URL: http://www.elsevier.nl/locate/entcs/volume73.html 24 pages

Addressed Term Rewriting Systems: Syntax, Semantics, and Pragmatics

[Extended Abstract]

Dan Dougherty

Worcester Polytechnic Institute, Worcester, MA, U.S.A.

Pierre Lescanne

École Normale Supérieure, Lyon, France

Luigi Liquori

INRIA Sophia Antipolis, France

Frédéric Lang

INRIA Rhone-Alpes, France

Abstract

We present a formalism called *Addressed Term Rewriting Systems*, which can be used to define the operational semantics of programming languages, especially those involving sharing, recursive computations and cyclic data structures. Addressed Term Rewriting Systems are therefore well suited for describing object-based languages, as for instance the family of languages called $\lambda\mathcal{O}bj^a$, involving both functional and object-based features.

1 Introduction

1.1 Addressed Calculi and Semantics of Sharing

Efficient implementations of lazy functional languages (and of computer algebras, theorem provers, etc.) require some sharing mechanism to avoid multiple computations of a single argument. A natural way to model this sharing in a symbolic calculus is to pass from a tree representation of terms to directed *graphs*. Such term graphs can be considered as a representation of program-expressions intermediate between abstract syntax trees and

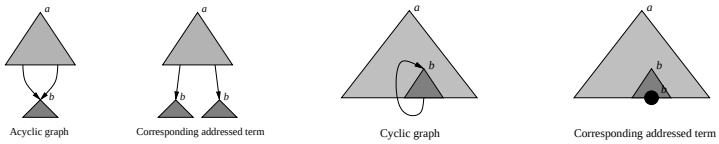

Figure 1. Sharing and Cycles Using Addresses

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 [BvEG+87,SPv93,Plu99,Blo01] for general treatments, and [Wad71,Tur79,AK94,AFM+95,AA95] for applications to λ-calculus and implementations.

In this paper we annotate terms, as trees, with *global addresses* in the spirit of [FF89,Ros96,BRL96]. Lévy [Lév80] and Maranget [Mar92] previously introduced *local addresses*; from the point of view of the operational semantics, global addresses describe better what is going in a computer or an abstract machine.

The formalisms of term-graph rewriting and addressed-term rewriting are fundamentally similar 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, 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 [DLL+02], addressed terms were studied in the context of *addressed term rewriting*, as an extension of classical first-order term rewriting. In addressed term rewriting we may rewrite simultaneously all sub-terms sharing a same address, mimicking what would happen in an implementation.

We also enrich the sharing with a special *back-pointer* to handle *cyclic graphs* [Ros96]. 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 Sections 2.

1.2 Suitability of Addressed TRS for describing an Object-based Framework

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" [FHM94,AC96,GH00,IPW01].

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

Several treatments of functional operational semantics exist in the literature [Lan64,Aug84,Kah87,MTH90]. Addressed Term Rewriting Systems (originally motivated by implementations of lazy functional programming languages [PJ87,PvE93]) are the foundation of the $\lambda\mathcal{O}bj^a$ framework [LLL99] for modeling object-oriented languages. The results in [LLL99] showed how to model $\lambda\mathcal{O}bj^a$ using Addressed Term Rewriting Systems, but with no formal presentation of those systems. Here we exposes 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 [Ros96,BRL96]. A useful way to understand the $\lambda\mathcal{O}bj^a$ framework 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 framework 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 system to work by presenting the three modules of rewriting rules that form the core of $\lambda\mathcal{O}bj^a$. For pedagogical sake, 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 [FHM94], and then we scale up to our $\lambda\mathcal{O}bj^a$. Section 4 presents a running object-based example in the $\lambda\mathcal{O}bj^a$ framework. Section 6 concludes. Section 5 show some theorems there address the relationship between $\lambda\mathcal{O}bj$ and $\lambda\mathcal{O}bj^a$.

For lack of space not all proof are presented here. A longer version of this paper containing full proofs and a large collection of functional and object-based and imperative examples concerning the object framework can be found in the technical reports and manuscript [LDLR99,DLL+02].

2 Addressed Term Rewriting Systems

In this section we introduce *addressed term rewriting systems* or ATRS in short. Classical term rewriting [DJ90,Klo90,BN98] 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:

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

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 6, we can observe that the object `p` and `q`

share a common structure addressed by b. This can be very easily formalized in the framework, since addresses are first-class citizens. See 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 introduced in memory via lazy evaluation of recursive code.

2.1 Addressed Terms

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

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

Definition 2.1 [Preterms]

(i) 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).

(ii) The location of an addressed preterm t, denoted by $loc(t)$, is defined by

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

and it is not defined on variables.

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

The definition of a preterm makes use of a special symbol $\bullet$ called a *back-pointer* and used to denote *cycles* [Ros96]. 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.

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 $fold(a)(t)$ as the *folding of preterms located at* a in t as follows:

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

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

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

We now proceed with the formal definition of *addressed terms* also called *admissible* preterms, or simply *terms*, for short, when there is no ambiguity. As already mentioned, addressed terms are preterms that denote term graphs.

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

unfolding.

Therefore, there are two notions to 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, $(\lambda \mathtt{x}.\mathtt{y})[\bullet^b/\mathtt{y}]^c$ has a dangling back-pointer, while $(\lambda \mathtt{x}.\mathtt{y})[\bullet^c/\mathtt{y}]^c$ 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 definition provides simultaneously two concepts:

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

Definition 2.3 [Dangling Addressed Terms] The set $DT(\Sigma)$ of *dangling addressed terms* is the smallest set that satisfies the following properties.

Variables. $\mathcal{X} \subseteq DT(\Sigma)$ and $X \mathbin{@} \_$ is nowhere defined.

Back-pointers. $\bullet^a \in DT(\Sigma)$ and $\bullet^a \mathbin{@} a \equiv \bullet^a$.

Expressions. For $t_1 \in DT(\Sigma), \ldots, t_n \in DT(\Sigma)$ such that: $b \in addr(t_i) \cap addr(t_j) \Rightarrow t_i \mathbin{@} b \equiv t_j \mathbin{@} b$, for a an address such that: $a \in addr(t_i) \Rightarrow t_i \mathbin{@} a \equiv \bullet^a$ and for $F \in \Sigma$ of arity n,

- $t \equiv F^a(t_1, \ldots, t_n) \in DT(\Sigma)$.
- $t \mathbin{@} a \equiv t$.
- $b \in addr(t_i) \setminus \{a\} \Rightarrow t \mathbin{@} b \equiv unfold(t)(t_i \mathbin{@} b)$.

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 $t \mathbin{@} \_$ 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 $a \in addr(t) \Rightarrow t \mathbin{@} a \not\equiv \bullet^a$. An admissible dangling addressed term will be simply denoted an *addressed term.*

Proposition 2.5 (In-terms Admissibility) *If t is an admissible term, and $a \in addr(t)$, then*

(i) *$t \mathbin{@} a$ is admissible, and*

(ii) *$\forall b \in addr(t \mathbin{@} a)$, we have $(t \mathbin{@} a) \mathbin{@} b \equiv t \mathbin{@} b$.*

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.

(i) *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.

(ii) *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).

(iii) *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.

(iv) For all a that occur both in t and u, the result of the rewriting step, say t', will have $t' @ a \equiv u @ a$, otherwise t' will be equal to t.

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

Definition 2.6 [Substitution, Matching, Unification]

(i) 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 $\langle \alpha; \sigma \rangle$.

(ii) 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:

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

(iii) We say that a term t *matches* a term p if there exists a substitution $\langle \alpha; \sigma \rangle$

such that $\langle \alpha; \sigma \rangle(p) \equiv t$.

(iv) 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 [DJ90] pp. 252) the places where replacement is performed are simply given by addresses instead of paths in the term.

Definition 2.7 [Replacement] Let t, u be addressed terms. The replacement generated by u in t, denoted by $repl(u)(t)$ is defined as follows:

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

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

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

Proposition 2.8 (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 2.9 [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 condition $loc(l) \equiv loc(r)$ says that l and r have the same top address, therefore l and r are not variables; the condition $var(r) \subseteq var(l)$ ensures that there is no creation of variables.

Definition 2.10 [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.

Note that, in general, we do not impose restrictions as linearity in addresses (*i.e.* the same address may occur twice), or acyclicity of l and r. However, $\lambda\mathcal{O}bj^a$ is *linear* in addresses (addresses occur only once) and patterns are never cyclic. Beside redirecting pointers, ATRS create *new* nodes. *Fresh renaming* insures that these new node addresses are not already used.

Definition 2.11 [Fresh Renaming]

(i) We denote by $dom(\varphi)$ and $rng(\varphi)$ the usual *domain* and *range* of a function φ.

(ii) A *renaming* is an injective address substitution.

(iii) 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 $dom(\alpha_{\text{fresh}}) = addr(r) \backslash addr(l)$ *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 2.12 (Substitution Admissibility) *Given an admissible term t that has a redex for the addressed rewriting rule $l \rightsquigarrow r$. Then*

(i) *A fresh renaming α_{fresh} exists for $l \rightsquigarrow r$ with respect to t.*

(ii) *$\langle \alpha \cup \alpha_{\text{fresh}}; \sigma \rangle(r)$ is admissible.*

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

Definition 2.13 [Rewriting] Let t be a term that we want to reduce at address a by rule $l \rightsquigarrow r$. Proceed as follows:

(i) Ensure $t \,@\, a$ is a redex. Let $\langle \alpha; \sigma \rangle(l) \triangleq t \,@\, a$.

(ii) Compute α_{fresh}, a fresh renaming for $l \rightsquigarrow r$ with respect to t.

(iii) Compute $u \equiv \langle \alpha \cup \alpha_{\text{fresh}}; \sigma \rangle(r)$.

(iv) 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 2.14 (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.*

2.3 Acyclic Mutation-free ATRS

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

Definition 2.15 [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 2.16 [Mappings]

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

$$\begin{aligned} \phi(X) &\triangleq X \\ \phi(F^a(t_1, \ldots, t_n)) &\triangleq F_\phi(\phi(t_1), \ldots, \phi(t_n)) \end{aligned}$$

- 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,

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

- 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:

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

Theorem 2.17 (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 $\phi(S) = \{\phi(l_i) \rightsquigarrow \phi(r_i) \mid i = 1..n\}$*

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

The purpose of this section is to describe the top level rules of the framework $\lambda\mathcal{O}bj^a$ as a framework strongly based on ATRS introduced in the previous section. The framework is described by a set of rules arranged in *modules*. The three modules are called respectively L, C, and F.

L is the *functional* module, and is essentially the calculus $\lambda\sigma^a_w$ of [BRL96]. This module alone defines the core of a purely functional programming language based on λ-calculus and weak reduction.

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

Figure 2. The Syntax of $\lambda\mathcal{O}bj^{\sigma}$

Basics for Substitutions

$$
\begin{array}{rl}
(MN)[s] \rightsquigarrow (M[s]\, N[s]) & \text{(App)} \\
\big((\lambda x.M)[s]\, U\big) \rightsquigarrow M[U/x \;;\; s] & \text{(Bw)} \\
x[U/y \;;\; s] \rightsquigarrow x[s] \quad x \not\equiv y & \text{(RVar)} \\
x[U/x \;;\; s] \rightsquigarrow U & \text{(FVar)} \\
\langle M \leftarrow m = N \rangle[s] \rightsquigarrow \langle M[s] \leftarrow m = N[s] \rangle & \text{(P)}
\end{array}
$$

Method Invocation

$$
\begin{array}{rl}
\langle\,\rangle[s] \rightsquigarrow \langle\,\rangle & \text{(NO)} \\
(M \Leftarrow m)[s] \rightsquigarrow (M[s] \Leftarrow m) & \text{(SP)} \\
(O \Leftarrow m) \rightsquigarrow Sel(O, m, O) & \text{(SA)} \\
Sel(\langle O \leftarrow m = U \rangle, m, V) \rightsquigarrow (U\, V) & \text{(SU)} \\
Sel(\langle O \leftarrow n = U \rangle, m, V) \rightsquigarrow Sel(O, m, V) \quad m \not\equiv n & \text{(NE)}
\end{array}
$$

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

$$M, N ::= \lambda x.M \mid MN \mid x \mid c \mid \langle\,\rangle \mid \langle M \leftarrow m = N\rangle \mid M \Leftarrow m \qquad \text{Code}$$

$$U, V ::= M[s]^a \mid (UV)^a \mid \qquad \text{Eval. Contexts}$$

$$(U \Leftarrow m)^a \mid \langle U \leftarrow m = V\rangle^a \mid [\![O]\!]^a \mid Sel^a\big(O, m, U\big) \mid \lceil U \rceil^a \mid \bullet^a$$

$$O ::= \langle\,\rangle^a \mid \langle O \leftarrow m = V\rangle^a \mid \bullet^a \qquad \text{Object Structures}$$

$$s ::= U/x\ ;\ s \mid \mathsf{id} \qquad \text{Substitutions}$$

Figure 4. The Syntax of $\lambda\mathcal{O}bj^a$

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

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

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:

(i) first we present the functional calculus $\lambda\mathcal{O}bj^\sigma$, intermediate between the calculus $\lambda\mathcal{O}bj$ of Fisher, Honsell and Mitchell [FHM94] and our $\lambda\mathcal{O}bj^a$.

(ii) Then we scale up over the full $\lambda\mathcal{O}bj^a$ as a *conservative extension* of $\lambda\mathcal{O}bj^\sigma$ in the sense that for an acyclic mutation-free term, computations in $\lambda\mathcal{O}bj^a$ and computations in $\lambda\mathcal{O}bj^\sigma$ return the same normal form. Since a $\lambda\mathcal{O}bj^a$-term yields a $\lambda\mathcal{O}bj^\sigma$-term by erasing addresses and indirections, one corollary of this conservativeness is *address-irrelevance*, *i.e.* the observation that the program layout in memory cannot affect the eventual result of the computation. This is an example of how an informal reasoning about implementations can be translated in $\lambda\mathcal{O}bj^a$ and formally justified.

3.1 Syntax of $\lambda\mathcal{O}bj^\sigma$

$\lambda\mathcal{O}bj^\sigma$ does not use addresses (see the syntax in Figure 2). The syntax of $\lambda\mathcal{O}bj^\sigma$ is presented in Figure 3; the reader will note that terms of this calculus are terms of $\lambda\mathcal{O}bj^a$ without the addresses, indirections, and object identities, and the rules are properly contained in those of modules $\mathsf{L} + \mathsf{C} + \mathsf{F}$ of $\lambda\mathcal{O}bj^a$. 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 change simultaneously). The second and third categories define dynamic entities, or inner structures: the *evaluation contexts*, and the *internal*

The Module L

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

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

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

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

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

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

The Module C

$$\langle\,\rangle[s]^a \rightsquigarrow [\![\langle\,\rangle^b]\!]^a \qquad \text{(NO)}$$

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

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

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

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

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

The Module F

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

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

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

Figure 5. The Modules L and C and F

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.

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

The syntax of $\lambda\mathcal{O}bj^a$ is summarized in Figure 4. 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 and object structures have explicit addresses.

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:

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

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

The Code Category.

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

- Pure λ-terms, constructed from abstractions, applications, variables, and constants. This allows the definition of higher-order functions.
- Objects, constructed from the empty object $\langle\,\rangle$ and a functional update operator $\langle \_ \leftarrow \_ \rangle$. An informal semantics of the update operator is given in Section 4. 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 $(UV)^a$, $(U \Leftarrow m)^a$, and $\langle U \leftarrow m = V \rangle^a$, are the evaluation contexts associated with the corresponding code constructors. Direct sub-terms of these evaluation contexts are themselves evaluation contexts instead of code.
- *Objects*, of the form $[\![O]\!]^a$, represent evaluated objects whose internal object structure is O and whose object identity is a. In other words, the address a plays the role of an *entry point* or *handle* to the object structure O, as illustrated by Figure 6.
- The term $Sel^a(O, m, U)$ is the evaluation context associated to a method-lookup, *i.e.*, the scanning of the object structure O to find the method m, and apply it to the object U. It is an auxiliary operator invoked when one sends a message to an object.

- The term $\lceil U \rceil^a$ denotes an indirection from the address a to the root of the addressed term U. The operator $\lceil \_ \rceil^a$ has no denotational meaning. It is introduced to make the right-hand side stay at the same address as the left-hand side. Indeed in some cases this has to be enforced. *e.g.* rule (FVAR). This gives account of phenomena well-known by implementors. Rules like (AppRed), (LCop) and (FRed) remove those indirections.
- *Back-references*, of the form $\bullet^a$ represents a *back-pointer* intended to denote cycles as explained in Section 2.

Internal Objects.

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

Remark 3.1 [ATRS-based preterms of $\lambda\mathcal{O}bj^a$] The concrete syntax of $\lambda\mathcal{O}bj^a$ of Figure 4 is consistent with the preterm definition in two ways:

(i) Symbols in the signature may also be infix (like *e.g.*, $(\_ \Leftarrow \_)$), bracketing (like *e.g.*, $[\![\_]\!]$), mixfix (like $\_[\_]$), or even "invisible" (as is traditional for application, represented by juxtaposition). In these cases, we have chosen to write the address outside brackets and parentheses.

(ii) We shall use $\lambda\mathcal{O}bj^a$ sort-specific variable names.

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

It is clear that not all preterms denote term graphs, since this may lead to inconsistency in the sharing. For instance, the preterm $\big((\![\langle\,\rangle^a]\!]^b \Leftarrow \mathtt{m})^a\, [\![\langle\,\rangle^a]\!]^b\big)^c$ is inconsistent, because location a is both labeled by $\langle\,\rangle$ and $\big(\_ \Leftarrow \_\big)$. The preterm $\big((\![\langle\,\rangle^a]\!]^b \Leftarrow \mathtt{m})^c\, [\![\langle\,\rangle^e]\!]^b\big)^d$ 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 $\big((\![\langle\,\rangle^a]\!]^b \Leftarrow \mathtt{m})^c\, [\![\langle\,\rangle^a]\!]^b\big)^d$ denotes a *legal* term graph with four nodes, respectively, at addresses a, b, c, and d [1].

[1] 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 $[\![\langle\,\rangle^a]\!]^b$, and hence will be rejected by a suitable sound type system or by a run-time exception.

Moreover, the nodes at addresses a and b, respectively labeled by $\langle\,\rangle$ and $[\![\_]\!]$, are shared in the corresponding graph since they have several occurrences in the term. These are the distinction captured by the well-formedness constraints defined in section 2.1. The rules of $\lambda\mathcal{O}bj^a$ as a computational-engine are defined in Figure 5.

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. The informal meaning of the reduction rules are defined in [LLL99], while a more formal explanation is given in the more complete [DLL+01].

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

Here we propose examples to help understanding the framework. We first give an example showing a functional object that extends itself [GHL98] with a field `n` upon reception of message `m`.

Example 4.1 [An Object which "self-inflicts" an Extension] Let

$$\texttt{self\_ext} \triangleq \langle\ \langle\,\rangle \leftarrow \texttt{add\_n} = \underbrace{\lambda\texttt{self}.\langle \texttt{self} \leftarrow \texttt{n} = \lambda\texttt{s}.1\rangle}_{N}\ \rangle.$$

The reduction of $M \triangleq (\texttt{self\_ext} \Leftarrow \texttt{add\_n})$ in $\lambda\mathcal{O}bj^a$ starting from an empty substitution is as follows:

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

In (1), two steps are performed to distribute the environment inside the extension, using rules (SP), and (FP). In (2), the empty object is given an object-structure and an object identity (NO). In (3), this new object is functionally extended (FC), hence it shares the structure of the former object but

has a new object-identity. In (4), and (5), two steps (SA) (SU) perform the look up of method `add_n`. In (6) we apply (Bw). In (7), the environment is distributed inside the functional extension (FP). In (8), (FVar) replaces `self` by the object it refers to, setting an indirection from h to b. In (9) the indirection is eliminated (FRed). Step (10) is another functional extension (FC). There is no redex in the last term of the reduction, *i.e.* it is in normal form.

Sharing of structures appears in the above example, since *e.g.* $[\![O]\!]^b$ turns out to have several occurrences in some of the terms of the derivation.

4.1 Object Representations in Figures 6

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.

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

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

After instantiation, the object `pixel` is located at an address, say a, and its object structure starts at address b, see Figure 6 (top). In what follows, we will derive three other objects from `pixel` and discuss the variations of how this may be done below.

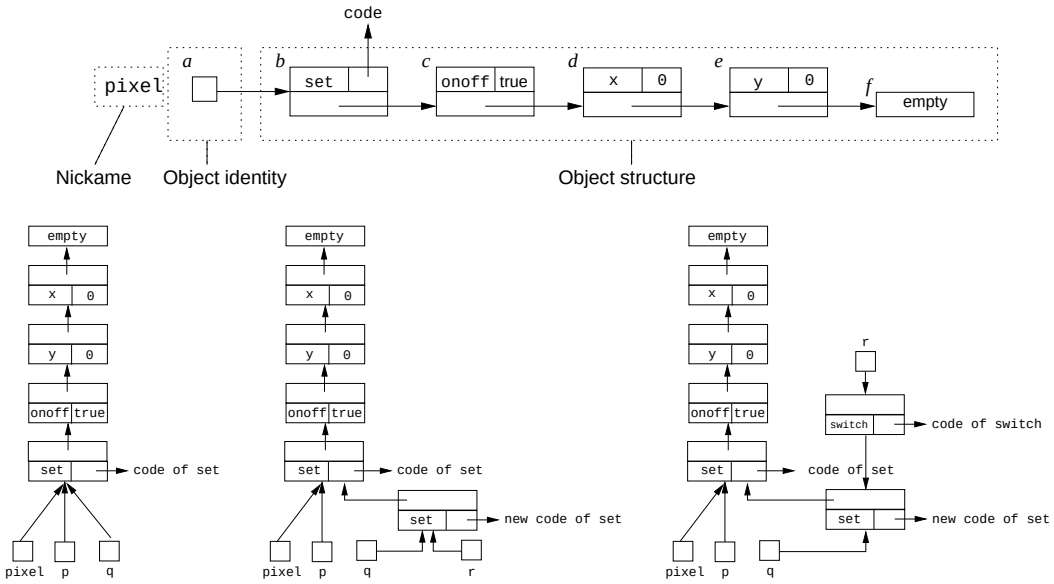

Figure 6. An Object Pixel (top), two Clones `p` and `q` (left), the Memory after `(1,2)` (right) and `(3)` (bottom).

4.2 Cloning

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 $(\lambda x.B)A$.)

```
let p = pixel in let q = p in q
```

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 6 (left). We might stress here that `p` and `q` should not be thought of as aliases of `pixel` as 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 framework when we *override* the `set` method of the clone `q` of `pixel`, and we extend a clone `r` of (the modified) `q` with a new method `switch`.

```
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 6 (middle) shows the state of the memory after the execution of the instructions `(1,2)`. Note that after `(1)` the object `q` refers to a new object-structure, obtained by chaining the new body for `set` with the old object-structure. As such, when the overridden `set` method is invoked, thanks to dynamic binding, the newer body will be executed since it will hide the older one. This dynamic binding is embodied in the treatment of the method-lookup rules (SU) and (NE) from Module `C` as described in Section 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, `(1)` only changes the object-structure of `q` without changing its object-identity. This is the sense in which our `clone` operator really does implement shallow copying rather than aliasing, even though there is no duplication of object-structure at the time that `clone` is evaluated.

This implementation model performs side effects in a very restricted and controlled way. Figure 6 (right), finally, shows the final state of memory after the execution of the instruction `(3)`. 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 [AC96], 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 object identities by the bracketing symbol $[\![\_]\!]$, the object `p` and `q`, presented in Figure 6, will be represented by the following addressed terms.

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

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

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 (for obvious lack of space) 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 2.3 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 acyclic $\lambda\mathcal{O}bj^a$-terms that erases addresses, indirection nodes ($\lceil\_\rceil^a$), and object identities ($[\![\_]\!]^a$), and leaves all the other symbols unchanged. Each term $\phi(U)$ is a term of $\lambda\mathcal{O}bj^\sigma$.*

Then we show a simulation result.

Theorem 5.2 ($\lambda\mathcal{O}bj^{\sigma}$ Simulates $\lambda\mathcal{O}bj^{a}$) *Let U be an acyclic $\lambda\mathcal{O}bj^{a}$-term. If $U \leadsto V$ in $\mathsf{L} + \mathsf{C} + \mathsf{F}$, then $\phi(U) \leadsto^{*} \phi(V)$ in $\lambda\mathcal{O}bj^{\sigma}$.*

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

Theorem 5.3 (Completeness of $\lambda\mathcal{O}bj^{a}$ w.r.t. $\lambda\mathcal{O}bj^{\sigma}$) *If $M \leadsto^{*} N$ 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 $M[\,]^{a} \leadsto^{*} U$ in $\mathsf{L} + \mathsf{C} + \mathsf{F}$ of $\lambda\mathcal{O}bj^{a}$.*

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 5.4 (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.*

6 Conclusions

We have presented the theory of addressed term rewriting systems and detailed its use as a foundation for $\lambda\mathcal{O}bj^{a}$, a framework to describe object-based calculi. This case study of $\lambda\mathcal{O}bj^{a}$ shows how ATRSs 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.

The ATRS setting is a congenial one to analyze *strategies* in rewriting-based implementations. For example the approach for functional languages studied in [BRL96] should be generalizable to $\lambda\mathcal{O}bj^{a}$: from a very general point of view, a strategy is a binary relation between addressed terms and addresses. The addresses, in relation with a given term, determines which redexes of the term has to be reduced next (note that in a given term at a given address, at most one rule applies).

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 [LLL99], to include a type system consistent with object-oriented feature with the ability to type objects extending themselves, following [GHL98].

The applied techniques in our framework could be also be applied in the setting of fixed-size objects like the Abadi and Cardelli's Object Calculus [AC96].

During the workshop Francois-Régis Sinot raised an interesting question about linearity of addresses. Although $\lambda\mathcal{O}bj^{a}$ is linear in addresses, we may consider whether to relax this constraint. Although it is well known that allowing non-linearity in terms can break confluence in ordinary term rewriting systems, it is not clear if non-linearity in address will break confluence

in $\lambda\mathcal{O}bj^a$. Allowing non-linearity could have positive benefits, like reasoning about term equality in a finer way. As an example we could design the following terms

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

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

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

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 $x = y$, or simply not legal.

Enriching our framework with constant address is also another improvement suggested by Sinot. This could, *e.g.* allow term of the following shape:

$$\mathsf{private\_eq}(x^{\mathtt{FRXX0004}}, y^{\mathtt{FRXX0004}}) \to \mathsf{true}$$

where `FRXX0004` 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 post-workshop by Francois-Régis Sinot were useful and greatly appreciated.

References

[AA95] Z. M. Ariola and Arvind. Properties of a First-order Functional Language with Sharing. *Theoretical Computer Science*, 146(1–2):69–108, 1995.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AFM+95] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and Ph. Wadler. A Call-By-Need Lambda Calculus. In *Proc. of POPL*, pages 233–246. ACM Press, 1995.

[AK94] Z. M. Ariola and J. W. Klop. Cyclic Lambda Graph Rewriting. In *Proc of LICS*, pages 416–425. IEEE Computer Society Press, 1994.

[Aug84] L. Augustson. A Compiler for Lazy ML. In *Symposium on Lisp and Functional Programming*, pages 218–227. ACM Press, 1984.

[Blo01] S. C. Blom. *Term Graph Rewriting, Syntax and Semantics*. PhD thesis, Vrije Universiteit, Amsterdam, 2001.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[BRL96] Z.-E.-A. Benaissa, K.H. Rose, and P. Lescanne. Modeling Sharing and Recursion for Weak Reduction Strategies using Explicit Substitution. In *Proc. of PLILP*, volume 1140 of *LNCS*, pages 393–407. Springer-Verlag, 1996.

[BvEG+87] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term Graph Rewriting. In *Proc. of PARLE*, volume 259 of *LNCS*, pages 141–158. Springer-Verlag, 1987.

[DJ90] N. Dershowitz and J.-P. Jouannaud. *Handbook of Theoretical Computer Science*, volume B, chapter 6: Rewrite Systems, pages 244–320. Elsevier Science Publishers, 1990.

[DLL+01] D. Dougherty, F. Lang, P. Lescanne, L. Liquori, and K. Rose. A Generic Object-Calculus based on Addressed Term Rewriting Systems. In P. Lescanne, editor, *Proc. of WESTAPP'01, Fourth International Workshop on Explicit Substitutions: Theory and Applications to Programs and Proofs*, pages 6–25. Logic Group Preprint series No 210. Utrecht University, the Netherlands, 2001.

[DLL+02] D. Dougherty, F. Lang, P. Lescanne, L. Liquori, and K. Rose. A Generic Object-calculus Based on Addressed Term Rewriting Systems. Technical Report RR-4549, INRIA, 2002. http://www.inria.fr/rrrt/rr-4549.html.

[EEKR99] H. Ehrig, G. Engels, H-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, 1999. Vol 2: Applications, Languages and Tools.

[FF89] M. Felleisen and D. P. Friedman. A Syntactic Theory of Sequential State. *Theoretical Computer Science*, 69:243–287, 1989.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[GH00] A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In Uwe Nestmann and Benjamin C. Pierce, editors, *Electronic Notes in Theoretical Computer Science*, volume 16. Elsevier, 2000.

[GHL98] P. Di Gianantonio, F. Honsell, and L. Liquori. A Lambda Calculus of Objects with Self-inflicted Extension. In *Proc. of OOPSLA*, pages 166–178. ACM Press, 1998.

[IPW01] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[Kah87] G. Kahn. Natural Semantics. Technical Report RR-87-601, INRIA, 1987.

[Klo90] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, chapter 6. Oxford University Press, 1990.

[Lan64] P. J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6, 1964.

[LDLR99] F. Lang, D. Dougherty, P. Lescanne, and K. Rose. Addressed Term Rewriting Systems. Technical Report RR 1999-30, LIP, ENS, Lyon, 1999.

[Lév80] J.-J. Lévy. Optimal Reductions in the Lambda-calculus. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191. Academic Press, 1980.

[LLL99] F. Lang, P. Lescanne, and L. Liquori. A Framework for Defining Object-Calculi (Extended Abstract). In *Proc. of FM*, volume 1709 of *LNCS*, pages 963–982. Springer-Verlag, 1999.

[Mar92] L. Maranget. Optimal Derivations in Weak Lambda Calculi and in Orthogonal Rewriting Systems. In *Proc. of POPL*, pages 255–268, 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[PJ87] S. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, 1987.

[Plu99] D. Plump. *Term Graph Rewriting*, chapter 1, pages 3–61. World Scientific, 1999. in [EEKR99].

[PvE93] M. J. Plasmeijer and M. C. D. J. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* International Computer Science Series. Addison-Wesley, 1993.

[Ros96] K. H. Rose. *Operational Reduction Models for Functional Programming Languages.* PhD thesis, DIKU, København, Denmark, 1996. DIKU report 96/1.

[SPv93] R. Sleep, R. Plasmeijer, and M. C. D. J. van Eekelen, editors. *Term Graph Rewriting. Theory and Practice.* John Wiley Publishers, 1993.

[Tur79] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software Practice and Experience*, 9:31–49, 1979.

[Wad71] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus.* PhD thesis, Oxford, 1971.