A Generic Object-Calculus Based on Addressed Term Rewriting Systems

DAN DOUGHERTY
Wesleyan University
Middletown, CT 06459, USA

FRÉDÉRIC LANG
INRIA Rhône-Alpes
Montbonnot, F 38334 St Ismier, France

PIERRE LESCANNE
École Normale Supérieure de Lyon
46 Allée d'Italie, F 69364 Lyon, France

LUIGI LIQUORI
INRIA Lorraine
Campus Scientifique BP 239, F 54506 Vandoeuvre-lès-Nancy, France

KRISTOFFER ROSE
KRISTOFFER ROSE
IBM T. J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA

Abstract

We describe the foundations of $\lambda\mathcal{O}bj^a$, a framework, or generic calculus, for modeling object-oriented programming languages. This framework provides a setting for a formal operational semantics of object based languages, in the style of the Lambda Calculus of Objects of Fisher, Honsell, and Mitchell. As a formalism for specification, $\lambda\mathcal{O}bj^a$ is arranged in *modules*, permitting a natural classification of many object-based calculi according to their features. In particular, there are modules for calculi of non-mutable objects (*i.e.*, functional object-calculi) and for calculi of mutable objects (*i.e.*, imperative object-calculi). As a computational formalism, $\lambda\mathcal{O}bj^a$ is based on rewriting rules. However, classical first-order term rewriting systems are not appropriate to reflect aspects of implementation practice such as sharing, cycles in data structures and mutation. Therefore, the notion of *addressed terms* and the corresponding notion of *addressed term rewriting* are developed.

Contents

1 Introduction 3
2 A Simple Example Exploiting Object Inheritance 8
2.1 Cloning 9
2.2 Illustrating an Imperative Calculus 10
2.3 Illustrating a Functional Calculus 12

3 Rewriting with Addresses 12
3.1 Sharing 13
3.2 Cycles and Mutation 14
3.3 Syntax of Addressed Terms 15
3.4 Rewriting 15
4 Modules and Top Level Rules of $\lambda\mathcal{O}bj^a$ 16
4.1 Syntax of $\lambda\mathcal{O}bj^a$ 17
4.2 Architecture of $\lambda\mathcal{O}bj^a$ 19
5 Examples in $\lambda\mathcal{O}bj^a$ 22
6 Meta-Theory of $\lambda\mathcal{O}bj^a$ 24
6.1 Addressed Terms 25
6.2 Addressed Term Rewriting 30
6.3 Acyclic Mutation-free ATRS 34
6.4 The Calculus $\lambda\mathcal{O}bj^\sigma$ and its Relation with $\lambda\mathcal{O}bj^a$ 38
7 Conclusions 40
References 42

List of Figures

1 Sharing and Cycles Using Addresses 7
2 The Object Pixel 9
3 The Clones `p` and `q` 10
4 The Memory Structure after `(1,2)` 11
5 The Memory Structure after `(3)` 11
6 Detail from Figure 4 14
7 A Loop in the Store 14
8 The Syntax of $\lambda\mathcal{O}bj^a$ 18
9 The Top Level Rules of $\lambda\mathcal{O}bj^a$ 20
10 The Module `K` 22
11 A Loop in the Store 25
12 The Term t and the Preterm u 28
13 The Syntax of $\lambda\mathcal{O}bj^\sigma$ 38
14 The Rules of $\lambda\mathcal{O}bj^\sigma$ 39

1 Introduction

Recent years have seen a great deal of research aimed at providing a rigorous foundation for object-oriented programming languages. In many cases, this work has taken the form of "object-calculi" (Fisher *et al.*, 1994; Abadi & Cardelli, 1996).

Such calculi can be understood in two ways. On the one hand, the formal system is a specification of the semantics of the language, and can be used as a framework for classifying language design choices, to provide a setting for investigating type systems, or to support a denotational semantics. Alternatively, we may treat an object-calculus as an intermediate language into which user code (in a high-level object-oriented language) may be translated, and from which an implementation (in machine language) may be derived.

In this paper, we present the calculus $\lambda\mathcal{O}bj^a$ in which one can give a formal specification and an operational semantics for a variety of object-based programming languages. In fact, $\lambda\mathcal{O}bj^a$, introduced in (Lang *et al.*, 1999a), is a *generic framework*, leading to an easy *classification* of object-based languages and their semantics, making a clear distinction between functional and imperative languages, *i.e.*, languages with non-mutable objects and languages with mutable objects, or according to Okasaki's terminology (Okasaki, 1998): languages with persistent objects and languages with ephemeral objects.

The calculus $\lambda\mathcal{O}bj^a$ is based on λ-calculus, but we do *not* restrict our attention to so-called "functional" object-oriented calculi. A key feature of our approach is the representation of programs as *addressed terms* (Lang *et al.*, 1999b), which support reasoning about mutation. Since $\lambda\mathcal{O}bj^a$ contains the λ-calculus explicitly, we therefore have a modular and uniform treatment of both functional and imperative programming.

Many treatments of functional operational semantics exist in the literature (Landin, 1964; Augustsson, 1984; Kahn, 1987; Milner *et al.*, 1990). To go further and accommodate imperative operations one can use the traditional *stack and store* approach (Plotkin, 1981; Felleisen & Friedman, 1989; Tofte, 1990; Mason & Talcott, 1991; Felleisen & Hieb, 1992; Wright & Felleisen, 1994; Abadi & Cardelli, 1996; Bono & Fisher, 1998). The greater complexity of the presentation of the latter works might lead one to conclude—wrongly—that implementing functional languages is easy in comparison with imperative languages. Such a false impression may be due in part to the fact that typical operational semantics formalisms are based on algebras: this makes them good at abstracting away the complexity of the algebraic structures used in functional languages, but ill-suited to express the non-algebraic structure of imperative data structures. The novelty of $\lambda\mathcal{O}bj^a$ is that it provides a *homogeneous* approach to both functional and imperative aspects of programming languages, in the sense the two semantics are treated in the same way using addressed terms, with only a minimal sacrifice in the permitted algebraic structures. Indeed, the addressed terms used were originally introduced to describe sharing behavior for functional programming languages (Rose, 1996; Benaissa *et al.*, 1996).

From another point of view, the use of addressed terms suggests a bridge between

the operational and denotational semantics for a language. This is not a direction we pursue in detail in this paper but the main idea is as follows. A traditional denotational semantics for imperative languages involves the *store*, *i.e.*, a function from locations to values, as one of the key domains. The store typically has no explicit representation in the programming language and it has no structure beyond being a function space. But if we identify locations with addresses in an addressed-term representation of an execution state, we may see the store as intimately bound up with the program expression. Indeed, the term "program expression" is now misleading, since we have at hand an expression modeling an *execution state*, which embodies a function from addresses to sub-expressions whose values in turn comprise the store. Note that the semantics now need not refer to the entire store but only that finite part concerning addresses explicit in the execution state. Furthermore the store inherits a structure, induced by the execution state's tree-structure; for example there is now a notion of one store-location occurrence being within the scope of another.

All of this leads to a new and rather subtle relationship between operational and denotational semantics. In a sense we have identified a new level of abstraction, more general and robust than the machine level, yet more concrete and operational than a purely mathematical treatment *à la* λ-calculus.

Specifically, the calculus $\lambda\mathcal{O}bj^a$ enjoys the following properties:

- It is *faithful to implementation* in the sense that each transition in the system corresponds to a constant-cost operation in the execution of code on a machine. This permits reasoning about resource usage and the actual cost of certain implementation choices.
- It is *a formal system* which can support a careful analysis of some fundamental properties of object-oriented languages, such as type-safety and observational equivalence.

With regard to the first point, $\lambda\mathcal{O}bj^a$ gives an explicit account of substitution, sharing, and redirection. The inclusion of explicit indirection nodes is a crucial innovation here. Indirection nodes allow us to give a more realistic treatment of the so-called collapsing rules of term graph rewriting (rules that rewrite a term to one of its proper sub-terms): more detailed discussion will be found in Sections 3.4 and 4.2.

The framework $\lambda\mathcal{O}bj^a$ is not a monolithic formal calculus. It is defined in terms of a set of five *modules* (L, C, F, I, and K), each of which captures a particular aspect of object-calculi. Indeed, the modules are sets of rules which describe, in "small-steps", the transformations of the objects, whereas the strategies (such as call-by-value, call-by-name, etc) describe how these rules are invoked giving the general evolution of the whole program. Usually in the description of an operational semantics, strategies and small steps are tightly coupled. In our approach they are independent. As a consequence, we get the genericity of $\lambda\mathcal{O}bj^a$, in the sense that many semantics can be instantiated in our framework to conform to specific wishes. A specific calculus is therefore a combination of *modules plus a suitable strategy.* Thus, we choose to not code strategies into the framework itself and we postpone

discussion of specific strategies for future work. Also, for the sake of simplicity, we will not here explore issues such as privacy or encapsulation.

A useful way to understand the current project is by analogy with graph-reduction as an implementation-calculus for functional programming. Comparing $\lambda\mathcal{O}bj^a$ with the state of the art implementation techniques of functional programming (FP) and object oriented programming (OOP) gives the following correspondence:

Paradigm	$\lambda\mathcal{O}bj^a$ **fragment**	**Implementation techniques**
Pure FP	$\lambda\mathcal{O}bj^a$ (L)	Graph Rewriting; Explicit Substitutions
Pure FP+OOP	$\lambda\mathcal{O}bj^a$ (L+C+F)	Graph Rewriting; Explicit Substitutions
Imp. FP+OOP	$\lambda\mathcal{O}bj^a$ (full)	Stack & Store

In the remainder of this introduction we provide historical context for our focus on *object-based* languages and our use of *explicit substitutions* and *addressed term rewriting* in the formal system.

Object-based Languages

The monograph (Abadi & Cardelli, 1996) makes the case for the study of object-based languages both as examples of a novel object-oriented style of programming and as a way of implementing class-based languages. In object-based languages there is no notion of class: the inheritance takes place at the object level. Objects are built "from scratch" or by inheriting the methods and fields from other objects (sometimes called *prototypes*). Examples of object-based language are in Self (Ungar & Smith, 1987), Obliq (Cardelli, 1995), Kevo (Tailvalsaari, 1992), Cecil (Chambers, 1993), Moby (Fisher *et al.*, 2000; Fisher & Reppy, 2000; Fisher & Reppy, 1999) and O-{1,2,3} (Abadi & Cardelli, 1996).

Among the proposals firmly setting the theoretical foundation of object-based languages, two of the most successful are the *Object Calculus* of Abadi and Cardelli (Abadi & Cardelli, 1996) and the *Lambda Calculus of Objects* ($\lambda\mathcal{O}bj$) of Fisher, Honsell, and Mitchell (Fisher *et al.*, 1994).

$\lambda\mathcal{O}bj$ is an untyped λ-calculus enriched with object primitives. Objects are untyped and a new object can be created by modifying and/or extending an existing prototype object. The result is a new object which inherits all the methods and fields of the prototype. The consistency of dynamic object-extension with a sound type-system was one of the main goals of $\lambda\mathcal{O}bj$. This calculus is computationally complete, since the λ-calculus is built in the calculus itself.

The calculus $\lambda\mathcal{O}bj^+$ (Gianantonio *et al.*, 1998; Ciaffaglione *et al.*, 2001) is an extension of $\lambda\mathcal{O}bj$ with a new small-step semantics, and a new type system; the type soundness result ensures that a typed program "cannot-go-wrong". In particular, $\lambda\mathcal{O}bj^+$ allows typed objects to extend themselves upon the reception of a message.

Two classical problems we find in the literature regarding the implementation of imperative and flexible object-calculi are:

- The capacity to handle *loops in the store* (Abadi & Cardelli, 1996).

- The capacity to dynamically extend objects.

The system $\lambda\mathcal{O}bj^a$ studied here presents solutions to each of these problems.

Explicit Substitutions Calculi

Calculi of explicit substitutions give a finer description of the meta-operation of *substitution*, a fundamental notion in any programming language; see for instance (Abadi *et al.*, 1991; Lescanne, 1994; Bloo & Rose, 1995). Roughly speaking, an explicit substitution calculus fully includes the substitution operation as part of the syntax, adding suitable rewriting rules to deal with it. These calculi give a good model of the concept of "closures" that represent partially computed function applications. If combined with updating, closures can represent objects by considering the state of the object as what has been computed "so far" (Abelson *et al.*, 1985). In section 6.4, we present $\lambda\mathcal{O}bj^\sigma$ as an antecedent of $\lambda\mathcal{O}bj^a$. It adds explicit substitutions to $\lambda\mathcal{O}bj$ of (Fisher *et al.*, 1994) (but not addresses, the main innovation of the present paper). The reader who wishes a slow and stepwise introduction to $\lambda\mathcal{O}bj^a$ may wish to look first at that language.

Recently (Dougherty & Lescanne, 2001) a technical analysis of an explicit substitutions calculus via intersection types has yielded a refinement and strengthening of the classical theorem that leftmost reduction is a normalizing strategy in the λ-calculus. The latter theorem is the theoretical foundation for the correctness of the standard evaluation strategy for functional languages; see for example (Mitchell, 1996) Prop. 2.4.12.

Addressed Calculi and Semantics of Sharing

Efficient implementations of lazy functional languages (and of computer algebras, theorem provers, etc.) require some sharing mechanism to avoid multiple computations of a single argument. A natural way to model this sharing in a symbolic calculus is to pass from a tree representation of terms to directed *graphs*. Such term graphs can be considered as a representation of program-expressions intermediate between abstract syntax trees and concrete representations in memory, and term-graph rewriting provides a formal operational semantics of functional programming sensitive to sharing. There is a wealth of research on the theory and applications of term graphs; see for example (Barendregt *et al.*, 1987; Sleep *et al.*, 1993; Plump, 1999; Blom, 2001) for general treatments, and (Wadsworth, 1971; Turner, 1979; Ariola & Klop, 1994; Ariola *et al.*, 1995; Ariola & Arvind, 1995) for applications to λ-calculus and implementations.

However, representing and thinking with graphs can be delicate (observe that graphs differ from trees in that the latter naturally support definition and proof by structural induction). In this paper we will annotate terms, as trees, with *global addresses* à la (Felleisen & Friedman, 1989; Rose, 1996; Benaissa *et al.*, 1996). Lévy (Lévy, 1980) and Maranget (Maranget, 1992) previously introduced *local addresses*; from the point of view of the operational semantics, global addresses describe better what is going in a computer or an abstract machine.

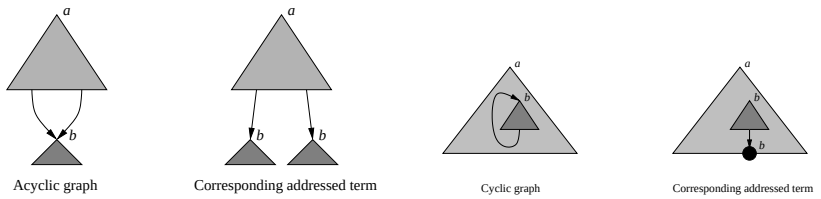

Fig. 1. Sharing and Cycles Using Addresses

With explicit global addresses we can keep track of the sharing that could be used in the implementation of a calculus. Sub-terms which share a common address represent the same sub-graphs, as suggested in Figure 1 (left), where a and b denote addresses. In (Lang *et al.*, 1999b), addressed terms were studied in the context of *addressed term rewriting*, as an extension of classical first-order term rewriting. In addressed term rewriting we may rewrite simultaneously all sub-terms sharing a same address, mimicking what would happen in an actual implementation.

The notion of computation on terms is expanded in the present paper to encompass computations performing mutation, still through rewriting rules.

We also enrich the sharing with a special *back-pointer* to handle *cyclic graphs* (Rose, 1996). Cycles are used in the functional language setting to represent infinite data-structures and (in some implementations) to represent recursive code; they are also interesting in the context of imperative object-oriented languages where *loops in the store* may be created by imperative updates through the use of `self` (or `this`), see Figure 11. The idea of the representation of cycles via addressed terms is rather natural: a cyclic path in a finite graph is fully determined by a prefix path ended by a "jump" to some node of the prefix path (represented with a back-pointer), as suggested in Figure 1 (right).

The formalisms of term-graph rewriting and addressed-term rewriting are fundamentally similar but we feel that the addressed-term setting has several advantages.

First, our intention is to define a calculus which is as close to actual implementations as possible, and the addresses in our terms really do correspond to memory references. To the extent that we are trying to build a bridge between theory and implementation we prefer this directness to the implicit coding inherent in a term-graph treatment.

Also, the relation between the value of subterm and the address attached to the subterm is precisely the *store*, so that the store is represented in the syntax itself. The most important consequence of our choice of formalism is the fact that we have a completely direct and transparent way to model *mutation*. Since memory references (*qua* addresses) are *first-class citizens* for us it is trivial to model changing the value at an address, by changing the subterm whose label is that address.

We should also note here that there is another active line of current research which aims to elucidate resource-usage in programming languages: this is the work which brings linear logic to bear on implementation questions. Notable examples here include (Gonthier *et al.*, 1992b; Gonthier *et al.*, 1992a; Asperti & Laneve,

1994; Mackie, 1995; Asperti & Laneve, 1996; Lang, 1998). These investigations are complementary to ours, but not really comparable. The intention there is to analyze phenomena such as garbage collection as part of the underlying *logic* of the language in question. But existing mainstream languages build in resource management in a way external to the underlying logic (if their design is based on logic at all). Questions of garbage collection and pointer-chaining are central to the task of mapping of a typical language to a machine, but are likely viewed by the programmer as well as the implementor at a different level from the meanings of the core language constructs. Our own goal is to better understand the relationship between code as written by the programmer and as executed on conventional architectures, by working in a system explicitly bridging these two extremes.

Outline of the Paper

In Section 2, we discuss by an example the main concepts that a generic calculus of objects has to take into account. In Section 3, we say how addressed term rewriting systems give solutions to the basic questions of object oriented languages, namely sharing, cycles and mutations. Section 4 presents the five modules of rewriting rules that form the core of $\lambda\mathcal{O}bj^a$. They are illustrated by some examples in Section 5. Section 6 details the framework of addressed term rewriting systems and establishes a general relation between addressed term rewriting systems and first-order term rewriting systems. This result is used to prove the correspondence between a subset of $\lambda\mathcal{O}bj^a$ and a calculus without addresses. Section 7 concludes and describes related and further works.

2 A Simple Example Exploiting Object Inheritance

The examples in this section embody certain choices about language design and implementation (such as "deep" *vs.* "shallow" copying, management of run-time storage, and so forth). It is important to stress that these choices are not tied to the formal calculus $\lambda\mathcal{O}bj^a$ which is the subject of this paper. Indeed, our main point is that $\lambda\mathcal{O}bj^a$ provides a foundation for a wide variety of language paradigms and language implementations. We hope that the examples are suggestive enough that it will be intuitively clear how to accommodate other design choices. The main body of the paper justifies that intuition. These schematic examples will be also useful to understand how objects are represented and how inheritance can be implemented in $\lambda\mathcal{O}bj^a$.

Reflecting implementation practice, in $\lambda\mathcal{O}bj^a$ we distinguish two distinct aspects of an object:

- *The object structure:* the actual list of methods/fields.
- *The object identity:* a pointer to the object structure.

We shall use the word "pointer" where others use "handle" or "reference". Objects can be bound to identifiers as "nicknames" (*e.g.*, `pixel`), but the only proper name of an object is its object identity: an object may have several nicknames but only one identity.

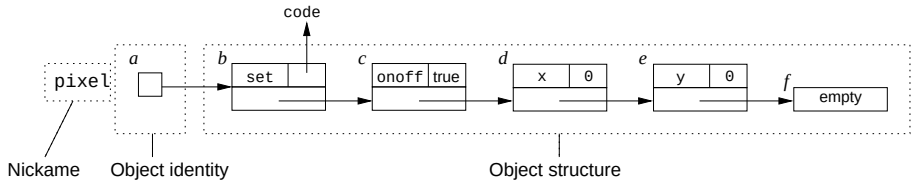

Fig. 2. The Object Pixel

Consider the following definition of a "pixel" prototype with three fields and one method. With a slight abuse of notation, we use ":=" for both assignment of an expression to a variable or the extension of an object with a new field or method and for overriding an existing field or method inside an object with a new value or body, respectively.

```
pixel = object {x     := 0;
                y     := 0;
                onoff := true;
                set   := (u,v,w){x := u; y := v; onoff := w;};
               }
```

After instantiation, the object `pixel` is located at an address, say a, and its object structure starts at address b, see Figure 2. In what follows, we will derive three other objects from `pixel` and discuss the variations of how this may be done below.

2.1 Cloning

The first two derived objects, nick-named `p` and `q`, are *clones* of `pixel`:

```
p := clone(pixel);
q := clone(p);
```

Object `p` shares the same object-structure as `pixel` but it has its own object-identity. Object `q` shares also the same object-structure as `pixel`, even if it is a clone of `p`. The effect is pictured in Figure 3. We might stress here that `p` and `q` should not be thought of as aliases of `pixel` as Figure 3 might suggest; this point will be clearer after the discussion of object overriding below. The semantics of the `clone` operator we illustrate here differs somewhat from that found in certain existing object-oriented programming languages like SmallTalk and Java. For example in Java there will be sharing between an object and its clone if an instance field of the original is itself a reference to an object. In Java a true deep clone of an object is in general not available via the built-in clone method, but may be provided by the programmer overriding the default method.

In the rest of this section, we discuss the differences between functional and imperative models of object-calculi, *i.e.*, between models with non-mutable and mutable objects.

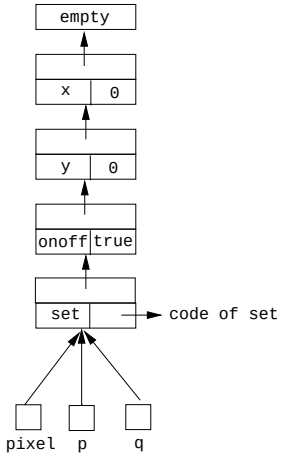

Fig. 3. The Clones `p` and `q`

2.2 Illustrating an Imperative Calculus

Imperative object-calculi have been shown to be fundamental in describing implementations of class-based languages. They are also essential as foundations of object-based programming languages like Obliq and Self. The main goal when one tries to define the semantics of an imperative object-based language is to say how an object can be modified while maintaining its object-identity. Particular attention must be paid to this when dealing with object extension. The semantics of the imperative update operation is subtle because of side-effects.

Here, we show what we want to model in our framework when we *override* the `set` method of the clone `q` of `pixel`, and we extend a clone `r` of (the modified) `q` with a new method `switch`.

```
q.set := (u,v,w){ x := x*u; y := y*v; onoff := w;}; (1)
r := clone(q);                                      (2)
r.switch := (){ onoff := not(onoff);};              (3)
```

Note that we have used a Java-like imperative syntax here to save parentheses.

Figure 4 shows the state of the memory after the execution of the instructions `(1,2)`. Note that after `(1)` the object `q` refers to a new object-structure, obtained by chaining the new body for `set` with the old object-structure. As such, when the overridden `set` method is invoked, thanks to dynamic binding, the newer body will be executed since it will hide the older one. This dynamic binding is embodied in the treatment of the method-lookup rules (SU) and (NE) from Module `C` as described in Section 4.

Observe that the override of the `set` method does not produce any side-effect on `p` and `pixel`; in fact, the code for `set` used by `pixel` and `p` will be just as before. Therefore, `(1)` only changes the object-structure of `q` without changing its object-identity. This is the sense in which our `clone` operator really does implement

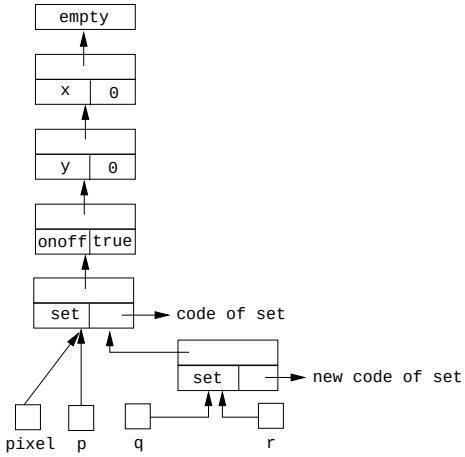

Fig. 4. The Memory Structure after `(1,2)`

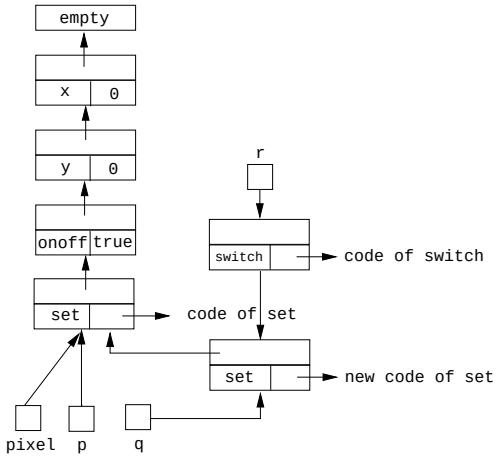

Fig. 5. The Memory Structure after `(3)`

shallow copying rather than aliasing, even though there is no duplication of object-structure at the time that `clone` is evaluated.

This implementation model performs side effects in a very restricted and controlled way. Figure 5, finally, shows the final state of memory after the execution of the instruction `(3)`. Observe that, in this case, the update operation, denoted by "`:=`", extend the object `r` with the `onoff` method. Again, the addition of the `switch` method changes only the object-structure of `r`.

In general, changing the nature of an object dynamically by adding a method or a field can be implemented by moving the object identity toward the new method/field (represented by a piece of code or a memory location) and to *chain it* to the original structure. This mechanism is used systematically also for method/field overriding but in practice (for optimization purposes) can be relaxed

for field overriding, where a more efficient *field look up and replacement* technique can be adopted. See for example the case of the Imperative Object Calculus in Chapter 10 of (Abadi & Cardelli, 1996), or observe that Java uses *static field lookup* to make the position of each field constant in the object.

This implementation model, however does not avoid the unfortunate *loop in the store*: an example of loop will be given Section 3.

2.3 Illustrating a Functional Calculus

Object-calculi can play a role as well in a purely functional setting, where there is no notion of *mutable state*. As said before, an "update" operation, denoted by "`:=`", can either override or extend an object with some fields or methods. In a functional setting, the update *always* produces another object with its proper object-identity since this ensures that all references to an object have the same meaning whether their evaluation is delayed or not. This property is also known as *referential transparency*. Thus, the result of an update must be a fresh object in the sense that it has a proper (new) object-identity.

The "imperative" definition of `pixel,p,q,r` in the previous example could have been written in a more traditional functional object-calculus as follows. Here `let x = A in B` is syntactic sugar for the functional application `(λx.B)A` and the `clone(_)` function is essentially an identity function since cloning in a purely functional setting is not relevant, due to the absence of mutation:

```
let p = clone(pixel) in
let q = clone(p).set := (u,v,w)
                 {((self.x := self.x*u).y := self.y*v).onoff := w} in
let r = (clone(q).switch := (){self.onoff := not(self.onoff);}) in r
```

which obviously reduces to:

```
(pixel.set := (u,v,w) {...}).switch := (){...}
```

Worth noticing is that the above code would be implemented, in a purely functional calculus, in the same way as Figure 5.

3 Rewriting with Addresses

In this section we introduce *addressed term rewriting systems* (ATRS) informally—in the context of object-oriented programming—by examining these issues in turn and the ways in which they are reflected in features of ATRS. Section 6 presents a more formal treatment.

The paradigm of *term rewriting* (Dershowitz & Jouannaud, 1990; Klop, 1990; Baader & Nipkow, 1998) provides a computational interpretation of first-order equational reasoning and is a very convenient and powerful tool to describe the operational semantics of simple calculi.

In addition, term rewriting systems are sufficiently flexible to model the operational semantics of functional programs, although at a high level, ignoring certain aspects of memory management, reduction strategy, and parameter-passing. They are widely used to formalize, prototype, and verify software.

However, as suggested in the introduction, classical term rewriting cannot easily express issues of sharing and mutation. Calculi which give an account of memory management often introduce some *ad-hoc* data-structure to model the memory, called *heap*, or *store*, together with access and update operations. However, the use of these structures necessitates restricting the calculus to a particular strategy. The aim of addressed term rewriting (and that of term graph rewriting) is to provide a mathematical model of computation which better reflects memory usage.

3.1 Sharing

Sharing has been extensively studied in the context of obtaining implementations of lazy functional programming languages (Peyton-Jones, 1987; Plasmeijer & van Eekelen, 1993), and the initial studies of sharing in the notations of term graph rewriting systems were indeed motivated by this application.

Sharing of computation. Consider the function square defined by

$$\mathsf{square}(x) = \mathsf{times}(x, x)$$

It is clear that an implementation of this function should not duplicate its input x in the expression $\mathsf{times}(x, x)$, but optimize this by only copying a *pointer* to the input. This not only saves memory but also makes it possible to *share future computations* on x, in particular when x is not already required to be a value, as in *e.g.*, lazy programming languages. Classical term representations do not permit us to express this sharing of the actual structure of x. However, the memory structures used for the computation of a program can be represented using addressed terms. For instance, the "program" $\mathsf{square}(\mathsf{square}(2))$ can be first instantiated in memory, provided locations a, b, c to each of its constructors, as the addressed term (or memory structure) $\mathsf{square}^a(\mathsf{square}^b(2^c))$. It can then be reduced as follows:

$$\begin{aligned}
\mathsf{square}^a(\mathsf{square}^b(2^c)) &\to \mathsf{times}^a(\mathsf{square}^b(2^c), \mathsf{square}^b(2^c)) \\
&\to \mathsf{times}^a(\mathsf{times}^b(2^c, 2^c), \mathsf{times}^b(2^c, 2^c)) \\
&\to \mathsf{times}^a(4^b, 4^b) \\
&\to 16^a,
\end{aligned}$$

where "$\to$" designates one step of shared computation (we are assuming that definitions to compute the function $\mathsf{times}(x, y)$ to the value $x * y$ exist for each x and y). The key point of a shared computation is that *all* terms which share a common address are reduced *simultaneously*. This corresponds to a *single computation step* on a small component of the memory.

Sharing of Object Structures. In object-oriented programming, the aim of sharing is not only to share *computations* as in the former example, but also to share

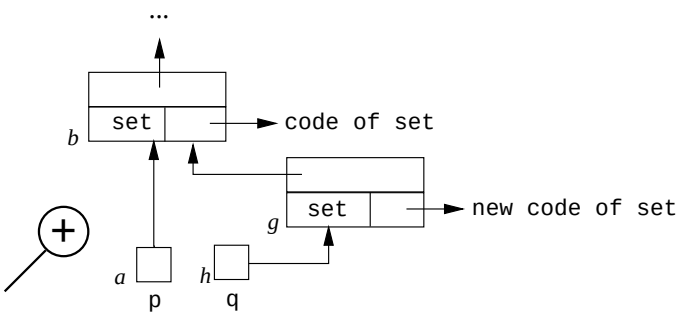

Fig. 6. Detail from Figure 4

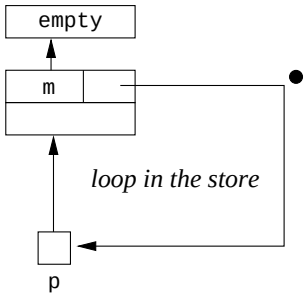

Fig. 7. A Loop in the Store

structures. Indeed, objects are typically structures which receive multiple pointers. Moreover, the delegation-based model of inheritance insists that object structures are shared between objects with different identities. As an example, if we "zoom" on Figure 4, we can observe that the object `p` and `q` share a *common structure*, addressed by b. This can be very easily formalized in the framework, since addresses are first-class citizens. See Example 2 in Section 5.

3.2 Cycles and Mutation

Cycles. Cycles are essential in functional programming when one wants to deal with infinite data-structures in an efficient way, as is the case in lazy functional programming languages. Cycles are also used, in some implementations, to save space in the code of recursive functions.

In the context of object programming languages, cycles can be also used to express *loops* introduced in the memory (the *store*) by the imperative operators. Figure 7 illustrates the situation where the body of method `m` references the object `p` (via `self`); a complete translation of such example in $\lambda\mathcal{O}bj^a$ will be presented in Example 3.

Mutation. Almost all object-oriented programming languages are not purely functional, but rather have some operations that may alter the state of objects without changing object's identity. An example of such a mutation was given in

Section 2.2 (Figures 4 and 5), where we see that the object denoted by p has had its structure altered by the addition of some methods, without changing its object identity. Note that the object p may be shared, and that all the expressions containing pointers to p undergo the mutation that happened at address a.

The key point of mutation in the setting of ATRS is the possibility to modify *in-place* the contents of any node at a given location, with respect to some precise rules. More details on computations performing mutation will be given in Section 4.

3.3 Syntax of Addressed Terms

An *addressed preterm* is a tree where each node receives a label (the operator symbol *e.g.*, times) and an address (*e.g.*, a). Of course we will not want to treat an arbitrary preterm as an addressed term, because an unconstrained preterm may denote a non-coherent memory structure. Roughly speaking, since each node in a memory has a *unique symbol* and a *unique list* of successor locations, it must be the case that all "sub-terms" at a given address in a term denote a *unique memory sub-structure*. For instance, the expression

$$\mathsf{times}^a(\mathsf{square}^b(2^c), \mathsf{square}^b(2^d))$$

(quite similar to the one presented in Subsection 3.1) is not admissible, because it designates that the unique son of the node at address b is both at addresses c and d. In contrast, the expression

$$\mathsf{times}^a(\mathsf{square}^b(2^c), \mathsf{square}^b(2^c))$$

is admissible.

Above we used the word *sub-term*, but in the presence of back-pointers, this notion does not really make sense. For example suppose t is an addressed term at address a containing a *back-pointer* $\bullet^a$. If one takes a sub-term of t in the usual sense, say at address b, one might obtain a term with a *dangling* back-pointer $\bullet^a$. Therefore when defining the term at address b, (denoted by $t \,@\, b$), one has to "expand" (or "unfold") $\bullet^a$ sufficiently to avoid dangling pointers. Because of this surgery, we do not call $t \,@\, b$ a *sub-term* but an *in-term*. Note that the relation "to be an in-term of" is not well-founded. Section 6 gives the details.

3.4 Rewriting

Addressed terms themselves formalize the data structures representing code and data in a machine; we now describe the operational semantics of the machine as rewriting of addressed terms.

A rewriting rule, denoted by $l \to r$, is a pair of addressed terms with variables. As for ordinary terms, such a rule induces a reduction relation on the set of addressed terms. This relation is defined with the help of a notion of a term *matching* another term. Roughly speaking, a term t matches l when the variables of l can be substituted by addressed terms, and its addresses by other addresses, resulting in t.

The intuition that substitution on addressed terms is almost the same as classical term substitution is sufficient to understand the following idea.

In an addressed rewriting rule $l \to r$, l and r must have a common address, say a, at their respective roots. The idea is that the rule describes how the node at this address has to be modified for computation. Other addresses reachable from a may be modified as well, and new nodes introduced by r.

Intuitively, given an addressed term t, the rewriting takes the following four steps:

1. *Find a redex in t, i.e.*, an in-term *matching* the left-hand side of a rule.
2. *Create fresh addresses, i.e.*, addresses not used in the current addressed term t, which will correspond to the new locations occurring in the right-hand side, but not in the left-hand side.
3. *Substitute the variables and addresses* of the right-hand side of the rule by their new values, as assigned by the matching of the left-hand side or created as fresh addresses. Let us call this new addressed term u.
4. For all a that occur both in t and u, replace in t the in-terms at address a by the in-term at address a in u.

The fact that both members of a rule must have the same address has non-trivial technical consequences. For example we cannot directly admit rules like $F^a(X) \to X$, called *collapsing* or *projection* rules. We can recover the effect of such rule by adding to the signature a unary function symbol, intuitively seen as an *indirection node* and written $\lceil\ \rceil$. This constraint is realistic as, in fact, it turns to be the technique used by all actual implementations to avoid searching the memory for pointers to redirect. With that, the above rule can be expressed as $F^a(X) \to \lceil X \rceil^a$. The use of explicit indirection nodes is motivated by our wish to explicit the constraints that every rewriting step must be as close as possible to what happens in a real implementation, so that the complexity of the rewriting and the complexity of the execution on a real machine are closely correlated. It is a simple and efficient way to avoid an unbounded, global, redirection.

The last operation in the above definition of rewriting corresponds to the simulation of updates *in-place* in a memory: all over the rewritten term, address contents are modified to give an account of the sharing and mutation. This operation is the key point of the following property: any reduction starting from an addressed term results in an addressed term, *i.e.*, the coherence of the underlying memory structures is preserved by the application of any rule, as will be established by Theorem 14 in Section 6.

Section 4 gives an intuition of the way rules are defined and used, in the particular setting of $\lambda\mathcal{O}bj^a$. Section 5 presents a rich collection of object expression in which some typical computations in $\lambda\mathcal{O}bj^a$ are modeled by (addressed) rewriting.

4 Modules and Top Level Rules of $\lambda\mathcal{O}bj^a$

The purpose of this section is to describe the top level rules of the framework $\lambda\mathcal{O}bj^a$. The framework is described by a set of rules arranged in *modules*. The five modules are called respectively L, C, F, I, and K.

L is the *functional* module, and is essentially the calculus $\lambda\sigma_w^a$ of (Benaissa *et al.*, 1996). This module alone defines the core of a purely functional programming language based on λ-calculus and weak reduction.

C is the *common object* module, and contains all the rules common to all instances of object calculi defined from $\lambda\mathcal{O}bj^a$. It contains rules for instantiation of objects and invocation of methods.

F is the module of *functional update*, containing the rules needed to implement object update that also changes object identity.

I is the module of *imperative update*, containing the rules needed to implement object update that does not change object identity.

K is a module which provides a dynamic semantics of the `clone` operator.

The set of rules $\mathsf{L} + \mathsf{C} + \mathsf{F}$ is the instance of $\lambda\mathcal{O}bj^a$ for non-mutable object calculi while $\mathsf{L} + \mathsf{C} + \mathsf{I} + \mathsf{K}$ is for mutable object calculi.

4.1 Syntax of $\lambda\mathcal{O}bj^a$

The syntax of $\lambda\mathcal{O}bj^a$ is summarized in Figure 8. The first category of expressions is the *code* of programs. Code contains all the constructs of the calculus $\lambda\mathcal{O}bj^+$ (Gianantonio *et al.*, 1998), plus an imperative update and a clone operator. Terms that define the code have no addresses, because code contains no environment and is not subject to any change during the computation (remember that addresses are meant to tell the computing engine which parts of the computation structure can change simultaneously). The second and third categories define dynamic entities, or inner structures: the *evaluation contexts*, and the *internal structure of objects* (or simply *object structures*). Terms in these two categories have explicit addresses. The last category defines *substitutions* also called *environments*, *i.e.*, lists of terms bound to variables, which are to be distributed and augmented over the code.

Notation. The " . " operator acts as a "cons" constructor for lists, with the environment id acting as the empty, or identity, environment. By analogy with traditional notation for lists we adopt the following aliases:

$$\begin{array}{rcl} M[\,]^a & \triangleq & M[\mathsf{id}]^a \\ M[U_1/x_1; \ldots; U_n/x_n]^a & \triangleq & M[U_1/x_1 \,.\, \ldots \,.\, U_n/x_n \,.\, \mathsf{id}]^a \end{array}$$

In what follows, we review all the four syntactic categories of $\lambda\mathcal{O}bj^a$:

The Code Category. Code terms, written M and N, provide the following constructs:

- Pure λ-terms, constructed from abstractions, applications, variables, and constants. This allows the definition of higher-order functions.
- Objects, constructed from the empty object $\langle\,\rangle$ and update operators: the functional $\langle \_ \leftarrow \_ \rangle$ and the imperative $\langle \_ \leftarrow\!: \_ \rangle$. An informal semantics of the update operators has been given in Section 2. As in (Gianantonio *et al.*, 1998), these operators can be understood as extension as well as override operators, since an override is handled as a particular case of extension.

$$
\begin{array}{lcll}
M, N & ::= & \lambda x.M \mid MN \mid x \mid c \mid \langle\,\rangle \mid M \Leftarrow m \mid & \text{Code} \\
 & & \langle M \leftarrow\!: \; m = N\rangle \mid \langle M \leftarrow m = N\rangle \mid \mathsf{clone}(x) & \\
U, V & ::= & M[s]^a \mid (UV)^a \mid & \text{Eval. Contexts} \\
 & & (U \Leftarrow m)^a \mid \langle U \leftarrow m = V\rangle^a \mid & \\
 & & \langle U \leftarrow\!: \; m = V\rangle^a \mid [\![O]\!]^a \mid Sel^a(O, m, U) \mid & \\
 & & \lceil U \rceil^a \mid \bullet^a & \\
O & ::= & \langle\,\rangle^a \mid \langle O \leftarrow m = V\rangle^a \mid \bullet^a & \text{Object Structures} \\
s & ::= & U/x \,.\, s \mid \mathsf{id} & \text{Substitutions}
\end{array}
$$

Fig. 8. The Syntax of $\lambda\mathcal{O}bj^a$

- Method invocation ($\_ \Leftarrow \_$).
- Cloning. The operator $\mathsf{clone}(x)$ creates a new object identity for the object pointed to by x but which still shares the same object structure as the object x itself (it is a "shallow copy" as discussed in Section 2).

Evaluation Contexts. These terms, written U and V, model *states of abstract machines.* Evaluation contexts contain an abstraction of the temporary structure needed to compute the result of an operation. They are given addresses as they denote dynamically instantiated data structures; they always denote a term closed under the distribution of an environment. There are the following evaluation contexts:

- *Closures*, of the form $M[s]^a$, are pairs of a code and an environment. Roughly speaking, s is a list of bindings for the free variables in the code M.
- The terms $(UV)^a$, $(U \Leftarrow m)^a$, $\langle U \leftarrow m = V\rangle^a$, and $\langle U \leftarrow\!: \; m = V\rangle^a$, are the evaluation contexts associated with the corresponding code constructors. Direct sub-terms of these evaluation contexts are themselves evaluation contexts instead of code.
- *Objects*, of the form $[\![O]\!]^a$, represent evaluated objects whose internal object structure is O and whose object identity is a. In other words, the address a plays the role of an *entry point* or *handle* to the object structure O, as illustrated by Figure 2.
- The term $Sel^a(O, m, U)$ is the evaluation context associated to a method-lookup, *i.e.*, the scanning of the object structure O to find the method m, and apply it to the object U. It is an auxiliary operator invoked when one sends a message to an object.
- The term $\lceil U \rceil^a$ denotes an indirection from the address a to the root of the addressed term U. The operator $\lceil \_ \rceil^a$ has no denotational meaning. It is

introduced to make the right-hand side stay at the same address as the left-hand side. Indeed in some cases this has to be enforced. *e.g.* rule (FVAR) and (IC). This gives account of phenomena well-known by implementors. Rules like (AppRed), (LCop), (FRed) and (IRed) remove those indirections.

- *Back-references*, of the form $\bullet^a$ represents a *back-pointer* intended to denote cycles as explained in Section 3.

Internal Objects. The crucial choice of $\lambda\mathcal{O}bj^a$ is the use of *internal objects*, written O, to model object structures in memory. They are persistent structures which may only be accessed through the address of an object, denoted by a in $[\![O]\!]^a$, and are never destroyed nor modified (but eventually removed by a garbage collector in implementations, of course). Since our calculus is inherently delegation-based, objects are implemented as linked lists (of fields/methods), but a more efficient array structure can be envisaged. Again, the potential presence of cycles means that object structures can contain occurrences of back-pointers $\bullet^a$.

The evaluation of a program, *i.e.*, a code term M, always starts in an empty environment, *i.e.*, as a closure $M[\,]^a$.

4.2 Architecture of $\lambda\mathcal{O}bj^a$

The rules of $\lambda\mathcal{O}bj^a$ as a computational-engine are defined in Figure 9.

Remark 1 (*On fresh addresses*)
We assume that all addresses occurring in right-hand sides but not in left-hand sides are *fresh*. This is a sound assumption relying on the formal definition of fresh addresses and addressed term rewriting (see Section 6), which ensures that clashes of addresses cannot occur.

In what follows, we will explain the rules, module by module.

The Module L. This module is very similar to the calculus $\lambda\sigma_w^a$ of (Benaissa *et al.*, 1996), a calculus of explicit substitution enriched with addresses, to which we have added explicit indirections. Module L hence defines the core of a very simple functional programming language.

Rule (App) tells how environments have to be distributed over applications: it creates two new evaluation contexts (closures) located at new fresh addresses b and c; each of these closures is reachable from address a, updated so as to contain an evaluation context of application. Note that the two occurrences of s in the right-hand side of the rule contain the same addressed sub-terms. This means that these sub-terms are shared.

Once a substitution reaches an abstraction, a redex can be contracted by applying rule (Bw) (the name comes from *Beta weak*, the name this rule is assigned in functional languages; a more appropriate pronunciation would be *Bind weakly*). This extends the substitution by adding a pair, binding the parameter of the abstraction to the argument of the application.

The Module L

$$(MN)[s]^a \rightarrow (M[s]^b\, N[s]^c)^a \quad \text{(App)}$$

$$\big((\lambda x.M)[s]^b\, U\big)^a \rightarrow M[U/x\ .\ s]^a \quad \text{(Bw)}$$

$$x[U/x\ .\ s]^a \rightarrow \lceil U \rceil^a \quad \text{(FVar)}$$

$$x[U/y\ .\ s]^a \rightarrow x[s]^a \quad x \not\equiv y \quad \text{(RVar)}$$

$$(\lceil U \rceil^b\, V)^a \rightarrow (U\, V)^a \quad \text{(AppRed)}$$

$$\lceil (\lambda x.M)[s]^b \rceil^a \rightarrow (\lambda x.M)[s]^a \quad \text{(LCop)}$$

The Module C

$$\langle\rangle[s]^a \rightarrow [\![\langle\rangle^b]\!]^a \quad \text{(NO)}$$

$$(M \Leftarrow m)[s]^a \rightarrow (M[s]^b \Leftarrow m)^a \quad \text{(SP)}$$

$$([\![O]\!]^b \Leftarrow m)^a \rightarrow Sel^a(O, m, [\![O]\!]^b) \quad \text{(SA)}$$

$$(\lceil U \rceil^b \Leftarrow m)^a \rightarrow (U \Leftarrow m)^a \quad \text{(SRed)}$$

$$Sel^a(\langle O \leftarrow m = U\rangle^b, m, V) \rightarrow (U\, V)^a \quad \text{(SU)}$$

$$Sel^a(\langle O \leftarrow n = U\rangle^b, m, V) \rightarrow Sel^a(O, m, V) \quad m \not\equiv n \quad \text{(NE)}$$

The Module F

$$\langle M \leftarrow m = N\rangle[s]^a \rightarrow \langle M[s]^b \leftarrow m = N[s]^c\rangle^a \quad \text{(FP)}$$

$$\langle [\![O]\!]^b \leftarrow m = V\rangle^a \rightarrow [\![\langle O \leftarrow m = V\rangle^c]\!]^a \quad \text{(FC)}$$

$$\langle \lceil U \rceil^b \leftarrow m = V\rangle^a \rightarrow \langle U \leftarrow m = V\rangle^a \quad \text{(FRed)}$$

The Module I

$$\langle M \leftarrow\!: m = N\rangle[s]^a \rightarrow \langle M[s]^b \leftarrow\!: m = N[s]^c\rangle^a \quad \text{(IP)}$$

$$\langle [\![O]\!]^b \leftarrow\!: m = V\rangle^a \rightarrow \lceil [\![\langle O \leftarrow m = V\rangle^c]\!]^b \rceil^a \quad \text{(IC)}$$

$$\langle \lceil U \rceil^b \leftarrow\!: m = V\rangle^a \rightarrow \langle U \leftarrow\!: m = V\rangle^a \quad \text{(IRed)}$$

All addresses occurring in right-hand sides but not in left-hand sides are *fresh*.

Fig. 9. The Top Level Rules of $\lambda\mathcal{O}bj^a$

Once a variable is reached by a substitution, a lookup has to be performed in the substitution to find the evaluation context to be substituted, *i.e.*, the one bound to the variable. This is described by rules (FVar), and (RVar). Note that, since modifications must be performed in place, and since U has its own address, the only simple way to get access to U from a is to set an indirection (denoted by a pair of $\lceil\ \rceil$-brackets) from a to the root of U.

The last two rules (AppRed), and (LCop) say how to get rid of indirections that could *block* the identification of redexes. Intuitively, we are here treating the situation in which address a has a redex, but one of its components is available only through a redirection. In this module, an indirection blocks a reduction if the indirected node is an abstraction, and the indirection node is the left argument of an application. We have two alternative ways to get rid of such indirections, modeling choices that may be made in an implementation:

1. Redirect from the address a to the root of U, as in rule (AppRed).
2. Copy the indirected abstraction node lying at address b, at the address of the indirection node a, as in rule (LCop). Note that the copy is only a copy of the node at b, not of a whole graph, since addresses in s and (implicit addresses) in $\lambda x.M$ do not change. Note as well that this copy may not cause a loss in the sharing of computation since an abstraction is already a value and can not be reduced further.

Finally observe that, in contrast to (Benaissa *et al.*, 1996), no rule is given which allows us to copy shared structures for applications and other closures. There are two reasons to do this: the first is that it could induce a *loss in the sharing* of computations since applications and closures are not values; the second (stronger) is that such closures can reduce to objects, and, as we will see later, a copy would have the same effect as a *clone* of object. We certainly do not want to have uncontrolled cloning of objects, particularly in the presence of imperative update. In fact, the way an implementer is going to handle redirections is an essential component of the design of an object oriented language. One main purpose of our approach is to make this pointer manipulation explicit in a rewriting framework.

The Common Object Module C. This module handles object instantiation and message sending. *Object instantiation* is defined by rule (NO) where an empty object is given an object identity. More sophisticated objects may then be obtained by functional or imperative updates, defined in modules F and I. *Message sending* is formalized by the five remaining rules, namely rule (SP), which propagates the environment into the receiver of the message, rule (SA), which performs the self-application, rules (SU) and (NE), which perform the method-lookup, and finally rule (SRed) which redirects a blocking indirection node. Note that there is no *copy* alternative to rule (SRed), since we still do not want to lose control of the cloning of objects.

The Functional Object Module F. This module gives the operational semantics of a calculus of non-mutable objects. It contains only three rules. Rule (FP) propagates

$$\mathsf{clone}(x)[U/y\ .\ s]^a \rightarrow \mathsf{clone}(x)[s]^a \quad x \not\equiv y \qquad \text{(SRVar)}$$

$$\mathsf{clone}(x)[[\![O]\!]^b/x\ .\ s]^a \rightarrow [\![O]\!]^a \qquad \text{(SFVar)}$$

$$\mathsf{clone}(x)[\lceil U \rceil^b/x\ .\ s]^a \rightarrow \mathsf{clone}(x)[U/x\ .\ s]^a \qquad \text{(CRed)}$$

Fig. 10. The Module K

substitutions over functional update operators, installing the evaluation context needed to proceed, while rule (FC) describes the actual update of an object of identity b. The update is not made in place at address b, hence no side effect is performed, but the result is a new object, with a new object identity a which used to be the address of the evaluation context that has led to this new object. This is why we call this operator *functional* or *non-mutating*. The last rule (FRed) is the way to get rid of blocking indirection nodes in the case of functional update.

The Imperative Object Module I. This module contains rules for the mutation of objects (imperative update) and cloning primitive. Imperative update is formalized in a way close to the functional update. Rule (IC) differs from rule (FC) in address management, as illustrated in Section 2. Indeed, look at address b in rule (IC). In the left-hand side, b is the identity of an object $[\![O]\!]$, when in the right-hand side it is the identity of the whole object modified by the rule. Since b may be shared from anywhere in the context of evaluation, this modification is observable *non-locally* as a *side effect* or *mutation*. Moreover, since the result of this transformation has to be accessible from address a, an indirection node is set from a to b. As described in Section 3, rule (IC) may create cycles because it is possible that the address b is a sub-address of V. Module I has also a rule that redirects blocking indirection nodes in the case of imperative extension, namely rule (IRed).

The Clone Module K. This module deals with object cloning and it is presented separately in Figure 10. The term $\mathsf{clone}(x)$ is a primitive for cloning, that performs a lookup in the environment as variable access, but that always creates a copy of the found object. As we said before, by copy, we mean a *shallow* copy that creates a new object identity for an existing object even though x and $\mathsf{clone}(x)$ share the same object structure. This operator belongs to the core of $\lambda\mathcal{O}bj^a$, but for readability we feel more appropriate to make it an independent module. Rules (SRVar) and (SFVar) lookup the object bound to x in the environment, in a way similar to rules (RVar) and (FVar). Once the object is found, it is given the new identity a (SFVar). Rule (CRed) gets rid of a blocking indirection by local redirection.

5 Examples in $\lambda\mathcal{O}bj^a$

Here we propose some examples that should aid the understanding of the framework.

We first give an example showing a functional object which extends itself (Gianantonio *et al.*, 1998) with a field `n` upon reception of message `m`.

Example 1 (*An Object which "self-inflicts" an Extension*)

Let $\texttt{self\_ext} \triangleq \langle\ \langle\rangle \leftarrow \texttt{add\_n} = \underbrace{\lambda\texttt{self}.\langle \texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.1\rangle}_{N}\ \rangle$. The reduction of $M \triangleq (\texttt{self\_ext} \Leftarrow \texttt{add\_n})$ in $\lambda\mathcal{O}bj^a$ starting from an empty substitution is as follows:

$$
\begin{aligned}
M[\,]^a &\to^* (\langle\langle\rangle[\,]^d \leftarrow \texttt{add\_n} = N[\,]^c\rangle^b \Leftarrow \texttt{add\_n})^a && (1)\\
&\to (\langle [\![\langle\rangle^e]\!]^d \leftarrow \texttt{add\_n} = N[\,]^c\rangle^b \Leftarrow \texttt{add\_n})^a && (2)\\
&\to (\underbrace{[\![\langle\langle\rangle^e \leftarrow \texttt{add\_n} = N[\,]^c\rangle^f]\!]^b}_{O} \Leftarrow \texttt{add\_n})^a && (3)\\
&\to Sel^a(O, \texttt{add\_n}, [\![O]\!]^b) && (4)\\
&\to ((\lambda\texttt{self}.\langle \texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.1\rangle)[\,]^c\ [\![O]\!]^b)^a && (5)\\
&\to \langle \texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.1\rangle[[\![O]\!]^b/\texttt{self}]^a && (6)\\
&\to \langle \texttt{self}[[\![O]\!]^b/\texttt{self}]^h \leftarrow \texttt{n} = (\lambda\texttt{s}.1)[[\![O]\!]^b/\texttt{self}]^g\rangle^a && (7)\\
&\to \langle \lceil [\![O]\!]^b \rceil^h \leftarrow \texttt{n} = (\lambda\texttt{s}.1)[[\![O]\!]^b/\texttt{self}]^g\rangle^a && (8)\\
&\to \langle [\![O]\!]^b \leftarrow \texttt{n} = (\lambda\texttt{s}.1)[[\![O]\!]^b/\texttt{self}]^g\rangle^a && (9)\\
&\to [\![\langle O \leftarrow \texttt{n} = (\lambda\texttt{self}.1)[[\![O]\!]^b/\texttt{self}]^g\rangle^h]\!]^a && (10)
\end{aligned}
$$

In (1), two steps are performed to distribute the environment inside the extension, using rules (SP), and (FP). In (2), the empty object is given an object-structure and an object identity (NO). In (3), this new object is functionally extended (FC), hence it shares the structure of the former object but has a new object-identity. In (4), and (5), two steps (SA) (SU) perform the look up of method `add_n`. In (6) we apply (Bw). In (7), the environment is distributed inside the functional extension (FP). In (8), (FVar) replaces `self` by the object it refers to, setting an indirection from h to b. In (9) the indirection is eliminated (FRed). Step (10) is another functional extension (FC). There is no redex in the last term of the reduction, *i.e.* it is in normal form.

Some sharing of structures appears in the example above, since *e.g.* $[\![O]\!]^b$ has several occurrences in certain terms of the derivation.

Example 2 (*Object Representations in Figures 4 and 6*)

Representing object structures with the constructors $\langle\rangle$ (the empty object), and $\langle \_ \leftarrow \_\rangle$ (the functional *cons* of an object with a method/field), and object identities by the bracketing symbol $[\![\_]\!]$, the object `p` and `q`, presented in Figures 4 and 6,

will be represented by the following addressed terms.

$$\mathtt{p} \triangleq [\![\ \langle\langle\langle\langle\langle\,\rangle^f \leftarrow \mathtt{y} = 0\rangle^e \leftarrow \mathtt{x} = 0\rangle^d \leftarrow \mathtt{onoff} = \mathtt{true}\rangle^c \leftarrow \mathtt{set} = \cdots\rangle^b]\!]^a$$

$$\mathtt{q} \triangleq [\![\langle\langle\langle\langle\langle\,\rangle^f \leftarrow \mathtt{y} = 0\rangle^e \leftarrow \mathtt{x} = 0\rangle^d \leftarrow \mathtt{onoff} = \mathtt{true}\rangle^c \leftarrow \mathtt{set} = \cdots\rangle^b \leftarrow \mathtt{set} = \cdots\rangle^g]\!]^h$$

The use of the same addresses b, c, d, e, f in $\mathtt{p}$ as in $\mathtt{q}$ denotes the sharing between both object structures while g, h, are unshared and new locations.

Let us look on an example how we can express cycles, hence loops in the framework.

Example 3 (*Loop in the store of (Abadi & Cardelli, 1996)*)
Consider an object $\mathtt{o}$ which contains one single method, namely $\mathtt{m}$. The method $\mathtt{m}$ overrides itself. In $\lambda\mathcal{O}bj^a$, we represent methods with λ-abstractions whose first parameter $\mathtt{s}$ denotes the object itself. The object $\mathtt{o}$ is then:

$$[\![\langle\langle\,\rangle^b \leftarrow \mathtt{m} = \underbrace{(\lambda\mathtt{s}.\langle\mathtt{s} \leftarrow\!: \mathtt{m} = \lambda\mathtt{s}'.\mathtt{s}\rangle)}_{N}\,[\,]^c\rangle^d]\!]^a$$

where $\langle\_ \leftarrow\!: \_\rangle$ denotes the imperative *cons* of an object with a method/field, and $N[\,]^c$ refers to the evaluation context at address c given by code N in the identity environment $[\,]$. When $\mathtt{m}$ is invoked on the object $\mathtt{o}$, it "self-inflicts" an override of $\mathtt{m}$ with a new body in which $\mathtt{s}$ is now bound to $\mathtt{o}$ itself.

The result of this operation could be expressed as the *infinite term* defined by the fixed point equation:

$$\mathtt{o} \triangleq [\![\langle\langle\,\rangle^b \leftarrow \mathtt{m} = N[\,]^c\rangle^d \leftarrow \mathtt{m} = (\lambda\mathtt{s}'.\mathtt{s})[\mathtt{o}/\mathtt{s}]^e\rangle^f]\!]^a \quad (\textit{bad})$$

Here, $[\mathtt{o}/\mathtt{s}]$ says that in the λ-abstraction $(\lambda\mathtt{s}'.\mathtt{s})$, the free variable $\mathtt{s}$ is bound to $\mathtt{o}$. This is not the approach taken in the framework $\lambda\mathcal{O}bj^a$: rather than having to deal with an infinite term (or a fixed point equation), we adopt the *back-pointer* $\bullet$ labeled with the same address as $\mathtt{o}$.

The following legal term in $\lambda\mathcal{O}bj^a$ is, the addressed term representing $\mathtt{o}$, in which $\bullet^a$ denotes a back-pointer to the addressed term at location a, namely $\mathtt{o}$ itself:

$$\mathtt{o} \triangleq [\![\langle\langle\,\rangle^b \leftarrow \mathtt{m} = N[\,]^c\rangle^d \leftarrow \mathtt{m} = (\lambda\mathtt{s}'.\mathtt{s})[\bullet^a/\mathtt{s}]^e\rangle^f]\!]^a \quad (\textit{good})$$

Figure 11 gives a graphical illustration of the loop.

6 Meta-Theory of $\lambda\mathcal{O}bj^a$

The computational engine underneath the framework $\lambda\mathcal{O}bj^a$ is essentially based on *Addressed Term Rewriting Systems* (ATRS) (Lang *et al.*, 1999b); ATRS were introduced as a framework which can account for computation with *sharing*, *cycles*, and *mutation*. ATRS enjoy the following features:

- They permit one to model the "geometry" of an implementation, including aspects of sharing and mutation.

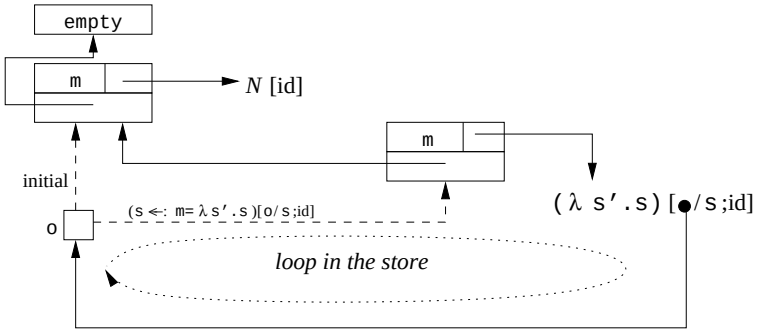

Fig. 11. A Loop in the Store

- They permit a straightforward representation of cyclic data via "back-pointers".
- They enjoy bounded complexity of rewriting steps by eliminating implicit pointer redirection.

In this sense, ATRS provide a handy tool for the definition of the formal operational semantics of $\lambda\mathcal{O}bj^a$.

Part of this Section is mainly inspired from (Lang *et al.*, 1999b) in which the interested reader may find some further examples.

6.1 Addressed Terms

Addressed terms are first order terms labeled by operator symbols and decorated with addresses. They satisfy well-formedness constraints ensuring that every addressed term represents a connected piece of a store. Moreover, the label of each node sets the number of its successors. More abstractly, addressed terms denote *term graphs*, as the *largest tree unfolding of the graph without repetition of addresses in any path*. Addresses intuitively denote *node locations* in memory. Identical subtrees occurring at different paths can thus have the same address corresponding to the fact that the two occurrences are *shared*.

The definition is in two stages: the first stage defines the basic inductive term structure, called *preterms*, while the second stage just restricts preterms to well-formed preterms, or addressed terms.

Definition 1 (*Preterms*)

1. Let Σ be a term signature, and $\bullet$ a special symbol of arity zero (a constant). Let $\mathcal{A}$ be an enumerable set of *addresses* denoted by $a, b, c, \ldots$, and $\mathcal{X}$ an enumerable set of *variables*, denoted by $X, Y, Z, \ldots$ An *addressed preterm* t over Σ is either a variable X, or $\bullet^a$ where a is an address, or an expression of the form $F^a(t_1, \ldots, t_n)$ where $F \in \Sigma$ (the label) has arity $n \geq 0$, a is an address, and each t_i is an addressed preterm (inductively).
2. The location of an addressed preterm t, denoted by $loc(t)$, is defined by

$$loc\big(F^a(t_1, \ldots, t_n)\big) \triangleq loc(\bullet^a) \triangleq a,$$

and it is not defined on variables.

3. The set of variables and addresses occurring within a preterm t is denoted by $var(t)$ and $addr(t)$, respectively, and defined in the obvious way.

Remark 2
Note that the concrete syntax of $\lambda\mathcal{O}bj^a$ of Figure 8 extends the above scheme in two ways:

1. Symbols in the signature may also be infix (like *e.g.*, $(\_ \Leftarrow \_)$), bracketing (like *e.g.*, $[\![\_]\!]$), mixfix (like $\_[\_]$), or even "invisible" (as is traditional for application, represented by juxtaposition). In these cases, we have chosen to write the address outside brackets and parentheses.
2. We shall use $\lambda\mathcal{O}bj^a$ sort-specific variable names.

For example we write $(UV)^a$ instead of $\mathsf{apply}^a(X,Y)$ and $M[s]^a$ instead of $\mathsf{closure}^a(X,Y)$ (substituting U for X, etc.). Indeed, we shall leave the names of $\lambda\mathcal{O}bj^a$ function symbols, such as apply and closure alluded to above, unspecified.

It is clear that not all preterms denote term graphs, since this may lead to inconsistency in the sharing. For instance, the preterm

$$\big((\llbracket\langle\rangle^a\rrbracket^b \Leftarrow \mathtt{m})^a \llbracket\langle\rangle^a\rrbracket^b\big)^c$$

is inconsistent, because location a is both labeled by $\langle\rangle$ and $(\_ \Leftarrow \_)$. The preterm

$$\big((\llbracket\langle\rangle^a\rrbracket^b \Leftarrow \mathtt{m})^c \llbracket\langle\rangle^e\rrbracket^b\big)^d$$

is inconsistent as well, because the node at location b has its successor at both locations a and e, which is impossible for a term graph. On the contrary, the preterm

$$\big((\llbracket\langle\rangle^a\rrbracket^b \Leftarrow \mathtt{m})^c \llbracket\langle\rangle^a\rrbracket^b\big)^d$$

denotes a *legal* term graph with four nodes, respectively, at addresses a, b, c, and d[1]. Moreover, the nodes at addresses a and b, respectively labeled by $\langle\rangle$ and $[\![\_]\!]$, are shared in the corresponding graph since they have several occurrences in the term. The well-formedness constraints filter preterms which denote term graphs from preterms which do not. Only the former are called *addressed terms.*

The definition of a preterm makes use of a special symbol denoted by $\bullet$, and called a *back-pointer.* The back-pointer is also present in the definition of the syntax of $\lambda\mathcal{O}bj^a$, see Figure 8. The purpose of this symbol is to denote *cycles.* Having a simple representation of cycles is an interesting feature for specifying imperative object calculi, because one can create cycles in the memory by doing imperative updates of objects. Classical rewriting, or algebraic specification tools, lack the provision of a representation of cycles. ATRS representation of cyclic graphs inherits from the work of Rose (Rose, 1996) in using the so-called *back-pointer* representation. A back-pointer $\bullet^a$ in an addressed term must be such that a is an address occurring on

[1] Observe that computation with this term leads to a *method-not-found* error since the invoked method m does not belong to the object $\llbracket\langle\rangle^a\rrbracket^b$, and hence will be rejected by a suitable sound type system or by a run-time exception.

the path from the root of the addressed term to the back-pointer node. It simply indicates at which address one has to branch (or point back) to go on along an infinite path. For instance, the addressed term

$$\llbracket \langle\langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda \mathtt{x}.\mathtt{y})[\bullet^b/\mathtt{y}]^c\rangle^d \rrbracket^b$$

denotes a cyclic object which refers to itself in the environment and whose cycle originates at address b. Note that $\bullet$ is considered as a special symbol in the sense that it is not a label. In the previous addressed term, the label at address b is $\llbracket \_ \rrbracket$. Given the previous informal definitions, one could argue that there may be several addressed terms denoting a same cyclic term graph. In fact, there may even be infinitely many. Indeed,

$$\begin{aligned}
&\llbracket \langle\langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda x.\mathtt{y})[\bullet^b/\mathtt{y}]^c\rangle^d \rrbracket^b \\
&\llbracket \langle\langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda \mathtt{x}.\mathtt{y})[\llbracket \bullet^d \rrbracket^b/\mathtt{y}]^c\rangle^d \rrbracket^b \\
&\llbracket \langle\langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda \mathtt{x}.\mathtt{y})[\llbracket \langle\langle\,\rangle^a \leftarrow \mathtt{m} = \bullet^c\rangle^d \rrbracket^b/\mathtt{y}]^c\rangle^d \rrbracket^b \\
&\ldots
\end{aligned}$$

are just the first three of an infinite sequence of preterms that all denote the same term graph, corresponding to different levels of *unfolding* of addresses b, c, and d. However, it is clear that there is a smallest (with respect to the size of the addressed term) representation of this term graph, namely the first one. In the following, we will work modulo the smallest representations of cyclic term graphs.

An essential operation that we must have on addressed (pre)terms is the *unfolding* that allows seeing, on demand, what is beyond a back-pointer. Unfolding can therefore be seen as a *lazy operator* that traverses one step deeper in a cyclic graph. It is accompanied with its dual, called *folding*, that allows giving a minimal representation of cycles. Note however that folding and unfolding operations have *no operational meaning* in an actual implementation (hence *no operational cost*) but they are essential in order to represent correctly transformations between addressed terms.

Definition 2 (*Folding and Unfolding*)

Folding. Let t be a preterm, and a be an address. We define $fold(a)(t)$ as the *folding of preterms located at* a in t as follows:

$$\begin{aligned}
fold(a)(X) &\triangleq X \\
fold(a)(\bullet^b) &\triangleq \bullet^b \\
fold(a)\big(F^a(t_1,\ldots,t_n)\big) &\triangleq \bullet^a \\
fold(a)\big(F^b(t_1,\ldots,t_n)\big) &\triangleq F^b\big(fold(a)(t_1),\ldots,fold(a)(t_n)\big) \quad \text{if } a \not\equiv b
\end{aligned}$$

Unfolding. Let s and t be preterms, such that $loc(s) \equiv a$ (therefore defined), and a does not occur in t except as the address of $\bullet^a$. We define $unfold(s)(t)$ as the

$$t \triangleq \langle [\![\langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda \mathtt{x}.\mathtt{y})[\bullet^b/\mathtt{y}]^c\rangle^e]\!]^b \leftarrow \mathtt{n} = (\lambda \mathtt{x}.\mathtt{y})[[\![\langle\,\rangle^a \leftarrow \mathtt{m} = \bullet^c\rangle^e]\!]^b/\mathtt{y}]^c\rangle^d$$

$$u \triangleq \langle [\![\langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda \mathtt{x}.\mathtt{y})[\bullet^b/\mathtt{y}]^c\rangle^e]\!]^b \leftarrow \mathtt{n} = (\lambda \mathtt{x}.\mathtt{y})[\bullet^b/\mathtt{y}]^c\rangle^d$$

Fig. 12. The Term t and the Preterm u

unfolding of $\bullet^a$ *by* s *in* t as follows:

$$\begin{aligned}
&\mathit{unfold}(s)(X) \triangleq X \\
&\mathit{unfold}(s)(\bullet^b) \triangleq \begin{cases} s & \text{if } a \equiv b \\ \bullet^b & \text{otherwise} \end{cases} \\
&\mathit{unfold}(s)\big(F^b(t_1,\ldots,t_m)\big) \triangleq F^b\big(t'_1,\ldots,t'_m\big) \quad \text{where} \quad s' \triangleq \mathit{fold}(b)(s) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad t'_1 \triangleq \mathit{unfold}(s')(t_1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \ldots \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad t'_m \triangleq \mathit{unfold}(s')(t_m)
\end{aligned}$$

We now proceed with the formal definition of *addressed terms* also called *admissible* preterms, or simply *terms*, for short, when there is no ambiguity. As already mentioned, addressed terms are preterms which denote term graphs. First, we give the reader an intuition of the problems raised by this constraint.

As an example, in Figure 12, the preterm t is an addressed term, whereas the preterm u is not an addressed term because the address b does not occur on the path from the root of the term to the second occurrence of $\bullet^b$. Similarly, a sub-term of an addressed term, in the usual sense, is not an addressed term. For instance, $\bullet^b$ is not an addressed term although t, which contains it, is.

This example shows us that the usual sub-term relation is not the one we need. A specific notion of term at a given address *in* an addressed term, abbreviated *in-term*, has the intended property and is given next. This notion tells us that the unique in-term of t located at address b is

$$[\![\langle\,\rangle^a \leftarrow \mathtt{m} = (\lambda \mathtt{x}.\mathtt{y})[\bullet^b/\mathtt{y}]^c\rangle^e]\!]^b$$

which is an addressed term. Similarly, the unique in-term of t at address c is

$$(\lambda \mathtt{x}.\mathtt{y})[[\![\langle\,\rangle^a \leftarrow \mathtt{m} = \bullet^c\rangle^e]\!]^b/\mathtt{y}]^c$$

although t has three distinct sub-terms at address c, namely u, $\bullet^c$, and $(\lambda \mathtt{x}.\mathtt{y})[\bullet^b/\mathtt{y}]^c$.

The notion of in-term helps to define addressed terms. The definition of addressed terms takes two steps: the first step is the definition of *dangling terms*, that are the sub-terms, in the usual sense, of actual addressed terms. Simultaneously, we define the notion of a dangling term, say s, at a given address, say a, in a dangling term, say t. When the dangling term t (*i.e.* the "out"-term) is known, we just call s an in-term. For a dangling term t, its in-terms are denoted by the function $t\,@\,\_$, read "t at address _", which returns a minimal and consistent representation of terms at each address, using the unfolding.

Therefore, there are two notions to distinguish: on the one hand the usual well-

founded notion of "sub-term", and on the other hand the (no longer well-founded) notion of "term in another term", or "in-term". In other words, although it is not the case that a term is a proper sub-term of itself, it may be the case that a term is a proper in-term of itself or that a term is an in-term of one of its in-terms, due to cycles. The functions $t_i \,@\, \_$ are also used during the construction to check that all parts of the same term are consistent, mainly that all in-terms that share a same address are all the same dangling terms.

Dangling terms may have back-pointers which do not point anywhere because there is no node with the same address "above" in the term. The latter are called *dangling back-pointers*. For instance,

$$(\lambda \mathtt{x}.\mathtt{y})[\bullet^b/\mathtt{y}]^c$$

has a dangling back-pointer, while

$$(\lambda \mathtt{x}.\mathtt{y})[[\![\langle\langle\,\rangle^a \leftarrow \mathtt{m} = \bullet^c\rangle^e]\!]^b/\mathtt{y}]^c$$

has none. The second step of the definition restricts the addressed terms to the dangling terms which do not have dangling back-pointers.

The following definition provides simultaneously two concepts:

- The dangling terms.
- The function $t \,@\, \_$ from $addr(t)$ to dangling in-terms. $t \,@\, a$ returns the in-term of t at address a.

Definition 3 (Dangling Addressed Terms)

Variables. Every $X \in \mathcal{X}$ is a dangling term. Since $addr(X) = \emptyset$, $X \,@\, \_$ is nowhere defined.

Back-pointers. $\bullet^a$ is a dangling term such that $\bullet^a \,@\, a \equiv \bullet^a$.

Expressions. Let $t_1, \ldots, t_n$ be dangling addressed terms ($n \geq 0$) and a be an address such that:

1. $\forall b \in addr(t_i) \cap addr(t_j)$, we have $t_i \,@\, b \equiv t_j \,@\, b$, and
2. $a \in addr(t_i)$ only if $t_i \,@\, a \equiv \bullet^a$.

Then, given $F \in \Sigma$ of arity n,

- $F^a(t_1, \ldots, t_n)$ is a dangling term.
- $t \,@\, \_$ is defined by:
  - $t \,@\, a \equiv t$.
  - $\forall b \in addr(t) \setminus \{a\}$, we have $t \,@\, b \equiv unfold(t)(t_i \,@\, b)$, where t_i is any of $t_1, \ldots, t_n$ containing b.

The "admissible" addressed terms can now be defined as those where all $\bullet^a$ *do* point back to something in t such that a complete (possibly infinite) unfolding of the term exists. The only way we can observe this with the $t \,@\, \_$ function is through checking that no $\bullet^a$ can "escape" because this cannot happen when it points back to something.

Definition 4 (Addressed Terms)

A dangling addressed term t is *admissible* if

$$\forall a \in addr(t), \text{ we have } t @ a \not\equiv \bullet^a$$

The *addressed terms* denote the admissible dangling addressed terms.

Proposition 5 (*In-terms Admissibility*)
If t is an admissible term, and $a \in addr(t)$, then

1. $t @ a$ is admissible, and
2. $\forall b \in addr(t @ a)$, we have $(t @ a) @ b \equiv t @ b$.

Proof
By definition of addressed terms: it follows from how the function $t @ \_$ is well-defined. □

6.2 Addressed Term Rewriting

Given the representation of term graphs by addressed terms, how do we compute? First of all, the computation on an addressed term must return an addressed term (not just a preterm). In other words, the computation model (here addressed term rewriting) must take into account the sharing information given by the addresses, and must be defined as the *smallest rewriting relation preserving admissibility between addressed terms.* Hence, a computation has to take place simultaneously at several places in the addressed term, namely at the places located at the same address. This simultaneous update of terms corresponds to the update of a location in the memory in a real implementation.

In an ATRS, a rewriting rule is *a pair of open addressed terms* (*i.e.*, containing variables), both located at the same location. The way addressed term rewriting proceeds on an addressed term t is not so different from the way usual term rewriting does. There are four steps.

1. *Find a redex in t, i.e.* an in-term *matching* the left-hand side of a rule. Intuitively, an addressed term matching is the same as a classical term matching, except there is a new kind of variables, called addresses, which can only be substituted by addresses.
2. *Create fresh addresses, i.e.* addresses not used in the current addressed term t, which will correspond to the locations occurring in the right-hand side, but not in the left-hand side (*i.e.* the new locations).
3. *Substitute the variables and addresses* of the right-hand side of the rule Sby their new values, as assigned by the matching of the left-hand side or created as fresh addresses. Let us call this new addressed term u.
4. For all a that occur both in t and u, the result of the rewriting step, say t', will have $t' @ a \equiv u @ a$, otherwise t' will be equal to t.

We give the formal definition of matching and replacement, and then we define rewriting precisely.

Definition 6 (*Substitution, Matching, Unification*)

1. Mappings from addresses to addresses are called *address substitutions.* Mappings from variables to addressed terms are called *variable substitutions.* A pair of an address substitution α and a variable substitution σ is called a *substitution*, and it is denoted by $\langle \alpha ; \sigma \rangle$.
2. Let $\langle \alpha ; \sigma \rangle$ be a substitution and p a term such that $addr(p) \subseteq dom(\alpha)$ and $var(p) \subseteq dom(\sigma)$. The application of $\langle \alpha ; \sigma \rangle$ to p, denoted by $\langle \alpha ; \sigma \rangle(p)$, is defined inductively as follows:

$$\langle \alpha ; \sigma \rangle(\bullet^a) \triangleq \bullet^{\alpha(a)}$$

$$\langle \alpha ; \sigma \rangle(X) \triangleq \sigma(X)$$

$$\langle \alpha ; \sigma \rangle\big(F^a(p_1, \ldots, p_m)\big) \triangleq F^{\alpha(a)}(q_1, \ldots, q_m) \text{ and } q_i \triangleq fold(\alpha(a))\big(\langle \alpha ; \sigma \rangle(p_i)\big)$$

3. We say that a term t *matches* a term p if there exists a substitution $\langle \alpha ; \sigma \rangle$ such that $\langle \alpha ; \sigma \rangle(p) \equiv t$.
4. We say that two terms t and u *unify* if there exists a substitution $\langle \alpha ; \sigma \rangle$ and an addressed term v such that $v \equiv \langle \alpha ; \sigma \rangle(t) \equiv \langle \alpha ; \sigma \rangle(u)$.

Example 4

1. The term $(\llbracket \langle \rangle^a \rrbracket^d \Leftarrow \mathtt{m})^b$ matches $(\llbracket O \rrbracket^b \Leftarrow \mathtt{n})^a$ with substitution

$$\langle \{a \mapsto b, b \mapsto d\} ; \{\mathtt{m} \mapsto \mathtt{n}, O \mapsto \langle \rangle^a\} \rangle$$

2. The term $\mathtt{z}[\bullet^b/\mathtt{z}]^b$ matches $\mathtt{x}[U/\mathtt{x}; s]^a$ with substitution

$$\langle \{a \mapsto b\} ; \{\mathtt{x} \mapsto \mathtt{z}, U \mapsto \mathtt{z}[\bullet^b/\mathtt{z}]^b ; s \mapsto [\,]\} \rangle$$

   Note that the range of the obtained variable substitution consists of addressed terms, as required by the definition of a substitution.

We now define *replacement.* The replacement function operates on terms. Given a term, it changes some of its in-terms at given locations by other terms with the same address. Unlike classical term rewriting (see for instance (Dershowitz & Jouannaud, 1990) pp. 252) the places where replacement is performed are simply given by addresses instead of paths in the term.

Definition 7 (Replacement)
Let t, u be addressed terms. The replacement generated by u in t, denoted by $repl(u)(t)$ is defined as follows:

$$repl(u)(X) \triangleq X$$

$$repl(u)(\bullet^a) \triangleq \begin{cases} u \mathbin{@} a & \text{if } a \in addr(u) \\ \bullet^a & \text{otherwise,} \end{cases}$$

$$repl(u)\big(F^a(t_1, \ldots, t_m)\big) \triangleq \begin{cases} u \mathbin{@} a & \text{if } a \in addr(u) \\ F^a\big(repl(u)(t_1), \ldots, repl(u)(t_m)\big) & \text{otherwise} \end{cases}$$

Proposition 8 (Replacement Admissibility)

If t and u are addressed terms, then $repl(u)(t)$ is an addressed term.

Proof
By induction on the structure of t as a dangling term. We show more generally that if t and u are dangling terms whose dangling addresses are in any set $\mathcal{A}$, then $repl(u)(t)$ is a dangling term whose dangling addresses are in $\mathcal{A}$. Since an addressed term is a dangling term without dangling address, the intended result follows. □

Example 5

1. Let t be $\mathtt{x}[\langle\,\rangle[\,]^a/\mathtt{z};\langle\,\rangle[\,]^a/\mathtt{y}]^b$, and u be $[\![\langle\,\rangle^c]\!]^a$.
   The replacement generated by u in t gives $\mathtt{x}[[\![\langle\,\rangle^c]\!]^a/\mathtt{x};[\![\langle\,\rangle^c]\!]^a/\mathtt{y}]^b$.
2. Let t be
   $$\mathtt{x}[([\![\langle\,\rangle^c]\!]^a \leftarrow\!: \mathtt{m} = (\lambda \mathtt{s}.\mathtt{s})[\,]^d)^e/\mathtt{x};[\![\langle\,\rangle^c]\!]^a/\mathtt{y}]^b$$
   and u be
   $$\lceil [\![\langle\langle\,\rangle^c \leftarrow \mathtt{m} = (\lambda \mathtt{s}.\mathtt{s})[\,]^d\rangle^f]\!]^a \rceil^e.$$
   The replacement generated by u in t gives
   $$\mathtt{x}[u/\mathtt{x};[\![\langle\langle\,\rangle^c \leftarrow \mathtt{m} = (\lambda \mathtt{s}.\mathtt{s})[\,]^d\rangle^f]\!]^a/\mathtt{y}]^b.$$

We now define the notions of redex and rewriting.

Definition 9 (*Addressed Rewriting Rule*)
An addressed rewriting rule over Σ is a pair of addressed terms (l, r) over Σ, written $l \to r$, such that $loc(l) \equiv loc(r)$ (same top address, therefore l and r are not variables), and $var(r) \subseteq var(l)$ (no creation of variables). Moreover, if there are addresses a, b in $addr(l) \cap addr(r)$ such that $l \,@\, a$ and $l \,@\, b$ are unifiable, then $r \,@\, a$ and $r \,@\, b$ must be unifiable with the same unifier.

Definition 10 (*Redex*)
A term t *is a redex* for a rule $l \to r$, if t matches l. A term t *has a redex*, if there exists an address $a \in addr(t)$ such that $t \,@\, a$ is a redex.

Note that, in general, we do not impose restrictions as linearity in addresses (*i.e.* the same address may occur twice), or acyclicity of l and r. However, $\lambda\mathcal{O}bj^a$ is *linear* in addresses (addresses occur only once) and patterns are never cyclic. Cycles may only be introduced by the means of imperative update.

Beside redirecting pointers, ATRS create *new* nodes. *Fresh renaming* insures that these new node addresses are not already used.

Definition 11 (*Fresh Renaming*)

1. We denote by $dom(\varphi)$ and $rng(\varphi)$ the usual *domain* and *range* of a function φ.
2. A *renaming* is an injective address substitution.
3. Let t be a term having a redex for the addressed rewriting rule $l \to r$. A renaming α_{fresh} is *fresh* for $l \to r$ with respect to t if
   $$dom(\alpha_{\text{fresh}}) = addr(r) \setminus addr(l)$$
   i.e. the renaming renames each newly introduced address to avoid capture, and $rng(\alpha_{\text{fresh}}) \cap addr(t) = \emptyset$, *i.e.* the chosen addresses are not present in t.

Proposition 12 (*Substitution Admissibility*)
Given an admissible term t that has a redex for the addressed rewriting rule $l \to r$. Then

1. A fresh renaming α_{fresh} exists for $l \to r$ with respect to t.
2. $\langle \alpha \cup \alpha_{\text{fresh}}; \sigma \rangle(r)$ is admissible.

Proof
The admissibility of t and l ensures that the substitution $\langle \alpha; \sigma \rangle$ satisfies some well-formedness property, in particular the set $rng(\sigma)$ is a set of mutually admissible terms in the sense that the parts they share together are consistent (or in other words, the preterm obtained by giving these terms a common root, with a fresh address, is an addressed term).

The use of α_{fresh} both ensures that all addresses of r are in the domain of the substitution, and that their images by α will not clash with existing addresses.

The definition of substitution takes care in maintaining admissibility for such substitutions, in particular the management of back-pointers. These properties are sufficient to ensure the admissibility of $\langle \alpha \cup \alpha_{\text{fresh}}; \sigma \rangle(r)$. □

At this point, we have given all the definitions needed to specify rewriting.

Definition 13 (*Rewriting*)
Let t be a term which we want to reduce at address a by rule $l \to r$. Proceed as follows:

1. Ensure $t @ a$ is a redex. Let $\langle \alpha; \sigma \rangle(l) \triangleq t @ a$.
2. Compute α_{fresh}, a fresh renaming for $l \to r$ with respect to t.
3. Compute $u \equiv \langle \alpha \cup \alpha_{\text{fresh}}; \sigma \rangle(r)$.
4. The result s of rewriting t by rule $l \to r$ at address a is $repl(u)(t)$. We write the reduction $t \to s$, defining "$\to$" as the relation of all such rewritings.

Theorem 14 (*Closure under Rewriting*)
Let R be an addressed term rewriting system and t be an addressed term. If $t \to u$ in R then u is also an addressed term.

Proof
The proof essentially walks through the steps of Definition 13, showing that each step preserves admissibility.

Let $\mathcal{R}$ be an addressed term rewriting system and t be an addressed term rewritten by the rule $l \to r$. All of t, l, and r, are admissible. For the rewrite to be defined, we furthermore know that t has a redex

$$t' \triangleq t @ a \equiv \langle \alpha; \sigma \rangle(l)$$

By Proposition 5, t' is admissible and by Proposition 12, we can find a renaming α_{fresh} that is fresh for $l \to r$ with respect to t' and we know that $u \equiv \langle \alpha \cup \alpha_{\text{fresh}}; \sigma \rangle(r)$ is admissible. Proposition 8 finally ensures that $repl(u)(t)$ is admissible. □

6.3 Acyclic Mutation-free ATRS

In this subsection, we consider a particular sub-class of ATRS, namely the ATRS involving *no cycles* and *no mutation*. We show that this particular class of ATRS is sound to simulate Term Rewriting Systems.

Definition 15 (*Acyclicity and Mutation-freeness*)

- An addressed term is called *acyclic* if it contains no occurrence of $\bullet$.
- An ATRS rule $l \to r$ is called *acyclic* if l and r are acyclic.
- An ATRS is called *acyclic* if all its rules are acyclic.
- An ATRS rule $l \to r$ is called *mutation-free* if,

$$\forall\ a \in (addr(l) \cap addr(r)) \setminus \{loc(l)\},\ \text{ we have }\ l @ a \equiv r @ a.$$

- An ATRS is called *mutation-free* if all its rules are mutation-free.

The following definition aims at making a relation between an ATRS and Term Rewriting System. We define mappings from addressed terms to algebraic terms, and from addressed terms to algebraic contexts.

Definition 16 (*Mappings*)

- An ATRS to TRS mapping is a homomorphism ϕ from acyclic addressed preterms to finite terms such that, for some function set $\{F_\phi \mid F \in \Sigma\}$ where each F_ϕ is either a projection or a constructor:

$$\phi(X) \triangleq X$$
$$\phi(F^a(t_1, \ldots, t_n)) \triangleq F_\phi(\phi(t_1), \ldots, \phi(t_n))$$

- Given an ATRS to TRS mapping ϕ, and an address a, we define ϕ_a as a mapping from addressed preterms to multi-hole contexts, such that all subterms at address a (if any) are replaced with holes, written $\Diamond$. More formally,

$$\phi_a(X) \triangleq X$$
$$\phi_a(F^b(t_1, \ldots, t_n)) \triangleq \begin{cases} \Diamond & \text{if } a \equiv b \\ F_\phi(\phi_a(t_1), \ldots, \phi_a(t_n)) & \text{otherwise} \end{cases}$$

- Given a context C containing zero or more holes, we write $C[t]$ the term obtained by filling all holes in C with t.
- Given an ATRS to TRS mapping ϕ, we define the mapping ϕ_s from addressed terms substitutions to term substitutions as follows:

$$\phi_s(\sigma)(X) \triangleq \begin{cases} \phi(\sigma(X)) & \text{if } X \in dom(\sigma) \\ X & \text{otherwise} \end{cases}$$

Lemma 17 (*Mappings and Contexts*)
Let t be an acyclic addressed term, and a an address. Then $\phi(t) \equiv \phi_a(t)[\phi(t @ a)]$.

Proof
The case $a \notin addr(t)$ is obvious since there is no hole to fill. Now let $a \in addr(t)$. We prove the lemma by structural induction on t.

- t is obviously not a variable since it must contain at least address a.
- Let t be $F^b(t_1, \ldots, t_n)$. The case $a \equiv b$ is trivial. Otherwise,

$$\phi(t) \equiv F_\phi(\phi(t_1), \ldots, \phi(t_n))$$

For each t_i, either $a \notin addr(t_i)$ and then $\phi(t_i) \equiv \phi_a(t_i)$, or $a \in addr(t_i)$ and then by induction hypothesis $\phi(t_i) \equiv \phi_a(t_i)[\phi(t_i @ a)]$. Since t is acyclic, then $t_i @ a \equiv t @ a$, hence $\phi(t_i) \equiv \phi_a(t_i)[\phi(t @ a)]$. Therefore, $\phi(t) \equiv F_\phi(\phi_a(t_1), \ldots, \phi_a(t_n))[\phi(t @ a)] \equiv \phi_a(t)[\phi(t @ a)]$ as desired.

□

Lemma 18 (*Replacements and Contexts*)
Let t and u be acyclic addressed terms where $a \triangleq loc(u)$ is the only address in $addr(t) \cap addr(u)$ such that $u @ a \not\equiv t @ a$. Then

1. $repl(u)(t)$ is acyclic.
2. $\phi(repl(u)(t)) \equiv \phi_a(t)[\phi(u)]$.

Proof

1. Trivial according to the definition of replacement (no folding).
2. By structural induction on t.
   - If t is a variable X, then

$$\phi(repl(u)(X)) \equiv \phi(X) \equiv X \equiv X[\phi(u)] \equiv \phi_a(X)[\phi(u)]$$

   - Let t be $F^b(t_1, \ldots, t_n)$. Two cases are to be considered:

   (a) $a \equiv b$: then

$$\phi(repl(u)(F^a(t_1, \ldots, t_n))) \equiv \phi(u) \equiv \Diamond[\phi(u)] \equiv \phi_a(t)[\phi(u)]$$

   (b) $a \not\equiv b$: note that since $repl(u)(t)$ is acyclic, then

$$\phi(repl(u)(F^b(t_1, \ldots, t_n))) \equiv \phi(F^b(repl(u)(t_1), \ldots, repl(u)(t_n))$$

   (no folding). Hence, by induction hypothesis,

$$\begin{aligned}\phi(repl(u)(F^b(t_1, \ldots, t_n))) &\equiv F_\phi(\phi_a(t_1)[\phi(u)], \ldots, \phi_a(t_n)[\phi(u)]) \\ &\equiv F_\phi(\phi_a(t_1), \ldots, \phi_a(t_n))[\phi(u)] \\ &\equiv \phi_a(t)[\phi(u)]\end{aligned}$$

□

Lemma 19 (*Mapping and Substitution*)
If $\langle\alpha;\sigma\rangle(t)$ is acyclic, then $\phi(\langle\alpha;\sigma\rangle(t)) \equiv \phi_s(\sigma)(\phi(t))$.

Proof
By structural induction on t.

- If $t \equiv X$, then

$$\phi(\langle\alpha;\sigma\rangle(X)) \equiv \phi(\sigma(X)) \equiv \phi_s(\sigma)(X) \equiv \phi_s(\sigma)(\phi(X))$$

- If $t \equiv F^a(t_1, \ldots, t_n)$, then

$$\phi(\langle\alpha;\sigma\rangle(t)) \equiv \phi(F^{\alpha(a)}(\langle\alpha;\sigma\rangle(t_1), \ldots, \langle\alpha;\sigma\rangle(t_n)))$$

since $\langle\alpha;\sigma\rangle(t)$ is acyclic (fold is unnecessary). Hence,

$$\phi(\langle\alpha;\sigma\rangle(t)) \equiv F_\phi(\phi(\langle\alpha;\sigma\rangle(t_1)), \ldots, \phi(\langle\alpha;\sigma\rangle(t_n)))$$

On the other hand,

$$\phi_s(\sigma)(\phi(t)) \equiv \phi_s(\sigma)(F_\phi(\phi(t_1), \ldots, \phi(t_n))) \equiv F_\phi(\phi_s(\sigma)(\phi(t_1)), \ldots, \phi_s(\sigma)(\phi(t_n)))$$

By induction hypothesis, $\phi(\langle\alpha;\sigma\rangle(t_i)) \equiv \phi_s(\sigma)(\phi(t_i))$.

□

Lemma 20 (*In-term Substitution*)
Let $\langle\alpha;\sigma\rangle(t)$ be an acyclic addressed term, and let $b \in addr(t)$, such that $\alpha(b) \equiv a$. Then $\langle\alpha;\sigma\rangle(t) @ a \equiv \langle\alpha;\sigma\rangle(t @ b)$.

Proof
By structural induction on t.

- t cannot be a variable since it must contain address b.
- If t be $F^c(t_1, \ldots, t_n)$, then we consider two cases:
  1. $b \equiv c$:

$$\langle\alpha;\sigma\rangle(t) @ a \equiv F^a(\langle\alpha;\sigma\rangle(t_1), \ldots, \langle\alpha;\sigma\rangle(t_n)) \equiv \langle\alpha;\sigma\rangle(t @ b)$$

  2. $b \not\equiv c$:

$$\langle\alpha;\sigma\rangle(t) @ a \equiv F^{\alpha(c)}(\langle\alpha;\sigma\rangle(t_1), \ldots, \langle\alpha;\sigma\rangle(t_n)) @ a$$

There must be some t_i such that

$$F^{\alpha(c)}(\langle\alpha;\sigma\rangle(t_1), \ldots, \langle\alpha;\sigma\rangle(t_n)) @ a \equiv \langle\alpha;\sigma\rangle(t_i) @ a$$

Hence, by induction hypothesis,

$$\langle\alpha;\sigma\rangle(t) @ a \equiv \langle\alpha;\sigma\rangle(t_i @ b) \equiv \langle\alpha;\sigma\rangle(t @ b)$$

□

Lemma 21 (*In-terms Conservation*)
Let $l \to r$ be an acyclic mutation-free rule, t an acyclic addressed term, and b an address in t such that $t @ b \equiv \langle\alpha;\sigma\rangle(l)$ (*i.e.*, $t @ b$ is a redex). Let α_f be a fresh address renaming for $l \to r$ *w.r.t.* t and $u \triangleq \langle\alpha \cup \alpha_f;\sigma\rangle(r)$.

1. u is acyclic.
2. $\forall a \in (addr(t) \cap addr(u)) \setminus \{b\}, t @ a \equiv u @ a$.

Proof

1. We show the acyclicity of u by contradiction. Assume u is cyclic. Since r is acyclic, there exists a sub-term of r of the form $F^c(t_1, \ldots, t_n)$ such that for one of the t_i, $\langle\alpha;\sigma\rangle(t_i)$ contains $\alpha(c)$ (*e.g.*, there is a fold that produces a $\bullet$).

Obviously, c is neither a fresh address nor the location of l and r, otherwise t_i would necessarily contain c *i.e.*, r would be cyclic.
Hence, c is another address of l. We know by hypothesis that $l @ c \equiv r @ c$ *i.e.*, that $l @ c \equiv F^c(t_1, \ldots, t_n)$, and that t is acyclic *i.e.*, that $\langle \alpha; \sigma \rangle(t_i)$ does not contain $\alpha(c)$. Obviously, this is also true for $\langle \alpha \cup \alpha_f; \sigma \rangle(t_i)$ since t_i does not contain fresh addresses (it belongs to l). This contradicts the hypothesis. We conclude that $\langle \alpha \cup \alpha_f; \sigma \rangle(r)$ is acyclic.

2. Assume there is $a \in (addr(t) \cap addr(u)) \setminus \{b\}$ such that $t @ a \not\equiv u @ a$. Then a may have three distinct origins:
   (a) a is a fresh address in u. Obviously, this is not possible since by hypothesis $a \in addr(t)$.
   (b) There is an address $c \in addr(l) \cap addr(r)$ such that $\alpha(c) \equiv a$. In this case,
   $$u @ a \equiv \langle \alpha \cup \alpha_f; \sigma \rangle(r) @ a \equiv \langle \alpha \cup \alpha_f; \sigma \rangle(r @ c)$$
   from Lemma 20 and acyclicity of u. Similarly,
   $$t @ a \equiv \langle \alpha; \sigma \rangle(l) @ a \equiv \langle \alpha; \sigma \rangle(l @ c)$$
   From the hypothesis, we know that $r @ c \equiv l @ c$, hence $r @ c$ contains no fresh address *i.e.*, $u @ a \equiv \langle \alpha; \sigma \rangle(r @ c) \equiv \langle \alpha; \sigma \rangle(l @ c) \equiv t @ a$, which contradicts the hypothesis.
   (c) There is no address of l mapping to a. Since t is acyclic, $\langle \alpha; \sigma \rangle(l)$ makes no folding *i.e.*, for all sub-term of l of the form $F^c(t_1, \ldots, t_n)$, we have
   $$\langle \alpha; \sigma \rangle(F^c(t_1, \ldots, t_n)) \equiv F^{\alpha(c)}(\langle \alpha; \sigma \rangle(t_1), \ldots, \langle \alpha; \sigma \rangle(t_n))$$
   Hence, there must be a variable X in l such that $\sigma(X)$ contains a. According to the previous observation, $t @ a \equiv \sigma(X) @ a$. Since $\langle \alpha; \sigma \rangle(r)$ is acyclic, it must also be the case that $u @ a \equiv \sigma(X) @ a$.

□

Theorem 22 (*TRS Simulation*)
Let $S = \{l_i \to r_i \mid i = 1..n\}$ be an acyclic mutation-free ATRS, and t an acyclic term. If $t \to u$ in S, then $\phi(t) \to^+ \phi(u)$ in the system
$$\phi(S) = \{\phi(l_i) \to \phi(r_i) \mid i = 1..n\}$$

Proof
From the definition of addressed term rewriting, we have that $t \to repl(t)(u)$ where there are $a, \alpha, \alpha_f, \sigma, l, r$ such that $t @ a \equiv \langle \alpha; \sigma \rangle(l)$, and $u \equiv \langle \alpha \cup \alpha_f; \sigma \rangle(r)$. From Lemma 17, we have $\phi(t) \equiv \phi_a(t)[\phi(t @ a)]$. Note that $a \in addr(t)$, hence $\phi_a(t)$ contains at least one hole. On the other hand, from Lemmas 18 and 21, we have $\phi(repl(u)(t)) \equiv \phi_a(t)[\phi(u)]$. We just have to show that $\phi(t @ a) \to \phi(u)$ by rule $\phi(l) \to \phi(r)$. This is immediate from Lemma 19 since $\phi(t @ a) \equiv \phi(\langle \alpha; \sigma \rangle(l)) \equiv \phi_s(\sigma)(\phi(l))$, and $\phi(u) \equiv \phi(\langle \alpha \cup \alpha_f; \sigma \rangle(r)) \equiv \phi_s(\sigma)(\phi(r))$, and obviously, $\phi_s(\sigma)(\phi(l)) \to \phi_s(\sigma)(\phi(r))$, by rule $\phi(l) \to \phi(r)$. □

$$
\begin{array}{lll}
M, N & ::= & \lambda x.M \mid MN \mid x \mid c \mid \\
 & & \langle\rangle \mid \langle M \leftarrow m = N\rangle \mid M \Leftarrow m \qquad \text{(Code)} \\
U, V & ::= & M[s] \mid UV \mid \\
 & & U \Leftarrow m \mid \langle U \Leftarrow m = V\rangle \mid \mathit{Sel}(O, m, U) \qquad \text{(Eval. Contexts)} \\
O & ::= & \langle\rangle \mid \langle O \Leftarrow m = V\rangle \qquad \text{(Object Structures)} \\
s & ::= & U/x \,.\, s \mid \mathsf{id} \qquad \text{(Substitutions)}
\end{array}
$$

Fig. 13. The Syntax of $\lambda\mathcal{O}bj^{\sigma}$

6.4 The Calculus $\lambda\mathcal{O}bj^{\sigma}$ and its Relation with $\lambda\mathcal{O}bj^{a}$

The calculus $\lambda\mathcal{O}bj^{\sigma}$ mentioned in the introduction is intermediate between the calculus $\lambda\mathcal{O}bj$ of Fisher, Honsell and Mitchell (Fisher *et al.*, 1994) and our $\lambda\mathcal{O}bj^{a}$. It does not use addresses (see the syntax in Figure 13) and so in particular it does not model mutation. The syntax of $\lambda\mathcal{O}bj^{\sigma}$ is presented in Figure 14; the reader will note that terms of this calculus are terms of $\lambda\mathcal{O}bj^{a}$ without the addresses, indirections, and object identities, and the rules are those of modules $\mathsf{L} + \mathsf{C} + \mathsf{F}$ of $\lambda\mathcal{O}bj^{a}$.

In this section we establish some fundamental results about the relationship between $\lambda\mathcal{O}bj^{\sigma}$ and $\lambda\mathcal{O}bj^{a}$: informally we say that $\lambda\mathcal{O}bj^{a}$ is a *conservative extension* of $\lambda\mathcal{O}bj^{\sigma}$ in the sense that for an acyclic mutation-free term, computations in $\lambda\mathcal{O}bj^{a}$ and computations in $\lambda\mathcal{O}bj^{\sigma}$ return the same normal form. Since a $\lambda\mathcal{O}bj^{a}$-term yields a $\lambda\mathcal{O}bj^{\sigma}$-term by erasing addresses and indirections, one corollary of this conservativeness is *address-irrelevance*, *i.e.* the observation that the program layout in memory cannot affect the eventual result of the computation. This is an example of how an informal reasoning about implementations can be translated in $\lambda\mathcal{O}bj^{a}$ and formally justified.

As a first step we note that the results presented in Section 6.3 are applicable to $\lambda\mathcal{O}bj^{\sigma}$.

Definition 23 (*Mapping $\lambda\mathcal{O}bj^{a}$ to $\lambda\mathcal{O}bj^{\sigma}$*)
Let ϕ be the mapping from acyclic $\lambda\mathcal{O}bj^{a}$-terms which do not contain the $\leftarrow$: operator and the $\mathsf{clone}(x)$ term, and which erases addresses, indirection nodes ($\lceil\_\rceil^{a}$), and object identities ($[\![\_]\!]^{a}$), and leaves all the other symbols unchanged. Each term $\phi(U)$ is a term of $\lambda\mathcal{O}bj^{\sigma}$.

Theorem 24 ($\lambda\mathcal{O}bj^{\sigma}$ *Simulates* $\lambda\mathcal{O}bj^{a}$)
Let U be an acyclic $\lambda\mathcal{O}bj^{a}$-term which does not contain the $\leftarrow$: and $\mathsf{clone}(x)$ symbols. If $U \to V$ in $\mathsf{L} + \mathsf{C} + \mathrm{F}$, then $\phi(U) \to^{*} \phi(V)$ in $\lambda\mathcal{O}bj^{\sigma}$.

Proof
It relies on Theorem 22. Just notice that each rule $l \to r$ of modules $\mathsf{L} + \mathsf{C} + \mathrm{F}$ is acyclic and mutation-free, and that either it maps to a rule $\phi(l) \to \phi(r)$ which belongs to $\lambda\mathcal{O}bj^{\sigma}$, or it is such that $\phi(l) \equiv \phi(r)$. □

Basics for Substitutions

$$(MN)[s] \to (M[s]\,N[s]) \qquad \text{(App)}$$

$$\big((\lambda x.M)[s]\,U\big) \to M[U/x\,.\,s] \qquad \text{(Bw)}$$

$$x[U/y\,.\,s] \to x[s] \quad x \not\equiv y \qquad \text{(RVar)}$$

$$x[U/x\,.\,s] \to U \qquad \text{(FVar)}$$

$$\langle M \leftarrow m = N\rangle[s] \to \langle M[s] \leftarrow m = N[s]\rangle \qquad \text{(P)}$$

Method Invocation

$$\langle\,\rangle[s] \to \langle\,\rangle \qquad \text{(NO)}$$

$$(M \Leftarrow m)[s] \to (M[s] \Leftarrow m) \qquad \text{(SP)}$$

$$(O \Leftarrow m) \to Sel(O, m, O) \qquad \text{(SA)}$$

$$Sel(\langle O \leftarrow m = U\rangle, m, V) \to (U\,V) \qquad \text{(SU)}$$

$$Sel(\langle O \leftarrow n = U\rangle, m, V) \to Sel(O, m, V) \quad m \not\equiv n \qquad \text{(NE)}$$

Fig. 14. The Rules of $\lambda\mathcal{O}bj^{\sigma}$

Another issue, tackled by the following theorem, is to prove that all normal forms of $\lambda\mathcal{O}bj^{\sigma}$ can also be obtained in $\mathsf{L} + \mathsf{C} + \mathsf{F}$ of $\lambda\mathcal{O}bj^{a}$.

Theorem 25 (*Completeness of $\lambda\mathcal{O}bj^{a}$ w.r.t. $\lambda\mathcal{O}bj^{\sigma}$*)
If $M \to^{*} N$ in $\lambda\mathcal{O}bj^{\sigma}$, such that N is a normal form, then there is some U such that $\phi(U) \equiv N$ and $M[\,]^{a} \to^{*} U$ in $\mathsf{L} + \mathsf{C} + \mathsf{F}$ of $\lambda\mathcal{O}bj^{a}$.

Proof
The result follows from the facts that

1. Each rule of $\lambda\mathcal{O}bj^{\sigma}$ is mapped by a rule of $\mathsf{L} + \mathsf{C} + \mathsf{F}$.
2. Rules of $\lambda\mathcal{O}bj^{a}$ are left linear in addresses, hence whenever a $\lambda\mathcal{O}bj^{\sigma}$-term matches the left-hand side of a rule, whatever the addresses of a similar addressed term are, it matches the left-hand side of a rule in $\mathsf{L} + \mathsf{C} + \mathsf{F}$ of $\lambda\mathcal{O}bj^{a}$.
3. Whenever an acyclic $\lambda\mathcal{O}bj^{a}$-term U contains an indirection node $\lceil\_\rceil^{a}$, this node may be eliminated using one of rules (AppRed), (LCop), (SRed), or (FRed) (hence, indirection nodes cannot permanently *hide* some redexes).

4. Object identities $[\![\_]\!]^a$ are harmless as well since they systematically occur on top of $\lambda\mathcal{O}bj^a$-terms of sort O, and only there.

□

The last issue is to show that $\mathsf{L} + \mathsf{C} + \mathsf{F}$ of $\lambda\mathcal{O}bj^a$ does not introduce non-termination *w.r.t.* $\lambda\mathcal{O}bj^\sigma$.

Theorem 26 (*Preservation of Strong Normalization*)
If M is a strongly normalizing $\lambda\mathcal{O}bj^\sigma$-term, then all $\lambda\mathcal{O}bj^a$-term U such that $\phi(U) \equiv M$ is also strongly normalizing.

Proof
This relies on two facts: first, Theorem 22 states that one step in $\mathsf{L} + \mathsf{C} + \mathsf{F}$ of $\lambda\mathcal{O}bj^a$ maps to at least one step in $\phi(\mathsf{L} + \mathsf{C} + \mathsf{F})$ (that is $\lambda\mathcal{O}bj^\sigma$ plus a set of rules of the form $M \to M$); second, the system of rules $l \to r$ for which $\phi(l) \equiv \phi(r)$ (AppRed, LCop, NO, SP, SRed, FP, FC, FRed) is Noetherian. The second fact can be shown using the following functions π and δ, and verifying that whenever $U \to V$ by one of these rules, then $\pi(U)$ is greater than $\pi(V)$:

$$\begin{aligned}
\pi(M[s]^a) &\triangleq \delta(M) \times 2 \\
\pi((UV)^a) &\triangleq \pi(U) + \pi(V) \\
\pi((U \Leftarrow m)^a) &\triangleq \pi(U) \\
\pi(\langle U \leftarrow m = V\rangle^a) &\triangleq \pi(U) + \pi(V) \\
\pi([\![O]\!]^a) &\triangleq 1 \\
\pi(\lceil U \rceil^a) &\triangleq \pi(U) + 1
\end{aligned}$$

$$\begin{aligned}
\delta(MN) &\triangleq \delta(M) + \delta(N) + 1 \\
\delta(\langle M \leftarrow m = N\rangle) &\triangleq \delta(M) + \delta(N) + 1 \\
\delta(M \Leftarrow m) &\triangleq \delta(M) + 1 \\
\delta(M) &\triangleq 1 \text{ in all other cases}
\end{aligned}$$

Both considerations permit to conclude that if U had an infinite computation in $\mathsf{L} + \mathsf{C} + \mathsf{F}$ of $\lambda\mathcal{O}bj^a$, then $\phi(U)$ would have an infinite computation in $\lambda\mathcal{O}bj^\sigma$. □

7 Conclusions

We have presented a framework to describe many object-based calculi. To our knowledge, this framework has no equivalent in the literature; it has the following features:

- It is computationally complete since the λ-calculus is explicitly built-in to the language of expressions.
- It gives an account of the delegation-based techniques of inheritance.
- It is compatible with dynamic object extension and self-extension in the style of (Gianantonio *et al.*, 1998).

- It is generic, due to the partition of rules in independent modules, which can be combined to model (for example) functional *vs.* imperative implementations.
- It supports the analysis of implementations at the level of resource usage, as it models sharing of computations and sharing of storage, and each computation-step in the calculus corresponds to a constant-cost computation in practice.
- It is founded on a novel and mathematically precise theory, *i.e.*, addressed term rewriting systems.

Furthermore, $\lambda\mathcal{O}bj^a$ is generic in the sense that many strategies may be implemented. We have not given any definition of particular strategies since we do not want to privilege one strategy over another.

The approach for functional languages studied in (Benaissa *et al.*, 1996) should be generalizable to $\lambda\mathcal{O}bj^a$: from a very general point of view, a strategy is a binary relation between addressed terms and addresses. The addresses, in relation with a given term, determines which redexes of the term has to be reduced next (note that in a given term at a given address, at most one rule applies). This is a restriction *w.r.t.* the calculus in which not all the redexes may be reduced. If this relation is a one-to-many relation, the strategy is *non-deterministic*. If this relation is a function, then the strategy is *deterministic and sequential*. If this function is computable, then the strategy is *computable*.

Implementors and designers of languages are usually interested in some subclass of the computable strategies, that follows some locality principle—namely that a lot of reductions happen in a small connected part of the whole structure before "jumping" to another distant part. The definition of such strategies—which includes the usual call-by-value, call-by-name, call-by-reference, *etc.*—can be expressed using a very simple set of inference rules (those rules will be collected in another module of $\lambda\mathcal{O}bj^a$ not presented in this paper). These rules can be combined, as basic building blocks, provided possible conditions on their application, to define a lot of strategies.

We plan to extend $\lambda\mathcal{O}bj^a$ to handle the embedding-based technique of inheritance, following (Lang *et al.*, 1999a), to include a type system consistent with imperative features and with the ability to type objects extending themselves, following (Gianantonio *et al.*, 1998). Finally a prototype of $\lambda\mathcal{O}bj^a$, should make it easy to embed specific calculi and to make experiments on the design of realistic object oriented languages.

Acknowledgements

Suggestions by Maribel Fernandez are gratefully acknowledged: they were very helpful in improving the technical presentation of the paper.

The present version of this paper has been deeply influenced by comments and remarks of anonymous referees. In particular, Section 6 was almost completely rewritten following their advises. Therefore the authors feel strongly indebted to the referees.

A "proof-of-concept" version of this paper was presented at the RTA's satellite workshop Westapp (Dougherty *et al.*, 2001); the final version of the paper was also

influenced by the many questions and suggestions raised by the audience during and after the talk.

References

Abadi, M., & Cardelli, L. (1996). *A Theory of Objects.* Springer-Verlag.

Abadi, M., Cardelli, L., Curien, P.-L., & Lévy, J.-J. (1991). Explicit Substitutions. *Journal of Functional Programming*, **1**(4), 375–416.

Abelson, H., Sussman, G. J., & Sussman, J. (1985). *Structure and Interpretation of Computer Programs.* MIT Press.

Ariola, Z. M., & Arvind. (1995). Properties of a First-order Functional Language with Sharing. *Theoretical Computer Science*, **146**(1–2), 69–108.

Ariola, Z. M., & Klop, J. W. (1994). Cyclic Lambda Graph Rewriting. *Pages 416–425 of: Proc of LICS.* IEEE Computer Society Press.

Ariola, Z. M., Felleisen, M., Maraist, J., Odersky, M., & Wadler, Ph. (1995). A Call-By-Need Lambda Calculus. *Pages 233–246 of: Proc. of POPL.* ACM Press.

Asperti, A., & Laneve, C. (1994). Interaction Systems I: The Theory of Optimal Reductions. *Mathematical Structures in Computer Science*, **4**(4), 457–504.

Asperti, A., & Laneve, C. (1996). Interaction Systems II: The Practice of Optimal Reductions. *Theoretical Computer Science*, **159**.

Augustson, L. (1984). A Compiler for Lazy ML. *Pages 218–227 of: Symposium on lisp and functional programming.* ACM Press.

Baader, F., & Nipkow, T. (1998). *Term Rewriting and All That.* Cambridge University Press.

Barendregt, H. P., van Eekelen, M. C. J. D., Glauert, J. R. W., Kennaway, J. R., Plasmeijer, M. J., & Sleep, M. R. (1987). Term Graph Rewriting. *Pages 141–158 of: Proc. of PARLE.* Lecture Notes in Computer Science, no. 259. Springer-Verlag.

Benaissa, Z.-E.-A., Rose, K.H., & Lescanne, P. (1996). Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution. *Pages 393–407 of: Proc. of PLILP.* Lecture Notes in Computer Science, no. 1140. Springer-Verlag.

Blom, S. C. (2001). *Term Graph Rewriting, Syntax and Semantics.* Ph.D. thesis, Vrije Universiteit, Amsterdam.

Bloo, R., & Rose, K. H. (1995). Preservation of Strong Normalisation in Named Lambda Calculi with Explicit Substitution and Garbage Collection. *Pages 62–72 of: Computer Science in the Netherlands.*

Bono, V., & Fisher, K. (1998). An Imperative, First-Order Calculus with Object Extension. *Pages 462–497 of: European Conference for Object-Oriented Programming.* Lecture Notes in Computer Science, no. 1445. Springer-Verlag.

Cardelli, L. (1995). A Language with Distributed Scope. *Computing systems*, **8**(1), 27–59.

Chambers, C. (1993). *The Cecil Language Specification, and Rationale.* Tech. rept. 93-03-05. Department of Computer Science and Engineering, University of Washington, USA.

Ciaffaglione, A., Gianantonio, P. Di, Honsell, F., & Liquori, L. (2001). *Toward a Typed Foundation of Object Reclassification.* In preparation.

Dershowitz, N., & Jouannaud, J.-P. (1990). *Handbook of Theoretical Computer Science.* Vol. B. Elsevier Science Publishers. Chap. 6: Rewrite Systems, pages 244–320.

Dougherty, D., & Lescanne, P. (2001). Reductions, Intersection Types, and Explicit Substitutions. *Pages 121–135 of: Proc. of TLCA.* Lecture Notes in Computer Science, no. 2044. Springer-Verlag.

Dougherty, D., Lang, F., Lescanne, P., Liquori, L., & Rose, K. (2001). A Generic Object-Calculus based on Addressed Term Rewriting Systems. *Pages 6–25 of:* Lescanne, P. (ed), *Proc. of WESTAPP'01, fourth international workshop on explicit substitutions: Theory and applications to programs and proofs.* Logic Group Preprint series No 210. Utrecht University, the Netherlands.

Ehrig, H., Engels, G., Kreowski, H-J., & Rozenberg, G. (eds). (1999). *Handbook of Graph Grammars and Computing by Graph Transformation Vol 2: Applications, Languages and Tools.* World Scientific.

Felleisen, M., & Friedman, D. P. (1989). A Syntactic Theory of Sequential State. *Theoretical Computer Science*, **69**, 243–287.

Felleisen, M., & Hieb, R. (1992). The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, **102**.

Fisher, K., & Reppy, J. H. (1999). The Design of a Class Mechanism for Moby. *Pages 37–49 of: Proc. of PLDI.* SIGPLAN Notices, vol. 34. ACM Press.

Fisher, K., & Reppy, J. H. (2000). Extending Moby with Inheritance-based Subtyping. *Pages 83–107 of: Proc. of ECOOP.* Lecture Notes in Computer Science, vol. 1850. Springer-Verlag.

Fisher, K., Honsell, F., & Mitchell, J. C. (1994). A Lambda Calculus of Objects and Method Specialization. *Nordic journal of computing*, **1**(1), 3–37.

Fisher, K., Grossmann, D., MacQueen, D., Pucella, R., Reppy, J., Riecke, J., & Weirich, S. (2000). *The Moby Programming Language.* `http://www.cs.bell-labs.com/who/jhr/moby/index.html`.

Gianantonio, P. Di, Honsell, F., & Liquori, L. (1998). A Lambda Calculus of Objects with Self-inflicted Extension. *Pages 166–178 of: Proc. of OOPSLA.* ACM Press.

Gonthier, G., Abadi, M., & Lévy, J.-J. (1992a). Linear Logic Without Boxes. *Pages 223–34 of: Proc. of LICS.* IEEE Computer Society Press.

Gonthier, G., Abadi, M., & Lévy, J.-J. (1992b). The Geometry of Optimal Lambda Reduction. *Proc. of POPL*, 15–26.

Kahn, G. (1987). *Natural Semantics.* Tech. rept. RR-87-601. Institut National de Recherche en Informatique et en Automatique, Sophia Antipolis, France.

Klop, J. W. (1990). Term Rewriting Systems. *Chap. 6 of:* Abramsky, S., Gabbay, D., & Maibaum, T. (eds), *Handbook of logic in computer science*, vol. 1. Oxford University Press.

Landin, P. J. (1964). The Mechanical Evaluation of Expressions. *Computer Journal*, **6**.

Lang, F. (1998). *Modèles de la β-réduction pour les Implantations.* Ph.D. thesis, École Normale Supérieure de Lyon.

Lang, F., Lescanne, P., & Liquori, L. (1999a). A Framework for Defining Object-Calculi (Extended Abstract). *Pages 963–982 of: Proc. of FM.* Lecture Notes in Computer Science, no. 1709. Springer-Verlag.

Lang, F., Dougherty, D., Lescanne, P., & Rose, K. (1999b). *Addressed Term Rewriting Systems.* Tech. rept. RR 1999-30. Laboratoire de l'Informatique du Parallèlisme, ENS de Lyon, France.

Lescanne, P. (1994). From $\lambda\sigma$ to $\lambda\upsilon$, a Journey Through Calculi of Explicit Substitutions. *Pages 60–69 of: Principles of Programming Languages.*

Lévy, J.-J. (1980). Optimal Reductions in the Lambda-calculus. *Pages 159–191 of:* Seldin, J. P., & Hindley, J. R. (eds), *To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism.* Academic Press.

Mackie, I. (1995). The Geometry of Interaction Machine. *Pages 198–208 of: Proc. of POPL.* ACM Press.

Maranget, L. (1992). Optimal Derivations in Weak Lambda Calculi and in Orthogonal Rewriting Systems. *Pages 255–268 of: Proc. of POPL.*

Mason, I., & Talcott, C. (1991). Equivalence in Functional Languages with Effects. *Journal of Functional Programming*, **1**(3), 287–327.

Milner, R., Tofte, M., & Harper, R. (1990). *The Definition of Standard ML.* MIT Press.

Mitchell, J. C. (1996). *Foundations for Programming Languages.* MIT Press.

Okasaki, Ch. (1998). *Purely Functional Data Structures.* Cambridge University Press.

Peyton-Jones, S. (1987). *The Implementation of Functional Programming Languages.* Prentice Hall.

Plasmeijer, M. J., & van Eekelen, M. C. D. J. (1993). *Functional Programming and Parallel Graph Rewriting.* International Computer Science Series. Addison-Wesley.

Plotkin, G. (1981). *A Structural Approach to Operational Semantics.* Tech. rept. DAIMI FN-19. Computer Science Department, Aarhus University, Denmark.

Plump, D. (1999). *Term Graph Rewriting.* World Scientific. in (Ehrig *et al.*, 1999). Chap. 1, pages 3–61.

Rose, K. H. (1996). *Operational Reduction Models for Functional Programming Languages.* Ph.D. thesis, DIKU, Kobenhavn, Denmark. DIKU report 96/1.

Sleep, R., Plasmeijer, R., & van Eekelen, M. C. D. J. (eds). (1993). *Term Graph Rewriting. Theory and Practice.* John Wiley Publishers.

Tailvalsaari, A. (1992). *Kevo, a Prototype-based Object-oriented Language based on Concatenation and Modules Operations.* Tech. rept. LACIR 92-02. University of Victoria, Canada.

Tofte, M. (1990). Type Inference for Polymorphic References. *Information and Computation*, **89**(1), 1–34.

Turner, D. A. (1979). A New Implementation Technique for Applicative Languages. *Software practice and experience*, **9**, 31–49.

Ungar, D., & Smith, R. B. (1987). Self: The Power of Simplicity. *Pages 227–241 of: Proc. of OOPSLA.* ACM Press.

Wadsworth, C. P. (1971). *Semantics and Pragmatics of the Lambda Calculus.* Ph.D. thesis, Oxford.

Wright, A. K., & Felleisen, M. (1994). A Syntactic Approach to Type Soundness. *Information and Computation*, **115**(1).