Under consideration for publication in Math. Struct. in Comp. Science

Addressed Term Rewriting Systems: Application to a Typed Object Calculus

Daniel J. Dougherty[1] and Pierre Lescanne[2] and Luigi Liquori[3]

[1] *Department of Computer Science, Worcester Polytechnic Institute Worcester, MA 01609 USA.*

[2] *Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon 46, Allée d'Italie, 69364 Lyon 07, FRANCE.*

[3] *INRIA Sophia Antipolis FR-06902 FRANCE.*

Received 14 October 2005; Revised 26 March 2006

We present a formalism called *Addressed Term Rewriting Systems,* which can be used to model implementations of theorem proving, symbolic computation, and programming languages, especially aspects of sharing, recursive computations and cyclic data structures. Addressed Term Rewriting Systems are therefore well suited for describing object-based languages, and as an example we present a language called $\lambda\mathcal{O}bj^a$, incorporating both functional and object-based features. As a case study in how reasoning about languages is supported in the ATRS formalism a type system for $\lambda\mathcal{O}bj^a$ is defined and a type soundness result is proved.

1. Introduction

1.1. *Addressed Calculi and Semantics of Sharing*

Efficient implementations of functional programming languages, computer algebra systems, theorem provers, and similar systems 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 & Klop 1996, Ariola, et al. 1995, Ariola & Arvind 1995) for applications to λ-calculus and implementations.

This paper bridges two approaches to the implementation of functionally or declarative programming language, namely *graph rewriting* and *term rewriting*. The former provides an abstract approach for an efficient implementation whereas the latter is closer

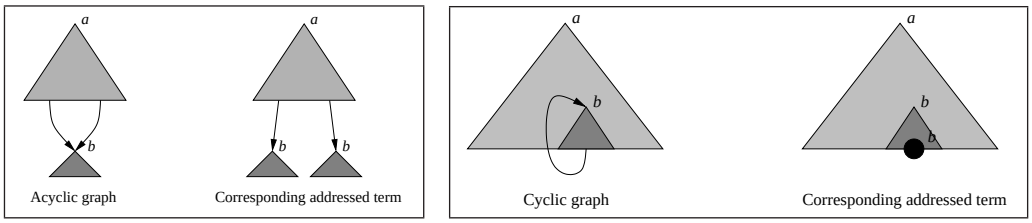

Figure 1. Sharing and Cycles Using Addresses

to ordinary mathematical notation and so stays closer to our intuition. The theory here applies to languages such as pure ML (Milner, et al. 1997, Leroy, et al. 2004), Haskell (Peyton Jones, et al. 2003), Clean (Plasmeijer & van Eekelen 2001), logic languages like Prolog (Kowalski 1979) or rule-based languages like Maude (Clavel, et al. 2003), Elan (Borovansky, et al. 1998) or Cafe-OBJ (Diaconescu & Futatsugi 1998).

In this paper we annotate terms, as trees, with *global addresses* in the spirit of (Felleisen & Friedman 1989, Rose 1996, Benaissa, et al. 1996). The approach to labeled term rewriting systems of Ohlebusch (Ohlebusch 2001) is similar to our formalism (in the acyclic case); 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 on in a computer or an abstract machine.

Labeling intermediate terms with addresses is a novel technique for the analysis of the implementation of functional languages. A standard treatment would consist of translating a big-step rewriting semantics into an abstract machine (Diehl, et al. 2000). In our formalism addresses are part of the evaluation-terms and they are *de facto* decorated with locations. As a consequence the small-step semantics given by rewriting and the abstract machines are strictly coupled: the thesis is that machine implementation can be accurately modeled by term rewriting.

The formalisms of term-graph rewriting and addressed-term rewriting share much similarity but we feel that the addressed-term setting has several advantages. Our intention is to define a calculus that is as close to actual implementations as possible with the view of terms as tree-like structure, 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.

With explicit global addresses we can keep track of the sharing that can be used in the implementation of a calculus. Sub-terms that share a common address represent the same sub-graphs, as suggested in Figure 1 (left), where a and b denote addresses. In (Dougherty, et al. 2002), 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 rewrite simultaneously all sub-terms sharing a same address, mimicking what would happen in an implementation.

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`). 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 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 Section 2.

1.2. *Suitability of Addressed TRS for describing an Object-based Formalism*

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, Gordon & Hankin 2000, Igarashi, et al. 2001).

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 formalism 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.

Several treatments of functional operational semantics exist in the literature (Landin 1964, Augustson 1984, Kahn 1987, Milner, et al. 1990). Addressed Term Rewriting Systems (originally motivated by implementations of lazy functional programming languages (Peyton-Jones 1987, Plasmeijer & van Eekelen 1993)) are the foundation of the $\lambda\mathcal{O}bj^a$ formalism (Lang, et al. 1999) for modeling object-oriented languages. The results in (Lang et al. 1999) showed how to model $\lambda\mathcal{O}bj^a$ using Addressed Term Rewriting Systems, but with no formal presentation of those systems. Here we expose the graph-based machinery underneath the rewriting semantics of $\lambda\mathcal{O}bj^a$. To our knowledge, term-graph rewriting has been little explored in the context of the analysis of object-based programming.

The novelty of $\lambda\mathcal{O}bj^a$ is that it provides a *homogeneous* approach to both functional and object-oriented 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).

A useful way to understand the $\lambda\mathcal{O}bj^a$ formalism is by analogy with graph-reduction as an implementation-calculus for functional programming. Comparing $\lambda\mathcal{O}bj^a$ with the implementation techniques of functional programming (FP) and object oriented programming (OOP) gives the following correspondence. The $\lambda\mathcal{O}bj^a$ "modules" L, C, and F are defined in section 3.

Paradigm	$\lambda\mathcal{O}bj^a$ **fragment**	**Powered by**
Pure FP	$\lambda\mathcal{O}bj^a$ (L)	ATRS
Pure FP+OOP	$\lambda\mathcal{O}bj^a$ (L+C+F)	ATRS

1.3. *Outline of the Paper*

The paper is organized as follows: Section 2 details the formalism of addressed term rewriting systems and establishes a general relation between addressed term rewriting systems and first-order term rewriting systems. Section 3 puts addressed term rewriting systems to work by presenting the three modules of rewriting rules that form the core of $\lambda\mathcal{O}bj^a$. For pedagogical convenience we proceed in two steps: first we present the calculus $\lambda\mathcal{O}bj^\sigma$, intermediate between the calculus $\lambda\mathcal{O}bj$ of Fisher, Honsell and Mitchell (Fisher et al. 1994) and our full calculus $\lambda\mathcal{O}bj^a$, then we scale up to $\lambda\mathcal{O}bj^a$itself. Section 4 presents a running object-based example in the $\lambda\mathcal{O}bj^a$ formalism. Section 5 addresses the relationship between $\lambda\mathcal{O}bj$ and $\lambda\mathcal{O}bj^a$. Section 6 proposes a type system for $\lambda\mathcal{O}bj^a$. Section 7 concludes. In appendix A we give a connection between a kind of ATRS's and TRS's.

This paper is an extended version of (Dougherty, et al. 2005) and originated in research presented in (Lang et al. 1999).

2. Addressed Term Rewriting Systems

In this section we introduce *addressed term rewriting systems* or ATRS. Classical term rewriting (Dershowitz & Jouannaud 1990, Klop 1990, Baader & Nipkow 1998) cannot easily express issues of sharing and mutation. Calculi that 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 that reflects memory usage and is robust enough to be independent of the rewriting strategy.

Sharing of computation. Consider the reduction $\mathsf{square}(x) \to \mathsf{times}(x, x)$. In order to share subterms, addresses are inserted in terms making them *addressed terms*. For instance if we are to compute $\mathsf{square}(\mathsf{square}(2))$, we attach addresses a, b, c to the individual subterms. This yields $\mathsf{square}^a(\mathsf{square}^b(2^c))$ which can then be reduced as follows:

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

where "$\rightsquigarrow$" designates a one step reduction with sharing. The key point of a shared computation is that *all* terms that share a common address are reduced *simultaneously*.

Sharing of Object Structures. It is important not only to share *computations*, but also to share *structures*. Indeed, objects are typically structures that receive multiple pointers. As an example, if we "zoom" on Figure 7, we can observe that the objects p and q share

a common structure addressed by b. This can be very easily formalized in the formalism, since addresses are first-class citizens. See Section 4.

Cycles. Cycles are essential in functional programming when one deals with infinite data-structures, as in lazy functional programming languages. Cycles are also used to save space in the code of recursive functions. Moreover in the context of object programming languages, cycles can be used to express *loops* which can be introduced in memory via lazy evaluation of recursive code.

2.1. *Addressed Terms*

Addressed terms are first order terms whose occurrences of operator symbols are 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. 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 2.1 (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

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

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

Example 2.1. The following are preterms

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

The first is "ill-formed" in the sense that the two subterms labeled with b are not the same; the latter two do not suffer from this anomaly, and are examples of what we call an *admissible* term: see Definition 2.4. For the first preterm above we have

— $loc(\mathsf{times}^b(2^d, 2^c))) = loc(\mathsf{times}^b(2^c, 2^c))) = b$,
— $var(\mathsf{times}^a(\mathsf{times}^b(x, x), \mathsf{times}^b(x, x))) = \{x\}$,
— $addr(\mathsf{times}^a(\mathsf{times}^b(2^c, 2^c), \mathsf{times}^b(2^c, 2^c))) = \{a, b, c\}$,

Addresses signify locations used to store a concrete term in memory; an open preterm t (a preterm with variables) is a notation to capture the family of concrete instantiations of t, and so the variables in t itself are never realized themselves. For this reason we do not assign addresses to variables in a term; they are meant to be replaced by addressed terms lying at any address.

The definition of a preterm makes use of a special symbol $\bullet$ called a *back-pointer* and used to denote *cycles* (Rose 1996). A back-pointer $\bullet^a$ in an addressed term must be such that a is an address occurring on 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.

Example 2.2. The infinite list made of 0's will be represented as $\mathsf{cons}^a(0^b, \bullet^a)$ and we can look for the 5^{th} element, by applying the function nth, as follows: $\mathsf{nth}^c(5^b, \mathsf{cons}^a(0^b, \bullet^a))$.

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 2.2 (Folding and Unfolding).

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

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

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

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

Given an address a, the intended meaning of *folding* is to replace all the subterm at

address a by $\bullet^a$, when *unfolding* does somewhat the opposite as it replaces $\bullet^a$ by the term it corresponds to (*i.e.* at the same address) above it.

Example 2.3. $\mathit{fold}(a)(\mathsf{nth}^c(5^b, \mathsf{cons}^a(0^b, \bullet^a))) = \mathsf{nth}^c(5^b, \bullet^a)$.
$\mathit{unfold}(\mathsf{cons}^a(0^b, \bullet^a))(\mathsf{nth}^c(5^b, \mathsf{cons}^a(0^b, \bullet^a))) = \mathsf{nth}^c(5^b, \mathsf{cons}^a(0^b, \mathsf{cons}^a(0^b, \bullet^a)))$

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 that denote term-graphs. From the point of view of addresses, terms are preterms where at each address one finds basically the same term, *i.e.* the same term modulo an appropriate treatment of $\bullet$ or back pointers.

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 $t \mathbin{@} \_$, 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 be distinguished: 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 $t_i \mathbin{@} \_$ 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 that 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, $\mathsf{times}^a(\bullet^c, \bullet^c)$ has a dangling back-pointer, while $\mathsf{cons}^a(0^b, \bullet^a)$ has none. The second step of the definition restricts the addressed terms to the dangling terms that do not have dangling back-pointers. The following defines dangling terms and the function $t \mathbin{@} \_$ from $\mathit{addr}(t)$ to dangling in-terms. Intuitively, $t \mathbin{@} a$ returns the in-term of t at address a.

Definition 2.3 (Dangling Addressed Terms).
We define the set $DT(\Sigma)$ of *dangling addressed terms* inductively and simultaneously define the function $t \mathbin{@} \_$:

Variables.

— Each variable X is in $DT(\Sigma)$

— $X \mathbin{@} \_$ is nowhere defined.

Back-pointers.

— $\bullet^a \in DT(\Sigma)$

— $\bullet^a \mathbin{@} a \equiv \bullet^a$

Expressions.

— $t \equiv F^a(t_1, \ldots, t_n) \in DT(\Sigma)$ provided that
  – $F \in \Sigma$ of arity n,
  – $t_1 \in DT(\Sigma), \ldots, t_n \in DT(\Sigma)$ satisfy: $b \in addr(t_i) \cap addr(t_j) \Rightarrow t_i @ b \equiv t_j @ b$, and
  – a is an address satisfying $a \in addr(t_i) \Rightarrow t_i @ a \equiv \bullet^a$
— $t @ a \equiv t$; and if $b \in addr(t_i)$, $b \neq a$, $t @ b \equiv unfold(t)(t_i @ b)$.

Example 2.4. Let $t = \mathsf{cons}^a(0^c, \mathsf{cons}^b(1^d, \bullet^a))$. Then $t@b = \mathsf{cons}^b(1^d, \mathsf{cons}^a(0^c, \mathsf{cons}^b(1^d, \bullet^a)))$. We say that $\mathsf{cons}^b(1^d, \bullet^a)$ is a dangling term.

Admissible addressed terms are 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 2.4 (Addressed Term).
A dangling addressed term t is *admissible* if $a \in addr(t) \Rightarrow t @ a \not\equiv \bullet^a$. An admissible dangling addressed term will be simply denoted an *addressed term.*

Example 2.5.

— The following are admissible terms:
  – $\mathsf{times}^a(\mathsf{times}^b(2^c, 2^c), \mathsf{times}^b(2^c, 2^c))$,
  – $\mathsf{nth}^c(5^b, \mathsf{cons}^a(0^b, \bullet^a))$,
  – $\mathsf{cons}^a(0^c, \mathsf{cons}^b(1^d, \bullet^a))$.

It is easy to see that an in-term of an admissible term is admissible.

2.2. *Addressed Term Rewriting*

The reduction of 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) 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; conceptually 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 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, 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. The reader may find it useful to look ahead to section 2.3 where we provide examples of rewriting with ATRSs: this will help motivate the following technical definitions.

Definition 2.5 (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:

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

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 $v \equiv \langle \alpha; \sigma \rangle(t) \equiv \langle \alpha; \sigma \rangle(u)$.

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 2.6 (Replacement).
Let t, u be addressed terms. The replacement generated by u in t, denoted by $repl(u)(t)$

is defined as follows:

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

The replacement of addressed term is the generalization of the usual notion of replacement for terms that takes account of addresses in two respects: terms at same address must be replaced simultaneously bullets must be unfolded appropriately to avoid dangling pointers.

An easy induction on the structure of dangling terms shows show

Proposition 2.1 (Replacement Admissibility).
If t and u are addressed terms, then $repl(u)(t)$ is an addressed term.

We now define the notions of redex and rewriting.

Definition 2.7 (Addressed Rewriting Rule).
An addressed rewriting rule over Σ is a pair of addressed terms (l, r) over Σ, written $l \rightsquigarrow r$, such that $loc(l) \equiv loc(r)$ and $var(r) \subseteq var(l)$. 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.

The square redex, given at the beginning of this section, is an example of addressed rewriting rule. This example is somewhat simple, since rules may in general contain addresses, but Figure 6 contains plenty of examples of rules with addresses.

The condition $var(r) \subseteq var(l)$ ensures that there is no creation of variables. The condition $loc(l) \equiv loc(r)$ says that l and r have the same top address, therefore l and r are not variables.

Remark 2.1 (The right-hand side restriction). In the definition of rewriting rule we forbid the right-hand side to be a variable. It may appear that this limits th expressive power of ATRS's. On the contrary this is an interesting feature of ATRS's which highlights our attention to implementation concerns. Suppose for example that we were to have a rule $F^a(x) \to x$, to be applied in an in-term $F^a(t^b)$ of a term. (Remember that variables have no addresses.) Then $F^a(t^b)$ would be rewritten to t^b. Do we intend that t^b is also at address a? If yes, this means we have to have a means to redirect all pointers to a toward b. Surely such acrobatics should not be part of a term-rewriting system – nor an implementation of rewriting! So we forbid such situations. We can still encode the rule capturing the equation $F(x) = x$, by using indirection nodes (see the paragraph on evaluation contexts in Section 3.2). In the case of the above rule $F^a(x) \to x$, the trick would be to introduce a unary operator $\lceil\_\rceil^a$ (the indirection) and replace the rule by $F^a(x) \to \lceil x \rceil^a$ which then fulfills the restriction. Therefore other rules have to be

introduced to remove the indirection, for instance a rule $G^b(\lceil x \rceil^a) \to G^b(x)$, which also fulfills the restriction.

Definition 2.8 (Redex).
A term t *is a redex* for a rule $l \rightsquigarrow r$, 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.

In the square example at the beginning of this section, both $\mathsf{square}^a(\mathsf{square}^b(2^c))$ and $\mathsf{square}^b(2^c)$ are redexes. 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. Besides redirecting pointers, ATRS create *new* nodes. *Fresh renaming* insures that these new node addresses are not already used.

Definition 2.9 (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 $l \rightsquigarrow r$. A renaming α_{fresh} is *fresh* for $l \rightsquigarrow r$ 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 2.2 (Substitution Admissibility).
Given an admissible term t that has a redex for the addressed rewriting rule $l \rightsquigarrow r$. Then

1 A fresh renaming α_{fresh} exists for $l \rightsquigarrow r$ 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 2.10 (Rewriting).
Let t be a term that we want to reduce at address a by rule $l \rightsquigarrow r$. 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 $l \rightsquigarrow r$ 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 $l \rightsquigarrow r$ at address a is $repl(u)(t)$. We write the reduction $t \rightsquigarrow s$, defining "$\rightsquigarrow$" as the relation of all such rewritings.

Theorem 2.1 (Closure under Rewriting).
Let R be an addressed term rewriting system and t be an addressed term. If $t \rightsquigarrow u$ in R then u is also an addressed term.

Proof. The proof essentially walks through the steps of Definition 2.10, 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 $l \rightsquigarrow r$. All of t, l, and r, are admissible. For the rewrite to be defined, we furthermore know that t has a redex

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

The term t' is admissible and by Proposition 2.2, we can find a renaming α_{fresh} that is fresh for $l \rightsquigarrow r$ with respect to t' and we know that $u \equiv \langle \alpha \cup \alpha_{\text{fresh}}; \sigma \rangle(r)$ is admissible. Proposition 2.1 finally ensures that $repl(u)(t)$ is admissible. □

Actually ATRS generalize TRS and we show in appendix A that acyclic ATRS which do not mutate terms, i.e., which do not replace a subterm by another subterm in place actually simulate TRS's.

2.3. *Rewriting examples*

Example 2.6. As an easy application of the definition of rewriting we can revisit our square example, and see that $\mathsf{times}^a(\mathsf{square}^b(2^c), \mathsf{square}^b(2^c))$ rewrites to $\mathsf{times}^a(\mathsf{times}^b(2^c, 2^c), \mathsf{times}^b(2^c, 2^c))$.

Example 2.7 (Cycles). Consider the term $t = H^c(F^b(G^a(\bullet^b)), G^a(F^b(\bullet^a)))$, and $l \to r = G^a(x) \to F^a(x)$. We want to reduce t at address a. We first compute the matching $\langle \alpha; \sigma \rangle = \langle \{a \mapsto a\}; \{x \mapsto F^b(G^a(\bullet^b))\} \rangle$. Since r does not contain fresh variables, then $\sigma_{\mathsf{fresh}} = \varnothing$. $s = \langle \alpha; \sigma \rangle(F^a(x)) = F^a(F^b(\bullet^a))$. The result of rewriting is:

$$repl(\{a \mapsto F^a(F^b(\bullet^a)), b \mapsto F^b(F^a(\bullet^b))\})(t) = H^c(F^b(F^a(\bullet^b)), F^a(F^b(\bullet^a)))$$

The corresponding transformation on graphs is shown in Figure 2.

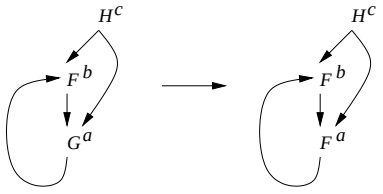

Figure 2. Graph Rewrite equivalent to Addressed Term Rewrite of Example 2.7.

Example 2.8. Consider the term $t = G^a(H^b(\bullet^a))$, and the rule

$$l \to r = H^a(G^b(x)) \to H^a(x).$$

We want to reduce t at address b. Note that there is a redex at address b even though it does not appear syntactically. Indeed, $t @ b = H^b(G^a(\bullet^b))$, and

$$\boldsymbol{match}(H^a(G^b(x)))(H^b(G^a(\bullet^b))) = \langle\{a \mapsto b, b \mapsto a\}; \{x \mapsto H^b(G^a(\bullet^b))\}\rangle.$$

We denote this matching by $\langle\alpha;\sigma\rangle$. Note that $\sigma_{\mathsf{fresh}} = \varnothing$ and $s = \langle\alpha;\sigma\rangle(H^a(x)) = H^b(\langle\alpha;\{x \mapsto \bullet^b\}\rangle(x)) = H^b(\bullet^b)$. The result of rewriting t is

$$\mathit{repl}(\{b \mapsto H^b(\bullet^b)\})(t) = G(H^b(\bullet^b))$$

Example 2.9 (Mutation). The following shows how side effects are performed, using the well-known example of setcar.

Consider the symbols nil of arity 0, and cons of arity 2, the constructors of lists. car, of arity 1 is the operator which returns the first element of a list, and setcar, of arity 2, the operator which affects a new value to the first element of the list, performing a side effect. We also have an infinite number of symbols denoted by $1, 2, \ldots, n, \ldots$, and the operation plus of arity 2. At last, we have the special symbol $\lceil\ \rceil$ of arity 1 denoting an indirection node. The rules for the definition of car and setcar are the following:

$$\mathsf{setcar}^a(\mathsf{cons}^b(y,z),x) \to \lceil\mathsf{cons}^b(x,z)\rceil^a \qquad \text{(Set-Car)}$$
$$\mathsf{setcar}^a(\lceil z\rceil^b,x) \to \mathsf{setcar}^a(z,x) \qquad \text{(Elim-Ind)}$$
$$\mathsf{car}^a(\mathsf{cons}^b(n^c,z)) \to \lceil n^c\rceil^a \qquad \text{(Car-}n\text{)}$$

We omit a definition of plus here. Consider the term

$$t = \mathsf{plus}^e(\mathsf{car}^c(\mathsf{setcar}^a(\mathsf{cons}^b(1^f,\mathsf{nil}^h),2^g)),\mathsf{car}^d(\mathsf{cons}^b(1^f,\mathsf{nil}^h)))$$

in which the list $\mathsf{cons}^b(1^f,\mathsf{nil}^h)$ appears twice, which means that it is shared. Starting by applying the rule for setcar, the reduction proceeds as follows:

$$\begin{aligned}
t &\to \mathsf{plus}^e(\mathsf{car}^c(\lceil\mathsf{cons}^b(2^g,\mathsf{nil}^h)\rceil^a),\mathsf{car}^d(\mathsf{cons}^b(2^g,\mathsf{nil}^h))) && \text{(Set-Car)}\\
&\to \mathsf{plus}^e(\mathsf{car}^c(\mathsf{cons}^b(2^g,\mathsf{nil}^h)),\mathsf{car}^d(\mathsf{cons}^b(2^g,\mathsf{nil}^h))) && \text{(Elim-Ind)}\\
&\to \mathsf{plus}^e(2^c,\mathsf{car}^d(\mathsf{cons}^b(2^g,\mathsf{nil}^h))) && \text{(Car)}\\
&\to \mathsf{plus}^e(2^c,2^d) && \text{(Car)}\\
&\to 4^e && \text{(Plus)}
\end{aligned}$$

On the other hand, starting by applying the rule for car, the computation proceeds as follows:

$$\begin{aligned}
t &\to \mathsf{plus}^e(\mathsf{car}^c(\mathsf{setcar}^a(\mathsf{cons}^b(1^f,\mathsf{nil}^h),2^g)),1^d) && \text{(Car)}\\
&\to \mathsf{plus}^e(\mathsf{car}^c(\lceil\mathsf{cons}^b(2^g,\mathsf{nil}^h)\rceil^a),1^d) && \text{(Set-Car)}\\
&\to \mathsf{plus}^e(\mathsf{car}^c(\mathsf{cons}^b(2^g,\mathsf{nil}^h)),1^d) && \text{(Elim-Ind)}\\
&\to \mathsf{plus}^e(2^c,1^d) && \text{(Car)}\\
&\to 3^e && \text{(Plus)}
\end{aligned}$$

The above example shows that ATRS may produce side effects. It is essential to be

able to determine easily whether a system has side effects or not. For instance, a side effect appears clearly in the rule (Set-Car), since the term $\mathsf{cons}^b(y,z)$ becomes $\mathsf{cons}^b(x,z)$ whereas the rewriting takes place at address a in a term of shape $\mathsf{setcar}^a(\Box,\Box)$. Rules which do not host such transformations have no side effect.

3. Modeling an Object-based Formalism via ATRS: $\lambda\mathcal{O}bj^a$

The purpose of this section is to describe the top level rules of the $\lambda\mathcal{O}bj^a$, a formalism strongly based on acyclic ATRS introduced in the previous section.

The formalism is described by a set of rules arranged in *modules*. The three modules are called respectively L, C, and F, and they will be introduced in Subsection 3.3.

L is the *functional* module, and is essentially the calculus $\lambda\sigma^a_w$ 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.

The set of rules $\mathsf{L}+\mathsf{C}+\mathsf{F}$ is the instance of $\lambda\mathcal{O}bj^a$ for functional object calculi.

We do this in two steps:

1 first we present the functional calculus $\lambda\mathcal{O}bj^\sigma$, intermediate between the calculus $\lambda\mathcal{O}bj$ of Fisher, Honsell and Mitchell (Fisher et al. 1994) and our $\lambda\mathcal{O}bj^a$,
2 then we scale up over the full $\lambda\mathcal{O}bj^a$.

By a slight abuse of terminology we may say that $\lambda\mathcal{O}bj^a$ is a *conservative extension* of $\lambda\mathcal{O}bj^\sigma$, in the sense that for computations in $\lambda\mathcal{O}bj^a$ and computations in $\lambda\mathcal{O}bj^\sigma$ return the same normal form (Theorems 5.1 and 5.2). 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.

3.1. *Syntax and operational semantics of* $\lambda\mathcal{O}bj^\sigma$

$\lambda\mathcal{O}bj^\sigma$ is a language without addresses (see the syntax in Figure 3). The rules of $\lambda\mathcal{O}bj^\sigma$ are presented in Figure 4. As noted earlier, terms of this calculus are terms of $\lambda\mathcal{O}bj^a$ without addresses and without indirection nodes. The rules of $\lambda\mathcal{O}bj^\sigma$ are properly contained in those of modules $\mathsf{L}+\mathsf{C}+\mathsf{F}$ of $\lambda\mathcal{O}bj^a$. One can also say that $\lambda\mathcal{O}bj^\sigma$ is the *lambda calculus of objects* proposed by Fisher, Honsell and Mitchell in (Fisher et al. 1994) enriched with explicit substitutions. The first category of expressions is the *code* of programs. 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 or have to

M, N	$::=$	$\lambda x.M \mid MN \mid x \mid c \mid$	
		$\langle\,\rangle \mid \langle M \leftarrow \mathtt{m} = N\rangle \mid M \Leftarrow \mathtt{m}$	(Code)
U, V	$::=$	$M[s] \mid UV \mid$	
		$U \Leftarrow \mathtt{m} \mid \langle U \leftarrow \mathtt{m} = V\rangle \mid Sel\big(O, \mathtt{m}, U\big)$	(Eval. Contexts)
O	$::=$	$\langle\,\rangle \mid \langle O \leftarrow \mathtt{m} = V\rangle$	(Object Structures)
s	$::=$	$U/x\ ;\ s \ \mid\ \mathsf{id}$	(Substitutions)

Figure 3. The Syntax of $\lambda\mathcal{O}bj^{\sigma}$

Basics for Substitutions				
$(MN)[s]$	$\rightsquigarrow$	$(M[s]\,N[s])$		(App)
$\big((\lambda x.M)[s]\,U\big)$	$\rightsquigarrow M$	$[U/x\ ;\ s]$		(Bw)
$x[U/y\ ;\ s]$	$\rightsquigarrow$	$x[s]$	$x \not\equiv y$	(RVar)
$x[U/x\ ;\ s]$	$\rightsquigarrow$	U		(FVar)
$\langle M \leftarrow \mathtt{m} = N\rangle[s]$	$\rightsquigarrow$	$\langle M[s] \leftarrow \mathtt{m} = N[s]\rangle$		(P)
Method Invocation				
$(M \Leftarrow \mathtt{m})[s]$	$\rightsquigarrow$	$(M[s] \Leftarrow \mathtt{m})$		(SP)
$(O \Leftarrow \mathtt{m})$	$\rightsquigarrow$	$Sel(O, \mathtt{m}, O)$		(SA)
$Sel(\langle O \leftarrow \mathtt{m} = U\rangle, \mathtt{m}, V)$	$\rightsquigarrow$	$(U\,V)$		(SU)
$Sel(\langle O \leftarrow \mathtt{n} = U\rangle, \mathtt{m}, V)$	$\rightsquigarrow$	$Sel(O, \mathtt{m}, V)$	$\mathtt{m} \not\equiv \mathtt{n}$	(NE)

Figure 4. The Rules of $\lambda\mathcal{O}bj^{\sigma}$

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*). The last category defines *substitutions* also called *environments*, *i.e.*, lists of terms bound to variables, that 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:

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

The code category is essentially the Lambda Calculus of Objects of (Fisher et al. 1994), and its operational semantics is the one presented in (Gianantonio, et al. 1998) and briefly recalled below:

(Gianantonio et al. 1998) Operational Semantics.

$$(\lambda x.M)\,N \rightsquigarrow M\{N/x\} \qquad \text{(Beta)}$$

$$M \Leftarrow \mathtt{m} \rightsquigarrow Sel(M, \mathtt{m}, M) \qquad \text{(Select)}$$

$$Sel(\langle M \leftarrow \mathtt{m} = N\rangle, \mathtt{m}, P) \rightsquigarrow NP \qquad \text{(Success)}$$

$$Sel(\langle M \leftarrow \mathtt{n} = N\rangle, \mathtt{m}, P) \rightsquigarrow Sel(M, \mathtt{m}, P) \quad \mathtt{m} \neq \mathtt{n} \qquad \text{(Next)}$$

The main difference between $\lambda\mathcal{O}bj^\sigma$ and $\lambda\mathcal{O}bj$ (Fisher et al. 1994) lies in the use of a single operator $\leftarrow$ for building an object from an existing prototype. If the object M contains $\mathtt{m}$, then $\leftarrow$ denotes an object override, otherwise $\leftarrow$ denotes an object extension. The principal operation on objects is method invocation, whose reduction is defined by the (Select) rule. Sending a message $\mathtt{m}$ to an object M containing a method $\mathtt{m}$ reduces to $Sel(M, \mathtt{m}, M)$. The arguments of Sel in $Sel(M, \mathtt{m}, P)$ have the following intuitive meaning (in reverse order)

— P is the receiver (or recipient) of the message;
— $\mathtt{m}$ is the message we want to send to the receiver of the message;
— M is (or reduces to) a proper sub-object of the receiver of the message.

We note that the (Beta) rule above is given using the *meta substitution* (denoted by $\{N/x\}$), as opposed to the *explicit substitution* used in $\lambda\mathcal{O}bj^a$. Finally, the semantics presented in (Gianantonio et al. 1998) is not restricted to weak reduction ("no reduction under lambdas") as is that in $\lambda\mathcal{O}bj^a$.

By looking at the last two rewrite rules, one may note that the Sel function "scans" the recipient of the message until it finds the definition of the method we want to use. When it finds the body of the method, it applies this body to the recipient of the message. Notice that Sel somewhat destroys the object. In order to keep track of the objects on which the method when found has to be applied, a third parameter is added to Sel.

3.2. *Syntax of* $\lambda\mathcal{O}bj^a$

The syntax of $\lambda\mathcal{O}bj^a$ is summarized in Figure 5. As for $\lambda\mathcal{O}bj^\sigma$ terms that define the code have no addresses (the same for substitutions). In contrast, terms in evaluation contexts have explicit addresses.

"Fresh" addresses are often provided to evaluation contexts, when distributing the environment (a fresh address is an address unused in a global term). Intuitively, the address of an evaluation context is the address where the result of the computation will be

M, N	$::=$	$\lambda x.M \mid MN \mid x \mid c \mid \langle\,\rangle \mid \langle M \leftarrow \mathtt{m} = N\rangle \mid M \Leftarrow \mathtt{m}$	(Code)
U, V, W	$::=$	$M[s]^a \mid (UV)^a \mid \lceil U \rceil^a \mid$	(Eval. Contexts)
		$\langle U \leftarrow \mathtt{m} = V\rangle^a \mid (U \Leftarrow \mathtt{m})^a \mid Sel^a\big(O, \mathtt{m}, V\big) \mid$	
O	$::=$	$\langle\,\rangle \mid \langle O \leftarrow m = V\rangle$	Object Structures
s	$::=$	$U/x\ ;\ s \mid \mathsf{id}$	(Substitutions)

Figure 5. The Syntax of $\lambda\mathcal{O}bj^a$

The Module L			
$(MN)[s]^a$	$\rightsquigarrow$	$(M[s]^b\, N[s]^c)^a$	(App)
$\big((\lambda x.M)[s]^b\, U\big)^a$	$\rightsquigarrow$	$M[U/x\ ;\ s]^a$	(Bw)
$x[U/x\ ;\ s]^a$	$\rightsquigarrow$	$\lceil U \rceil^a$	(FVar)
$x[U/y\ ;\ s]^a$	$\rightsquigarrow$	$x[s]^a \quad x \not\equiv y$	(RVar)
$(\lceil U \rceil^b\, V)^a$	$\rightsquigarrow$	$(U\, V)^a$	(AppRed)
The Module C			
$(M \Leftarrow \mathtt{m})[s]^a$	$\rightsquigarrow$	$(M[s]^b \Leftarrow \mathtt{m})^a$	(SP)
$(U \Leftarrow \mathtt{m})^a$	$\rightsquigarrow$	$Sel^a(U, \mathtt{m}, U)$	(SA)
$(\lceil U \rceil^b \Leftarrow \mathtt{m})^a$	$\rightsquigarrow$	$(U \Leftarrow \mathtt{m})^a$	(SRed)
$Sel^a(\langle U \leftarrow \mathtt{m} = V\rangle^b, \mathtt{m}, W)$	$\rightsquigarrow$	$(V\, W)^a$	(SU)
$Sel^a(\langle U \leftarrow \mathtt{n} = V\rangle^b, \mathtt{m}, W)$	$\rightsquigarrow$	$Sel^a(U, \mathtt{m}, W) \quad \mathtt{m} \not\equiv \mathtt{n}$	(NE)
$Sel^a(\lceil U \rceil^b, \mathtt{m}, V)$	$\rightsquigarrow$	$Sel^a(U, \mathtt{m}, V)$	(SelRed)
The Module F			
$\langle M \leftarrow \mathtt{m} = N\rangle[s]^a$	$\rightsquigarrow$	$\langle M[s]^b \leftarrow \mathtt{m} = N[s]^c\rangle^a$	(FP)
$\langle \lceil U \rceil^b \leftarrow \mathtt{m} = V\rangle^a$	$\rightsquigarrow$	$\langle U \leftarrow \mathtt{m} = V\rangle^a$	(FRed)

Figure 6. The Modules L and C and F

stored. A *closure* $M[s]^a$ is analogous to a closure in a λ-calculus, but it is given an address, and the terms in s are also addressed terms. The evaluation context $\lceil O \rceil^a$ represents an object whose *internal* object-structure is O and whose object-identity is $\lceil\ \rceil^a$. In other words, the address a plays the role of an *entry point* of the object-structure O. $(O \hookleftarrow m)^a$ is the evaluation context associated to a method-lookup *i.e.*, the scanning of the object-

structure to find the method m. $\hookleftarrow$ is an auxiliary operator, reminiscent to the selection operator *Sel* of $\lambda\mathcal{O}bj^{+}$, invoked when one sends a message to an object. We recall that the term $\bullet^a$ is a *back pointer*. From object point of view they allow to create "loops in the store". Only $\bullet^a$ can occur inside a term having the same address a, therefore generalizing our informal notion of admissible term and simultaneous rewriting.

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 a functional update operator $\langle \_ \leftarrow \_ \rangle$. In a functional setting, this operator can be understood as extension as well as override operator, since an override is handled as a particular case of extension.
— Method invocation ($\_ \Leftarrow \_$).

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 $(UV)^a$, $(U \Leftarrow \mathtt{m})^a$, and $\langle U \leftarrow \mathtt{m} = V\rangle^a$, are the evaluation contexts associated with the corresponding code constructors. Direct sub-terms of these evaluation contexts are themselves evaluation contexts instead of code.
— The term $Sel^a(U, \mathtt{m}, V)$ is the evaluation context associated to a method-lookup, *i.e.*, the scanning of the context U to find the method $\mathtt{m}$, and apply it to the context V. 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). This gives account of phenomena well-known by implementers. Rules like (AppRed), (SRed) and (FRed) remove those indirections.

Remark 3.1 (ATRS-based preterms of $\lambda\mathcal{O}bj^a$).
The concrete syntax of $\lambda\mathcal{O}bj^a$ of Figure 5 can be viewed as syntactic sugar over the definition of ATRS preterm in two ways:

1 Symbols in the signature may also be infix (like *e.g.*, ($\_ \Leftarrow \_$)), bracketing (like *e.g.*, $\lceil \_ \rceil$), 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. 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.).
2 We shall use $\lambda\mathcal{O}bj^a$ sort-specific variable names.

It is clear that not all preterms denote term-graphs, since this may lead to inconsistency in the sharing. For instance, the preterm $\big((\langle\rangle[\,]^b \Leftarrow \mathtt{m})^a \langle\rangle[\,]^a\big)^c$ is inconsistent, because location a is both labeled by $\langle\rangle[\,]$ and $\big(\_ \Leftarrow \_\big)$. The preterm $\big((\lceil\langle\rangle[\,]^a\rceil^b \Leftarrow \mathtt{m})^c \lceil\langle\rangle[\,]^e\rceil^b\big)^d$ 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 $\big((\lceil\langle\rangle[\,]^a\rceil^b \Leftarrow \mathtt{m})^c \lceil\langle\rangle[\,]^a\rceil^b\big)^d$ denotes a *legal* term-graph with four nodes, respectively, at addresses a, b, c, and d[†]. Moreover, the nodes at addresses a and b, corresponding respectively to $\langle\rangle[\,]$ and $\langle\rangle[\,]^a$, are shared in the corresponding graph since they have several occurrences in the term. These are the distinction captured by the well-formedness constraints defined in section 2.1.

3.3. *Operational Semantics of $\lambda\mathcal{O}bj^a$*

The rules of $\lambda\mathcal{O}bj^a$ as a computational-engine are defined in Figure 6. First of all observe that *there are no rules for code expressions*: in fact every code expression is evaluated starting from an empty substitution (you may think this empty substitution as the empty return continuation) and as such code belongs directly to the evaluation contexts category. In what follows we give a brief explanation of the operational semantic:

Remark 3.2 (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 2), which ensures that clashes of addresses cannot occur.

3.4. *The intuition behind the rules*

The set of thirteen rules is divided in three modules. Notice the specificity of ATRS's, namely that in each rule the left-hand side lies at the same address as this of the right-hand side. To avoid losing this property and prevent bad effects, an indirection $\lceil\,\rceil^a$ is created when necessary. In $\mathsf{L} + \mathsf{C} + \mathsf{F}$ this is the case once, namely for (FVar). In some other rules, namely (AppRed), (SRed), (SelRed), and (FRed) the indirection is removed and the pointer is redirected. Notice that this can only be done inside a term, since a pointer has to be redirected and one has to know where this pointer comes from. In the name of these rules the suffix "Red" stands for redirection.

$\lambda\mathcal{O}bj^a$ is based on extending a calculus of explicit substitutions and L is precisely that calculus which was introduced first by Benaissa et al. (Benaissa et al. 1996), itself derived from Bloo and Rose's $\lambda\mathtt{x}$ (Bloo & Rose 1995) and Curien's calculus of closures $\lambda\rho$ (Curien 1991), by adding addresses. Like Rose and Bloo and unlike Curien, who works on de Bruijn's indices, $\mathsf{L} + \mathsf{C} + \mathsf{F}$ works on explicit names. As those calculi L has a rule

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

(Bw) which looks for a β-redex and produces an extension of the environment. Rule (App) distributes the closure in an application. Note that (App) creates new nodes in the tree structure of the term, which is to say that it creates new addresses. But note that there is no duplication of s: this has its own set of addresses, replicated in the tree structure, but the repetition of these addresses merely indicates explicitly the sharing in memory. This comment is important since *this is the place where sharing actually takes place in object languages*. (Rvar) is when the code is a variable x. If that variable x mismatches the variable y of the first pair in the environment, the access is performed on the tail of the list. (FVar) returns the evaluation context associated with the variable x when the variable in the first pair in the environment is precisely x. The term on the left side lies at address a, but this is not the case for the evaluation context, so a redirection toward this evaluation context forces this evaluation context to be accessed through the address a and therefore the right-hand side lies also at address a. (AppRed) removes an indirection in a application. It does that only on the left part of the application, in order to avoid hiding a λ-abstraction and to enable a potential later application of (Bw). Since this indirection removal takes places inside a term, it can be easily implemented by a redirection.

Module C speaks about objects and method invocations. Like (App), (SP) is a rule that distributes an environment through a structure. Notice since $\mathtt{m}$ is just a label, the environment s has no effect on it. The left-hand side of (SA) is a method invocation. The intuitions behind *Sel* were given earlier. (SRed) deals with a redirection. Since, on the left-hand side, the method $\mathtt{m}$ is applied on an indirection, this indirection is removed in the left-hand side. This removal leads in an actual implementation to a pointer redirection. (SelRed) also deals with indirection. (SRed) can be considered as redundant with it, but this is left to the implementer's. (SU) and (NE) are for method selection. In (SU) the method is found and its body is applied to the actual object. In (NE) the method is not found and searched further.

Module F is about method extension. Notice that the semantics does not speak about method overriding, since when a method is added to an object which has already a method with the same label, only the last added method will be selected later one. Therefore this addition has the effect of a method overriding. Like (App) and (SP), (FP) distributes an environment through a structure. (FC) says that when one extends an evaluated object by a new method the result is an evaluated object. (FRed) is again an indirection removal leading to a redirection.

Remark 3.3.
Notice that *normal forms* are as follows. $((\lambda x.M)[s])^a$ and $(\langle M[s_1]^b \leftarrow \mathtt{m} = N[s_2]^c\rangle)^a$ correspond to values computed by programs. $x[\ ]^a$ corresponds to an *illegal memory access* and $Sel^a(\langle\ \rangle[s]^b, \mathtt{m}, U)$ corresponds to *method not found*.

4. ATRS at Work: an Example in $\lambda\mathcal{O}bj^a$

Here we propose examples to help understanding the formalism. We first give an example showing a point object with a set method.

Example 4.1 (A Point Object). Let

$$\mathtt{point} \triangleq \langle \underbrace{\langle\, \langle\,\rangle \leftarrow \mathtt{n} = \lambda \mathtt{s}.0\rangle}_{P} \leftarrow \mathtt{set1} = \underbrace{\lambda \mathtt{self}.\langle \mathtt{self} \leftarrow \mathtt{n} = \lambda \mathtt{s}.1\rangle}_{N} \,\rangle.$$

The reduction of $M \triangleq (\mathtt{point} \Leftarrow \mathtt{set1})$ in $\lambda\mathcal{O}bj^a$ starting from an empty substitution is as follows:

$$\begin{aligned}
M[\,]^a &\rightsquigarrow^* (\langle P[\,]^d \leftarrow \mathtt{set1} = N[\,]^c\rangle^b \Leftarrow \mathtt{set1})^a && (1)\\
&\rightsquigarrow Sel^a(P[\,]^d \leftarrow \mathtt{set1} = N[\,]^c,\quad \mathtt{set1},\quad \mathtt{point}^b) && (2)\\
&\rightsquigarrow ((\lambda \mathtt{self}.\langle \mathtt{self} \leftarrow \mathtt{n} = \lambda \mathtt{s}.1\rangle)[\,]^c\quad \mathtt{point}^b)^a && (3)\\
&\rightsquigarrow \langle \mathtt{self} \leftarrow \mathtt{n} = \lambda \mathtt{s}.1\rangle[\mathtt{point}^b/\mathtt{self}]^a && (4)\\
&\rightsquigarrow \langle \mathtt{self}[\mathtt{point}^b/\mathtt{self}]^h \leftarrow \mathtt{n} = (\lambda \mathtt{s}.1)[\mathtt{point}^b/\mathtt{self}]^g\rangle^a && (5)\\
&\rightsquigarrow \langle \lceil \mathtt{point}^b \rceil^h \leftarrow \mathtt{n} = (\lambda \mathtt{s}.1)[\mathtt{point}^b/\mathtt{self}]^g\rangle^a && (6)\\
&\rightsquigarrow^2 \langle \mathtt{point}^b \leftarrow \mathtt{n} = (\lambda \mathtt{s}.1)[\mathtt{point}^b/\mathtt{self}]^g\rangle^a && (7)
\end{aligned}$$

In (1), two steps are performed to distribute the environment inside the extension, using rules (SP), and (FP). In (2), and (3), two steps (SA) (SU) perform the look up of method `set1`. In (4) we apply (Bw). In (5), the environment is distributed inside the functional extension by (FP). In (6), (FVar) replaces `self` by the object it refers to, setting an indirection from h to b. In (7) the indirection is eliminated by (FRed). There is no redex in the last term of the reduction, *i.e.* it is in normal form.

Sharing of structures appears in the above example, since *e.g.* $\mathtt{point}^b$ turns out to have several occurrences in some of the terms of the derivation.

4.1. *Object Representations in Figures 7*

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 $\lambda\mathcal{O}bj^a$ itself; $\lambda\mathcal{O}bj^a$ 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. 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.

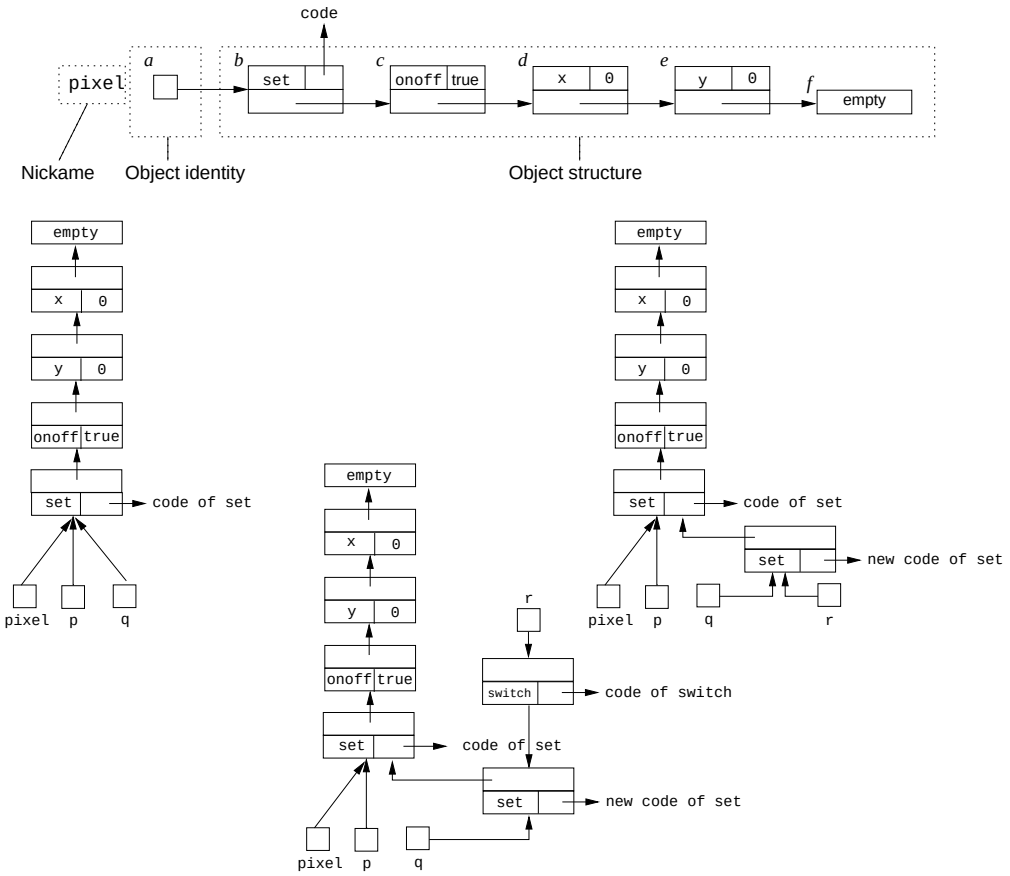

Figure 7. An Object Pixel (top), and the evolution of memory in Section 4.2

Consider the following definition of a "pixel" prototype with three fields and one method.

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

We make 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.

After instantiation, the object `pixel` is located at an address, say a, and its object structure starts at address b, see Figure 7 (top). In what follows, we will derive three other objects from `pixel` and discuss the variations of how this may be done below.

4.2. *Cloning: an example*

The first two derived objects, nick-named `p` and `q`, are *clones* of `pixel` (Here `let x = A in B` is syntactic sugar for the functional application `(λx.B)A`.)

```
let p = pixel in let q = p in q        //  clone
```

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 7 (left). We might stress here that `p` and `q` should not be thought of as aliases of `pixel` as the Figure might suggest; this point will be clearer after the discussion of object overriding below. Then, we show what we want to model in our formalism 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`.

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

which obviously reduces to: `(pixel.set:=(u,v,w){..}).switch:=(){..}`. Figure 7 (middle) shows the state of the memory after the execution of the first two `let` binding are executed (let us call these program points 1 and 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 3.

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, executing `let q = p.set` only changes the object-structure of `q` without changing its object-identity. This is the sense in which our `clone` operator really does implement 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. The last diagram in Figure 7 shows the final state of memory after the execution of the binding of `r`. 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 Object Calculus in Chapter 6-7 of (Abadi & Cardelli 1996), or observe that Java uses *static field lookup* to make the position of each field constant in the object.

4.3. *Implementing*

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 the object `p` and `q`, presented in Figure 7, will be represented by the following addressed terms.

$$\mathtt{p} \triangleq \lceil\langle\langle\langle\langle\,\rangle^{f} \leftarrow \mathtt{y} = 0\rangle^{e} \leftarrow \mathtt{x} = 0\rangle^{d} \leftarrow \mathtt{onoff} = \mathtt{true}\rangle^{c} \leftarrow \mathtt{set} = \ldots\rangle^{b}\rceil^{a}$$

$$\mathtt{q} \triangleq \Vert\lceil\langle\langle\langle\langle\langle\,\rangle^{f} \leftarrow \mathtt{y} = 0\rangle^{e} \leftarrow \mathtt{x} = 0\rangle^{d} \leftarrow \mathtt{onoff} = \mathtt{true}\rangle^{c} \leftarrow \mathtt{set} = \ldots\rceil^{b} \leftarrow \mathtt{set} = \ldots\rangle^{g}\Vert^{h}$$

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

5. Relation between $\lambda\mathcal{O}bj^{\sigma}$ and $\lambda\mathcal{O}bj^{a}$

In this section we just list some fundamental results about the relationship between $\lambda\mathcal{O}bj^{\sigma}$ and $\lambda\mathcal{O}bj^{a}$. As a first step we note that the results presented in Section A are applicable to $\lambda\mathcal{O}bj^{\sigma}$.

Lemma 5.1 (Mapping $\lambda\mathcal{O}bj^{a}$ to $\lambda\mathcal{O}bj^{\sigma}$).
Let ϕ be the mapping from $\lambda\mathcal{O}bj^{a}$-terms that erases addresses, indirection nodes ($\lceil \_ \rceil^{a}$), and leaves all the other symbols unchanged. Each term $\phi(U)$ is a term of $\lambda\mathcal{O}bj^{\sigma}$.

Proof. $\lambda\mathcal{O}bj^{a}$-terms are acyclic and the calculus is mutation-free. □

Then we show a simulation result.

Theorem 5.1 ($\lambda\mathcal{O}bj^{\sigma}$ Simulates $\lambda\mathcal{O}bj^{a}$).
Let U be an acyclic $\lambda\mathcal{O}bj^{a}$-term. If $U \rightsquigarrow V$ in $\mathsf{L} + \mathsf{C} + \mathsf{F}$, then either $\phi(U) \equiv \phi(V)$ or $\phi(U) \rightsquigarrow \phi(V)$ in $\lambda\mathcal{O}bj^{\sigma}$.

Proof. The proof relies on Theorem A.1. Just notice that each rule $l \rightsquigarrow r$ of modules $\mathsf{L} + \mathsf{C} + \mathrm{F}$ is acyclic and mutation-free, and that either it maps to a rule $\phi(l) \rightsquigarrow \phi(r)$ which belongs to $\lambda\mathcal{O}bj^{\sigma}$, or it is such that $\phi(l) \equiv \phi(r)$. □

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 5.2 (Completeness of $\lambda\mathcal{O}bj^{a}$ w.r.t. $\lambda\mathcal{O}bj^{\sigma}$).
If $M \rightsquigarrow^{*} N$ 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 $M[\,]^{a} \rightsquigarrow^{*} U$ 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), (SRed), or (FRed) (hence, indirection nodes cannot permanently *hide* some redexes).

□

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 5.3 (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. By Theorem 5.1 it suffices to show that the set of $\lambda\mathcal{O}bj^a$ rules $l \rightsquigarrow r$ for which $\phi(l) \equiv \phi(r)$ comprises a terminating system. This is the set of rules {(AppRed), (SRed), (SelRed), (FRed)}. But termination of this set is clear since each of these rules decreases the size of a term. □

It follows from the theorems of this section that modulo making addresses explicit, terms compute to the same normal forms in $\lambda\mathcal{O}bj^a$ and $\lambda\mathcal{O}bj^\sigma$.

6. A Type System for $\lambda\mathcal{O}bj^a$

In this section, we present a simple type system for $\lambda\mathcal{O}bj^a$, inspired by the systems of Fisher and Mitchell in (Fisher & Mitchell 1995) and Bono and Bugliesi in (Bono & Bugliesi 1999). This type system feature subtyping, which allow method bodies to operate uniformly over all objects having some minimum set of required methods.

6.1. *Types*

Types are built from type constants, type variables, a function space constructor, and two type abstractions called respectively `obj` and `pro`. Moreover, one introduces row of types designed for typing objects with methods.

The type expressions are described as follows. For simplicity we work with a single base type constant ι.

$$\sigma, \tau \quad ::= \quad \iota \mid t \mid \sigma \to \tau \mid \mathtt{pro}\, t.R \mid \mathtt{obj}\, t.R$$

$$R, R' \quad ::= \quad \langle\langle\rangle\rangle \mid \langle\langle R, \mathtt{m}{:}\sigma\rangle\rangle$$

Here σ and τ are meta-variables ranging over types. R, and R' are meta-variables ranging over rows.

Here are intuitions behind these concepts.

— ι is a constant, base, type.
— A *row* R is an *unordered* set of pairs *(method label, method type).*
— t are type-variables. Type variables play a role in type abstractions (`obj` or `pro`) which behave as a kind of fixed point. The type variable t when it occurs in an abstraction

$\texttt{obj}\,t.R(t)$ (or in an abstraction $\texttt{pro}\,t.R(t)$) is equivalent to `this` in a language like JAVA.

— $\texttt{pro}\,t.R$ is the type of prototypical objects that are modifiable via object override and object extension. On those objects, subtyping is forbidden, since subtyping is unsound when one allows objects to be extended (Abadi & Cardelli 1996, Liquori 1997).

— $\texttt{obj}\,t.R$ is the type of objects *of fixed size and behavior*; objects having this type can only receive messages. They can be, possibly, subsumed via subtyping; that is all.

6.2. *Contexts and Judgments*

Contexts are as follows

$$\Gamma ::= \varepsilon \mid \Gamma, x{:}\sigma \mid \Gamma, t <\!\!\# \sigma \mid \Gamma, a{:}\sigma$$

(treated as sets). Judgments are as follows

$$(1) \qquad \Gamma \vdash ok, \qquad \Gamma \vdash \sigma : ok, \quad \Gamma \vdash \sigma <\!\!\# \tau, \quad \Gamma \vdash \sigma <: \tau$$

$$(2) \qquad \Gamma \vdash M : \sigma, \quad \Gamma \vdash U : \sigma \qquad \Gamma \vdash a : \sigma$$

— $\Gamma \vdash ok$ means that Γ is well-formed;

— $\Gamma \vdash \sigma : ok$ means that in the context Γ, σ is well-formed;

— $\Gamma \vdash \sigma <\!\!\# \tau$ is the matching relation introduced by Bruce et al in (Bruce, et al. 1997) and further studied by Abadi and Cardelli in (Abadi & Cardelli 1995, Abadi & Cardelli 1996). It means that the type σ (a `pro`-type) has *more methods* than the type τ (another `pro`-type). Roughly speaking $\texttt{pro}\,t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\sigma}, \overline{\mathtt{n}}{:}\overline{\tau} \rangle\!\rangle <\!\!\# \texttt{pro}\,t.\langle\!\langle \overline{\mathtt{m}}{:}\overline{\sigma} \rangle\!\rangle$. Notice that in $\sigma <\!\!\# \tau$, σ cannot be an `obj`-type;

— $\Gamma \vdash \sigma <: \tau$ is subtyping. In conjunction with subsumption it allows using an object with a larger interface (*i.e.* with more methods) in any context expecting another object with a smaller interface (*i.e.* with less methods). This leads to an implicit form of polymorphism. In other words, σ can be the type of an update of an object of type τ. This judgment formalizes the ability to "inherit" method types;

— $\Gamma \vdash U : \sigma$ means that an evaluation context U has type σ in the context Γ;

— $\Gamma \vdash M : \sigma$ means that expression M has type σ in the context Γ;

— $\Gamma \vdash a : \sigma$ means that address a has type σ in the context Γ.

In defining the type rules it will be convenient to be able to refer to the set of method labels in a row.

Notation. If R is the row $\{\mathtt{m}_1{:}\sigma_1, \ldots, \mathtt{m}_k{:}\sigma_k\}$ then $\mathcal{M}(R)$ is the set of labels $\{\mathtt{m}_1, \ldots, \mathtt{m}_k\}$.

6.2.1. *Bureaucracy, Matching and Subtyping* Rules in Figure 8 assert the well-formedness of contexts and types, and serve to define the subtype and matching relations on types. A few notes are in order:

— In rule (Type−Pro) the assumption $t <\!\!\# \texttt{pro}\,t.\langle\!\langle\rangle\!\rangle$ makes t a `pro`-type a priori. (Type−Obj) says that if type $\texttt{pro}\,t.R$ is well-formed, its brother $\texttt{obj}\,t.R$ is also well-formed. The structure of `obj`-type really derives from this of `pro`-type.

Well-formed contexts

$$\frac{}{\varepsilon \vdash ok}\ (\text{Cont}-\varepsilon) \qquad \frac{\Gamma \vdash \sigma : ok \quad x \notin dom(\Gamma)}{\Gamma, x{:}\sigma \vdash ok}\ (\text{Cont}-\text{x})$$

$$\frac{\Gamma \vdash \sigma : ok \quad a \notin dom(\Gamma)}{\Gamma, a{:}\sigma \vdash ok}\ (\text{Cont}-\text{a}) \qquad \frac{\Gamma \vdash \sigma : ok \quad t \notin dom(\Gamma)}{\Gamma, t \lessdot\!\!\# \sigma \vdash ok}\ (\text{Cont}-\text{t})$$

Well-formed types

$$\frac{\Gamma \vdash ok}{\Gamma \vdash \iota : ok}\ (\text{Type}-\text{Const}) \qquad \frac{\Gamma, t \lessdot\!\!\# \sigma, \Delta \vdash ok}{\Gamma, t \lessdot\!\!\# \sigma, \Delta \vdash t : ok}\ (\text{Type}-\text{Var})$$

$$\frac{\Gamma \vdash \sigma : ok \quad \Gamma \vdash \tau : ok}{\Gamma \vdash \sigma \to \tau : ok}\ (\text{Type}-\text{Arrow}) \qquad \frac{\Gamma \vdash ok}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle\rangle\!\rangle : ok}\ (\text{Type}-\text{Pro}-\text{Empty})$$

$$\frac{\Gamma \vdash \mathtt{pro}\, t.R : ok \quad \mathtt{m} \notin \mathcal{M}(R) \quad \Gamma, t \lessdot\!\!\# \mathtt{pro}\, t.\langle\!\langle\rangle\!\rangle \vdash \sigma : ok}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle : ok}\ (\text{Type}-\text{Pro}) \qquad \frac{\Gamma \vdash \mathtt{pro}\, t.R : ok}{\Gamma \vdash \mathtt{obj}\, t.R : ok}\ (\text{Type}-\text{Obj})$$

Matching rules

$$\frac{\Gamma, t \lessdot\!\!\# \sigma, \Delta \vdash ok}{\Gamma, t \lessdot\!\!\# \sigma, \Delta \vdash t \lessdot\!\!\# \sigma}\ (\text{Match}-\text{Var}) \qquad \frac{\Gamma \vdash \sigma \lessdot\!\!\# \tau \quad \Gamma \vdash \tau \lessdot\!\!\# \rho}{\Gamma \vdash \sigma \lessdot\!\!\# \rho}\ (\text{Match}-\text{Trans})$$

$$\frac{\Gamma \vdash \sigma : ok \quad \sigma \text{ is a } \mathtt{pro}\, t. \text{ or a variable}}{\Gamma \vdash \sigma \lessdot\!\!\# \sigma}\ (\text{Match}-\text{Refl}) \qquad \frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle : ok}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle \lessdot\!\!\# \mathtt{pro}\, t.R}\ (\text{Match}-\text{Probject})$$

(Width) Subtyping rules

$$\frac{\Gamma \vdash \sigma <: \tau \quad \Gamma \vdash \tau <: \rho}{\Gamma \vdash \sigma <: \rho}\ (\text{Sub}-\text{Trans}) \qquad \frac{\Gamma \vdash \sigma : ok}{\Gamma \vdash \sigma <: \sigma}\ (\text{Sub}-\text{Refl})$$

$$\frac{\Gamma \vdash \sigma' <: \sigma \quad \Gamma \vdash \tau <: \tau'}{\Gamma \vdash \sigma \to \tau <: \sigma' \to \tau'}\ (\text{Sub}-\text{Arrow}) \qquad \frac{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle : ok}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle <: \mathtt{obj}\, t.R}\ (\text{Sub}-\text{Object})$$

$$\frac{\Gamma \vdash \mathtt{pro}\, t.R : ok \quad t \text{ covariant in } R}{\Gamma \vdash \mathtt{pro}\, t.R <: \mathtt{obj}\, t.R}\ (\text{Sub}-\text{Seal})$$

Figure 8. Type rules, part I

— The *matching* rules define the matching relation as defined in (Bruce et al. 1997). (Match−Var) says that only variable types and `pro`-types can be related by a matching $\lessdot\!\!\#$, while (Match−Probject) says that any extension of $\mathtt{pro}\, t.R$ matches $\mathtt{pro}\, t.R$.

— The *subtyping* rules are those expected in an object oriented calculus. An object with more methods can safely be subsumed (used) in any context expecting an object with fewer methods (Sub−Object).

— Rule (Sub−Seal) says that a `pro`-type can subsume an `obj`-type with the same interface. As explained in (Fisher & Mitchell 1995) this means that after the object has

Code

$$\frac{\Gamma \vdash ok}{\Gamma \vdash c : \iota}\ (\text{Const}) \qquad \frac{\Gamma, x{:}\sigma \vdash ok}{\Gamma, x{:}\sigma \vdash x : \sigma}\ (\text{Var})$$

$$\frac{\Gamma, x{:}\sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}\ (\text{Abs}) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \tau}\ (\text{Appl})$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash \sigma <: \tau}{\Gamma \vdash M : \tau}\ (\text{Sub}) \qquad \frac{\Gamma \vdash ok}{\Gamma \vdash \langle\,\rangle : \langle\!\langle\,\rangle\!\rangle}\ (\text{Empty})$$

$$\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash \tau <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle}{\Gamma \vdash M \Leftarrow \mathtt{m} : \sigma[\tau/t]}\ (\text{Call}-\text{Pro}) \qquad \frac{\Gamma \vdash M : \tau \quad \Gamma \vdash \tau <: \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle}{\Gamma \vdash M \Leftarrow \mathtt{m} : \sigma[\tau/t]}\ (\text{Call}-\text{Obj})$$

$$\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash \tau <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle \quad \Gamma, t <\!\# \tau \vdash N : t \to \sigma}{\Gamma \vdash \langle M \leftarrow \mathtt{m} = N\rangle : \tau}\ (\text{Over})$$

$$\frac{\Gamma \vdash M : \mathtt{pro}\, t.R \quad \Gamma, t <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle \vdash N : t \to \sigma \quad \mathtt{m} \notin \mathcal{M}(R)}{\Gamma \vdash \langle M \leftarrow \mathtt{m} = N\rangle : \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle}\ (\text{Ext})$$

Figure 9. Type rules, part II

Evaluation contexts

$$\frac{\Gamma, a{:}\sigma \vdash ok}{\Gamma, a{:}\sigma \vdash a : \sigma}\ (\text{Ev}-\text{Var})$$

$$\frac{\Gamma \vdash M : \sigma \quad \Gamma \vdash a : \sigma}{\Gamma \vdash M[\mathsf{id}]^a : \sigma}\ (\text{Ev}-\text{Clos}_1) \qquad \frac{\Gamma \vdash U : \tau \quad \Gamma, x{:}\tau \vdash M[s]^a : \sigma \quad \Gamma \vdash a : \sigma}{\Gamma \vdash M[U/x\ ;\ s]^a : \sigma}\ (\text{Ev}-\text{Clos}_2)$$

$$\frac{\Gamma \vdash a : \tau \quad \Gamma \vdash U : \sigma \to \tau \quad \Gamma \vdash V : \sigma}{\Gamma \vdash (UV)^a : \tau}\ (\text{Ev}-\text{Appl}) \qquad \frac{\Gamma \vdash a : \sigma[\tau/t] \quad \Gamma \vdash U : \tau \quad \Gamma \vdash \tau <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle}{\Gamma \vdash (U \Leftarrow \mathtt{m})^a : \sigma[\tau/t]}\ (\text{Ev}-\text{Send})$$

$$\frac{\Gamma \vdash U : \tau \quad \Gamma \vdash \tau <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle \quad \Gamma, t <\!\# \tau \vdash V : t \to \sigma \quad \Gamma \vdash a : \tau}{\Gamma \vdash \langle U \leftarrow \mathtt{m} = V\rangle^a : \tau}\ (\text{Ev}-\text{Over})$$

$$\frac{\Gamma \vdash U : \mathtt{pro}\, t.R \quad \Gamma, t <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle \vdash V : t \to \sigma \quad \Gamma \vdash a : \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle \quad \mathtt{m} \notin \mathcal{M}(R)}{\Gamma \vdash \langle U \leftarrow \mathtt{m} = V\rangle^a : \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle}\ (\text{Ev}-\text{Ext})$$

$$\frac{\Gamma \vdash U : \tau \quad \Gamma \vdash \tau <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma\rangle\!\rangle \quad \Gamma \vdash V : \rho \quad \Gamma \vdash \rho <\!\# \tau \quad \Gamma \vdash a : \sigma[\rho/t]}{\Gamma \vdash \mathit{Sel}^a\big(U, \mathtt{m}, V\big) : \sigma[\rho/t]}\ (\text{Ev}-\text{Lookup})$$

$$\frac{\Gamma \vdash U : \sigma \quad \Gamma \vdash a : \sigma}{\Gamma \vdash \lceil U \rceil^a : \sigma}\ (\text{Ev}-\text{Id}) \qquad \frac{\Gamma \vdash a : \mathtt{pro}\, t.\langle\!\langle\rangle\!\rangle}{\Gamma \vdash \langle\,\rangle^a : \mathtt{pro}\, t.\langle\!\langle\rangle\!\rangle}\ (\text{Ev}-\text{Empty})$$

Figure 10. Type rules, part III

been "sealed" it can no longer be modified, *i.e.*, a sealed object cannot be overridden or extended. Moreover sealing is irreversible. Roughly speaking, this corresponds to "freezing" the object structure and makes it a message receiver only. This agrees with the class-based philosophy that stipulates that when an object has been created, it belongs to its class and it can only send and receive messages or be passed as a parameter. In class-based languages, once created, the structure of an object can no longer be modified.

6.2.2. *Type Rules for Code (Object Expressions)* These are the rules in Figure 9.

— rules for constants, variables, abstraction, application and subsumption are as usual in the λ-calculus with subtyping;
— rule (Empty) give a type to an empty object;
— rule (Call−Pro), (Call−Obj) give a type to a method call; in both cases we require the receiver be typed with a $\mathtt{pro}\,t.$-type (resp. $\mathtt{obj}\,t.$-type) that contains method $\mathtt{m}$;
— rule (Over) deals with object override: we require the object to be overridden M be typed with a $\mathtt{pro}\,t.$-type containing method $\mathtt{m}$ and that the new body N be typed in a context Γ enriched with a match-binding for the variable denoting the type of the object itself (*i.e.* `this`);
— rule (Ext) deals with object extension: we require the object to be extended M be typed with a $\mathtt{pro}\,t.$-type that does not contains the new method $\mathtt{m}$ and that the new body N be typed in a context Γ enriched with a match-binding for the variable denoting the type of the object itself (*i.e.* `self`).

6.2.3. *Type Rules for Evaluation Contexts* These are the rules in Figure 10.

— The typing of addresses is monomorphic, *i.e.* an address attached to a some evaluation context (or even object structure) of type τ must have type τ, *i.e.* we forbid subtyping on addresses; this practice is quite common in functional languages with references, where store is considered to be monomorphic.
— The rules (Ev−Closure$_{1,2}$) rules are inherited from calculi involving explicit substitutions.
— All the other rules present no surprises; the typing of the address enrich every premises in every rule.

It is worth to notice that the main point in the evaluation rules is how to evaluate address. In our formalism the rationale is as follows: if an address a explicitly surrounding an expression, then it has the same type as the expression itself. This, in principle can be generalized to every ATRS-based formalism, not only the object-based ones.

Example 6.1 (Subtyping for Code Reuse, (Fisher & Mitchell 1995)).
Consider the function

$$\mathtt{avg} \triangleq \lambda p_1.\lambda p_2.((p_1 \Leftarrow \mathtt{x}) + (p_2 \Leftarrow \mathtt{x}))/2$$

of type $\mathtt{obj}\,t.\langle\!\langle \mathtt{x{:}int} \rangle\!\rangle \to \mathtt{obj}\,t.\langle\!\langle \mathtt{x{:}int} \rangle\!\rangle \to \mathtt{int}$. Now consider two classical objects like:

$$\begin{aligned} \mathtt{point} &\triangleq \langle \mathtt{x} = \lambda\mathtt{self}.3, \mathtt{move} = \lambda\mathtt{self}.\lambda\mathtt{dx}.\langle \mathtt{self} \leftarrow \mathtt{x} = \lambda\mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} \rangle \rangle \\ \mathtt{cpoint} &\triangleq \langle \mathtt{point} \leftarrow \mathtt{col} = \lambda\mathtt{self}.\mathtt{red} \rangle \end{aligned}$$

As in (Fisher & Mitchell 1995) we can assign to those object the types

$$\begin{array}{rcl} \vdash \texttt{point} & : & \texttt{pro}\, t.\langle\!\langle \texttt{x:int}, \texttt{move:int} \to t \rangle\!\rangle \\ \vdash \texttt{cpoint} & : & \texttt{pro}\, t.\langle\!\langle \texttt{x:int}, \texttt{move:int} \to t, \texttt{col:colors} \rangle\!\rangle \end{array}$$

but also, via subtyping, the type $\texttt{obj}\, t.\langle\!\langle \texttt{x:int} \rangle\!\rangle$. As such we can type

$$\vdash \texttt{avg point cpoint} : \texttt{int}$$

6.3. *Subject Reduction and Type Soundness*

As an example of reasoning about implementations in the context of addressed term rewriting, we prove the subject reduction property and type soundness for typed $\lambda\mathcal{O}bj^a$. In the following, we denote by $\Gamma \vdash A$ any derivable judgment in our system.

As usual in explicit substitutions, we may assume, without loss of generality, that for any substitution of the form

$$[U_1/x_1\ ;\ U_2/x_2\ ;\ \ldots U_n/x_n],$$

for each $1 \leq i \leq n$, and $i \leq j \leq n$, the variable x_i is not free in U_j.

Lemma 6.1 (Auxiliary).

1 If $\Gamma \vdash \sigma <\!\!\# \tau$ or $\Gamma \vdash \sigma <: \tau$, then $\Gamma \vdash \sigma : ok$ and $\Gamma \vdash \tau : ok$.
2 If $\Gamma \vdash M : \tau$ or $\Gamma \vdash U : \tau$, then $\Gamma \vdash \tau : ok$.
3 If $\Gamma \vdash A$, then $Var(A) \subseteq Dom(\Gamma)$.
4 If $\Gamma \vdash A$, and $\Gamma, \Delta \vdash ok$, then $\Gamma, \Delta \vdash A$, *i.e.* the rule $\dfrac{\Gamma \vdash A \quad \Gamma, \Delta \vdash ok}{\Gamma, \Delta \vdash A}$ is admissible.
5 If $\Gamma, x{:}\tau, \Delta \vdash A$, and $\Gamma \vdash \sigma <: \tau$, then $\Gamma, x{:}\sigma, \Delta \vdash A$.

Proof. By routine induction on the structure of derivations. Point (4) use point (3). □

Lemma 6.2 (Generation).

1 $\Gamma \vdash \lambda x.M : \tau \to \sigma$ if and only if $\Gamma, x{:}\tau \vdash M : \sigma$.
2 $\Gamma \vdash MN : \sigma$ if and only if there exists τ such that $\Gamma, x{:}\tau \vdash M : \tau \to \sigma$, and $\Gamma \vdash N : \tau$.
3 $\Gamma \vdash (UV)^a : \sigma$, if and only if there exists τ such that $\Gamma \vdash U : \tau \to \sigma$ and $\Gamma \vdash V : \tau$, and $\Gamma \vdash a : \sigma$.
4 $\Gamma \vdash M[U_1/x_1\ ;\ U_2/x_2\ ;\ \ldots\ ;\ U_n/x_n]^a : \sigma$ if and only if there exist $\tau_1, \ldots, \tau_n$ such that for $1 \leq i \leq n-1$, we have $\Gamma, x_1{:}\tau_1, \ldots, x_i{:}\tau_i \vdash U_{i+1} : \tau_{i+1}$, and $\Gamma, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash M : \sigma$, and $\Gamma \vdash a : \sigma$.
5 $\Gamma \vdash \langle U \leftarrow \texttt{m} = V \rangle^a : \tau$ if and only if one of those cases are satisfied:
   (a) there exists R, σ such that $\Gamma \vdash U : \tau$ and $\Gamma \vdash \tau <\!\!\# \texttt{pro}\, t.\langle\!\langle R, \texttt{m:}\sigma \rangle\!\rangle$ and $\Gamma, t <\!\!\# \tau \vdash V : t \to \sigma$, and $\Gamma \vdash a : \tau$.
   (b) there exists R, σ such that $\Gamma \vdash U : \texttt{pro}\, t.R$ and $\Gamma, t <\!\!\# \texttt{pro}\, t.\langle\!\langle R, \texttt{m:}\sigma \rangle\!\rangle \vdash V : t \to \sigma$, and $\Gamma \vdash a : \texttt{pro}\, t.\langle\!\langle R, \texttt{m:}\sigma \rangle\!\rangle$ and $\texttt{m} \notin \mathcal{M}(R)$.

Proof. In each case the "if" direction is immediate from the shape of the typing rules. For the "only if" direction, the only complication is the presence of the non syntax directed rule (Sub); but a routine induction on the structure of the derivation suffices. □

Lemma 6.3 (Substitution).

1 If $\Gamma, x{:}\sigma, \Delta \vdash M[s]^a : \tau$, and $\Gamma \vdash N : \sigma$, then $\Gamma, \Delta \vdash M[N/x\ ;\ s]^a : \tau$.
2 If $\Gamma, t \lessdot\# \sigma, \Delta \vdash A$, and $\Gamma \vdash \tau \lessdot\# \sigma$, then $\Gamma, \Delta[\tau/t] \vdash A[\tau/t]$.

Proof. Both parts can be proved by induction on the structure of the derivations. □

We are now ready to prove the Subject Reduction property. Let $Addr(U)$ denote the set of addresses in U.

Theorem 6.1 (Subject Reduction).
If $\Gamma \vdash U : \sigma$, and $U \rightsquigarrow U'$, then for some Δ with $Dom(\Delta) = Addr(U') \setminus Addr(U)$, $\Gamma, \Delta \vdash U' : \sigma$.

Proof. We organize the proof by cases over the reduction rule applied.

Case:

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

Let the explicit substitution $[s]$ be $[U_1/x_1\ ;\ U_2/x_2\ ;\ \ldots U_n/x_n]$. Given that $\Gamma \vdash (MN)[s]^a : \sigma$, by inspecting the rules we see that it must be that $\Gamma \vdash a : \sigma$. By Lemma 6.2(4), there exists $\tau_1, \ldots, \tau_n$ such that for $1 \leq i \leq n-1$, we have that

$$\Gamma, x_1{:}\tau_1, \ldots, x_i{:}\tau_i \vdash U_{i+1} : \tau_{i+1} \quad \text{and} \quad \Gamma, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash MN : \sigma$$

By Lemma 6.2(2) there exists a type τ such that

$$\Gamma, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash M : \tau \to \sigma \quad \text{and} \quad \Gamma, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash N : \tau$$

Apply rules (Ev$-$Clos$_{1,2}$) to get

$$\Gamma, b{:}\tau \to \sigma \vdash M[s]^b : \tau \to \sigma \quad \text{and} \quad \Gamma, c{:}\tau \vdash N[s]^c : \tau$$

and so, using weakening on contexts, apply rule (Ev$-$Appl) to get

$$\Gamma, b{:}\tau \to \sigma, c{:}\tau \vdash (M[s]^b\ N[s]^c)^a : \sigma$$

as desired.

Case:

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

We must have that $\Gamma \vdash a : \sigma$, and by Lemma 6.2(3) there exists τ such that

$$\Gamma \vdash (\lambda x.M)[s]^b : \tau \to \sigma \quad \text{and} \quad \Gamma \vdash U : \tau.$$

Writing $[s]$ as $[U_1/x_1\ ;\ \ldots\ ;\ U_n/x_n]$, we have by Lemma 6.2(4) that there exists types $\tau_1, \ldots, \tau_n$ such that for $1 \leq i \leq n-1$, we have

$$\Gamma, x_1{:}\tau_1, \ldots, x_i{:}\tau_i \vdash U_{i+1} : \tau_{i+1} \quad \text{and} \quad \Gamma, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash \lambda x.M : \tau \to \sigma.$$

By Lemma 6.2(1) we get

$$\Gamma, x_1{:}\tau_1, \ldots, x_n{:}\tau_n, x{:}\tau \vdash M : \sigma.$$

Since

$$\Gamma, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash U : \tau$$

we may apply rules (Ev−Clos$_{1,2}$) to get

$$\Gamma \vdash M[U/x\ ;\ s]^a : \sigma$$

as desired.

Case:

$$x[U/x\ ;\ s]^a \quad \rightsquigarrow \quad \lceil U \rceil^a \qquad\qquad \text{(FVar)}$$

Let $[s]$ be $[U_1/x_1\ ;\ \ldots\ ;\ U_n/x_n]$. We must have that $\Gamma \vdash a : \sigma$. By Lemma 6.2(4), there exists $\tau, \tau_1, \ldots, \tau_n$ such that

$$\Gamma \vdash U : \tau$$

and for $1 \leq i \leq n-1$ we have

$$\Gamma, x_1{:}\tau_1, \ldots, x_i{:}\tau_i, x{:}\tau \vdash U_{i+1} : \tau_{i+1} \quad \text{and} \quad \Gamma, x_1{:}\tau_1, \ldots, x_n{:}\tau_n, x{:}\tau \vdash x : \sigma.$$

It follows from the last typing that $\tau <: \sigma$. Therefore

$$\Gamma \vdash U : \sigma$$

and so we may apply rule (Ev−id) to get

$$\Gamma \vdash \lceil U \rceil^a : \sigma.$$

as desired.

Case:

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

Let $[s]$ be $[U_1/x_1\ ;\ \ldots\ ;\ U_n/x_n]$. We must have that $\Gamma \vdash a : \sigma$. By Lemma 6.2(4) there exist $\tau, \tau_1, \ldots, \tau_n$ such that

$$\Gamma \vdash U : \tau$$

and for $1 \leq i \leq n-1$ we have

$$\Gamma, x_1{:}\tau_1, \ldots, x_i{:}\tau_i, y{:}\tau \vdash U_{i+1} : \tau_{i+1} \quad \text{and} \quad \Gamma, x_1{:}\tau_1, \ldots, x_n{:}\tau_n, y{:}\tau \vdash x : \sigma$$

Since $x \not\equiv y$ we get

$$\Gamma, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash x : \sigma$$

apply rules (Ev−Clos$_{1,2}$) to get

$$\Gamma \vdash x[s]^a : \sigma.$$

as desired.

Case:

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

We must have that $\Gamma \vdash a : \sigma$. By Lemma 6.2(3) there exists a type τ such that

$$\Gamma \vdash \lceil U \rceil^b : \tau \to \sigma \quad \text{and} \quad \Gamma \vdash V : \tau$$

By inspecting the typing rules we see that it must be the case that

$$\Gamma \vdash U : \tau \to \sigma$$

apply rule (Ev−Appl) to get

$$\Gamma \vdash (UV)^a : \sigma.$$

as desired.

Case:

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

Let $[s]$ be $[U_1/x_1\ ;\ \ldots\ ;\ U_n/x_n]$, and let $\Delta \triangleq x_1{:}\tau_1, \ldots, x_n{:}\tau_n, x{:}\tau$. We must have that $\Gamma \vdash a : \sigma$. By Lemma 6.2(4), there exist $\tau_1, \ldots, \tau_n$ such that, for $1 \leq i \leq n-1$, we get

$$\Gamma, \Delta \vdash U_{i+1} : \tau_{i+1} \quad \text{and} \quad \Gamma, \Delta \vdash (M \Leftarrow \mathtt{m}) : \sigma$$

The last typing ended with rule (Call−Obj), so it must be that σ is of the form $\delta[\tau/t]$ such that

$$\Gamma, \Delta \vdash M : \tau \quad \text{and} \quad \Gamma, \Delta \vdash \tau <\!\!\# \, \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\delta \rangle\!\rangle$$

Since

$$\Gamma \vdash \tau <\!\!\# \, \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\delta \rangle\!\rangle$$

apply rules (Ev−Clos$_{1,2}$), using weakening on contexts, to get

$$\Gamma, b{:}\tau \vdash M[s]^b : \tau$$

and then rule (Ev−Send) to obtain

$$\Gamma, b{:}\tau \vdash (M[s]^b \Leftarrow \mathtt{m})^a : \sigma$$

as desired.

Case:

$$(U \Leftarrow \mathtt{m})^a \quad \rightsquigarrow \quad Sel^a(U, \mathtt{m}, U) \qquad \text{(SA)}$$

We must have that $\Gamma \vdash a : \sigma$. Since the typing of $(U \Leftarrow \mathtt{m})^a$ ends with an application of rule (Ev−Send), then the type σ is of the form $\delta[\tau/t]$ such that

$$\Gamma \vdash U : \tau \quad \text{and} \quad \Gamma \vdash \tau <\!\!\# \, \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\delta \rangle\!\rangle$$

Rule (Ev−Lookup) is available with $\rho \equiv \tau$. The result type there is $\sigma \equiv \delta[\tau/t]$; thus

$$\Gamma \vdash Sel^a(U, \mathtt{m}, U) : \sigma$$

as desired.

Case:

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

Easy, since if $\lceil U \rceil^b$ gets type σ under a given environment then so does U.

Case:

$$Sel^a(\langle U \leftarrow \mathtt{m} = V \rangle^b, \mathtt{m}, W) \quad \rightsquigarrow \quad (V\,W)^a \qquad \text{(SU)}$$

We must have that $\Gamma \vdash a : \sigma$. The are two cases to consider according the $\leftarrow$ operator is used an override or an extension operator. We consider the first case the second being similar. By Lemma 6.2(5) we have

$$\Gamma \vdash \langle U \leftarrow \mathtt{m} = V \rangle^b : \tau \quad \text{and} \quad \Gamma \vdash \tau <\!\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\delta \rangle\!\rangle \quad \text{and} \quad \Gamma \vdash W : \rho \quad \text{and} \quad \Gamma \vdash \rho <\!\!\# \,\tau.$$

By Lemma 6.2 again we get

$$\Gamma, t <\!\!\# \,\tau \vdash V : t \to \delta.$$

The type σ of a is of the form $\delta[\rho/t]$. By Substitution Lemma (2) we get

$$\Gamma \vdash V : \rho \to \delta.$$

Apply rule (Sub) to get $\Gamma \vdash W : \rho$ and rule (Appl) to get

$$\Gamma \vdash (V\,W)^a : \delta[\rho/t]$$

as desired.

Case:

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

We must have that $\Gamma \vdash a : \sigma$. The are two cases to consider according the $\leftarrow$ operator is used an override or an extension operator. We consider the first case the second being similar. As in the previous case we have

$$\Gamma \vdash \langle U \leftarrow \mathtt{m} = V \rangle^b : \tau \quad \text{and} \quad \Gamma \vdash \tau <\!\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\delta \rangle\!\rangle \quad \text{and} \quad \Gamma \vdash W : \rho \quad \text{and} \quad \Gamma \vdash \rho <\!\!\# \,\tau.$$

Again we get

$$\Gamma \vdash U : \tau$$

The type σ of a is of the form $\delta[\rho/t]$. Apply rule (Ev$-$Lookup) to get

$$\Gamma \vdash Sel^a(U, \mathtt{m}, W) : \delta[\rho/t]$$

as desired.

Case:

$$Sel^a(\lceil U \rceil^b, \mathtt{m}, V) \quad \rightsquigarrow \quad Sel^a(U, \mathtt{m}, V) \qquad \text{(SelRed)}$$

Easy, since if $\lceil U \rceil^b$ gets type σ under a given environment then so does U.

Case:

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

Let $[s]$ be $[U_1/x_1\ ;\ \ldots\ ;\ U_n/x_n]$, and let $\Delta \triangleq x_1{:}\tau_1, \ldots, x_n{:}\tau_n$. We must have that $\Gamma \vdash a : \sigma$. We know that $\Gamma, \Delta \vdash \langle M \leftarrow \mathtt{m} = N\rangle : \sigma$. There are two cases as to the last rule in this derivation: either the last applied rule is (Over) or (Ext). In the former case we have that

$$\Gamma, \Delta \vdash M : \sigma \quad \text{and} \quad \Gamma, \Delta \vdash \sigma \lessdot\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\tau\rangle\!\rangle \quad \text{and} \quad \Gamma, \Delta \vdash N : t \to \tau$$

Then

$$\Gamma \vdash M[s] : \sigma \quad \text{and} \quad \Gamma \vdash \sigma \lessdot\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m}{:}\sigma_1\rangle\!\rangle \quad \text{and} \quad \Gamma \vdash N[s] : t \to \tau$$

and we can augment the context Γ with declarations $b{:}\sigma$ and $c{:}t \to \tau$ to obtain the desired result. If the last applied rule is (Ext), then the proof is similar.

Case:

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

Easy, since if $\lceil U \rceil^b$ gets type σ under a given environment then so does U.

□

Type Soundness follows from the Subject Reduction theorem. Recall Remark 3.3 concerning the normal forms of $\lambda\mathcal{O}bj^a$.

Theorem 6.2 (Type Soundness). The evaluation of a typable $\lambda\mathcal{O}bj^a$ expression never results in *method not found*.

Formally: if $\Gamma \vdash U : \sigma$, and $U \rightsquigarrow^* U'$, then U' is not of the form $Sel^a(\langle\ \rangle[s]^b, m, U)$.

Proof. Immediate from the Subject Reduction theorem, since $Sel^a(\langle\ \rangle[s]^b, m, U)$ is not typable. □

7. Conclusions

We have presented the initial steps in a theory of addressed term rewriting systems and detailed its use as a foundation for $\lambda\mathcal{O}bj^a$, a formalism to describe object-based calculi. This case study of $\lambda\mathcal{O}bj^a$ shows how ATRS's can support the analysis of implementations at the level of resource usage, modeling sharing of computations and sharing of storage, where each computation step in the calculus corresponds to a constant-cost computation in practice.

There is much to be developed in the pure theory of ATRS's to be sure: fundamental questions concerning confluence, termination, critical lemmas, and so forth all deserve investigation. The ATRS setting is a congenial one to analyze *strategies* in rewriting-based implementations. For example 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 redex of the term has to be reduced next (note that in a given term at a given address, at most one rule applies).

The calculus $\lambda\mathcal{O}bj^a$ itself is the basis for future work: we plan to extend $\lambda\mathcal{O}bj^a$ to handle the embedding-based technique of inheritance, following (Lang et al. 1999), to include a type system consistent with object-oriented features with the ability to type objects extending themselves, following (Gianantonio et al. 1998).

The applied techniques in our formalism could be also be applied in the setting of fixed-size objects like the Abadi and Cardelli's Object Calculus (Abadi & Cardelli 1996).

Although $\lambda\mathcal{O}bj^a$ is linear in addresses, we may consider whether to relax this constraint. Allowing non-linearity could have benefits such as supporting reasoning about term equality in a finer way. As an example we could design the following terms

$$\mathsf{eq}(x^a, x^a) \quad \to \quad \mathsf{true} \tag{1}$$

$$\mathsf{eq}(x^a, x^b) \quad \to \quad \mathsf{true} \tag{2}$$

$$\mathsf{eq}(x^a, y^a) \quad \to \quad \mathsf{true} \tag{3}$$

The first rewriting could correspond to physical equality (same object at the same address), while the second could correspond to a form of structural equality (same object in two different locations). The third equation could, *e.g.* be fired only if $x = y$.

Enriching our formalism with constant addresses is also another improvement. This could, *e.g.* allow terms of the following shape:

$$\mathsf{private\_eq}(x^{\texttt{F9EE0004}}, y^{\texttt{F9EE0004}}) \quad \to \quad \mathsf{true}$$

where `F9EE0004` is a constant address (in hexadecimal form).

Finally, a prototype of $\lambda\mathcal{O}bj^a$ will make it possible to embed specific calculi and to make experiments on the design of realistic object oriented languages.

Acknowledgments. Suggestions by Maribel Fernandez are gratefully acknowledged: they were very helpful in improving the paper. Moreover, the authors are sincerely grateful to all anonymous referees for their useful comments. Discussions with Francois-Régis Sinot were useful and are greatly appreciated.

References

M. Abadi & L. Cardelli (1995). 'On Subtyping and Matching'. In *European Conference for Object-Oriented Programming*, pp. 145–167.

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

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

Z. M. Ariola, et al. (1995). 'A Call-By-Need Lambda Calculus'. In *Proc. of POPL*, pp. 233–246. ACM Press.

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

Z. M. Ariola & J. W. Klop (1996). 'Equational term graph rewriting'. *Fundam. Inf.* **26**(3-4):207–240.

L. Augustson (1984). 'A Compiler for Lazy ML'. In *Symposium on Lisp and Functional Programming*, pp. 218–227. ACM Press.

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

H. P. Barendregt, et al. (1987). 'Term Graph Rewriting'. In *Proc. of PARLE*, vol. 259 of *LNCS*, pp. 141–158. Springer-Verlag.

Z.-E.-A. Benaissa, et al. (1996). 'Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution'. In *Proc. of PLILP*, vol. 1140 of *LNCS*, pp. 393–407. Springer-Verlag.

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

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

V. Bono & M. Bugliesi (1999). 'Matching for the lambda calculus of objects'. *Theoretical Computer Science* **212**(1–2):101–140.

P. Borovansky, et al. (1998). 'An Overview of ELAN'. In C. & H. Kirchner (eds.), *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications*, Pont-A-Mousson, France.

K. B. Bruce, et al. (1997). 'Subtyping is not a good "match" for object-oriented languages'. In *European Conference for Object-Oriented Programming*, vol. 1241 of *Lecture Notes in Computer Science*, pp. 104–127. Springer-Verlag.

M. Clavel, et al. (2003). 'The Maude 2.0 System'. In R. Nieuwenhuis (ed.), *Rewriting Techniques and Applications (RTA 2003)*, no. 2706 in Lecture Notes in Computer Science, pp. 76–87. Springer-Verlag.

P.-L. Curien (1991). 'An abstract framework for environment machines'. *Theoretical Computer Science* **82**:389–402.

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

R. Diaconescu & K. Futatsugi (1998). *Cafe Obj Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, vol. 6 of *Amast Series in Computing.* World Scientific.

S. Diehl, et al. (2000). 'Abstract machines for programming language implementation'. *Future Generation Computer Systems* **16**(7):739–751.

D. Dougherty, et al. (2002). 'A Generic Object-Calculus Based on Addressed Term Rewriting Systems'. Tech. Rep. RR-4549, INRIA.
`http://www.inria.fr/rrrt/rr-4549.html`.

D. J. Dougherty, et al. (2005). 'Addressed Term Rewriting Systems: Syntax, Semantics, and Pragmatics: Extended Abstract.'. *Electr. Notes Theor. Comput. Sci.* **127**(5):57–82.

H. Ehrig, et al. (eds.) (1999). *Handbook of Graph Grammars and Computing by Graph Transformation.* World Scientific. Vol 2: Applications, Languages and Tools.

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

K. Fisher, et al. (1994). 'A Lambda Calculus of Objects and Method Specialization'. *Nordic Journal of Computing* **1**(1):3–37.

K. Fisher & J. Mitchell (1995). 'A Delegation-based Object Calculus with Subtyping'. In *Fundamentals of Computation Theory (FCT'95).* Springer LNCS.

P. D. Gianantonio, et al. (1998). 'A Lambda Calculus of Objects with Self-inflicted Extension'. In *Proc. of OOPSLA*, pp. 166–178. ACM Press.

A. D. Gordon & P. D. Hankin (2000). 'A Concurrent Object Calculus: Reduction and Typing'. In U. Nestmann & B. C. Pierce (eds.), *Electronic Notes in Theoretical Computer Science*, vol. 16. Elsevier.

A. Igarashi, et al. (2001). 'Featherweight Java: a minimal core calculus for Java and GJ'. *ACM Trans. Program. Lang. Syst.* **23**(3):396–450.

G. Kahn (1987). 'Natural Semantics'. Tech. Rep. RR-87-601, INRIA.

J. W. Klop (1990). 'Term Rewriting Systems'. In S. Abramsky, D. Gabbay, & T. Maibaum (eds.), *Handbook of Logic in Computer Science*, vol. 1, chap. 6. Oxford University Press.

R. Kowalski (1979). *Logic for Problem Solving.* Artificial Intelligence Series, North Holland.

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

F. Lang, et al. (1999). 'A Framework for Defining Object-Calculi (Extended Abstract)'. In *Proc. of FM*, vol. 1709 of *LNCS*, pp. 963–982. Springer-Verlag.

X. Leroy, et al. (2004). *The Objective Caml system release 3.09 Documentation and user's manual.* Institut National de Recherche en Informatique et en Automatique.

J.-J. Lévy (1980). 'Optimal Reductions in the Lambda-calculus'. In J. P. Seldin & J. R. Hindley (eds.), *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 159–191. Academic Press.

L. Liquori (1997). 'An Extended Theory of Primitive Objects: FIrst Order System.'. In *ECOOP*, pp. 146–169.

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

R. Milner, et al. (1990). *The Definition of Standard ML.* MIT Press.

R. Milner, et al. (1997). *The Definition of Standard ML: Revised 1997.* The MIT Press.

E. Ohlebusch (2001). 'Implementing conditional term rewriting by graph rewriting.'. *Theor. Comput. Sci.* **262**(1):311–331.

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

S. Peyton Jones, et al. (2003). *Haskell 98 Language and Libraries: the Revised Report.* Cambridge University Press.

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

R. Plasmeijer & M. van Eekelen (2001). 'Concurrent Clean Language Report (version 2.0)'. `http://www.cs.ru.nl/~clean/contents/contents.html`.

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

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

R. Sleep, et al. (eds.) (1993). *Term Graph Rewriting.* John Wiley Publishers.

D. A. Turner (1979). 'A New Implementation Technique for Applicative Languages'. *Software Practice and Experience* **9**:31–49.

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

Appendix A. Acyclic Mutation-free ATRS

In this section, we consider a particular sub-class of ATRS, namely the ATRS involving *no cycles* and *no mutation*. Having no cycle means that there is no "pointer" from a node

to a node above it. We have noticed that such a back pointer is represented by a symbol $\bullet$. Thus an acyclic addressed term has no occurrence of $\bullet$. A mutation (or a side effect) is when an in-term which is not the in-term where the rewriting takes place is changed into another in-term. Due to the use of addresses this would be possible. The absence of mutation is made precise by the condition that addresses in the redexes cannot be changed. We show that the class of acyclic and mutation free ATRS is *sound* to simulate Term Rewriting Systems.

Definition A.1 (Acyclicity and Mutation-freeness).

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

$$a \in (addr(l) \cap addr(r)) \setminus \{loc(l)\} \Rightarrow l @ a \equiv r @ a.$$

— 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 A.2 (Mappings).

— An ATRS to TRS mapping is a homomorphism ϕ from acyclic addressed preterms to finite terms which preserves variables and constructors (the set of constructors in the TRS is $\{F_\phi \mid F \in \Sigma\}$):

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

— 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 sub-terms 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:

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

Example A.1. The address term $\mathsf{times}^a(\mathsf{square}^b(2^c), \mathsf{square}^b(2^c))$ is acyclic. The rule

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

is mutation-free as left-hand side and right-hand side contain no internal addresses. One has: $\phi_b(\mathsf{times}^a(\mathsf{square}^b(2^c), \mathsf{square}^b(2^c))) = \mathsf{times}_\phi(\Diamond, \Diamond)$.

Lemma A.1 (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 A.2 (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 A.3 (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 A.4 (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 A.5 (In-terms Conservation).
Let $l \rightsquigarrow r$ 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 $l \rightsquigarrow r$ *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 $l \mathbin{@} c \equiv r \mathbin{@} c$ *i.e.*, that $l \mathbin{@} c \equiv F^c(t_1, \ldots, t_n)$, 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 $a \in (\mathit{addr}(t) \cap \mathit{addr}(u)) \setminus \{b\}$ such that $t \mathbin{@} a \not\equiv u \mathbin{@} a$. Then a may have three distinct origins:

(a) a is a fresh address in u. Obviously, this is not possible since by hypothesis $a \in \mathit{addr}(t)$.

(b) There is an address $c \in \mathit{addr}(l) \cap \mathit{addr}(r)$ such that $\alpha(c) \equiv a$. In this case,

$$u \mathbin{@} a \equiv \langle\alpha \cup \alpha_f;\sigma\rangle(r) \mathbin{@} a \equiv \langle\alpha \cup \alpha_f;\sigma\rangle(r \mathbin{@} c)$$

from Lemma A.4 and acyclicity of u. Similarly,

$$t \mathbin{@} a \equiv \langle\alpha;\sigma\rangle(l) \mathbin{@} a \equiv \langle\alpha;\sigma\rangle(l \mathbin{@} c)$$

From the hypothesis, we know that $r \mathbin{@} c \equiv l \mathbin{@} c$, hence $r \mathbin{@} c$ contains no fresh address *i.e.*, $u \mathbin{@} a \equiv \langle\alpha;\sigma\rangle(r \mathbin{@} c) \equiv \langle\alpha;\sigma\rangle(l \mathbin{@} c) \equiv t \mathbin{@} a$, 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, $t \mathbin{@} a \equiv \sigma(X) \mathbin{@} a$. Since $\langle\alpha;\sigma\rangle(r)$ is acyclic, it must also be the case that $u \mathbin{@} a \equiv \sigma(X) \mathbin{@} a$.

□

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

Proof. From the definition of addressed term rewriting, we have that $t \rightsquigarrow \mathit{repl}(t)(u)$ where there are $a, \alpha, \alpha_f, \sigma, l, r$ such that $t \mathbin{@} a \equiv \langle\alpha;\sigma\rangle(l)$, and $u \equiv \langle\alpha \cup \alpha_f;\sigma\rangle(r)$. From Lemma A.1, we have $\phi(t) \equiv \phi_a(t)[\phi(t \mathbin{@} a)]$. Note that $a \in \mathit{addr}(t)$, hence $\phi_a(t)$ contains at least one hole. On the other hand, from Lemmas A.2 and A.5, we have $\phi(\mathit{repl}(u)(t)) \equiv \phi_a(t)[\phi(u)]$. We just have to show that $\phi(t \mathbin{@} a) \rightsquigarrow \phi(u)$ by rule $\phi(l) \rightsquigarrow \phi(r)$. This is immediate from Lemma A.3 since $\phi(t \mathbin{@} a) \equiv \phi(\langle\alpha;\sigma\rangle(l)) \equiv \phi_s(\sigma)(\phi(l))$,

and $\phi(u) \equiv \phi(\langle \alpha \cup \alpha_f; \sigma \rangle(r)) \equiv \phi_s(\sigma)(\phi(r))$, and obviously, $\phi_s(\sigma)(\phi(l)) \rightsquigarrow \phi_s(\sigma)(\phi(r))$, by rule $\phi(l) \rightsquigarrow \phi(r)$. □