N}$. In either case, we have to prove $\rho_1 \vdash y_1^{n_1} \twoheadrightarrow x^n$ for some ρ_1. That is, the proof tree of $\rho \vdash x^n \twoheadrightarrow x^n$ strictly includes a proof tree of $\rho_1 \vdash y_1^{n_1} \twoheadrightarrow x^n$. Note that ρ is a suffix of ρ_1 (if e is of form $y_1^{n_1}$ then $\rho_1 = \rho$, in the other case $\rho_1 = \sigma :: \rho$ for some σ).

   We may apply to $y_1^{n_1}$ the same reasoning and we obtain that the proof tree of $\rho \vdash x^n \twoheadrightarrow x^n$ must contain a proof tree of $\rho_2 \vdash y_2^{n_2} \twoheadrightarrow x^n$ with ρ a suffix of ρ_2. We may continue in this way, but because the proof tree of $\rho \vdash x^n \twoheadrightarrow x^n$ is finite, we obtain that for some j: $y_j \equiv x$. Then we have just established that the proof tree of $\rho \vdash x^n \twoheadrightarrow x^n$ strictly contains a proof tree of $\rho_j \vdash x^n \twoheadrightarrow x^n$ with ρ a suffix of ρ_j.

   We distinguish two cases:

   (A) If $\rho_j = \rho$, then the proof tree of $\rho \vdash x^n \twoheadrightarrow x^n$ strictly contains itself. So it cannot be finite.

   (B) Let $\rho^{(1)} = \rho_j$. If $\rho^{(1)} \neq \rho$, then x^n cannot be in $\rho^{(1)}$-clear form because x^n is not in ρ-clear form and ρ is a suffix of $\rho^{(1)}$. We may apply the same reasoning as before and we have in the proof tree the statements: $\rho^{(2)} \vdash x^n \twoheadrightarrow x^n$, $\rho^{(3)} \vdash x^n \twoheadrightarrow x^n$, ... We cannot iterate this procedure indefinitely because the length $|\rho'|$ of an environment ρ in a proof tree is bounded by the height of the proof tree (more precisely: $|\rho'| \leq d + |\rho|$ for any environment ρ' in the proof tree of $\rho \vdash x^n \twoheadrightarrow x^n$). Then the number of possible environments with suffix ρ is bounded and we must have $\rho^{(k)} = \rho$ for some k. Then we conclude as in the case (A).

2. Induction step. We suppose the property true for any sub-term of u.

(A) Let $u \equiv \{\overrightarrow{x = e}\}$, $\rho' = \{\overrightarrow{x = e}\} \uplus \rho$ and e'_j such that: $\rho' \vdash e_j \twoheadrightarrow e'_j$. u is not in ρ-clear form, thus there is a i such that e_i is not in ρ'-clear form. Then e_i is not in ρ'-normal form by induction hypothesis, that is: $\rho' \vdash e_i \twoheadrightarrow e'_i$ with $e_i \not\equiv e'_i$. But then, $\{\overrightarrow{x = e}\} \not\equiv \{\overrightarrow{x = e'}\}$, or in other word, u is not in ρ-normal form

(B) Let $u \equiv v_1 \# v_2$. The proof is similar to the previous case, using only environment ρ and one of v_1 or v_2 instead of e_i.

(C) Let $u \equiv v_1 \centerdot v_2$. The proof is similar to the previous case.

Lastly, we have to establish that if u is not in a stable form, then u cannot be in a normal form. The proof is simple: if u is not in stable form, there exists a subterm of u of form $s \centerdot e_2$ with $s \in \text{Sys}$ or of form $s \# s'$ with $s, s' \in \text{Sys}$. For these terms, the rules that can be applied are $\centerdot_2$ and $\#_2$ respectively. The second rule relates a merge with a system. The first one cannot relate $s \centerdot e_2$ to $s \centerdot e_2$ by an argument similar to the base case of the previous proof (it implies an infinite proof tree). ■

A.4. Commutativity in a system

Proof: Obviously, if $s \in \text{Sys}$ then $s \uplus \rho = \alpha(s) \uplus \rho$. Then, the ordering of the equations in a system is used nowhere in the reduction rules and the assertion is obvious. ■

A.5. Commutativity of the Merge

Proof: If $\rho \vdash u \twoheadrightarrow \{\overrightarrow{x = e}\}$ and $\rho \vdash v \twoheadrightarrow \{\overrightarrow{x' = e'}\}$ and $\rho \vdash u \# v \twoheadrightarrow w$, then we have $\rho \vdash \{\overrightarrow{x = e}, \overrightarrow{x' = e'}\} \twoheadrightarrow w$ and if $\rho \vdash v \# u \twoheadrightarrow w'$, then $\rho \vdash \{\overrightarrow{x' = e'}, \overrightarrow{x = e}\} \twoheadrightarrow w'$. But, by use of the previous proposition, we have: $w \equiv \alpha(w')$ with α the permutation such that $\{\overrightarrow{x = e}, \overrightarrow{x' = e'}\} = \alpha(\{\overrightarrow{x' = e'}, \overrightarrow{x = e}\})$. The result follows. ■

Notes

1. The reader, which is probably acquainted with the de Bruijn representation of λ-terms may think of such a feature. But great care should be taken here: the escape operator defined in the amalgams ***does not behave like de Bruijn indexes***. Even if we use the same syntactic construction to express a scope escaping mechanism, its semantics is totally different. Where de Bruijn indexes require the *lift* of λ-terms for substitution, the amalgams require ***not to lift*** the free references in the substitution process (see section 4.7.1) so that *name capture* may happen. In the λ-calculus, name capture has to be avoided while it is the central computation mechanism of the amalgams.
2. All examples given in this paper have been processed using this evaluator. The current version is available at `ftp://ftp.lami.univ-evry.fr/pub/archi/michel/Research/amald.tar.gz`.

References

1. Davide Ancona. An algebraic framework for separate compilation. Technical Report DISI-TR-97-10, Dipartimento di Informatica e Scienze dell'Informazione - Genova, 1997.
2. Davide Ancona and Elena Zucca. An algebraic approach to mixins and modularity. In Michael Hanus and Mario Rodríguez-Artalejo, editors, *Algebraic and Logic Programming, 5th International Conference, ALP'96*, volume 1139 of *lncs*, pages 179–193, Aachen, Germany, 25–27 September 1996. Springer.
3. Davide Ancona and Elena Zucca. A theory of mixin modules: basic and derived operators. Technical Report DISI-TR-96-24, Dipartimento di Informatica e Scienze dell'Informazione - Genova, 1996.
4. John Backus. Can programming be liberated from the von neumann style ? A functional style and its algebra of programs. *Com. ACM*, 21:613–641, August 1978.
5. Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.
6. Gilad Bracha and William Cook. Mixin-based inheritance. *ACM SIGPLAN Notices*, 25(10):303–311, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).
7. Rod Burstall and Butler Lampson. A kernel language for modules and abstract data types. Technical Report 1, DEC SRC, September 1984.
8. Luca Cardelli. A semantics of multiple inheritance. In G. Kahn, D. MacQueen, and G. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–67. Springer-Verlag, 1984. Full version in *Information and Computation* 76(2/3):138–164, 1988.
9. Laurent Dami. Functions and names, without name capture. Technical report, CUI Genève, 1994.
10. Laurent Dami. *Software Composition: Towards an Integration of Functional and Object-Oriented Approaches.* Ph.D. thesis, University of Geneva, 1994.
11. Laurent Dami. Functions, records and compatibility in the λN-calculus. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 153–174. Prentice Hall, 1995.
12. Laurent Dami. A lambda-calculus for dynamic binding. *Theoretical Computer Science*, 192(2), February 1998.
13. N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter Rewrite systems, pages 244–320. Elsevier Science, 1990.
14. Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin, TX*, pages 331–347, New York, NY, 1984. ACM.
15. Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 262–273, Philadelphia, Pennsylvania, 24–26 May 1996.
16. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 19–21 January 1998.
17. Jacques Garrigue. Dynamic binding and lexical binding in a transformation calculus. In *Proceedings of the Fuji International Workshop on Functional and Logic Programming.* World Scientific, 1995.
18. Jacques Garrigue and H. Aït-Kaci. The typed polymorphic label-selective lambda-calculus. In *Principles of Programming Languages*, Portland, 1994.
19. Jean-Louis Giavitto. A synchronous data-flow language for massively parallel computer. In D. J. Evans, G. R. Joubert, and H. Liddell, editors, *Proc. of Int. Conf. on Parallel Computing (ParCo'91)*, pages 391–397, London, 3-6 September 1991. North-Holland.
20. Masatomo Hashimoto and Atsushi Ohori. A typed context calculus. Technical Report RIMS-1098, Research Institute for Mathematical Sciences, Kyoto University, August 1996.
21. W. Wilson Ho and Ronald A. Olsson. An approach to genuine dynamic linking. *Software, Practice and Experience*, 21(4):375–390, April 1991.
22. Suresh Jagannathan. A programming language supporting first-class parallel environments. Technical Report 434, MIT LCS, 1989.

23. Gilles Kahn. The semantics of a simple language for parallel programming. In *proceedings of IFIP Congress'74*, pages 471–475. North-Holland, 1974.
24. Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.
25. Sonya E. Keene. *Object-Oriented Programming in Common Lisp.* Addison Wesley, Reading, Mass., 1989.
26. John Lamping. A unified system of parameterization for programming languages. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 316–326. ACM, ACM, July 1988.
27. Shinn-Der Lee and Daniel P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 479–492, Charleston, South Carolina, January 1993.
28. Shinn-Der Lee and Daniel P. Friedman. Enriching the lambda calculus with contexts: toward a theory of incremental program construction. In *International Conference on Functional Programming.* ACM, May 1996.
29. Marc Ban Limberghen and Tom Mens. Encapsultation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.
30. Olivier Michel. Design and implementation of 81/2, a declarative data-parallel language. *Computer Languages*, 22(2/3):165–179, 1996. special issue on Parallel Logic Programming.
31. Olivier Michel. Introducing dynamicity in the data-parallel language 81/2. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *EuroPar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Sciences*, pages 678–686. Springer-Verlag, August 1996.
32. Olivier Michel. *Représentations dynamiques de l'espace dans un langage déclaratif de simulation.* PhD thesis, Université de Paris-Sud, centre d'Orsay, December 1996. `ftp://ftp.lri.fr/LRI/articles/michel/thesis.ps.gz`.
33. Olivier Michel. A straightforward translation of D0L Systems in the declarative data-parallel language 81/2. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *EuroPar'96 Parallel Processing*, volume 1123 of *Lecture Notes in Computer Sciences*, pages 714–718. Springer-Verlag, August 1996.
34. Olivier Michel, Dominique De Vito, and Jean-Paul Sansonnet. 81/2 : data-parallelism and data-flow. In E. Ashcroft, editor, *Intensional Programming II: Proc. of the 9th Int. Symp. on Lucid and Intensional Programming.* World Scientific, May 1996.
35. Olivier Michel and Jean-Louis Giavitto. Amalgams: Names and name capture in a declarative framework. Technical Report 32, LaMI – Université d'Évry Val d'Essonne, January 1998.
36. Robin Milner. *A Calculus for Communicating Processes*, volume 92 of *Lecture Notes in Computer Science.* Springer Verlag, 1980.
37. Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Computer Science Dept., University of Edinburgh, October 1991.
38. Robin Milner. Elements of interaction: Turing award lecture. *Communications of the ACM*, 36(1):78–89, January 1993.
39. David A. Moon. Object-oriented programming with flavors. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21, pages 1–8, New York, NY, November 1986. ACM Press.
40. Martin Odersky. A syntactic theory of local names. Research Report YALEU/DCS/RR-965, Department of Computer Science, Yale University, May 1993.
41. François Rouaix. *Caml Applets User Guide.* INRIA Rocquencourt, April 1996.
42. Jay M. Sipelstein and Guy Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.
43. Sun Microsystems. *The Java^{TM} Language Specification*, 1.0 beta edition, October 1995.
44. G. L. Tesler and H. J. Enea. A language design for concurrent processes. In *AFIPS Conference Proceedings*, volume 32, pages 403–408, 1968.

45. Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection*, pages 111–134. Elsevier Sci. Publishers B.V. (North Holland), 1988. Also to appear in Lisp and Symbolic Computation.