Addressed Term Rewriting Systems

Frédéric Lang, Daniel Dougherty, Pierre Lescanne, Kristoffer Rose

Laboratoire de l'Informatique du Parallélisme

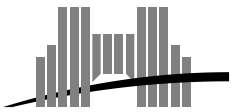
École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON n° 5668

Addressed Term Rewriting Systems

Frédéric Lang
Daniel Dougherty
Pierre Lescanne
Kristoffer Rose

June 3, 1999

Research Report N° RR 1999-30



École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`

Addressed Term Rewriting Systems

Frédéric Lang
Daniel Dougherty
Pierre Lescanne
Kristoffer Rose

June 3, 1999

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 *mutation*. 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 rewriting 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.

Keywords: Rewriting, Addresses, Mutation, Sharing, Cycles

Résumé

Nous proposons les *Systèmes de Réécritures de Termes à Adresses* (ATRS) comme solution au problème de définition d'un cadre simple et formel qui permet de rendre compte de calculs mettant en œuvre du *partage*, des *cycles*, et des *mutations*. Les ATRS sont caractérisés par la combinaison de trois aspects : ils sont basés sur une représentation des graphes par des termes, où l'induction structurelle est utile, ils mettent en œuvre une représentation minimale des structures de données cyclique par «pointeur arrière», et ils assurent une complexité bornée des pas de réécriture en éliminant les redirections de pointeurs implicites. Dans cet article, nous développons ceci, et nous montrons qu'il s'agit d'un compromis très utile entre la réécriture de graphes, moins abstraite, et la réécriture équationnelle, plus abstraite.

Mots-clés: Réécriture, Adresses, Mutation, Partage, Cycles

Contents

1 Introduction

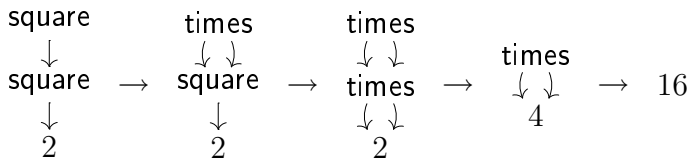
Term Rewriting Systems, TRS [DJ90, Klo92, BN98], are widely used to formalize, prototype, and verify software. However, they do not easily express some of the useful notions that one finds in software implementations, namely sharing, cycles, and mutation. In this introduction we explain each of these and the solution we propose before giving the plan for the rest of the paper.

1.1 Sharing

This is an essential notion to give an account of efficiency in the implementation of TRS. Sharing has mostly been studied in the context of obtaining efficient implementations of lazy

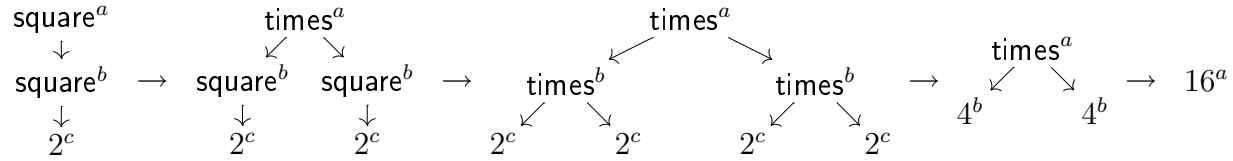
functional programming languages [PJ87, PvE93], and the initial studies of sharing in the context of *term graph rewriting systems*, TGRS [BVEG+87, Plu], were indeed motivated by this application.

Consider TRS rules such as $\mathsf{square}(x) \to \mathsf{times}(x, x)$. In principle the rule copies a subterm of the input but real implementations optimize this by only copying a *pointer* to the term. This not only saves memory but also makes it possible to *share future computations* on the referred subterm. We can compute, for example,



(assuming a rule to rewrite $\mathsf{times}(x, y)$ to the value $x \times y$ exists for each x and y).

Our solution is simply to use the original term representation but *annotate* it with an "address" per pointer target. For example, the above rewrite sequence can be represented as follows, using a, b, c, ... for addresses:

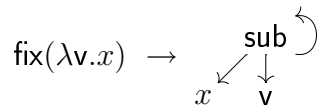


where each reduction affects all subterms with the same address *simultaneously*.

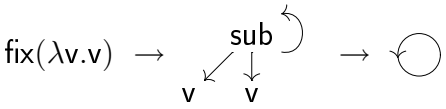
1.2 Cycles

These are essential when one wants to deal with infinite data structures in an efficient way as is the case, for example, for implementing recursive equations in lazy functional programming languages (as above), or logic programming language (Prolog, Lambda Prolog, *etc.*) without the "occur check".

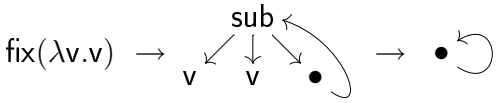
A typical TRS rule that we would like to implement with a cycle is the notorious fixed point unfolding rule $\mathsf{fix}(\lambda \mathsf{v}.x) \to \mathsf{sub}(x, \mathsf{v}, \mathsf{fix}(\lambda \mathsf{v}.x))$ implemented by



with v denoting a bound variable. However, given that substitution usually includes rules like $\mathsf{sub}(\mathsf{v}, \mathsf{v}, x) \to x$ for each v this makes it possible to rewrite



The result is called a *black hole* because it is a graph without any nodes and in particular no root (that can be used to observe it). The solution, also adopted by [AK96], is to require all "back-pointers" to use a special indirection node, written "•", replacing the above rewrite with

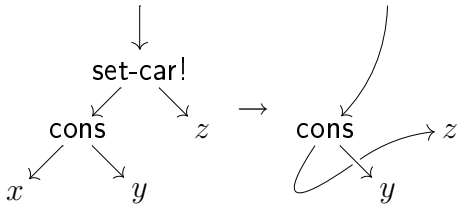


This also illustrates the view that cycles are "self-sharing" or, as it is often expressed, exhibit *vertical* sharing (in contrast to usual sharing called *horizontal*).

1.3 Mutation

This is a basic notion in many programming languages, for instance, in many object oriented programming languages, changing the value of a field in a shared object performs the side effect of "updating" in a non-local way that field for all other objects referring to it.

A classical example of mutation is the set-car! operator of the Scheme [CR91] language (originating from the rplaca primitive of Lisp). This operator has the graph rewriting rule



(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 mutation 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"*. Mutation was, 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" [Ros96b] suffer from having a very low-level notion of reduction which thus makes generic algebraic reasoning difficult.

1.4 Synthesis

Our main contribution 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

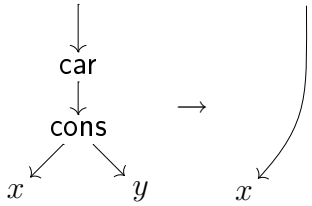
current node denotes the head of a list), and a vector of *successor addresses* (for instance, the vector containing the address of the next cell of the list) which make reference to the connected nodes, obtained through the state itself. In other words, a state can be seen as a cyclic term graph. However, 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.

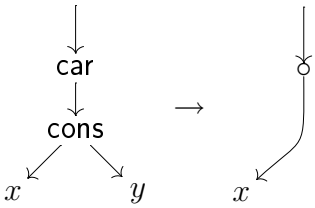
With respect to the first issue, we note that memory addresses are modeled by "node identity" in the TGRS model. Thus, it is difficult to model a "move" of data in memory since updated parts of a graph due to rewriting are new graph nodes, with new identities, *i.e.* new addresses. Hence, TGRS cannot model non-functional issues such as mutation. 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 mutation" in the sense that only the parts of a specification that actually *break* locality need 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 so-called *collapsing rules* because those normally require redirection to function. Take for example the rule for car of Scheme:



Applying this rule requires redirecting all *other* pointers to the car node to point to x otherwise we "loose sharing".

Our solution is to simply *forbid* collapsing rules and propose the use of indirection nodes as follows: example the rule for car of Scheme:



which prevents a need for redirection. The solution also resolves the "black hole" problem as a black hole is just a cyclic chain of indirection nodes, in our setting. Moreover, with explicit

indirection nodes to implement redirection we can claim that "counting rewrite steps" has a direct correspondance with the real complexity of implementation. In particular, the way indirection nodes are handled in a real implementation can be dealt with in our framework, which allows to reason formally on the different choices of implementation one would be faced by, and to express properties of the consequences of these choices. For instance, one can imagine to prove that a system is *optimal* in its treatment of indirection nodes by never creating chains of indirections of length greater than one. Philosophically, this can be seen as the natural continuation of the wish of explicitness we have already exploited in our group, while working on lambda-calculi with explicit substitution (see [Les94, Ros96a] for surveys).

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.* one is that it allows a very rigorous definition of rewriting, while providing an intuitive correspondance between graphs and first order terms.

1.5 Plan

After some preliminary definitions in Section 2, we define addressed terms in Section 3. In Section 4, we define matching and subterm replacement on addressed terms, notions which we then use in Section 5 to define rewriting of addressed terms. In Section 6, we detail the set-car! mutation example above as an ATRS whch leads to Section 7 where we characterize mutations 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 sets, functions denoted as sets, tuples, and term graphs.

Definition 1 (Sets, Functions, Tuples).

1. Sets are defined as usual: the empty set is denoted by $\varnothing$, union by $\cup$, and set membership by $\in$.

2. Using comma as in S, e where S is a set, denotes the *disjoint adjunction* of e to S, *i.e.*, requires that $e \notin S$. $\{e_1, \ldots, e_n\}$ denotes $S = \varnothing, e_1, e_2, \ldots, e_n$ which we will say is a set of *size* n, written $|S|$.

3. Let D, C be sets. A partial function Φ from D to C is denoted by a set of pairs from $D \times C$ written $d \mapsto c$, such that for all d there exists at most one c such that $d \mapsto c \in \Phi$. The predicate *fun* is defined on sets of pairs, where $fun(S)$ is satisfied if and only if S is a function. The *domain*, $dom(\Phi)$, resp. the *range*, $rng(\Phi)$, of the function Φ is the set $\{d \mid d \mapsto c \in \Phi\}$, resp. $\{c \mid d \mapsto c \in \Phi\}$. The application of a function Φ to an

argument $d \in dom(\Phi)$ is written $\Phi(d)$ and equals c, where $d \mapsto c \in \Phi$. The notation $\Psi = \Phi, d \mapsto c$ means the extension of Φ into Ψ such that $\Psi(x) = \Phi(x)$ if $x \in dom(\Phi)$ and $\Psi(d) = c$. It requires that $d \notin dom(\Phi)$.

4. For a set S, S^n denotes the set of n-tuples with elements from S, written $\langle s_1; \ldots; s_n \rangle$. Formally an n-tuple is a map from $\{1, \ldots, n\}$ to S so $\langle s_1; \ldots; s_n \rangle(i) = s_i$ if $1 \leq i \leq n$. $S^0 = \varnothing$ is written $\langle\rangle$, and S^* denotes the union of the n-tuples for all natural n.

Term graphs (as defined in the setting of TGRS [BVEG+87, Plu]) are rooted, directed, and oriented 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 Σ be a signature, with a function *arity* which associates a natural number to each symbol of Σ. A *term graph* over Σ is a quadruple $(N; r; lab; succ)$ where N is a finite set of *node identifiers*, $r \in N$ is a distinguished node (identifier) called the *root*, $lab : N \mapsto \Sigma$ maps each node to its *label*, $succ : N \mapsto N^*$ maps each node to its *successor tuple*, and it is required for all nodes $n \in N$ that $succ(n) \in N^{arity(lab(n))}$.

As will be shown, the structures we define in this paper, namely addressed terms, represent term graphs. However, the rewriting relation which we will define on them is different from rewriting of TGRS. In the following we sometimes denote term graphs simply as graphs.

3 Addressed Terms

Addressed terms, to be defined in this section, are terms decorated with addresses, which correspond to term graphs in a natural way. But not all terms decorated with addresses will represent graphs; we first define addressed *preterms*, then give admissibility conditions that a preterm must satisfy in order to be an addressed *term*.

3.1 Preterms

Definition 3 (Preterms).

1. Let Σ be a term signature as in Definition 2, and $\bullet$ a special symbol of arity zero. 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 $F \in \Sigma$ has arity $n \geq 0$, a is an address, and each t_i is an addressed preterm (inductively).

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

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

   It is not defined on variables.

3. The set of variables of a preterm t is denoted by $var(t)$ and defined in the obvious way.

4. The set of addresses of a preterm t denoted by $addr(t)$ is defined as follows:

$$addr\big(F^a(t_1,\ldots,t_m)\big) = \{a\} \cup \bigcup_{i=1}^{m} addr(t_i)$$
$$addr(\bullet^a) = \{a\}$$
$$addr(x) = \varnothing$$

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 subpreterms 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 subpreterms 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.

3.2 Cycles

To model cyclic memory structures with this "one occurrence per path" principle we would need infinitary trees as pursued by [KKSdV95] for which one needs not only infinity of terms but also of rewriting sequences to capture that all the infinite number of copies may need to be rewritten to reestablish a term that models a graph. This approach provides 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. Alternatively the condition is also similar to the definition of "rational trees" which are known to be implementable in a finite setting as cyclic graphs [Cou83] but with that approach we loose the possibility of observing the cycles.

Figure 1: Graph of $\mathsf{succ}(\mathsf{succ}(\cdots))$.

Our solution is the special symbol $\bullet$ called *back-pointer* used to give a representation of cyclic graphs as terms. Occurring as a leaf in a term, $\bullet^a$ is a reference to an ancestor node with address a. In an admissible term there will be exactly one such ancestor. For instance, the preterm $\mathsf{succ}^a(\bullet^a)$ denotes the cyclic graph in figure 1 (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 there is no ancestor node of $\bullet^b$ having address b.

Definition 4 (Vertical Admissibility). We call the *address context* of a subpreterm s in a preterm t the set of addresses of the ancestor nodes of s in t. 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 subpreterm 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.

3.3 Terms

The dual notion, *horizontal* admissibility, is the property that all subpreterms with the same address "beside" each other in the graph actually contain the same subterm. Defining this is a bit subtle so we first give some examples.

Figure 2: Graph of $\mathsf{plus}^a(\mathsf{succ}^b(\mathsf{zero}^c), \mathsf{succ}^b(\mathsf{zero}^c))$

Example 1. Given symbols zero of arity 0, succ of arity 1, plus and mult of arity 2. Examples of preterms with this signature are $t_1 = \mathsf{zero}^a$, $t_2 = \mathsf{succ}^a(x)$, $t_3 = \mathsf{plus}^c(\mathsf{zero}^a, \mathsf{zero}^b)$, $t_4 = \mathsf{plus}^a(\mathsf{succ}^b(\mathsf{zero}^c), \mathsf{succ}^b(\mathsf{zero}^c))$, $t_5 = \mathsf{plus}^a(\mathsf{zero}^a, \mathsf{succ}^b(\mathsf{zero}^a))$, $t_6 = \mathsf{mult}^b(\bullet^a, \mathsf{succ}^a(\mathsf{zero}^b))$, $t_7 = \mathsf{succ}^a(\mathsf{succ}^a(\bullet^a))$. Only the first four of these preterms denote graphs. Actually t_1, t_2, and t_3 denote trees, and t_4 is the graph in figure 2. The last three, t_5, t_6, and t_7, are *non-admissible*: t_5 because the address a is bound to both zero^a and t_5 itself; t_6 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_7 because a smaller representation of the same graph can be written, namely $\mathsf{succ}^a(\bullet^a)$ discussed above. (t_5, t_6, and t_7 are not even vertically admissible.)

In order to define exactly what it means to "contain" the same subpreterm we need to be able to compare subpreterms. This is done by defining the notion of *subterm* by extracting for each address the term corresponding to the subgraph rooted at that address. 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 for insisting on a term structure: we have to unfold whenever one needs to observe the full subterm where it contains a cycle, because we cannot know by local observation in an addressed term what a $\bullet$ stands for. For instance, take the term $t = \mathsf{succ}^a(\bullet^a)$. 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; unfolding 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.

Example 2. The addressed term $t = \mathsf{plus}^c(\mathsf{succ}^b(\mathsf{pred}^a(\bullet^b)), \mathsf{pred}^a(\mathsf{succ}^b(\bullet^a)))$ which represents the left hand side of the graph rewriting of Figure 3 in Section 6, has the subterms t, $\mathsf{succ}^b(\mathsf{pred}^a(\bullet^b))$, and $\mathsf{pred}^a(\mathsf{succ}^b(\bullet^a))$. One sees the following: (1) for every address of t there is one subterm rooted at that address, (2) every subterm is "extended" to describe a complete graph (which corresponds to the subgraph *reachable* from the root address), and (3) the subterms of a subterm is the set of subterms that are rooted at the addresses of the subterm. (1) will be confirmed by Proposition 8 while (2) and (3) will be confirmed by Proposition 13.

Informal definition 5 (Terms). *A preterm is* admissible *when it is vertically admissible, and when it is possible to build an* inventory*, which is a function mapping every address to the unique subterm at that address (using folding and unfolding adequately). Admissible preterms are called* terms*; for a term t the (unique and admissible) subterm "at" each address $a \in addr(t)$ is denoted $t\,@\,a$.*

This last definition is incomplete as presented here, since we do not say how to unfold to obtain the right subterms: a formal presentation is developed in the following. 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 since it then suffices to verify that the minimal representations of two terms at the same location are syntactically the same. Notice, however, this is not equivalent to saying that sub-*preterms* with the same address must be identical, as demonstrated by t of Example 2.

Example 3. The inventory of t_4 of Example 1 is

$$\{a \mapsto \mathsf{plus}^a(\mathsf{succ}^b(\mathsf{zero}^c), \mathsf{succ}^b(\mathsf{zero}^c)), b \mapsto \mathsf{succ}^b(\mathsf{zero}^c), c \mapsto \mathsf{zero}^c\}$$

The inventory of t in Example 2 is the function mapping $loc(s) \mapsto s$ for each listed subterm s.

To make the definition of terms precise we detail the mentioned notions of "folding" and "unfolding".

Definition 6 (Folding and Unfolding).

Folding. $fold(a)(t)$, where t is a preterm, and a an address, is the *folding of preterms located at a* in t, defined as follows:

$$\begin{aligned} fold(a)(x) &= x \\ fold(a)(\bullet^b) &= \bullet^b \\ fold(a)\big(F^b(t_1, \ldots, t_n)\big) &= \begin{cases} \bullet^b & \text{if } a = b \\ F^b\big(fold(a)(t_1), \ldots, fold(a)(t_n)\big) & \text{otherwise} \end{cases} \end{aligned}$$

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

$$\begin{aligned} unfold(s)(x) &= x \\ unfold(s)(\bullet^b) &= \begin{cases} s & \text{if } a = b \\ \bullet^b & \text{otherwise} \end{cases} \\ unfold(s)\big(F^b(t_1, \ldots, t_m)\big) &= F^b\big(unfold(s')(t_1), \ldots, unfold(s')(t_m)\big) \\ &\quad \text{where } s' = fold(b)(s) \end{aligned}$$

Example 4. Let t be the preterm $\mathsf{plus}^a(\mathsf{succ}^b(\bullet^a), \mathsf{succ}^b(\bullet^a))$, let s be the preterm $fold(b)(t)$, and let u be the preterm $unfold(w)(s)$ where $w = \mathsf{succ}^b(\mathsf{plus}^a(\bullet^b, \bullet^b))$. Then

$$\begin{aligned} s &= \mathsf{plus}^a\big(fold(b)(\mathsf{succ}^b(\bullet^a)), fold(b)(\mathsf{succ}^b(\bullet^a))\big) \\ &= \mathsf{plus}^a(\bullet^b, \bullet^b) \end{aligned}$$

$$\begin{aligned} u &= \mathsf{plus}^a(v, v) \\ &\qquad \text{where } v = unfold(fold(a)(\mathsf{succ}^b(\mathsf{plus}^a(\bullet^b, \bullet^b))))(\bullet^b) \\ &= \mathsf{plus}^a\big(unfold(\mathsf{succ}^b(\bullet^a))(\bullet^b), unfold(\mathsf{succ}^b(\bullet^a))(\bullet^b)\big) \\ &= \mathsf{plus}^a(\mathsf{succ}^b(\bullet^a), \mathsf{succ}^b(\bullet^a)) \\ &= t \end{aligned}$$

Thus addressed terms provide a representation of term graphs as first order finite terms with annotations. In this, they resemble term graphs in the "linear term" notation defined in [BVEG+87], with two major differences: (1) every node must have an address, and (2) if in a graph there exists 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 (for $n = 0$ this teaches us that with addressed terms we cannot represent "garbage", *i.e.*, subgraphs not reachable from the root).

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 *e.g.*, type checking on cyclic graphs easily possible. Second, as we want to use these addressed terms in a rewriting framework, rewriting strategies will play a main rôle and this presentation of cyclic terms will make the description of strategies easy and precise.

We now proceed with the formal definition of *addressed terms* (called *terms* for short), *i.e.*, preterms which denote graphs.

Definition 7 (Terms).

1. A preterm t is a (addressed) term (or is *admissible*) in the address context Γ, if there exists an inventory Δ *i.e.*, a partial function from addresses to terms, such that the expression $\Gamma \vdash t \rhd \Delta$ is derivable in the following system.

$$\frac{}{\Gamma \vdash x \rhd \varnothing}\ (TVar) \qquad \frac{}{\Gamma, a \vdash \bullet^a \rhd \{a \mapsto \bullet^a\}}\ (TBack)$$

$$\frac{\Gamma, a \vdash t_i \rhd \Delta_i}{\Gamma \vdash t \rhd \Delta}\ (TCons) \quad \text{where} \quad \begin{array}{l} \Delta = \bigcup_{i=1}^{m} (\mathit{unfold}(t) \circ \Delta_i) \cup \{a \mapsto t\}, \\ t = F^a(t_1, \ldots, t_m), \text{ and } \mathit{fun}(\Delta) \end{array}$$

   Note that it is clear from the rules, that if $\Gamma, a \vdash t_i \rhd \Delta_i$ succeeds, then t_i and Δ_i may only contain a as the address of $\bullet^a$ (provable by an easy induction on the structure of t). Therefore, the composition of $\mathit{unfold}(t)$ with Δ_i in the third rule is well defined.

2. A preterm t is a (addressed) term (or is admissible) if it is admissible in the empty context.

Note that if such a Δ exists for a given Γ, then it is unique and built deterministically, since the application of the rules is unambiguous and guided by the term syntax.

The rule (TVar) says that, in any address context, a variable is a term *i.e.*, denotes an open graph. The inventory of such a graph is obviously empty since it contains no address. (TBack) says that, for a back-pointer to denote a graph, its address has to belong to the address context. This means that, while going up in the context, one can find the node referred by this back-pointer. The inventory associated with such a back-pointer is obviously the one which binds it to the address a. Here, one has to understand that this back-pointer in the inventory will be unfolded while building the inventory. This can be seen in (TCons) rule, which says that a constructed preterm is admissible only if all its subterms are admissible, in a context now taking account of the address of the current term, and all this subterms give the same information on addresses they share. The built inventory is obtained as the union of all the sub-inventories, in which an unfold is performed on $\bullet^a$ so that it is now known that this back-pointer was a reference to the current term. Moreover, the current term is added to the inventory as the term at address a. Note that the address context Γ controls vertical admissibility, while the inventory Δ controls horizontal admissibility.

The inventory 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 later in this section). The difference is that in our approach the system defining the graph and its subgraphs is somewhat *stratified*. In other words, 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 inventory to be a function.

Proposition 8. *If* $\Gamma \vdash t \rhd \Delta$, *then* $addr(t) = dom(\Delta)$.

Proof. Let $\Gamma \vdash t \rhd \Delta$. We proceed by induction on the structure of t:

- if t is a variable, then $\Delta = \varnothing$ and $dom(\Delta) = \varnothing = addr(t)$;
- if $t = \bullet^a$, where $a \in \Gamma$, then $\Delta = \{a \mapsto \bullet^a\}$ and $dom(\Delta) = \{a\} = addr(\bullet^a)$;
- if $t = F^a(t_1, \ldots, t_m)$, where $\Gamma, a \vdash t_i \rhd \Delta_i$ for each t_i, then by induction hypothesis we have $dom(\Delta_i) = addr(t_i)$; therefore,

$$\begin{aligned}
dom(\Delta) &= dom\big(\bigcup_{i=1}^{m}(unfold(t) \circ \Delta_i) \cup \{a \mapsto t\}\big) \\
&= \bigcup_{i=1}^{m} dom(unfold(t) \circ \Delta_i) \cup dom(\{a \mapsto t\}) \\
&= \bigcup_{i=1}^{m} dom(\Delta_i) \cup \{a\} \\
&= \bigcup_{i=1}^{m} addr(t_i) \cup \{a\} && \text{by I.H.} \\
&= addr(t) && \text{by definition}
\end{aligned}$$

□

The following lemmas are structural lemmas which will be used many times in our proofs of theorems. Lemmas 9 and 10 say that *fold*() and *unfold*() can be used as inverses of each other. Lemmas 11 and 12 show that, under some conditions, folded and unfolded admissible terms (in some context) are admissible terms.

Lemma 9. *If* $unfold(s)(t)$ *is defined, and* $a = loc(s)$, *then* $\big(fold(a) \circ unfold(s)\big)(t) = t$.

Proof. By structural induction on t.

- $t = x$. The result is immediate;
- $t = \bullet^b$. There are two cases to consider:
  1. $b = a$. Then $fold(a)(unfold(s)(\bullet^a)) = fold(a)(s) = \bullet^a$.
  2. $b \neq a$. Then $fold(a)(unfold(s)(\bullet^b)) = \bullet^b$.
- $t = F^b(t_1, \ldots, t_m)$. From the condition of applicability of *unfold*() on t, we have $b \neq a$. Therefore, $unfold(s) \circ fold(a)$ may be distributed to the t_i, and then the I.H. applies. Note that the top address of s is obviously the same as the top address of $fold(a)(s)$.

□

Lemma 10. *Let* $\Gamma \vdash t \rhd \Delta$. *If* $a \notin dom(\Delta)$ *or* $\Delta(a) = s$ *then* $\big(unfold(s) \circ fold(a)\big)(t) = t$.

Proof. The case $a \notin \mathit{dom}(\Delta)$ is trivial, because this simply means that $a \notin \mathit{addr}(t)$ (see Proposition 8), hence $\mathit{unfold}(s)(\mathit{fold}(a)(t)) = \mathit{unfold}(s)(t) = t$. The case $\Delta(a) = s$ is proven by structural induction on t:

- $t = x$. There is neither such a nor s.
- $t = \bullet^a$. Note that the case $t = \bullet^b$ does not have to be considered since otherwise we would have $a \notin \mathit{dom}(\Delta)$. Note also that in this case, $a \in \Gamma$ and $s = \bullet^a$. Hence,

$$\mathit{unfold}(\bullet^a)(\mathit{fold}(a)(\bullet^a)) = \bullet^a.$$

- $t = F^b(t_1, \dots, t_m)$. We must consider two cases.
  1. $a = b$. Then $s = \Delta(a) = t$ and $\mathit{unfold}(t)(\mathit{fold}(a)(t)) = \mathit{unfold}(t)(\bullet^a) = t$.
  2. $a \neq b$. Then for each t_i where a occurs, we have $\Gamma, a \vdash t_i \rhd \Delta_i$, $\Delta_i(a) = s'$, and $\mathit{unfold}(t)(s') = s$. Therefore,

$$\mathit{fold}(b)(\mathit{unfold}(t)(s')) = \mathit{fold}(b)(s).$$

  Lemma 9 then gives $s' = \mathit{fold}(b)(s)$. Hence, by I.H.,

$$\mathit{unfold}(\mathit{fold}(b)(s))(\mathit{fold}(a)(t_i)) = t_i.$$

  Therefore,

$$\begin{aligned} \mathit{unfold}(s)(\mathit{fold}(a)(t)) &= F^b\big(\dots, \mathit{unfold}(\mathit{fold}(b)(s))(\mathit{fold}(a)(t_i)), \dots\big) \\ &= F^b(\dots, t_i, \dots) \\ &= t \end{aligned}$$

□

Lemma 11. *Let $a \notin \Gamma$. If $\Gamma \vdash t \rhd \Delta$ then $\Gamma, a \vdash \mathit{fold}(a)(t) \rhd \Delta'$ with $\Delta' \subseteq \mathit{fold}(a) \circ \Delta$.*

Proof. By structural induction on t.

- $t = x$. $\Gamma \vdash t \rhd \varnothing$, and the result is trivial.
- $t = \bullet^b$. Note that, since $a \notin \Gamma$, then $b \neq a$. $\Gamma \vdash \bullet^b \rhd \{b \mapsto \bullet^b\}$, and trivially

$$\Gamma, a \vdash \bullet^b \rhd \{b \mapsto \bullet^b\} = \{b \mapsto \mathit{fold}(a)(\bullet^b)\}.$$

- $t = F^b(t_1, \dots, t_m)$. We must consider two cases.
  1. $b = a$. Then $\mathit{fold}(a)(t) = \bullet^a$, hence

$$\Gamma, a \vdash \mathit{fold}(a)(t) \rhd \{a \mapsto \bullet^a\} = \mathit{fold}(a) \circ \{a \mapsto t\} \subseteq \mathit{fold}(a) \circ \Delta.$$

2. $b \neq a$. Then

$$\mathit{fold}(a)(t) = F^b\big(\mathit{fold}(a)(t_1), \ldots, \mathit{fold}(a)(t_m)\big).$$

By definition,

$$\Delta = \bigcup_{i=1}^{m} (\mathit{unfold}(t) \circ \Delta_i) \cup \{b \mapsto t\}.$$

By I.H.,

$$\Gamma, a \vdash \mathit{fold}(a)(t_i) \rhd \Delta_i' \subseteq \mathit{fold}(a) \circ \Delta_i.$$

Hence

$$\begin{aligned}\Gamma, a \vdash \mathit{fold}(a)(t) \rhd & \mathit{unfold}(\mathit{fold}(a)(t)) \circ \Delta_i' \cup \{b \mapsto \mathit{fold}(a)(t)\} \\ & \subseteq \bigcup_{i=1}^{m} (\mathit{unfold}(\mathit{fold}(a)(t)) \circ \mathit{fold}(a) \circ \Delta_i) \cup (\mathit{fold}(a) \circ \{b \mapsto t\}) \\ & = \mathit{fold}(a) \circ \big(\bigcup_{i=1}^{m} (\mathit{unfold}(t) \circ \Delta_i) \cup \{b \mapsto t\}\big) \\ & = \mathit{fold}(a) \circ \Delta\end{aligned}$$

□

Lemma 12. *If* $\Gamma, a \vdash t \rhd \Delta$, $\Gamma \vdash s \rhd \Phi$ *and* $\mathit{fun}\big((\mathit{unfold}(s) \circ \Delta) \cup \Phi\big)$ *then*

$$\Gamma \vdash \mathit{unfold}(s)(t) \rhd \Delta' \text{ with } \Delta' \subseteq (\mathit{unfold}(s) \circ \Delta) \cup \Phi.$$

Proof. By induction on the structure of t.

- $t = x$. Then $\Delta = \varnothing = \Delta' \subseteq \mathit{unfold}(s) \circ \varnothing = \varnothing$.
- $t = \bullet^a$. Then $\Delta = \{a \mapsto \bullet^a\}$.
  - If $\mathit{loc}(s) \neq a$, then $\Delta' = \Delta = \mathit{unfold}(s) \circ \Delta \subseteq (\mathit{unfold}(s) \circ \Delta) \cup \Phi$.
  - If $\mathit{loc}(s) = a$ then $\Delta' = \Phi \subseteq (\mathit{unfold}(s) \circ \Delta) \cup \Phi$.
- $t = F^b(t_1, \ldots, t_m)$.
  - $b = a$. Then $\mathit{unfold}(s)(t) = s$, and $\Gamma \vdash s \rhd \Phi \subseteq (\mathit{unfold}(s) \circ \Delta) \cup \Phi$.
  - $b \neq a$. Then

$$\mathit{unfold}(s)(t) = F^b\big(\mathit{unfold}(\mathit{fold}(b)(s))(t_1), \ldots, \mathit{unfold}(\mathit{fold}(b)(s))(t_m)\big).$$

By definition, for each t_i,

$$\Gamma, a, b \vdash t_i \rhd \Delta_i.$$

By Lemma 11, we have

$$\Gamma, b \vdash \mathit{fold}(b)(s) \rhd \Phi' \subseteq \mathit{fold}(b) \circ \Phi.$$

Since $\mathit{fun}\big((\mathit{unfold}(s) \circ \Delta) \cup \Phi\big)$ is satisfied, then for each i,

$$\mathit{fun}\big((\mathit{unfold}(s) \circ \Delta_i) \cup \Phi\big)$$

is satisfied. Hence, the I.H. applies on the t_i *i.e.*,

$$\Gamma, b \vdash \mathit{unfold}(\mathit{fold}(b)(s))(t_i) \rhd \Delta'_i \subseteq (\mathit{unfold}(\mathit{fold}(b)(s)) \circ \Delta_i) \cup (\mathit{fold}(b) \circ \Phi).$$

Therefore,

$$\begin{aligned}\Delta' &= \bigcup_{i=1}^{m} (\mathit{unfold}(\mathit{unfold}(s)(t)) \circ \Delta'_i) \cup \{b \mapsto \mathit{unfold}(s)(t)\} \\ &\subseteq \bigcup_{i=1}^{m} \mathit{unfold}(\mathit{unfold}(s)(t)) \circ \big((\mathit{unfold}(\mathit{fold}(b)(s)) \circ \Delta_i) \cup (\mathit{fold}(b) \circ \Phi)\big) \\ &\qquad \cup \{b \mapsto \mathit{unfold}(s)(t)\}\end{aligned}$$

Since $\Delta = \bigcup_{i=1}^{m} (\mathit{unfold}(t) \circ \Delta_i) \cup \{b \mapsto t\}$, then

$$\mathit{unfold}(t) \circ \Delta_i \subseteq \Delta$$

i.e., by Lemma 9,

$$\Delta_i \subseteq \mathit{fold}(b) \circ \Delta.$$

Hence,

$$\begin{aligned}\Delta' &\subseteq \mathit{unfold}(\mathit{unfold}(s)(t)) \circ \big((\mathit{unfold}(\mathit{fold}(b)(s)) \circ \mathit{fold}(b) \circ \Delta) \cup (\mathit{fold}(b) \circ \Phi)\big) \\ &\qquad \cup \{b \mapsto \mathit{unfold}(s)(t)\} \\ &= \big(\mathit{unfold}(\mathit{unfold}(s)(t)) \circ \mathit{fold}(b) \circ \big((\mathit{unfold}(s) \circ \Delta) \cup \Phi\big)\big) \\ &\qquad \cup \{b \mapsto \mathit{unfold}(s)(t)\} \\ &= (\mathit{unfold}(s) \circ \Delta) \cup \Phi \cup \{b \mapsto \mathit{unfold}(s)(t)\} \quad \text{by Lemma 10, see below} \\ &= (\mathit{unfold}(s) \circ \Delta) \cup \Phi \quad \text{since } \{b \mapsto t\} \subseteq \Delta\end{aligned}$$

The use of Lemma 10 is justified by the fact that:

* From the hypothesis, $(\mathit{unfold}(s) \circ \Delta) \cup \Phi$ is a function, and then, if defined, b has a unique image.
* $\Delta(b) = t$, hence $\big((\mathit{unfold}(s) \circ \Delta) \cup \Phi\big)(b) = \mathit{unfold}(s)(t)$.

□

We use the structural properties to establish that the constructed inventory is always defined where it should be.

Proposition 13. *If* $\Gamma \vdash t \rhd \Delta$, *and* $a \in \mathit{addr}(t)$, *then* $\Gamma \vdash \Delta(a) \rhd \Phi$ *with* $\Phi \subseteq \Delta$.

Proof. By induction on the structure of t.

- t is a variable. Then no such a exists in $\mathit{addr}(t)$ which is empty.

- $t = \bullet^b$. Then $a = b$, $\Delta(a) = \bullet^b$, and the result is given by an hypothesis.
- $t = F^b(t_1, \ldots, t_m)$. We have to consider two cases:
  1. $b = a$. Then $\Delta(a) = t$ and the result is given by an hypothesis.
  2. $b \neq a$. Then there exists some i such that $a \in addr(t_i)$. Since $\Gamma, b \vdash t_i \rhd \Delta_i$, we have by I.H.
  $$\Gamma, b \vdash \Delta_i(a) \rhd \Phi_i \subseteq \Delta_i.$$
  Since $\Delta = \bigcup_{i=1}^{m}(unfold(t) \circ \Delta_i) \cup \{b \mapsto t\}$ and $fun(\Delta)$, then
  $$\Delta(a) = unfold(t)(\Delta_i(a)).$$
  Since $\Phi_i \subseteq \Delta_i$, then
  $$unfold(t) \circ \Phi_i \subseteq unfold(t) \circ \Delta_i.$$
  Since $\Delta = \bigcup_{i=1}^{m}(unfold(t) \circ \Delta_i) \cup \{b \mapsto t\}$, then
  $$unfold(t) \circ \Phi_i \subseteq \Delta.$$
  Hence, since $fun(\Delta)$ is satisfied, so is $fun\big((unfold(t) \circ \Phi_i) \cup \Delta\big)$, and therefore Lemma 12 applies, giving
  $$\Gamma \vdash \Delta(a) = unfold(t)(\Delta_i(a)) \rhd \Phi \subseteq unfold(t) \circ \Phi_i \subseteq \Delta.$$

□

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 2.

Definition 14. Given a term t such that $\varnothing \vdash t \rhd \Delta$, and given $a \in addr(t)$, $t\,@\,a$ is $\Delta(a)$.

Corollary 15. *Let t be an admissible term and $a \in addr(t)$. $t\,@\,a$ exists, is unique, and is admissible.*

Proof. By Propositions 8 and 13. □

3.4 Relation to Term Graphs

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, see Definition 22). The last issue treated here is the formalisation of this correspondance, specifically with the term graph notion of Definition 2.

Definition 16. Given a ground addressed term t and a term graph g over the same signature Σ. We say they *correspond*, written $t \succcurlyeq g$, if we can derive $\vdash t \succcurlyeq g, \varnothing$ with the following inference system:

$$\frac{}{\vdash \bullet^a \succcurlyeq (\varnothing; a; \varnothing; \varnothing), \{a\}} \quad (GBack)$$

$$\frac{\vdash t_i \succcurlyeq (A_i; a_i; lab_i; succ_i), B_i \quad fun(lab) \quad fun(succ)}{\vdash F^a(t_1, \ldots, t_n) \succcurlyeq (A; a; lab; succ), B} \quad (GCons)$$

$$\begin{aligned} \text{where } A &= \textstyle\bigcup_{1 \le i \le n} A_i \cup \{a\}, \\ B &= \textstyle\bigcup_{1 \le i \le n} B_i \setminus \{a\}, \\ lab &= \textstyle\bigcup_{1 \le i \le n} lab_i \cup \{a \mapsto F\}, \\ \text{and } succ &= \textstyle\bigcup_{1 \le i \le n} succ_i \cup \{a \mapsto \langle loc(t_1), \ldots, loc(t_n)\rangle\} \end{aligned}$$

Theorem 17. *For any signature Σ the relation $\succcurlyeq$ is a total function from the ground addressed terms to term graphs (over that signature).*

Proof. For any derivation of $\vdash t \succcurlyeq (A; a; lab; succ), B$, the following properties are seen to be invariant (by an easy structural induction over derivation trees):

- $A \cap B = \varnothing$ and $A \cup B = addr(t)$.
- $fun(lab)$ and $fun(succ)$ with $dom(lab) = dom(succ) = A$.
- $\forall_{b \in B}\ t/b = \{(\bullet; \langle\rangle)\}$.
- $\forall_{a \in A}\ t/a = \{(lab(a); \mathsf{succ}(a))\}$.

where t/a returns a set of pairs of an element of Σ and a tuple of addresses, which denotes the set of *parallel flat subterms at address a*, defined by

$$\bullet^a/b = \begin{cases} \{(\bullet; \langle\rangle)\} & \text{if } a = b \\ \varnothing & \text{otherwise} \end{cases}$$

$$F^a(t_1, \ldots, t_n)/b = \begin{cases} \{(F; \langle loc(t_1); \ldots; loc(t_n)\rangle)\} & \text{if } a = b \\ \bigcup_{1 \le i \le n} (t_i/b) & \text{otherwise} \end{cases}$$

(this is needed since we do not know that t is a term). The theorem now follows from the observation that $B = \varnothing$ implies that

- t is admissible, and
- g is a term graph.

□

What this says is that we can map each addressed term into a unique term graph and furthermore that we can model all possible term graphs with addressed terms. This justifies our claim that addressed terms are exactly the preterms that model graphs.

Corollary 18. *Ground addressed terms (modulo address renaming) are isomorphic to term graphs (modulo node reidentification).*

Proof. Follows directly from theorem 17 by observing that the rules for $\succcurlyeq$ are deterministic both ways, and thus define a function, if the only observable of addresses is their equality or not. □

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 *matching* of addressed terms. The idea is that an addressed term t matches another (open) addressed term p, called the *pattern*, if there exists a graph homomorphism and a substitution from the graph denoted by p to the graph denoted by t. 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.

4.1 Matching

Informal definition 19 (Matching). *We say that a term t* matches *another term p if one can build a pair of functions $(\alpha;\sigma)$ such that α is a homomorphic function from addresses to addresses, σ is a function from variables to terms (defining the substitution), and these functions applied to p return t.*

The informal definition is made precise by introducing "prehomomorphisms" in a manner similar to what we did for terms.

Definition 20 (Matching).

1. Functions from addresses to addresses are called *prehomomorphisms*.[1] Injective prehomomorphisms are called *renamings*. Functions which bind variables to preterms are called *substitutions*. A *matching* is a pair of a prehomomorphism α and a substitution σ, denoted by $(\alpha;\sigma)$.

[1] Such functions serve to define homomorphisms on graphs, since they can add sharing in the graph, but never remove such sharing. However, the application of (the extension to terms of) a prehomomorphism to an addressed term does not always result in an addressed term, which is the reason why we do not call them *homomorphisms*. However, the prehomomorphisms computed by the match function will be real homomorphisms.

2. Let p and t be two terms. t *matches* p with *prematching* Π if there exists a matching $(\alpha;\sigma)$ such that $match(\Pi;p;t) = (\alpha;\sigma)$ can be obtained from the following system.

$$\frac{match(\Pi, a \mapsto b; p_i; t'_i) = (\alpha_i;\sigma_i) \quad fun(\alpha \cup \{a \mapsto b\}) \quad fun(\sigma)}{match(\Pi; F^a(p_1,\ldots,p_m); F^b(t_1,\ldots,t_m)) = (\alpha \cup \{a \mapsto b\};\sigma)} \ (MCons)$$

$$\text{where} \quad (\alpha;\sigma) = (\textstyle\bigcup_{i=1}^m \alpha_i; \bigcup_{i=1}^m \sigma_i),$$
$$t'_i = unfold(F^b(t_1,\ldots,t_m))(t_i)$$

$$\frac{loc(t) = b}{match(\Pi, a \mapsto b; \bullet^a; t) = (\{a \mapsto b\};\varnothing)} \ (MBack)$$

$$\frac{}{match(\Pi; x; t) = (\varnothing; \{x \mapsto t\})} \ (MVar)$$

Note that in the definition of $match(\Pi;p;t)$, the facts that p is a term in the context $dom(\Pi)$, and that t is a term in the empty context, are invariant. We say that t *matches* p if t matches p with an empty prematching.

3. A matching $(\alpha;\sigma)$ *fits* an admissible pattern p in an address context Γ if

   - $dom(\alpha) \supseteq \Gamma \cup addr(p)$ *i.e.*, every address of p or of its context has an image by α, and
   - $dom(\sigma) \supseteq var(p)$ *i.e.*, every variable of p has an image by σ, and
   - $\forall t \in rng(\sigma), \exists \Delta_t$ such that
     - $\alpha(\Gamma) \vdash t \rhd \Delta_t$, where $\alpha(\Gamma) = \{\alpha(a) \mid a \in \Gamma\}$, and
     - $fun(\bigcup_{t \in rng(\sigma)} \Delta_t)$ *i.e.*, the terms of $rng(\sigma)$ are, in some sense, *mutually coherent* in that they form together a coherent inventory.

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 the particularity of our matching procedure is that it may be *unfolded on demand* governed by the structure of the pattern. Here is an example.

Example 5. Matching the pattern $\mathsf{succ}^a(\mathsf{succ}^b(x))$ against the term $\mathsf{succ}^c(\bullet^c)$ works since, due to the shape of the pattern, one has to unfold the back-pointer in the term only once. The match is ($\{a \mapsto c, b \mapsto c\}$; $\{x \mapsto \mathsf{succ}^c(\bullet^c)\}$).

Example 6. 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

$$(\alpha;\sigma) = (\{a \mapsto c, b \mapsto d\}; \{x \mapsto \mathsf{succ}^d(\bullet^d)\}).$$

With this principle the pattern itself can also be cyclic.

Example 7. The term $\mathsf{succ}^c(\bullet^c)$ matches both the pattern $\mathsf{succ}^a(\mathsf{succ}^b(\bullet^b))$ and the pattern $\mathsf{succ}^a(\mathsf{succ}^b(\bullet^a))$ with matching $(\{a \mapsto c, b \mapsto c\};\varnothing)$.

Lemma 21. *If* $match(\varnothing; p; t) = (\alpha; \sigma)$ *then* $(\alpha; \sigma)$ *fits* p.

Proof. By induction on the structure of p.

- $p = x$. The fact is trivial.
- $p = \bullet^a$. The fact is trivial.
- $p = F^a(p_1, \ldots, p_m)$. Let t be $F^b(t_1, \ldots, t_m)$.
  By definition, $match(\Pi, a \mapsto b; p_i; unfold(t)(t_i)) = (\alpha_i; \sigma_i)$ hence, by I.H., $(\alpha_i; \sigma_i)$ fits p_i in $dom(\Pi), a$. Therefore, it is clear that $(\bigcup_{i=1}^m \alpha_i; \bigcup_{i=1}^m \sigma_i)$ fits p, in particular due to the definition of *addr* and *var*, and due to Lemma 11.

□

Definition 22 (Application of a matching).

1. Let $(\alpha; \sigma)$ be a matching fitting a term p. The application of $(\alpha; \sigma)$ to p, denoted by $sub(\alpha; \sigma)(p)$, is defined as follows:

$$\begin{aligned} sub(\alpha; \sigma)(\bullet^a) &= \bullet^{\alpha(a)} \\ sub(\alpha; \sigma)(x) &= \sigma(x) \\ sub(\alpha; \sigma)\big(F^a(p_1, \ldots, p_m)\big) &= F^{\alpha(a)}\big(\tau(p_1), \ldots, \tau(p_m)\big) \\ &\qquad \text{where } \tau = fold(\alpha(a)) \circ sub(\alpha; \sigma) \end{aligned}$$

2. We extend the application of matchings to inventories as follows:

$$sub(\alpha; \sigma)(\Delta) = \{\alpha(a) \mapsto sub(\alpha; \sigma)(p) \mid a \mapsto p \in \Delta\}$$

   Therefore, the application of the matching to the inventory is not the composition of the matching with the inventory. Note as a consequence that, contrary to a composition, the result of $sub(\alpha; \sigma)(\Delta)$ may be non functional even if Δ is a function. See however Lemma 24 which gives sufficient conditions on Δ and $(\alpha; \sigma)$ for $sub(\alpha; \sigma)(\Delta)$ to be a function.

3. Two terms p and t are *equivalent modulo renaming*, or *isomorphic*[2] if and only if there exists an α and a σ, both injective and such that $rng(\sigma) \subseteq \mathcal{X}$ (*i.e.*, α and σ are renamings) such that $sub(\alpha; \sigma)(p) = t$. It is clear that, indeed, renaming equivalence defined this way is an equivalence relation: symmetry is given by the fact that α and σ are invertible; reflexivity by the fact that the identity is reflexive (choose α as the identity on the addresses of p and σ as the identity on its variables); and transitivity by the fact that the composition of injective functions from a set to itself is an injective function of the same type.

[2]Because they define isomorphic graphs.

The application of a matching to a pattern preserves admissibility: the homomorphism α changes the addresses of the pattern while the substitution σ replaces the variables, but foldings are performed accordingly (exactly like unfoldings were performed while matching), whenever admissibility could be broken. For instance, a naive application of the matching to the pattern of Example 5 would return the non admissible preterm $\mathsf{succ}^c(\mathsf{succ}^c(\mathsf{succ}^c(\bullet^c)))$. The way we have defined it, it returns the admissible term $\mathsf{succ}^c(\bullet^c)$. This makes our matching quite similar to the matching employed for equational graph rewriting [AK96] but significantly more general than the matching of TGRS.

The following proposition expresses that the matching computed by the match function is indeed a function allowing to compute the term from the pattern.

Proposition 23. *If* $match(\Pi; p; t) = (\alpha; \sigma)$ *then* $sub(\alpha; \sigma)(p) = t$.

Proof. By structural induction on p.

- If $p = \bullet^a$ and $t = \bullet^b$ then $(\alpha; \sigma) = (\{a \mapsto b\}; \varnothing)$. Obviously, $sub(\{a \mapsto b\}; \varnothing)(\bullet^a) = \bullet^b$.
- If $p = x$ then $(\alpha; \sigma) = (\varnothing; \{x \mapsto t\})$, and obviously, $sub(\varnothing; \{x \mapsto t\})(x) = t$.
- If $p = F^a(p_1, \dots, p_m)$, then $t = F^b(t_1, \dots, t_m)$, and

  $$(\alpha; \sigma) = (\bigcup_{i=1}^{m} \alpha_i, a \mapsto b; \bigcup_{i=1}^{m} \sigma_i),$$

  where $match(\Pi, a \mapsto b; p_i; unfold(t)(t_i)) = (\alpha_i; \sigma_i)$. Therefore, by I.H.

  $$sub(\alpha; \sigma)(p_i) = unfold(t)(t_i).$$

  Then,

  $$\begin{aligned} sub(\alpha; \sigma)(p) &= sub(\alpha; \sigma)\big(F^a(p_1, \dots, p_m)\big) \\ &= F^b\big((fold(b) \circ sub(\alpha; \sigma))(p_1), \dots, (fold(b) \circ sub(\alpha; \sigma))(p_m)\big) \\ &= F^b\big(fold(b)(unfold(t)(t_1)), \dots, fold(b)(unfold(t)(t_m))\big) \\ &= F^b(t_1, \dots, t_m) \qquad \text{by Lemma 9} \end{aligned}$$

□

Lemma 24. *Let* t *be a term of inventory* Δ *in* Γ*, and* $(\alpha; \sigma)$ *be a matching fitting* t*.* $sub(\alpha; \sigma)(\Delta)$ *is a function.*

Proof. By induction on t.

- If t is a variable, then $\Delta = \varnothing = sub(\alpha; \sigma)(\Delta)$.
- If $t = \bullet^a$, then $\Delta = \{a \mapsto \bullet^a\}$. $sub(\alpha; \sigma)(\Delta) = \{b \mapsto \bullet^b\}$ where $\alpha(a) = b$, which is a function.

- If $t = F^a(t_1, \ldots, t_m)$, then $\Delta = \bigcup_{i=1}^{m}(\mathit{unfold}(t) \circ \Delta_i) \cup \{a \mapsto t\}$. Since $(\alpha;\sigma)$ fits t, it follows that $(\alpha;\sigma)$ fits each t_i. Hence, by I.H., we have

  $$\mathit{fun}\big(\mathit{sub}(\alpha;\sigma)(\Delta_i)\big).$$

  Since $\mathit{dom}(\mathit{unfold}(t) \circ \Delta_i) = \mathit{dom}(\Delta_i)$, it is clear that

  $$\mathit{fun}\big(\mathit{sub}(\alpha;\sigma)(\mathit{unfold}(t) \circ \Delta_i)\big)$$

  is also satisfied. It is hence also clear that

  $$\mathit{fun}\big(\mathit{sub}(\alpha;\sigma)(\bigcup_{i=1}^{m}(\mathit{unfold}(t) \circ \Delta_i) \cup \{a \mapsto t\})\big)$$

  is satisfied.

□

Lemma 25. $(\mathit{fold}(\alpha(a)) \circ \mathit{sub}(\alpha;\sigma) \circ \mathit{fold}(a))(t) = (\mathit{fold}(\alpha(a)) \circ \mathit{sub}(\alpha;\sigma))(t)$.

Proof. By induction on the structure of t.

- $t = x$. Both sides result in $\mathit{fold}(\alpha(a))(\sigma(x))$.
- $t = \bullet^b$. Both sides result in $\bullet^{\alpha(b)}$, even with $b = a$.
- $t = F^a(t_1, \ldots, t_m)$. The induction hypothesis applies on the t_i, which yields the intended result.

□

Lemma 26. *Let Γ be an address context and α a prehomomorphism such that $\mathit{dom}(\alpha) \subseteq \Gamma$. Let $\alpha(\Gamma)$ denote $\bigcup_{a\in\Gamma}\alpha(a)$, and ϕ be $\bigcirc_{a\in\Gamma}\mathit{fold}(a)$ where $\bigcirc$ denotes the composition of a set of commuting functions, which is obviously the case of fold. Consider the following:*

- $\Gamma \vdash p \rhd \Delta$,
- $(\alpha;\sigma)$ *fits* p *in* Γ,
- *each* $t \in \mathit{rng}(\sigma)$ *has an inventory* Δ_t *in* $\alpha(\Gamma)$.

Under these prerequisites, the following holds:

$$\alpha(\Gamma) \vdash (\phi \circ \mathit{sub}(\alpha;\sigma))(p) \rhd \Delta' \text{ where } D' \subseteq \phi \circ (\mathit{sub}(\alpha;\sigma)(\Delta) \cup \bigcup_{t\in\mathit{rng}(\sigma)} \Delta_t).$$

Proof. By induction on the structure of p. The existence of the Δ_t is by definition of fitting.

- $p = x$. Obviously, $\sigma(x) \in \mathit{rng}(\sigma)$. Hence by definition, and by Lemma 11,

  $$\alpha(\Gamma) \vdash (\phi \circ \sigma)(x) \rhd \Phi$$

  where $\Phi \subseteq \phi \circ \Delta_{\sigma(x)} \subseteq \phi \circ (\mathit{sub}(\alpha;\sigma)(\Delta) \cup \bigcup_{t\in\mathit{rng}(\sigma)} \Delta_t)$.

- $p = \bullet^a$. Then $(\phi \circ sub(\alpha;\sigma))(p) = \bullet^{\alpha(a)}$. By definition, Γ contains a, and $\Delta = \{a \mapsto \bullet^a\}$. Hence $\alpha(\Gamma)$ contains $\alpha(a)$ and $\Delta' = \{\alpha(a) \mapsto \bullet^{\alpha(a)}\} = (\phi \circ sub(\alpha;\sigma))(\Delta)$.

- $p = F^a(p_1, \ldots, p_m)$. Let ϕ' be $\phi \circ fold(\alpha(a))$ and Λ be $\bigcup_{t \in rng(\sigma)} \Delta_t$. Then,

$$\begin{aligned} t = (\phi \circ sub(\alpha;\sigma))(p) &= \phi(F^{\alpha(a)}\big(\ldots, (fold(\alpha(a)) \circ sub(\alpha;\sigma))(p_i), \ldots\big)) \\ &= F^{\alpha(a)}\big(\ldots, (\phi \circ fold(\alpha(a)) \circ sub(\alpha;\sigma))(p_i), \ldots\big) \\ &= F^{\alpha(a)}\big(\ldots, (\phi' \circ sub(\alpha;\sigma))(p_i), \ldots\big) \end{aligned}$$

  For each p_i we have $\Gamma, a \vdash p_i \rhd \Delta_i$. Moreover, it is clear that $(\alpha;\sigma)$ fits all the p_i, hence the I.H. applies, and

$$\alpha(\Gamma, a) \vdash (\phi' \circ sub(\alpha;\sigma))(p_i) \rhd \Delta_i'$$

  where $\Delta_i' \subseteq \phi' \circ (sub(\alpha;\sigma)(\Delta_i) \cup \Lambda)$. Hence,

$$\begin{aligned} \Delta' &= \bigcup_{i=1}^{m} (unfold(t) \circ \Delta_i') \cup \{\alpha(a) \mapsto t\} \quad \text{By definition} \\ &\subseteq \bigcup_{i=1}^{m} (unfold(t) \circ \phi' \circ (sub(\alpha;\sigma)(\Delta_i) \cup \Lambda)) \cup \{\alpha(a) \mapsto t\} \quad \text{By I.H.} \\ &= (unfold(t) \circ \phi' \circ (sub(\alpha;\sigma)(\bigcup_{i=1}^{m} (fold(a) \circ unfold(p) \circ \Delta_i)) \cup \Lambda) \cup \{\alpha(a) \mapsto t\} \\ &\quad \text{By Lemma 10} \\ &= (unfold(t) \circ \phi' \circ (sub(\alpha;\sigma)(\bigcup_{i=1}^{m} (unfold(p) \circ \Delta_i)) \cup \Lambda) \cup \{\alpha(a) \mapsto t\} \\ &\quad \text{By Lemma 25} \\ &= (unfold(t) \circ fold(\alpha(a)) \circ \phi \circ (sub(\alpha;\sigma)(\bigcup_{i=1}^{m} (unfold(p) \circ \Delta_i)) \cup \Lambda) \\ &\qquad \cup (\phi \circ sub(\alpha;\sigma))(\{a \mapsto p\}) \\ &= \phi \circ (sub(\alpha;\sigma)(\bigcup_{i=1}^{m} (unfold(p) \circ \Delta_i) \cup \Lambda)) \cup (\phi \circ sub(\alpha;\sigma))(\{a \mapsto p\}) \\ &\quad \text{By Lemma 9} \\ &= \phi \circ (sub(\alpha;\sigma)(\bigcup_{i=1}^{m} (unfold(p) \circ \Delta_i) \cup \{a \mapsto p\}) \cup \Lambda) \\ &= \phi \circ (sub(\alpha;\sigma)(\Delta) \cup \Lambda) \quad \text{By definition of } \Delta \end{aligned}$$

□

4.2 Replacement

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.

Informal definition 27 (Replacement). *A* replacement *is characterized by a function* Φ *from addresses to terms such that* $loc(\Phi(a)) = a$*. Applying it to a term* t *results in a term* t' *such that* t' *has the same location as* t*, and for all address* $a \in addr(t)$*:* $t' @ a = \Phi(a)$ *if* $a \in dom(\Phi)$*, otherwise* $t' @ a = (t @ a)'$*, where* $(t @ a)'$ *is the application of the replacement to* $t @ a$*.*

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

Example 8. Let t be $\mathsf{plus}^a(\mathsf{plus}^b(2^c, 2^c), \mathsf{plus}^b(2^c, 2^c))$ and Δ be $\{b \mapsto 4^b\}$. The replacement of Δ in t gives $\mathsf{plus}^a(4^b, 4^b)$.

Here is the formal definition of a replacement.

Definition 28 (Replacement). In the following, a *replacement* is characterized by a function Φ from addresses to terms such that $loc(\Phi(a)) = a$. $repl(\Phi)(t)$ is defined on terms as follows:

$$repl(\Phi)(x) = x$$

$$repl(\Phi)(\bullet^a) = \begin{cases} \Phi(a) & \text{if } a \in dom(\Phi) \\ \bullet^a & \text{otherwise,} \end{cases}$$

$$repl(\Phi)\big(F^a(t_1, \dots, t_m)\big) = \begin{cases} \Phi(a) & \text{if } a \in dom(\Phi) \\ F^a\big(repl(\Phi')(t_1), \dots, repl(\Phi')(t_m)\big) & \text{otherwise,} \\ \quad \text{where } \Phi' = fold(a) \circ \Phi & \end{cases}$$

Lemma 29. *If* $\Gamma \vdash t \rhd \Delta$ *and* $\Gamma \vdash u \rhd \Phi$*, then*

$$\Gamma \vdash repl(\Phi)(t) \rhd \Delta' \text{ where } \Delta' \subseteq (repl(\Phi) \circ \Delta) \cup \Phi.$$

Proof. By structural induction on t.

- $t = x$. This case is trivial since $\Delta = \varnothing = \Delta'$.
- $t = \bullet^a$.
  - First case: $a \in dom(\Phi)$, then $repl(\Phi)(t) = \Phi(a)$. By Proposition 13, we thus have that $\Gamma \vdash repl(\Phi)(t) \rhd \Phi' \subseteq \Phi$.

– Second case: $a \notin dom(\Phi)$, then $repl(\Phi)(t) = \bullet^a$ and

$$\Gamma \vdash repl(\Phi)(t) \rhd \Delta = repl(\Phi) \circ \Delta.$$

- $t = F^a(t_1, \ldots, t_m)$.
  - First case: $a \in dom(\Phi)$, then this case is similar to the first case of $\bullet^a$.
  - Second case: $a \notin dom(\Phi)$, then

$$repl(\Phi)(t) = F^a\big(repl(fold(a) \circ \Phi)(t_1), \ldots, repl(fold(a) \circ \Phi)(t_m)\big).$$

By definition of admissibility, each t_i satisfies $\Gamma, a \vdash t_i \rhd \Delta_i$. By Lemma 11, each element u of $rng(\Phi)$ satisfies

$$\Gamma, a \vdash fold(a)(u) \rhd \Phi' \subseteq fold(a) \circ \Phi.$$

Therefore, by I.H., each t_i satisfies

$$\Gamma, a \vdash repl(fold(a) \circ \Phi)(t_i) \rhd \Delta_i' \subseteq (repl(fold(a) \circ \Phi) \circ \Delta_i) \cup (fold(a) \circ \Phi).$$

Hence,

$$\begin{aligned}
\Delta' &= \bigcup_{i=1}^{m} unfold(repl(\Phi)(t)) \circ \Delta_i' \cup \{a \mapsto repl(\Phi)(t)\} \\
&\subseteq \bigcup_{i=1}^{m} unfold(repl(\Phi)(t)) \circ \big((repl(fold(a) \circ \Phi) \circ \Delta_i) \cup (fold(a) \circ \Phi)\big) \\
&\qquad \cup \{a \mapsto repl(\Phi)(t)\}
\end{aligned}$$

Since $\Delta = \bigcup_{i=1}^{m} (unfold(t) \circ \Delta_i) \cup \{a \mapsto t\}$, then $unfold(t) \circ \Delta_i \subseteq \Delta$, and by Lemma 9,

$$\Delta_i = fold(a) \circ unfold(t) \circ \Delta_i \subseteq fold(a) \circ \Delta.$$

Hence,

$$\begin{aligned}
\Delta' &\subseteq (unfold(repl(\Phi)(t)) \circ repl(fold(a) \circ \Phi) \circ fold(a) \circ \Delta) \\
&\qquad \cup (unfold(repl(\Phi)(t)) \circ fold(a) \circ \Phi) \cup \{a \mapsto repl(\Phi)(t)\} \\
&= (unfold(repl(\Phi)(t)) \circ fold(a) \circ repl(\Phi) \circ \Delta) \\
&\qquad \cup (unfold(repl(\Phi)(t)) \circ fold(a) \circ \Phi) \cup \{a \mapsto repl(\Phi)(t)\} \\
&= (repl(\Phi) \circ \Delta) \cup \Phi \cup \{a \mapsto repl(\Phi)(t)\} \quad \text{by Lemma 10, see below} \\
&= (repl(\Phi) \circ \Delta) \cup \Phi \quad \text{since } a \mapsto t \in \Delta \text{ by definition}
\end{aligned}$$

The use of Lemma 10 is justified by the following considerations:

1. $(repl(\Phi) \circ \Delta)(a) = repl(\Phi)(t)$, since $\Delta(a) = t$.
2. Either $\Phi(a) = repl(\Phi)(t)$, or $n \notin dom(\Phi)$. Indeed, otherwise Δ' is not a function since $dom\big((repl(\Phi) \circ \Delta) \cup \Phi\big) = dom(\Delta')$.

□

5 Addressed Term Rewriting Systems

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

Definition 30 (Rewriting Rule).

1. An addressed rewriting rule over Σ is a pair of addressed terms (l, r) over Σ, written $l \to r$, such that $loc(l) = loc(r)$ (same top address, therefore l and r are not variables), and $var(r) \subseteq var(l)$ (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 $a \in addr(t)$ 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 cyclic.

Example 9. We do not permit simple variables on any side so "collapsing" or "projection" rules such as $\mathsf{car}^a(\mathsf{cons}^b(x, y)) \to x$ are not possible. This can be fixed by adding to the signature a unary function symbol, intuitively seen as an *indirection node* and written $\lceil\_\rceil$. (This may seem like a messy solution but it is, in fact, the technique used by all real implementations to avoid having to search the memory for pointers to redirect.) With that we can express the above rule as $\mathsf{car}^a(\mathsf{cons}^b(x, y)) \to \lceil x \rceil^a$. This means, of course, that we must write rules like $\mathsf{car}^a(\lceil x \rceil^b) \to \mathsf{car}^a(x)$ to *redirect* the indirection nodes as needed.

A slighty more complex case is the TRS rule $\mathsf{proj}(\mathsf{succ}(x)) \to \mathsf{succ}(x)$. It would correspond to $\mathsf{proj}^a(\mathsf{succ}^b(x)) \to \mathsf{succ}^b(x)$ except this is not an addressed rewriting rule. Here we have to choose between two solutions, either indirection or copy: it may be implemented either by the rule $\mathsf{proj}^a(\mathsf{succ}^b(x)) \to \mathsf{succ}^a(x)$, which expresses the update of address a with a *copy* of the label and successor addresses of the node located at address b, or by the rule $\mathsf{proj}^a(\mathsf{succ}^b(x)) \to \lceil \mathsf{succ}^b(x) \rceil^a$, 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.

The remaining issue is how we create *new* nodes. The technique we have chosen is illustrated by the following example.

Example 10. $\mathsf{mult}^a(\mathsf{succ}^b(x), y) \to \mathsf{plus}^a(y, \mathsf{mult}^c(x, y))$ is an addressed rewriting rule. By the non-linearity of the right-hand side it expresses some sharing (y must be seen as a pointer). It furthermore creates a new node in the graph whose address is denoted by c and which was not present in the left-hand side.

The key definition formalizing "new" nodes is the notion of *fresh renaming* given next. It expresses the importance of giving these new nodes addresses which are not already used.

Definition 31 (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 $l \to r$ with respect to t is a renaming denoted by α_{fresh} such that $dom(\alpha_{fresh}) = addr(r) \setminus addr(l)$, *i.e.*, the fresh renaming renames each newly introduced address to avoid capture, and $rng(\alpha_{fresh}) \cap addr(t) = \varnothing$, *i.e.*, the chosen addresses are *fresh*, which means not present in t.

Lemma 32. *Let $l \to r$ be a rule. If $(\alpha; \sigma)$ fits l* w.r.t. *a term t, and α_{fresh} is a fresh renaming for $l \to r$* w.r.t. *t, then $(\alpha \cup \alpha_{fresh}; \sigma)$ fits r* w.r.t. *t.*

Proof. We check that the three conditions of Definition 20 (item 3) are satisfied:

1. By hypothesis, $dom(\alpha) \supseteq addr(l)$, and by definition, $dom(\alpha_{fresh}) = addr(r) \setminus addr(l)$. Hence,

$$dom(\alpha \cup \alpha_{fresh}) \supseteq addr(l) \cup (addr(r) \setminus addr(l)) = addr(l) \cup addr(r) \supseteq addr(r).$$

2. By hypothesis, $dom(\sigma) \supseteq var(l)$, and by definition, $var(l) \supset var(r)$, hence

$$dom(\sigma) \supseteq var(r).$$

3. The last condition remains satisfied from the hypothesis, since α_{fresh} does not rename addresses of t. Hence, for all Γ context in t, we have $(\alpha \cup \alpha_{fresh})(\Gamma) = \alpha(\Gamma)$.

□

Corollary 33. *Let $l \to r$ be a rule, and t a term. If $(\alpha; \sigma)$ fits l, and α_{fresh} is a fresh renaming for $l \to r$* w.r.t. *t, then $sub(\alpha \cup \alpha_{fresh}; \sigma)(r)$ is a term.*

Proof. Direct from Lemmas 26 and 32. □

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

Definition 34 (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 *i.e.*, there is a matching $(\alpha; \sigma)$, obtained by deriving

$$match(\varnothing; l; t \,@\, a) = (\alpha; \sigma)$$

2. Compute α_{fresh}, a fresh renaming for $l \to r$ with respect to t.

3. Compute $s = sub(\alpha \cup \alpha_{fresh}; \sigma)(r)$, and Φ such that $\varnothing \vdash s \rhd \Phi$ (which exists, according to Corollary 33).

4. The result t' of rewriting t by rule $l \to r$ at address a is $repl(\Phi)(t)$. We write the reduction $t \to t'$.

Addressed term rewriting is performed in a similar way as term rewriting: first find a redex, then remember 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. This corresponds to *memory allocation* in implementations.

Definition 35 (Addressed Term Rewriting Systems). An Addressed Term Rewriting System (ATRS) over a signature Σ is a set of addressed rewriting rules over Σ.

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

Corollary 36. *Let t be a term. If $t \to t'$ then t' is a term.*

Proof. Direct from Corollary 33. □

6 Examples

This section gives simple examples of ATRS, and of derivations.

6.1 Simple ATRS

Example 11. Consider the term $t = \mathsf{plus}^c(\mathsf{succ}^b(\mathsf{plus}^a(1^d, \bullet^b)), \mathsf{plus}^a(1^d, \mathsf{succ}^b(\bullet^a)))$, and let the rule $l \to r$ be $\mathsf{plus}^a(1^b, x) \to \mathsf{succ}^a(x)$. We want to reduce t at address a, following Definition 34. We first compute the matching

$$(\alpha; \sigma) = (\{a \mapsto a, b \mapsto d\}; \{x \mapsto \mathsf{succ}^b(\mathsf{plus}^a(1^d, \bullet^b))\}).$$

Since r does not contain fresh variables, then $\alpha_{fresh} = \varnothing$.

$$s = sub(\alpha; \sigma)(\mathsf{succ}^a(x)) = \mathsf{succ}^a(\mathsf{succ}^b(\bullet^a)).$$

The inventory of s is $\Delta = \{a \mapsto \mathsf{succ}^a(\mathsf{succ}^b(\bullet^a)), b \mapsto \mathsf{succ}^b(\mathsf{succ}^a(\bullet^b))\}$. The result of rewriting is:

$$repl(\Delta)(t) = \mathsf{plus}^c(\mathsf{succ}^b(\mathsf{succ}^a(\bullet^b)), \mathsf{succ}^a(\mathsf{succ}^b(\bullet^a)))$$

The equivalent, drawn as graphs, is shown in Figure 3.

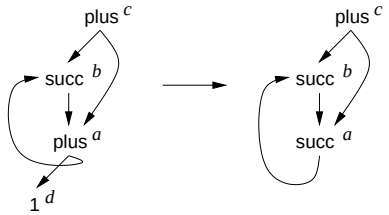


Figure 3: Graph rewriting step equivalent to addressed term rewriting step of Example 11.

Example 12. Consider the term $t = \lceil \mathsf{succ}^b(\bullet^a) \rceil^a$, and the rule

$$\mathsf{succ}^a(\lceil x \rceil^b) \to \mathsf{succ}^a(x)$$

which simply expresses the removal of an indirection node from a node labelled by succ. We want to reduce t at address b. Note that there is a redex at address b even though it does not appear syntactically. Indeed, $t @ b = \mathsf{succ}^b(\lceil \bullet^b \rceil^a)$, and

$$match(\varnothing; \mathsf{succ}^a(\lceil x \rceil^b); \mathsf{succ}^b(\lceil \bullet^b \rceil^a)) = (\{a \mapsto b, b \mapsto a\}; \{x \mapsto \mathsf{succ}^b(\lceil \bullet^b \rceil^a)\}).$$

We denote this matching by $(\alpha; \sigma)$. Note that $\alpha_{fresh} = \varnothing$.

$$s = sub(\alpha; \sigma)(\mathsf{succ}^a(x)) = \mathsf{succ}^b(\bullet^b).$$

The result of rewriting t is

$$repl(\{b \mapsto \mathsf{succ}^b(\bullet^b)\})(t) = \lceil (\mathsf{succ}^b(\bullet^b)) \rceil^a$$

The equivalent, drawn as graphs, is shown in Figure 4.

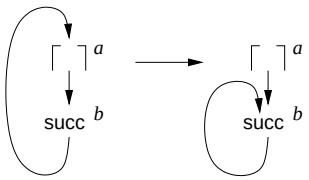


Figure 4: Graph rewriting step equivalent to addressed term rewriting step of Example 12.

6.2 Mutable Lists

The following shows how mutation is performed, by taking the famous example of set-car!.

Example 13. Consider the symbols nil of arity 0, and cons of arity 2, the constructors of lists. car, of arity 1 is the operator which returns the first element of a list, and set-car!, of arity 2, the operator which mutates a new value into the first element of the list. We also have an infinite number of symbols denoted by $1, 2, \ldots, n, \ldots$ of arity 0, and the operation plus of arity 2. The rules for the definition of car and set-car! are the following:

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

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

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

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

(We do not give the definition of plus since it is not the purpose of this example – suppose it is built-in.) 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 $\mathsf{set\text{-}car!}$, the reduction proceeds as follows:

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

On the other hand, starting by applying the rule (Car), the computation proceeds as follows:

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

The rules can even be used to convert horizontal sharing to vertical sharing:

$$\mathsf{set\text{-}car!}^a(\mathsf{cons}^b(\mathsf{nil}^c, \mathsf{nil}^d), \mathsf{cons}^b(\mathsf{nil}^c, \mathsf{nil}^d)) \;\to\; \lceil \mathsf{cons}^b(\bullet^b, \mathsf{nil}^d) \rceil^a$$

where the $\bullet$ appears by folding.

This example shows that ATRS may produce mutation. It is essential to be able to determine easily whether a system has mutation or not. For instance, a mutation 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 perform no mutation.

7 Mutation, Overlap, and Confluence

As shown in Example 13, non-overlapping ATRS may be non-confluent since they allow mutation. In this section, we characterize a subclass of the ATRS which are confluent. We call them *mutation-free non-overlapping ATRS*.

Definition 37 (Overlap). Let $\tau_1 = l_1 \to r_1$ and $\tau_2 = l_2 \to r_2$ be two rules, not necessarily distinct. τ_1 *overlaps* τ_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 with the additional requirement that if $\tau_1 = \tau_2$ then $a \neq loc(l_1)$ (to avoid trivial self-overlaps). A system in which no rules overlap is called *non-overlapping*.

Definition 38 (Mutation). A rule $l \to r$ performs *mutation* if and only if there exists an address a in $(addr(l) \cap addr(r)) \setminus \{loc(l)\}$ such that $l @ a$ is syntactically different from $r @ a$. A rule or an ATRS which does not perform mutation is called *mutation-free*.

The name "mutation-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 perform mutation.

Example 14. The rule (Set-Car) of Example 13 clearly performs a mutation, since $l @ b = \mathsf{cons}^b(y, z)$ while $r @ b = \mathsf{cons}^b(x, z)$. All the other rules are mutation-free. The system is non-overlapping.

Note that in ATRS, as usual in TGRS, identifying overlapping rules is a bit more tricky than in TRS, as shows the following example.

Example 15. Consider the following mutation-free system:

$$\mathsf{plus}^a(x, x) \to \mathsf{even}^a \tag{R1}$$

$$\mathsf{plus}^a(x, \mathsf{succ}^b(x)) \to \mathsf{odd}^a \tag{R2}$$

One would be tempted to say that (R1, R2) are non-overlapping, since the rules of the TRS obtained by erasing the addresses are clearly non-overlapping. However, the term $\mathsf{plus}^d(\mathsf{succ}^c(\bullet^c), \mathsf{succ}^c(\bullet^c))$ matches both left hand sides. (R1, R2) hence clearly overlap. (Indeed, all natural number is either odd or even, but this is no more the case with ordinals.)

Similarly, one has to be careful when checking whether a rule performs mutation, as shows the following example.

Example 16. Let (R3) be the rule

$$\mathsf{next}^a(\mathsf{succ}^b(\bullet^a)) \to \mathsf{succ}^a(\mathsf{succ}^b(\bullet^a)) \tag{R3}$$

Apparently, (R3) seems to be a mutation-free rule, because its left and right hand sides have the same sub-preterm at the same location, namely $\mathsf{succ}^b(\bullet^a)$. However, one has to remember the definition of term at a given location: In fact, there actually *is* a mutation, because one has $l @ b = \mathsf{succ}^b(\mathsf{next}^a(\bullet^b))$ and $r @ b = \mathsf{succ}^b(\mathsf{succ}^a(\bullet^b))$, which are clearly distinct. On the other hand, the following rule (R4), which creates a new node in the graph, is mutation free.

$$\mathsf{next}^a(\mathsf{succ}^b(\bullet^a)) \to \mathsf{succ}^a(\mathsf{succ}^c(\bullet^a)) \tag{R4}$$

Theorem 39. *Mutation-free non-overlapping ATRS are confluent (modulo address renaming).*

Proof outline. The conditions of mutation-freeness and non-overlapping are strong enough to ensure that the reduction of one redex may only eventually discard the other redex (in which case the fact it is discarded after being reduced or not is unimportant *w.r.t.* confluence), but never modify the part of the term matched by the other redex. □

8 Conclusion

We have defined in this paper the Addressed Term Rewriting Systems, giving an account of essential features of implementations. The main difference with TGRS is that implicit redirections are not allowed in ATRS due to the constraint that both members of a rule must be located at the same address. This allows us to reason formally about implementation choices such as indirection *vs* copy. Moreover, ATRS give an account of mutation. We have shown that rewriting is a closed relation on the set of terms, and that mutation-free non-overlapping ATRS are confluent. Moreover, we have given many examples illustrating the power of ATRS.

8.1 Further Work

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 mutation it may be associated to, whereas usually, systems with mutation 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 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 specific paper. It will be interesting to see if we get a notion of type similar to the one for TGRS [Bar95].

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

8.2 Acknowledgements

The second author[3] is grateful for support from the *École Normale Supérieure de Lyon* during his visit at the *Laboratoire de l'Informatique du Parallélisme* in the spring of 1999. The authors thank Zine-El-Abidine Benaissa [Ben97] who contributed to the elaboration of

[3]Department of Mathematics, Wesleyan University Middletown, CT 06459 USA ddougherty@wesleyan.edu

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.

[Bar95] E. Barendsen. *Types and Computations in Lambda Calculi and Graph Rewrite Systems*. PhD thesis, Katholieke Universiteit Nijmegen, 1995.

[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] W. Clinger and J. 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.

[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.

[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.

[Les94] P. Lescanne. From $\lambda\sigma$ to $\lambda\upsilon$, a journey through calculi of explicit substitutions. In *Proceedings of the 21st conference on Principles of Programming Languages*, pages 60–69, 1994.

[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.

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

[Ros93] K. H. Rose. Graph-based operational semantics of a lazy functional language. In M. R. Sleep, M. J. Plasmeijer, and M. C. D. J. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 22, pages 303–316. John Wiley, 1993.

[Ros96a] K. H. Rose. Explicit substitution – tutorial & survey. Lecture Series LS–96–3, BRICS, Dept. of Computer Science, University of Aarhus, Ny Munkegade (bld.540), 8000 Aarhus C, Denmark, September 1996.

[Ros96b] K. H. Rose. *Operational Reduction Models for Functional Programming Languages.* PhD thesis, DIKU, Universitetsparken 1, DK-2100 København Ø, 1996. DIKU report 96/1.

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