# Addressed Term Rewriting Systems*

Frédéric Lang, Daniel Dougherty, Pierre Lescanne, and Kristoffer Rose

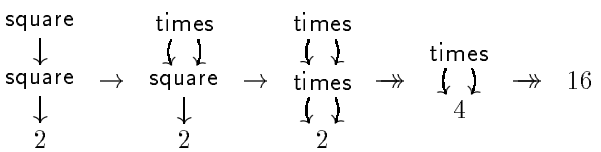
Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon
46, Allée d'Italie, F–69364 Lyon Cedex 07, FRANCE
E-mail: Frederic.Lang@ens-lyon.fr, *etc.*
URL: http://www.ens-lyon.fr/LIP/PLUME/plume.us.html
Phone: +33 (0)4 72 72 85 04 Fax: +33 (0)4 72 72 80 80

**Abstract.** We propose *Addressed Term Rewriting Systems* (ATRS) as a solution to the still-standing problem of finding a simple yet formally useful framework that can account for computation with sharing, cycles, and side effects. ATRS are characterised by the combination of three features: they work with terms where structural induction is useful, they use a minimal back-pointer representation of cyclic data, and they ensure a bounded complexity of rewrite steps by eliminating implicit pointer redirection. In the paper we develop this and show how it is a very useful compromise between less abstract term graph rewriting and the more abstract equational rewriting.

## 1 Introduction

Term Rewriting Systems (TRS, see [DJ90, Klo92, BN98]) are widely used to formalize, prototype, and prove software. However, they do not easily express some of the useful notions that one finds in software implementations, namely sharing, cycles, and side effects. In this introduction we explain each of these and how we are able to formalise them all in the following.

*Sharing.* An essential notion to give an account of efficiency in the implementation of TRS. Indeed, while TRS rules such as the usual rule for squaring a number $\mathsf{square}(x) \to \mathsf{times}(x, x)$ in principle copies a subterm of the input, real implementations optimize in only copying a *pointer* to this term. This not only saves memory but also makes it possible to *share future computations* on the referred subterm. For example, we can compute, provided that a rule to reduce $\mathsf{times}(x, y)$ to $x \times y$ exists,

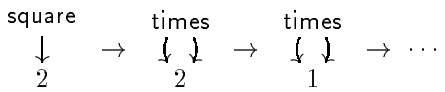



* Extended Abstract; full version available as [LDLR99].

Such sharing is, for example, essential in efficient implementations of lazy functional programming languages [PJ87], and the initial studies of sharing as "term graph rewriting" [BVEG+87] were indeed motivated by this application. However, one has to be careful. Consider the squaring TRS augmented with the rules $2 \to 1$, $1 \to 2$, and $\mathsf{times}(1,2) \to \mathsf{ok}$ (these are silly but similar situations do arise in practice). This TRS permits the reduction

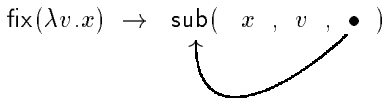
$$\mathsf{square}(2) \to \mathsf{times}(2,2) \to \mathsf{times}(1,2) \to \mathsf{ok}$$

If one uses sharing naively then the only possible reduction is the infinite

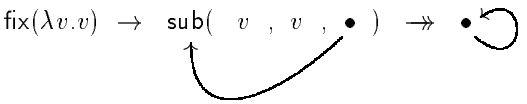



since it is impossible to "unshare" the copied node. These issues have been studied for term graph rewriting systems (TGRS, see [Plu] for a survey), in particular the *adequacy* of TGRS to "implement" TRS [Plu93, KKSdV94, AKP97, EJ98]. However, the issue of guaranteeing *efficient* implementation has not been addressed until this work. check reference list

*Cycles.* These are also essential since one wants to deal with infinite data structures. A typical TRS rule that we would like to implement with a cycle is the notorious fixed point unfolding rule $\mathsf{fix}(\lambda v.x) \to \mathsf{sub}(x, v, \mathsf{fix}(\lambda v.x))$ (with $v$ some variable and additional rules defining $\mathsf{sub}$). This also illustrates the view that cycles are "self-sharing" or, as it is often expressed, exhibit *vertical* sharing (in contrast to usual *horizontal* sharing). We would like to implement the above rule as something like



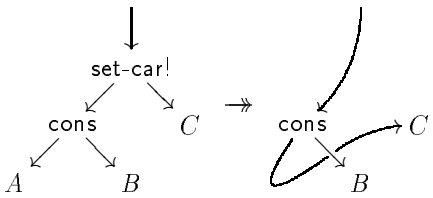

using a special "back-pointer" indirection node. This creates more problems, however, in particular we have to deal specially with the "black hole" problem corresponding to the reduction




(assuming that rules for $\mathsf{sub}$ are properly defined). The resulting graph is called a "black hole" because it does not make sense given the intuition that $\bullet$ is just notation for a backpointer. Again, this is a real problem that occurs in connection with recursive equations in lazy functional programming languages (Haskell, Clean, *etc.*), or logic programming language (Prolog, Lambda Prolog, *etc.*, where it is known as the "missing occur check"). The issue of cyclic structures has

also been studied for TGRS [Ros92, AK96] and the problem is also related to "rational trees" [Cou83] which can be implemented in a finite setting as cyclic graphs but as in the general case none of these discuss the efficiency of the rewriting. check references

*Side effects.* Side effects are a basic notion in many programming languages, thus it is important to provide a formal framework for their description. For instance, in many object oriented programming languages, changing the value of a field in a shared object performs a side effect: the behaviour of other references to the same object is affected by the "non-local" update. Another, more classical, example is the set-car! operator of the Scheme [CR91] language (originating from the rplaca primitive of Lisp). This operator has a graph rewrite rule like the one on the right (where we have tried to indicate graphically that there is "redirection" going on). This rule is worse than the previous ones because it involves *two* redirections that happen simultaneously. For comparison, the TGRS model can only describe a single redirection per reduction step. Describing side effects in conjunction with parallel computing was in fact one of the goals of the DACTL project [GKS89] from which TGRS emerges but it was never achieved. Similarly, [AK96] conclude their definition of Equational Term Graph Rewriting by *"Furthermore, we intend to study the suitability of equational graph rewriting for expressing side-effect operations"*. Side effects were, however, handled by [FF89] by introducing (what we call) *addresses*, to cite: *"we incorporate the store into the program component of the machine"*. This approach as well as the later "addressed explicit rewriting" [Ros96] suffer from having a very low-level notion of reduction which thus makes generic algebraic reasoning difficult.




*Synthesis.* Our main purpose is to unify the two approaches to give an algebraic and general account of computation with a *memory state* while preserving the nice properties of simple term rewriting sufficiently to permit usual inductive reasoning. The idea is the same as in store-based models: the state of a memory is seen as a map from addresses to nodes, where each node carries a piece of information in the form of a *label* (for instance telling that the current node denotes the head of a list), and a vector of *successor addresses* (for instance, the vector containing the address of the next cell of the list) which make reference to the connected nodes, obtained through the state itself. In other words, a state can be seen as a cyclic term graph. However, we eliminate the following structural problems with TGRS:

1. It is very hard to express *locality* of reduction with term graphs. Computers work by having a small "scratch area" in which they keep essentially a subterm of the entire problem. This can be seen as a small connected component

of a term graph but we then still cannot observe the *inductive structure* of the considered subterm.

2. The notion of *reduction step* is not directly realizable as a single operation in the computer. The culprit is the *redirection phase* of [BVEG+87]: this has an unbounded complexity in the sense that the number of pointers that have to be redirected depends on the context of the problem being solved.

With respect to the first issue, we note that, in the term graph model, memory addresses are modeled by "node identity". Thus, it is difficult to model a "move" of data in memory since updated parts of a graph due to rewriting are new graph nodes, with new identities, *i.e.* new addresses. Hence, TGRS cannot model non-functional issues such as side effects. We introduce addresses in a less obtrusive way: they are annotations on terms which survive updates due to rewriting. With this we can truly have "add-on side-effects" in the sense that only the parts of a specification that actually *breaks* locality needs to be specified as such.

With respect to the second issue, ATRS involve an explicit treatment of redirection by using *indirection nodes*, while implicit redirections, as described in most papers on TGRS, are simply not permitted. As a consequence we require a separate treatment of the so-called *collapsing rules* which resolves the "black hole" problem, as in our setting, a black hole is just a cyclic chain of indirection nodes. Moreover, with explicit indirection nodes to implement redirection we can claim that "counting rewrite steps" has a direct correspondance with the real complexity of implementations.

Finally, introducing addresses in the representation of a graph has some nice and useful consequences. In particular, it gives us the opportunity to represent graphs as trees (or terms) in a canonical way. Indeed, the addresses provide the required anotations, so that there is an isomorphism between graphs (or states) and addressed terms. The advantages of this new representation of graphs upon the Barendregt *et al.* ones is that it allows a very rigorous definition of rewriting, while providing an intuitive correspondance between graphs and first order terms, and the corresponding rewriting systems.

*Plan.* After some preliminary definitions in Section 2, we define in Section 3 addressed terms, and show their correspondance with term graphs. In Section 4, we define graph matching and graph replacement on addressed terms, notions which will be used in Section 5 to define the rewriting of addressed terms, and show that the set of addressed terms is closed under rewriting. In Section 6, we give the example of a system which implements the `set-car!` of Lisp, handling side effects. In Section 7, we characterize side effects, and establish a criterium of confluence for ATRS. We conclude and discuss further work in Section 8.

## 2 Preliminaries

In this section, we give some definitions of structures which will be widely used in the following, namely functions and tuples, and of term graphs.

**Definition 1 (Functions, Tuples).**

1. Let $D, C$ be sets. A partial function $\Phi$ from $D$ to $C$ is denoted by a set of pairs of $D \times C$ written $d \mapsto c$, such that for all $d$ there exists at most one $c$ such that $d \mapsto c \in \Phi$.
2. For a set $S$, $S^n$ denotes the set of $n$-tuples with elements from $S$, written $\langle s_1; \ldots; s_n \rangle$.

Term graphs [BVEG+87, Plu] are rooted, directed, oriented, and cyclic graphs, whose nodes are labelled by symbols of a signature such that the number of successors of each node is given by the arity of its label.

**Definition 2 (Term Graphs).** Let $\Sigma$ be a signature with a function $\mathsf{arity} : \Sigma \mapsto \mathbb{N}$. A *term graph* over $\Sigma$ is a quadruple $(N; r; \mathsf{lab}; \mathsf{succ})$ where $N$ is a finite set of *node identifiers*, $r \in N$ is a distinguished node (identifier) called the *root*, $\mathsf{lab} : N \mapsto \Sigma$ maps each node to its *label*, $\mathsf{succ} : N \mapsto N^*$ maps each node to its *successor tuple*, and it is required for all nodes $n \in N$ that $\mathsf{succ}(n) \in N^{\mathsf{arity}(\mathsf{lab}(n))}$.

In the following we call term graphs simply graphs.

## 3 Addressed Terms

An *addressed term* is a term decorated with addresses, which denotes a graph in a canonical way. Upon all terms decorated with addresses, which we call *addressed preterms*, some do not represent graphs. We first start to define those preterms, before giving the required conditions for such a preterm to be a term, that we call an *admissible* preterm.

**Definition 3 (Preterms).**

1. Let $\Sigma$ be a term signature as in Definition 2, and $\bullet$ a special symbol of arity zero. Let be an enumerable set of *addresses* denoted by $a, b, c, \ldots$, and an enumerable set of *variables*, denoted by $x, y, z, \ldots$ An *addressed preterm* $t$ over $\Sigma$ is either: a variable $x$; or an expression of the form $F^a(t_1, \ldots, t_n)$ where $F \in \Sigma$ has arity $n \geq 0$, and each $t_i$ is an addressed preterm; or $\bullet^a$.
2. The location of an addressed preterm $t$, denoted by $\mathsf{loc}(t)$, is defined by $\mathsf{loc}\big(F^a(t_1, \ldots, t_n)\big) = \mathsf{loc}(\bullet^a) = a$. It is not defined on variables.
3. The set of variables, resp. the set of addresses, of a term $t$ is denoted by $\mathsf{var}(t)$, resp. $\mathsf{addr}(t)$, and defined in the obvious way.

The notion of address is introduced to identify distinct nodes of the term as several occurrences of a unique node of the represented graph. Intuitively, if two subterms have a same address, this means that the subgraph they represent is shared *i.e.*, there are (at least) two paths from the root of the graph which go to this node (namely the paths of the occurrences of the address). It is clear that some coherence has to be preserved in the term such that it really does correspond to a graph. Specifically: if two subterms have the same address,

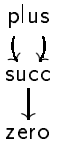
then they indeed should represent a unique subgraph and thus have the same "contents". The coherence conditions will be given later.

Yet, to model cyclic memory structures with this "one occurrence per path" principle we would need infinitary trees. This idea has been pursued [KKSdV95] and one needs not only infinity of terms but also of rewrite sequences to capture that all the infinite number of copies may need to be rewritten to reestablish a term that models a graph. The approach is a formally very clean way to achieve a model of cyclic structures but the loss of ordinary induction principles is a serious drawback when *using* the theory. The same holds, by the way, for the techniques used to reason about pointers in ML [Tof90, MT91] where one needs to use co-induction which is difficult to mix with "usual" term structure.

Our solution is the special symbol $\bullet$ called *back-pointer* used to give a representation of cyclic graphs as terms. Occurring in a term $t$, $\bullet^a$ denotes a reference to a node of $t$, which is located on the path going from the root of the term to itself, and has address $a$. On any cyclic path starting from the root in a cyclic graph, the first node visited twice is clearly identifiable and unique. This is this occurrence of the node which will appear as a back-pointer in the addressed term. For instance, the preterm $\mathsf{succ}^a(\bullet^a)$ denotes the cyclic graph on the right, (hence is a term), whose only node is given the address $a$. On the contrary, the addressed preterm $\mathsf{succ}^a(\bullet^b)$ does not represent any graph since $b$ has no other occurrence on the path from the root to $\bullet^b$, hence does not point back.

*Example 1.* Let zero of arity 0, succ of arity 1, plus and mult of arity 2. Examples of preterms with this signature are $t_1 = \mathsf{zero}^a$, $t_2 = \mathsf{succ}^a(x)$, $t_3 = \mathsf{plus}^c(\mathsf{zero}^a, \mathsf{zero}^b)$, $t_4 = \mathsf{succ}^a(\bullet^a)$, $t_5 = \mathsf{plus}^a(\mathsf{succ}^b(\mathsf{zero}^c), \mathsf{succ}^b(\mathsf{zero}^c))$, $t_6 = \mathsf{plus}^a(\mathsf{zero}^a, \mathsf{succ}^b(\mathsf{zero}^a))$, $t_7 = \mathsf{mult}^b(\bullet^a, \mathsf{succ}^a(\mathsf{zero}^b))$, $t_8 = \mathsf{succ}^a(\mathsf{succ}^a(\bullet^a))$. Only the first five of these preterms denote graphs. Actually $t_1, t_2$, and $t_3$ denote trees, $t_4$ is similar to the graph above and $t_5$ denotes the graph below. The last three, $t_6, t_7$, and $t_8$, are *non admissible*: $t_6$ because the address $a$ is bound to both $\mathsf{zero}^a$ and $t_6$ itself; $t_7$ because the address $a$ is bound to both $\bullet^a$ (which does not point back anywhere) and $\mathsf{succ}^a(\mathsf{zero}^b)$, and $b$ is also bound to different subgraphs; and $t_8$ because a smaller representation of the same graph can be written, namely $t_4$.

While handling addressed terms, it is necessary to maintain some coherence in their representation. To this aim, we need two basic operations which are called *folding* and *unfolding*. Folding consists in replacing a subterm by a $\bullet$, corresponding to the introduction of a cycle in the addressed term, *e.g.*, after some rewriting step. Unfolding is dual to folding, since it performs the replacement of a $\bullet$ by an addressed term at the same location. This is the price we pay[2] for insisting on a term structure: we have to unfold whenever one needs to observe the full subterm where it contains a




[2] A "conceptual"price, since in actual implementation it costs nothing.

cycle, because we cannot know by local observation in an addressed term what a $\bullet$ stands for. For instance, take the term $t = F^a(\bullet^a)$. It has a strict subterm $\bullet^a$, which in fact refers to $t$ itself. The way one gets back to $t$ from the back pointer is by unfolding it. Hence, we have a noetherian "subterm" relation, because our terms are finite; However we still need to be able to run arbitrarily far along the infinite paths of a corresponding graph; Unfold simply drives us back to this possibility, by expanding the back pointers.

The intuition is that folding and unfolding have no real operational meaning, but are used to keep the *representation* of a graph correct and unique. In fact, as we shall see later, the translation from an addressed term to a graph is invariant with respect to fold and unfold. We do not give the formal definition of folding and unfolding, which can be found in the long version of this paper.

Thus addressed terms provide a representation of cyclic graphs as first order finite terms with annotations. In this, they resemble term graphs in the linear notation defined in [BVEG+87], with two major differences: (1) every node has an address, and (2) if in a graph there exists more than one (say $n$) acyclic path leading to the same node (*i.e.*, to the same entry in a subgraph) this subgraph is duplicated $n$ times in the external representation of the addressed term.

Therefore, the label and successors of each node are locally accessible, independently of the graph structure, the sharing information being given by the addresses. For instance, the graph which would be represented by the linear term $\mathsf{mult}(m : \mathsf{plus}(1,2), m)$, where $m$ is the *name* of the node representing the root of $\mathsf{plus}(1,2)$, is represented by the addressed term $\mathsf{mult}^a(\mathsf{plus}^b(1^c, 2^d), \mathsf{plus}^b(1^c, 2^d))$ (uniquely, modulo an injective change of addresses, called *address renaming*). Of course, there is an exception for back pointers, for which unfolding is needed. This highlights the two advantages of the ATRS representation. First it makes structural induction on graphs possible, indeed induction on graphs boils down to induction on terms. As a consequence it makes type checking on cyclic graphs easily possible. Second, as we want to use these addressed terms in a rewriting framework, rewrite strategies will play a main rôle and this presentation of cyclic terms will make the description of strategies easy and precise.

We do not give the formal definition of term, which can be found in the long version of this paper. However, we give lighter definitions which gives the essence of admissibility.

**Definition 4 (Vertical Admissibility).** We call the *address context* of a subterm $s$ in a preterm $t$ the set of addresses occurring on the path from the root of $t$ to $s$. We say that a preterm is *vertically admissible* when (1) every $\bullet$ has an address which occurs in its address context, and (2) every other subterm has an address which does not occur in its address context.

The first condition ensures that every $\bullet$ is indeed a back-pointer *i.e.*, that one can get the referred term by looking up the context. The second condition ensures that one has a *minimal* representation *i.e.*, that a $\bullet$ is used as soon as it can for representing a cycle.

*Example 2.* In Example 1, $t_7$ is not vertically admissible because the first condition of the previous definition is not satisfied. $t_8$ is not vertically admissible because of a violation of the second condition.

**Definition 5 (Admissibility).** A preterm is *admissible* when it is vertically admissible, and when it is possible to build a *store* mapping every address to a unique term (using folding and unfolding adequately).

This last definition is slightly vague since we do not say how to build the store. Technically, however, it is very precise as one can see in the long version. Note only that the vertical admissibility condition implies a minimal representation of graphs as terms, which is what makes it easy to check *horizontal* admissibility *i.e.*, the fact that, for each address, a unique term can be retrieved, since it then suffices to verify that the minimal representations of two terms at the same location are syntactically the same.

*Example 3.* $\{a \mapsto \mathsf{plus}^a(\mathsf{succ}^b(\mathsf{zero}^c), \mathsf{succ}^b(\mathsf{zero}^c)), b \mapsto \mathsf{succ}^b(\mathsf{zero}^c), c \mapsto \mathsf{zero}^c\}$ is the store one can build for the term $t_5$ of Example 1. The term $t_6$ is not admissible because $a$ maps to both a term constructed with plus and a term contructed with zero.

The store of a term is analogous to the equational system defining its graph, as defined in [AK96], and similar to the term graph expression (the latter will be formalised last in this section). The difference is that in our approach the system defining the graph and its subgraphs is somewhat *stratified*. In other word, instead of associating a flat term to each address, one associates an addressed (sub)term, which could itself be represented as an equational system. The condition for an addressed preterm to denote a graph is therefore simple: the system must be solvable *i.e.*, an address must be bound to at most one subsystem. This is stated in our setting by the requirement for the store to be a function. Another helpful analogy is of course to see the store as a real (stratified) store on a machine.

*Example 4.* The addressed term $H^c(F^b(G^a(\bullet^b)), G^a(F^b(\bullet^a)))$ has the store:

$$\{a \mapsto G^a(F^b(\bullet^a)), b \mapsto F^b(G^a(\bullet^b)), c \mapsto H^c(F^b(G^a(\bullet^b)), G^a(F^b(\bullet^a)))\}$$

One sees the following (stated as propositions in the long version): (1) every address of $t$ has an entry in the store, and (2) every term of the range of the store is an admissible term, with the same store reduced to its set of addresses.

The following definition exploits the fact that, in an addressed term, in a given address context, there is a canonical term associated to each address, although there may be (syntactically) different terms which are located at this address, as shows Example 4.

**Definition 6.** Given an admissible term $t$, and given $a \in \mathsf{addr}(t)$, we write $t\,@\,a$ to denote the unique and admissible term located at address $a$ and obtained from the store.

It is clear that there is an isomorphism between addressed terms and graphs *i.e.*, from a ground and instantiated addressed term, one can build a unique graph, and from such a graph one can get back to the starting term (modulo a renaming of addresses).

**Theorem 7.** *We can map each addressed term into a unique term graph and we can model all possible term graphs with addressed terms i.e., addressed terms are exactly the preterms that model graphs.*

In the following, we only deal with admissible preterms (*i.e.*, terms), unless stated otherwise.

## 4 Matching and replacement of addressed terms

In this section, we define the *matching* of addressed terms, which corresponds to the matching of graphs. The idea is that an addressed term $t_1$ matches another (open) addressed term $t_2$, called the *pattern*, if there exists a graph homomorphism, and a substitution, from the graph denoted by $t_2$ to the graph denoted by $t_1$. Moreover, the matching operation builds and returns the homomorphism and the substitution. We also define *replacement*, an operation which simply expresses how a graph can be updated so that some of its nodes are changed. These two notions will play a major rôle in the definition of rewriting.

**Definition 8 (Matching).** We say that a term $t$ *matches* another term $p$ if one can build a pair of functions $\langle \alpha ; \sigma \rangle$ such that $\alpha$ is a function from addresses to addresses (defining the homomorphism), $\sigma$ is a function from variables to terms (defining the substitution), and these functions applied to $p$ return $t$.

Here again, the way one builds these functions and the way one applies them on a term is described in the long version. Note that in the definition of matching, $p$ plays the rôle of the *pattern*, and its inductive structure (as a term) is the key point of the matching algorithm, ensuring its termination. $t$ plays the rôle of the *instance*, and may be unfolded on demand, with respect to the structure of the pattern. For instance, matching the pattern $\mathsf{succ}^a(\mathsf{succ}^b(x))$ from the term $\mathsf{succ}^c(\bullet^c)$ makes no trouble, since due to the shape of the pattern, one has to unfold the back-pointer in the term only once. The matching built in this case is $\langle \{a \mapsto c, b \mapsto c\} ; \{x \mapsto \mathsf{succ}^c(\bullet^c)\} \rangle$. (This is thus quite similar to the matching employed for equational graph rewriting [?].)

*Example 5.* The term $\mathsf{plus}^c(\mathsf{succ}^d(\bullet^d), \mathsf{succ}^d(\bullet^d))$ matches $\mathsf{plus}^a(x, \mathsf{succ}^b(x))$ with $\langle \alpha ; \sigma \rangle = \langle \{a \mapsto c, b \mapsto d\} ; \{x \mapsto \mathsf{succ}^d(\bullet^d)\} \rangle$.

We now define *replacement*. The replacement function operates on terms. Given a term, it changes some of its subterms at given locations by other subterms with the same address.

**Definition 9 (Replacement).** A *replacement* is defined as the application of a store $\Delta$ to a term $t$, which results in a term $t'$ such that $t'$ has the same location as $t$, and for all address $a \in \mathsf{addr}(t)$: $t' @ a = \Delta(a)$ if $a \in \mathsf{dom}(\Delta)$, otherwise $t' @ a = (t @ a)'$, where $(t @ a)'$ is the application of $\Delta$ to $t @ a$.

Replacement operates like the replacement in a context of TRS, generalized to addressed term. The places where to perform a replacement is simply given by addresses instead of a path in the term.

*Example 6.* Let $t$ be $\mathsf{plus}^a(\mathsf{plus}^b(2^c, 2^c), \mathsf{plus}^b(2^c, 2^c))$ and $\Delta$ be $\{b \mapsto 4^b\}$. The replacement of $\Delta$ in $t$ gives $\mathsf{plus}^a(4^b, 4^b)$.

Kris: Say something about term graph matching which is also based on homeomorphic embedding ...

## 5 Addressed Term Rewriting Systems

We define in this section how rewriting is performed on addressed terms.

**Definition 10 (Rewriting Rule).**

1. An addressed rewriting rule over $\Sigma$ is a pair of addressed terms $(l, r)$ over $\Sigma$, written $l \to r$, such that $\mathsf{loc}(l) = \mathsf{loc}(r)$ (same top address, therefore $l$ and $r$ are not variables), and $\mathsf{var}(r) \subseteq \mathsf{var}(l)$ (no creation of variables).
2. A term $t$ *is a redex* for a rule $l \to r$ if $t$ matches $l$. A term $t$ *has a redex* if there exists an address $a \in \mathsf{addr}(t)$ such that $t @ a$ is a redex.

Note that we do not set any restriction of linearity in addresses, and that $l$ and $r$ may be eventually cyclic. Note also that projection rules, such as $F^a(x) \to x$ or $F^a(G^b(x)) \to G^b(x)$ are not addressed rewriting rules since both sides do not share the same address. However, we propose to add to the signature a unary function symbol, intuitively seen as an *indirection node* and written $\lceil\ \rceil$. Therefore, the TRS rule $F(x) \to x$ can be encoded by the ATRS rule $F^a(x) \to \lceil x \rceil^a$, and the TRS rule $F(G(x)) \to G(x)$ either by the rule $F^a(G^b(x)) \to G^a(x)$, which expresses the copy at address $a$ of the label and successor addresses of the node located at address $b$, namely $G(x)$, or by the rule $F^a(G^b(x)) \to \lceil G^b(x) \rceil^a$, which expresses an indirection from the node located at address $a$ to the node located at address $b$. The use of explicit indirection nodes is motivated by our wish to explicit the constraints that every rewriting step must be as close as possible to what happens in a real implementation. In particular the cost of the application of a rule must be almost constant in the implementation, so that the complexity of the rewriting and the complexity of the execution are tightened. Allowing projection rules would involve implicit redirections, whose cost depends on the number of references to the redirected node.

*Example 7.* $\mathsf{mult}^a(\mathsf{succ}^b(x), y) \to \mathsf{plus}^a(y, \mathsf{mult}^c(x, y))$ is an addressed rewriting rule. By its non right linearity, it expresses some sharing ($y$ must be seen as a pointer). It creates a new node in the graph, whose address is denoted by $c$ and which was not present in the left hand side. Moreover, unlike in classical graph rewriting, the node at address $b$ is only dereferenced, but not deleted.

The following definition introduces *fresh renaming*. Informally, a fresh renaming is required when new nodes are introduced by rewriting step. It is important to give these new nodes addresses which are not already used. Fresh renaming captures this intuition.

**Definition 11 (Fresh Renaming).** Let $l \to r$ be an addressed rewriting rule, and $t$ a term having a redex for this rule. A *fresh renaming* for $l \to r$ *w.r.t.* $t$ is an injective map from addresses to addresses (called a renaming) denoted by $\alpha_{\mathsf{fresh}}$ such that $\mathsf{dom}(\alpha_{\mathsf{fresh}}) = \mathsf{addr}(r) \setminus \mathsf{addr}(l)$ *i.e.*, the fresh renaming renames each newly introduced address to avoid capture, and $\mathsf{rng}(\alpha_{\mathsf{fresh}}) \cap \mathsf{addr}(t) = \varnothing$ *i.e.*, the chosen addresses are *fresh*, which means not present in $t$.

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

**Definition 12 (Rewriting).** Let $t$ be a term which we want to reduce at address $a$ by rule $l \to r$. Proceed as follows:

1. Ensure $t @ a$ is a redex and compute the matching $\langle \alpha ; \sigma \rangle$ from $l$ to $t @ a$.
2. Compute $\alpha_{\mathsf{fresh}}$, a fresh renaming for $l \to r$ *w.r.t.* $t$.
3. Compute $s$ as the result of applying the matching $\langle \alpha \cup \alpha_{\mathsf{fresh}} ; \sigma \rangle$ to $r$, and build the store $\Phi$ of the resulting term.
4. The result of rewriting $t$ by rule $l \to r$ at address $a$ is obtained as the replacement of $\Phi$ in $t$.

Addressed term rewriting is performed in a similar way as term rewriting: first find a redex, then remind its context, then compute the result, then replace it in the previous context. An extra step is performed here, which is the creation of fresh addresses. In some sense, term rewriting can be seen as a special case of addressed term rewriting, where all addresses are distinct (no sharing and no side effects) and where there are no back-pointers (no cycles).

The following proposition ensures that the set of addressed terms is closed under rewriting.

**Proposition 13.** *Let $t$ be a term. If $t \to t'$ then $t'$ is a term.*

Kris: Say something about term graph rewrite steps in the simplified case of non-root-destructive rules, c.f. [KKSdV94].

## 6 Modelling set-car!

The following shows how side effects are performed, by taking the famous example of set-car!. To our knowledge, it is the first time set-car! is defined in a setting of graph rewriting.

*Example 8.* The rules for the definition of car and set-car! are the following:

$$\mathsf{set\text{-}car!}^a(\mathsf{cons}^b(y,z),x) \to \lceil \mathsf{cons}^b(x,z) \rceil^a \qquad \text{(Set-Car)}$$

$$\mathsf{car}^a(\mathsf{cons}^b(n^c,z)) \to n^a \qquad \text{(Car-}n\text{)}$$

$$\mathsf{car}^a(\lceil z \rceil^b,x) \to \mathsf{car}^a(z,x) \qquad \text{(Elim-Ind)}$$

We do not give the definition of plus since it is not the purpose of this example. Suppose it is built-in. Note that this system is infinite, since there is an infinity of rules (Car-$n$), namely one for each $n \in \mathbb{N}$. Take the term

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

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 set-car!, the reduction proceeds as follows:

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

On the other hand, the reader may check that, starting by applying the rule for car, one gets $t \twoheadrightarrow 3^e$.

This 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 $\mathsf{set\text{-}car!}^a(\Box, \Box)$. Rules which do not host such transformations have no side effect.

## 7 Side effects, Overlapping, and Confluence

As shown in Example 8, non overlapping ATRS may be not confluent since they allow side effects. In this section, we characterize a subclass of the ATRS which are conjectured confluent. We call them *side effect free non overlapping ATRS*.

**Definition 14 (Overlapping, Side Effect).**

1. Let $\tau_1 = l_1 \to r_1$ and $\tau_2 = l_2 \to r_2$ be two rules, not necessarily distinct. $\tau_1$ *overlaps* with $\tau_2$ if and only if there exists a term $t$, and an address $a$ in $l_1$, such that $t$ matches both $l_1 \,@\, a$ and $l_2$, and if $\tau_1 = \tau_2$, then $a$ is not the location of members of the rule. A system in which no rules overlap is called *non overlapping*.
2. A rule $l \to r$ has a side effect if and only if there exists an address $a$ in $\mathsf{addr}(l) \cap \mathsf{addr}(r)$, distinct from the location of both members, such that $l \,@\, a$ is syntactically different from $r \,@\, a$. A rule or an ATRS with no side effects is called *side effect free*.

The name "side effect free" stems from the analogy with programming languages using a store: a rule which only modifies one place in the store (namely the location of the redex) does not have a side effect.

**Theorem 15.** *Side effect free non overlapping ATRS are confluent modulo address renaming.*

Kris: I think [KKSdV93] may be of interest here.

## 8 Conclusion

We have defined in this paper the Addressed Term Rewriting Systems, which have a non empty intersection with the Term Graph Rewriting Systems: In particular, implicit redirections are not allowed in ATRS due to the constraint that both members of a rule must be located at the same address. The new feature of ATRS is that they give an account of side effects. We have shown that rewriting is a closed relation on the set of terms, and conjectured that side effect free non orthogonal ATRS are confluent. Moreover, we have given many examples illustrating the power of ATRS.

## Further Work

Kris: TG-AT correspondence formalised, rewriting & matching still to go.

The term structure makes easy the definition of *rewriting strategies*, since it is simply a function which, given a term, returns an address, namely the next address where reduction must take place. This principle was used in earlier work on rewriting with addresses [BRL96] to define weak reduction of the lambda-calculus, and show how the classical strategies of functional programming languages, namely call-by-value, call-by-name, and call-by-need, can be defined. Since ATRS is a restriction on addressed rewriting we look forward to develop the notion of address-defined strategies in this framework. In particular it is interesting to notice that strategies can be built independently of the calculus with side effects it may be associated to, whereas usually, systems with side effects and their strategies are tightly coupled. Similarly it would be nice if the notion would generalise to include the "logical rules" used in [Ros93].

In the same spirit, we would like to give an account of types, and typing properties, of ATRS. This has already started to be studied in [LLL99], in the setting of a lambda-calculus with objects, defined as an extension of the weak lambda-calculus with explicit substitution and addresses of [BRL96], and of the Lambda-Calculus of Objects of [FHM94]. We have tried to prove that this calculus, implementing mutable states, had the property of *Type Soundness* (via *Subject Reduction*). We have not yet succeeded because of imprecisions in our specification of ATRS at that time, which has led to this more formal paper.

At last, we think that ATRS provide a good language (for both a computer and a computer scientist) to express graph rewriting rules with side effects, which could lead to a better understanding of the implementations of programming languages.

## Acknowledgements

The authors thank Zine-El-Abidine Benaissa [Ben97] who contributed to the elaboration of ATRS concepts, which helped to clarify many of the ideas of this paper. They also thank Luigi Liquori for useful dicussions and comments and for his delicious pasta.

## References


[AK96] Z.M. Ariola and J.W. Klop. Equational term graph rewriting. *Acta Informatica*, 1996.

[AKP97] Z.M. Ariola, J. W. Klop, and D. Plump. Confluent rewriting of bisimilar term graphs. In *Proc. Fourth Workshop on Expressiveness in Concurrency*, volume 7 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1997.

[Ben97] Z.-E.-A. Benaissa. *Les calculs de substitutions explicites comme fondement de l'implantation des langages fonctionnels*. PhD thesis, Université Henri Poincaré – Nancy 1, 1997.

[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*, number 1140 in Lecture Notes in Computer Science, 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 '87*, number 259 in Lecture Notes in Computer Science, pages 141–158. Springer-Verlag, 1987.

[Cou83] B. Courcelle. Fundamental Properties of Infinite Trees. *Theoretical Computer Science*, 25:95–169, 1983.

[CR91] William Clinger and Jonathan Rees, editors. Revised$^4$ report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 244–320. Elsevier, 1990.

[EJ98] R. Echahed and J.-C. Janodet. Admissible Graph Rewriting and Narrowing. In *Proceedings of JICSLP'98*, 1998.

[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.

[GKS89] J. R. W. Glauert, J. R. Kennaway, and M. R. Sleep. Final specification of Dactl. Report SYS–C88–11, University of East Anglia, Norwich, U.K., 1989.

[KKSdV93] J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. *An Infinitary Church-Rosser Property for Non-collapsing Orthogonal Term Rewriting Systems*, chapter 4. In Sleep et al. [SPvE93], 1993.

[KKSdV94] J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries. On the Adequacy of Graph Rewriting for Simulating Term Rewriting. *ACM TOPLAS*, 16(3):493–523, 1994.

[KKSdV95] J. R. Kennaway, J. W. Klop, M. R. Sleep, and F. J. de Vries. Transfinite reductions in orthogonal term rewriting systems. *Information and Computation*, 119(1):18–38, 1995.

[Klo92] J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 1, pages 1–116. Oxford University Press, 1992.

[LDLR99] F. Lang, D. Dougherty, P. Lescanne, and K.H. Rose. Addressed Term Rewriting Systems. RR 1999-30, Laboratoire de l'Informatique du Parallélisme, ENS Lyon, 1999. Avalaible by ftp at `www.ens-lyon.fr/pub/LIP/Rapports/RR/RR1999/RR1999-30.ps.Z`.

[LLL99] F. Lang, P. Lescanne, and L. Liquori. A Framework for Defining Object Calculi. In *Proc. of Formal Methods (FM'99)*, Lecture Notes in Computer Science. Springer Verlag, 1999. To appear.

[MT91] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87:209–220, 1991.

[PJ87] S. Peyton-Jones. *The implementation of functional programming languages*. Prentice Hall International, 1987.

[Plu] D. Plump. Term graph rewriting. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2. World Scientific. To appear.

[Plu93] D. Plump. Collapsed tree rewriting: Completeness, confluence and modularity. In *Proc. Conditional Term Rewriting Systems*, volume 656 of *Lecture Notes in Computer Science*, pages 97–112. Springer Verlag, 1993.

[Ros92] K. H. Rose. Explicit cyclic substitutions. In M. Rusinowitch and J.-L. Rémy, editors, *CTRS '92—3rd International Workshop on Conditional Term Rewriting Systems*, volume 656 of *LNCS*, pages 36–50, Pont-a-Mousson, France, July 1992. Springer-Verlag.

[Ros93] K.H. Rose. Graph-based operational semantics of a lazy functional language. In Sleep et al. [SPvE93], chapter 22, pages 303–316.

[Ros96] K. H. Rose. *Operational Reduction Models for Functional Programming Languages*. PhD thesis, DIKU, Universitetsparken 1, DK-2100 København Ø, 1996. DIKU report 96/1.

[SPvE93] M. R. Sleep, M. J. Plasmeijer, and M. C. D. J. van Eekelen, editors. *Term Graph Rewriting: Theory and Practice*. John Wiley, 1993.

[Tof90] M. Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

This article was typeset using the LaTeX macro package with the LLNCS2E class.