Director Strings Revisited

A generic approach to the efficient representation of free variables in higher-order rewriting

François-Régis Sinot[*]

LIX, École Polytechnique, 91128 Palaiseau, France
frs@lix.polytechnique.fr

Abstract

The representation of free variables is crucial for the efficiency of the implementation of various kinds of rewrite systems. We give an innovative, although very natural, representation of variables abstract enough to fit in many different frameworks and more satisfactory from an operational perspective than usual representations. This work also provides a generalisation of director strings for the λ-calculus [17].

1 Introduction

Many fields in computer science make use of a notion of variable and substitution. Such fields include (all sorts of) rewriting, the λ-calculus, programming languages, natural language processing, etc. Substitution is often the *éminence grise*, quoting Abadi et al. [1], of these formalisms because the intuitive meaning of it (replace a variable by "something") often hides potential problems, like name capture in presence of binders, and some intrinsic computational cost. To make the process of substitution *explicit* is then a natural alternative in order to put the computational cost of the substitution in the system itself [1]. But explicit substitutions are only half of the way and some complexity may still be hidden in the rules. In particular the *representation* of variables is of a crucial importance for the efficiency of the process of substitution as we will show in Section 2, and this is the topic we address here. More precisely, we propose a representation which is both efficient in the sense that it allows the usual operations in constant time and very natural since it is the dual (in a sense that will be made precise) of the usual representation. The first premises of this idea are due to Kennaway and Sleep [13], where director strings were used for combinatory reduction. This idea was then recently extended to closed reduction (cf. [10]) in [9] and to the full λ-calculus in [17, 11].

[*]Projet Logical, Pôle Commun de Recherche en Informatique du plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

Overview. The question we address will be introduced more precisely in Section 2, then Section 3 will be devoted to presenting the notion of *directors*, which is the representation of free variables we propose. The reduction then has to take these directors into account, we will see how in the cases of term rewrite systems (TRSs, Section 4) and combinatory reduction systems (CRSs, Section 5). Some more involved examples will be developed in Sections 6 and 7 and we will conclude in Section 8.

2 Position of the Problem

Term rewriting systems, higher-order rewriting systems, the λ-calculus and many other frameworks allow to express some kind of computation in a rather abstract way (as opposed to e.g. Turing machines). Abstraction is a pleasant feature of these systems, however the real cost of the computation may then be very difficult to estimate and may depend on the way the abstract system is implemented concretely.

There are some results about the intrinsic (i.e. implementation independent) computational complexity of some systems. For instance, a classical theorem of Statman states that the cost of a single β-step in any implementation of the λ-calculus may be non-elementary [18]. Hence β-reduction cannot be considered as an atomic step (as opposed to e.g. a single transition in a Turing machine), and this motivates a search for *more atomic* steps. In the case of the λ-calculus, a very natural idea introduced by Abadi *et al.* [1] is to put the implicit extensional notion of substitution in the system itself, hence making some choices about its definition and being more concrete, more *explicit* about the way things are done computationally. This has motivated a wide variety of work in the so-called field of *explicit substitutions* (for instance [1, 16, 6] without any pretension to be exhaustive).

More precisely, in the (traditional, implicit) λ-calculus, substitution is defined with definitional *equalities* such as [2]:

$$(t\ u)\{x \leftarrow v\} \overset{\Delta}{=} (t\{x \leftarrow v\})\ (u\{x \leftarrow v\}).$$

This is really an extensional definition: it is not intended to tell the way to *compute* the substitution, but only to *define* it. In particular, the expression $t\{x \leftarrow v\}$ is not at all in the syntax of terms: the substitution and its evaluation are both outside of the system. However, the misleading point is that this definition is constructive in the sense that it also gives an (intentional) algorithm, by simply orienting these equalities (from left to right) as rewrite rules on terms with explicit substitutions, in the following way (where $t[x \leftarrow v]$ is part of the syntax of terms *with explicit substitutions*):

$$(t\ u)[x \leftarrow v] \rightarrow (t[x \leftarrow v])\ (u[x \leftarrow v]).$$

This is the most natural (not to say naive) way to make substitution explicit in the λ-calculus. However, this approach may be unsatisfactory for the following reason: assume x appears free only in t and not in u, then we are probably making some useless work:

- to copy v;
- to propagate the substitution in $u[x \leftarrow v]$;

- to finally erase v when we realise it is not needed.

We say "probably" because what will really be computationally costly is still unclear in this (less, but still) abstract framework. For instance, with a call-by-name strategy, no reduction will be performed inside v hence there is no need to actually perform the duplication at this point, but only to duplicate a pointer to v (assuming such an artifact is available in the concrete framework).

This motivates to express the previous rewrite rule as a bunch of conditional rewrite rules (as in [10]) of the following flavour (where $\mathsf{fv}(t)$ denotes the set of *free variables* of t in the usual sense of the λ-calculus [2]):

$$(t\ u)[x \leftarrow v] \to (t[x \leftarrow v])\ u \quad \text{if } x \in \mathsf{fv}(t) \text{ and } x \notin \mathsf{fv}(u).$$

This solution avoids the previous defects, but now some computational cost may be hidden in the conditions on free variables. For instance, to compute the sets of free variables from scratch each time has a cost linear in the size of the terms. This is clearly too expensive for a single rewrite step. We hence have to be more clever about the *representation* of free variables, and this is the subject of this paper.

3 From Variables to Directors

In this section, we introduce the notion of directors as a representation of free variables. We will deal with reduction in Section 4 (TRSs) and 5 (CRSs), so we are mainly concerned with terms here. To make our point more concrete and easy to grasp, we consider terms as found in term rewriting systems (TRSs), although the ideas apply abstractly to any term algebra, and in particular to higher-order rewriting as will be shown in Section 5 for combinatory reduction systems.

Terms. Given a signature $(\mathcal{S}, \mathcal{F})$ and a set of variables $\mathcal{V}$, we consider the algebra of terms $\mathcal{T} = \mathcal{T}(\mathcal{F}, \mathcal{V})$ defined by [8, 19]:

$$t ::= x \mid f(t_1, \ldots, t_n) \text{ where } x \in \mathcal{V} \text{ and } f \in \mathcal{F} \text{ of arity } n$$

Free variables. The set of free variables of a term is usually defined in the following inductive way, as a function $\mathcal{T} \to \mathcal{P}(\mathcal{V})$ (where $\mathcal{P}(X)$ stands for the powerset of any set X):

$$\begin{cases} \mathsf{fv}(x) = \{x\} \\ \mathsf{fv}(f(t_1, \ldots, t_n)) = \mathsf{fv}(t_1) \cup \ldots \cup \mathsf{fv}(t_n) \end{cases}$$

This is a definition, but this also gives a (naive) algorithm to compute the set of free variables of a term in a bottom-up fashion: start from the leaves of the syntax tree representing the term (the variables) and, at each node, take the union of the sets of free variables in all the children (subterms) of the node.

However, this algorithm is linear in the size of the term which is the best we can hope without prior information, but this is still costly. If the information about free variables is of importance (as in the example of Section 2), it would certainly not be a good idea to perform the computation over and over again after each rewrite step. A better solution would be to keep the term together with its set of free variables, as a pair $(t, \mathsf{fv}(t))$, and ensure that reduction preserves that information i.e. define a reduction $\rightsquigarrow$ on pairs of terms and sets of free variables such that if $t \to u$ then $(t, \mathsf{fv}(t)) \rightsquigarrow (u, \mathsf{fv}(u))$.

In general, if $t \to u$, the information on $\mathsf{fv}(t)$ is not enough to compute $\mathsf{fv}(u)$: we thus need a richer representation. Knowing the rule $l \to r$ used and $\mathsf{fv}(t')$ for every t' subterm of t, the information will be easy to maintain, because a rewrite step is local in the sense that it does not alter deep enough subterms. The next candidate to represent terms is thus pairs $(t, \{\mathsf{fv}(t'), t' \text{ subterm of } t\})$.

Now this is slightly too much: in fact only the information about immediate subterms is necessary. If $t = f(t_1, \ldots, t_n)$, we need $\mathsf{fv}(t_i)$ for $1 \le i \le n$. If we define $\upsilon_t : \{1, \ldots, n\} \to \mathcal{P}(\mathcal{V})$ by $\upsilon_t(i) = \mathsf{fv}(t_i)$, then (t, υ_t) is a good representation because $\bigcup_{1\le i\le n} \upsilon_t(i) = \mathsf{fv}(t)$ (υ_t is enough to recover $\mathsf{fv}(t)$), and $\upsilon_t(i) = \bigcup_{1\le j\le m} \upsilon_{t_i}(j)$ (for the right m), so υ is also inductive. Note that υ_t is a function (in the mathematical sense), and we say that it a good representation because its domain is finite (and usually small) hence it can be efficiently represented by an array. We will usually assume that and omit the final step towards implementation.

The other way around. We now have a nice representation of the *sets* of free variables of a term. That means that, at each step of the substitution, we will still have to go through (some datatype representing) sets to test for membership. This is not what we want. For every position i in a symbol f of arity n, we have the information about the corresponding sets of free variables (of the subterm t_i), but what we indeed want is exactly the opposite: for each free variable x in the term $f(t_1, \ldots, t_n)$, we want to know the (maximal) set of positions i such that x is also free in t_i, so that we may propagate a substitution for x in these subterms only.

Let us write $\mathcal{L}_n = \mathcal{P}(\{1, \ldots, n\})$, and $\mathcal{L} = \amalg_{n\in\mathbb{N}}\mathcal{L}_n$, where $\amalg$ denotes the disjoint union. $\mathcal{L}$ is called the set of *immediate positions* or *locations*. If $S \in \mathcal{L}$, we allow ourselves to be explicit about the n such that $S \in \mathcal{L}_n$ by subscripting it as S_n. We abbreviate $\{1\}_1$ to $\downarrow$. Then we have: $\{1, \ldots, n\} \to \mathcal{P}(\mathcal{V}) \simeq \{1, \ldots, n\} \times \mathcal{V} \to 2 \simeq \mathcal{V} \times \{1, \ldots, n\} \to 2 \simeq \mathcal{V} \to \mathcal{P}(\{1, \ldots, n\}) = \mathcal{V} \to \mathcal{L}_n$ (where 2 is the set with two elements), i.e. υ_t in fact defines a relation between $\{1, \ldots, n\}$ and $\mathcal{V}$ and we may take the dual relation. Then it is only a matter of convenience (again, think of it as an array) to orient it as a function $\mathcal{V} \to \mathcal{L}_n$, that we denote σ_t and call the *director* of t (note that this corresponds to *director strings* in [17]). More formally, let's define the relation $\mathcal{R}$ by $i\,\mathcal{R}\,x \Leftrightarrow x \in \upsilon_t(i)$ and $\mathcal{R}^{-1}$ by $x\,\mathcal{R}^{-1}\,i \Leftrightarrow i\,\mathcal{R}\,x$. Then σ_t is the function $\mathcal{V} \to \mathcal{L}_n$ such that $x\,\mathcal{R}^{-1}\,i \Leftrightarrow i \in \sigma_t(x)$. Hence if $x \in \mathcal{V}$, $\sigma_t(x)$ gives the set of immediate positions of t where the variable x occurs free, and in particular where a substitution for x should be propagated.

Compilation. Given a (usual) term t, we call *compilation* the initial computation of σ i.e. $\sigma_{t'}(x)$ for every subterm t' of t and every variable x (note that $\sigma_{t'}(x) \neq \emptyset$ only for finitely many x's). The compilation is done once, then σ is assumed to be recomputed incrementally (i.e. not by compiling the term again) during the reduction process; this topic is discussed in Sections 4 and 5. We denote the compilation function by $[\![\cdot]\!]$, so that $[\![t]\!] = (t, \sigma)$. The intended low-level meaning is the following: at each node of the syntax tree of the term t, we attach an array indexed on (a finite subset of) $\mathcal{V}$ of lists of integers (positions). The compilation can be done by computing the sets of free variables of each subterm and taking the dual relation corresponding to v. Thinking of it as a matrix in $\{0,1\}$ indexed on $\{1,\ldots,n\}$ and $\mathcal{V}$, it is clear that no extra cost is introduced: one simply has to read the columns first instead of the rows. The whole process of compilation thus has a linear complexity, which is very acceptable since this is supposed to be done only once.

Example 1 To illustrate the idea, assume f and g are functional symbols, and x, y, z are variables. Let $t = f(x, y, g(x, z))$, then its director σ is given by (we omit σ_y and σ_z):

$$\begin{array}{llll} \sigma_t : & x \mapsto \{1,3\} & y \mapsto \{2\} & z \mapsto \{3\} \\ \sigma_{g(x,z)} : & x \mapsto \{1\} & y \mapsto \emptyset & z \mapsto \{2\} \\ \sigma_x : & x \mapsto \{1\} & y \mapsto \emptyset & z \mapsto \emptyset \end{array}$$

Readback. At the end of the reduction of an annotated term, we want to be able to read back the result as a standard term. We write this function of readback $(\!|\cdot|\!)$. In our case, we preserve the structure of terms and the names of variables so the readback is cheap and easy: simply forget σ, hence $(\!|(t,\sigma)|\!) = t$. Operationally speaking, we just ignore the extra information attached at each node.

Abstractly. For the sake of clarity, we considered some concrete term algebra along this section. However, very little depends on that concrete structure. In fact, our reasoning is general enough to fit with any framework equipped with an algebra of term defined in an inductive way with a notion of position and a notion of variables and of sets of free variables (defined in an inductive way on terms).

Alternative presentation. An alternative presentation, which is the one adopted in [17], consists in putting the directors directly in the syntax of terms, in the following way (pursuing with the same example):

$$\mathbf{t} ::= x \mid f(\mathbf{t}_1, \ldots, \mathbf{t}_n)^\sigma \text{ where } \sigma : \mathcal{V} \to \mathcal{L}_n.$$

This has the advantage of being explicit both about what part of (i.e. on which terms) σ has been computed and about where the information is located. For convenience, we prefer to leave that aspect implicit in the sequel, although this is exactly what we have in mind when we speak about (t,σ).

4 Maintaining Directors

In the previous section, we have obtained a representation of free variables suitable for the kind of problems exposed in Section 2. This would be of very little interest if this representation could not be maintained through reduction at a relatively low cost. We demonstrate in this section that it is not the case.

For the sake of clarity, we pursue the example of term rewrite systems (TRSs) [8, 19], although the results of this section will be subsumed in the next section dealing with combinatory reduction systems.

We consider already annotated terms (e.g. obtained from compilation, but not necessarily) and explain how to modify the reduction relation to preserve the correct directors.

Syntax. Terms are defined by:

$$t ::= x \mid f(t_1, \dots, t_n) \text{ where } x \in \mathcal{V}, f \in \mathcal{F} \text{ of arity } n$$

Annotated terms are then pairs (t, σ) (sometimes simply written $\mathbf{t}$) where σ is a partial function of type $\mathcal{T} \to \mathcal{V} \to \mathcal{L}$ (we write σ_t instead of $\sigma(t)$) such that:

- if $x \in \mathcal{V}$, $\sigma_x(y) = \begin{cases} \downarrow & \text{if } y = x, \\ \emptyset & \text{otherwise;} \end{cases}$
- if $f(t_1, \dots, t_n) \in \mathcal{T}$, $\forall i, 1 \le i \le n, \forall x \in \mathcal{V}, \sigma_{t_i}(x) \neq \emptyset \Rightarrow i \in \sigma_{f(t_1,\dots,t_n)}(x)$.

Remark 2 A more restrictive definition is given by replacing $\Rightarrow$ above by $\Leftrightarrow$. This alternative definition allows to recover *exactly* the set of free variables and gives a unique representation of terms. But on the other hand, this is more difficult to preserve by reduction. The definition chosen is some kind of *abstract interpretation*: a variable may suddenly disappear when traversing down a term, but at least, we are sure to reach them all. We discuss this point in more details below.

Lemma 3 Compilation as explained in the previous section gives (valid) annotated terms (even in the strong sense).

Reduction. In a term rewrite system, a rewrite rule is a pair $l \to r$ of terms, a rewrite system a set of rewrite rules, and we say that t rewrites to u if there is a rule $l \to r$, a position p in the syntax tree of t and a substitution ς such that $t|_p = l\varsigma$ and $u = t[r\varsigma]_p$ i.e. such that the subterm of t at position p is l (modulo a substitution) and such that u is t where the subterm at position p is replaced by r (modulo the same substitution).

In other words, rewriting is just a local replacement of l by r in t, without altering the context surrounding l. This is best seen graphically: on Figure 1, the only part of the term that changes is the middle one: the context and the substitution are the same. Note however that three types of things might happen in the box labelled $\omega - \delta - \epsilon$. The variables that are both free in l and in the support (see below) of ς may appear in r at different places (rearrangement), several times (duplication), or not at all (erasure). This observation will be of crucial importance, as will be explained later.

In a setting with directors, we first need to adapt the notion of substitution.

Definition 4 (Substitution) An (annotated) substitution ς is a total application from the set of variables to the set of (annotated) terms. We usually define a substitution on some variables only, with the convention that it is extended identically to $\mathcal{V}$ (i.e. $\varsigma(x) = x$ by default). The support of ς is the set of x such that $\varsigma(x) \neq x$.

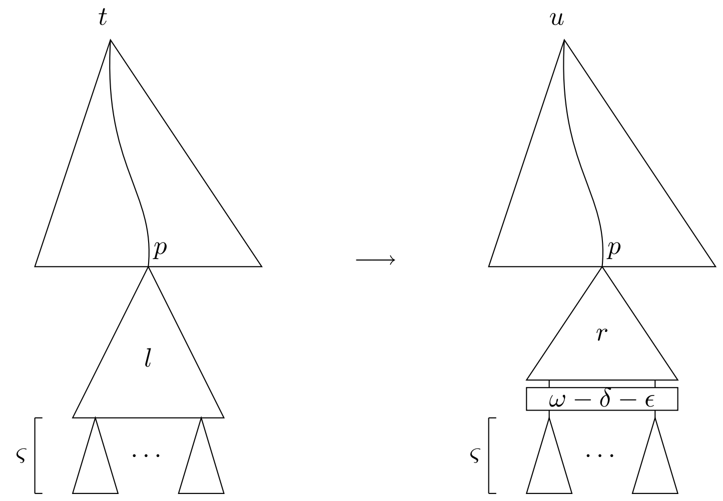

Figure 1: Reduction in a TRS

Then we would like to define a rewrite rule as a pair $\mathbf{l} \to \mathbf{r}$ of annotated terms, and say that $\mathbf{t}$ rewrites to $\mathbf{u}$ if there is a rule $\mathbf{l} \to \mathbf{r}$, a position p in the syntax tree of t and an annotated substitution ς such that $\mathbf{t}|_p = \mathbf{l}\varsigma$ and $\mathbf{u} = \mathbf{t}[\mathbf{r}\varsigma]_p$.

We thus need to define the result of the substitution $\mathbf{t}\varsigma$ of $\mathbf{t}$ with the substitution ς in presence of directors. We obviously want the following specification:

$$\mathbf{t}\varsigma = [\![(\!|\mathbf{t}|\!)\varsigma]\!]$$

where the substitution is done on the usual unlabelled terms, then the term is compiled again. This is clearly not satisfactory from an operational perspective, since we have to compile the term again and compilation is a rather costly operation.

Definition 5 Substitution on terms with directors can be defined by $(t, \sigma)\varsigma = (t\varsigma, \sigma')$ such that:

$$\begin{cases} \sigma'_x & = \sigma_{\varsigma(x)} \\ \sigma'_{f(t_1,\ldots,t_n)}(x) & = \displaystyle\bigcup_{y, \sigma_{\varsigma(y)}(x) \neq \emptyset} \sigma_{f(t_1,\ldots,t_n)}(y) \end{cases}$$

Proof This definition is valid since it fulfils the specification $\mathbf{t}\varsigma = [\![(\!|\mathbf{t}|\!)\varsigma]\!]$:

$$\begin{cases} \sigma'_x & = \sigma_{x\varsigma} = \sigma_{\varsigma(x)} \\ \sigma'_{f(t_1,\ldots,t_n)}(x) & = \sigma_{f(t_1,\ldots,t_n)\varsigma}(x) = \sigma_{f(\mathbf{t}_1\varsigma,\ldots,\mathbf{t}_n\varsigma)}(x) \\ & = \displaystyle\bigcup_{y, x \in \mathsf{fv}(y\varsigma)} \sigma_{f(t_1,\ldots,t_n)}(y) = \bigcup_{y, \sigma_{\varsigma(y)}(x) \neq \emptyset} \sigma_{f(t_1,\ldots,t_n)}(y) \end{cases}$$

Locality. At a given node, the recomputation of the attached director is thus feasible, according to the expression above. Having another glance at Figure 1, it is clear that we have to recompute the directors everywhere in the image of r in u, but this is independent from the size of the term. But do we also have to perform that recomputation everywhere else in the term ?

The free variables of a term depend only on the free variables of its subterms, and so is it for σ. Our first observation is thus that the subterms of r that are in the image of ς (at the bottom of the figure) keep the same directors, hence no recomputation is needed. The same argument also holds for any node in the upper part of the term (the context) which is not on the path towards p.

Now imagine that only rearrangement or duplication occur in the box $\omega - \delta - \epsilon$ (no erasing). Then at position p, the set of free variables is the same in u than in t (because the union is associative, commutative and idempotent). In other words, the directors also remain unchanged even on the path towards p, except maybe at position exactly p (because directors give information about the children of the corresponding node).

In case of erasing, the nodes above p (i.e. on the path towards p) should of course be updated if we want to keep the exact information about free variables up-to-date. This cost is proportional to the depth of p, which seems acceptable.

However, we may also take a slightly different approach, following [17]: we may restore the locality of rewriting by giving up the update along the path to p, at the price of uselessly propagating some substitutions that will be erased later. This is roughly not any more or less efficient, but it allows to count this cost at the top-level, and not to hide it in the reduction. In other words, the cost of a single step of reduction is independent of the size of the term, which is usually desirable; but some extra reductions may be needed.

Conditional TRSs. There is a slight paradox in our presentation. The introduction of directors is motivated by the need to have rewrite rules under certain conditions on the free variables of some subterms. Then the development is done for plain TRSs, that do not allow to express that kind of rules. But the paradox is only superficial: it is clear that adding such conditions does not lead to any problem, and that directors allow to resolve these conditions in constant time.

5 Combinatory Reduction Systems with Directors

We generalise the work done in previous sections to higher-order rewriting, thus taking into account the eventuality of binders. We adapt our formalism to that of combinatory reduction systems (CRSs) [15]. We define the notion of directors in full generality in CRSs and also in the particular case of explicit substitutions combinatory reduction systems (ESCRSs) [3], which are a better framework to fulfil our motivation to give a more realistic cost to the computation.

5.1 CRSs

We refer the reader to [14, 15] for a full presentation of CRSs. The important ideas are the use of metavariables with arity and of a generic abstraction. We give a short presentation of combinatory reduction systems in order to make the paper self-contained. The presentation is mostly taken from Klop *et al.* [15].

Term formation. Given a set of *variables* $\mathcal{V} = \{x, y, \ldots\}$, a set of *functional symbols* $\mathcal{F} = \{f^n, g^m, \ldots\}$, a set of *metavariables* (also with arity) $\mathcal{MV} = \{Z^n, Y^m, \ldots\}$, then the set of *metaterms* $\mathcal{MT}$ is defined by the following BNF-style grammar rule:

$$t ::= x \mid [x]t \mid f^n(t_1, \ldots, t_n) \mid Z^n(t_1, \ldots, t_n).$$

The superscript n on functional symbols and metavariables is called the *arity* of that symbol and is often omitted when it is clear from the context. We say that the construct $[x]t$ *abstracts* x in t, and the notions of *free* and *bound* variables are defined as usual according to this notion of abstraction (an occurrence of a variable x is bound if it is in the scope of an *abstractor* $[x]$ and free otherwise). A metaterm without any free variable is said to be *closed* (and *open* otherwise). Metaterms are considered equal modulo renaming of bound variables (*α-conversion*) and we will work under Barendregt's convention [2]. A *term* is a metaterm without any occurrence of a metavariable.

Rewrite rules. A *rewrite rule* is a pair $l \to r$ of closed metaterms where l is a *constructed* term (i.e. begins by a functional symbol) and such that the metavariables that occur in r also occur in l and the metavariables in l only occur in the form $Z(x_1, \ldots, x_n)$ where the x_i are pairwise distinct variables.

Reduction relation. Some care has to be taken to avoid name capture in the substitution, like in the λ-calculus and in contrast with TRSs. As usual, we say that t rewrites to u using rule $l \to r$, if there is a position p in the syntax tree of t and a substitution ς such that $t|_p = l\varsigma$ and $u = t[r\varsigma]_p$. Following Klop [15], we define substitutions as mappings assigning to an n-ary metavariable an n-ary *substitute*:

$$\varsigma(Z^n) = \underline{\lambda}(x_1, \ldots, x_n).t.$$

Substitutions are homomorphically extended to metaterms as follows (also recall that n-ary metavariables are not metaterms unless $n = 0$, they need arguments):

$$\begin{cases} x\varsigma = x \\ ([x]t)\varsigma = [x](t\varsigma) \\ f(t_1, \ldots, t_n)\varsigma = f(t_1\varsigma, \ldots, t_n\varsigma) \\ Z(t_1, \ldots, t_n)\varsigma = \varsigma(Z)(t_1\varsigma, \ldots, t_n\varsigma) \end{cases}$$

Substitutes immediately generate a simultaneous substitution in the following way: if $\varsigma(Z) = \underline{\lambda}(x_1, \ldots, x_n).t$, then $\varsigma(Z)(t_1, \ldots, t_n) = t\{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$ (with the usual implicit notion of substitution modulo α-conversion).

With substitutes, we are close to the familiar ground of λ-calculus, hence to avoid variable clashes, we may just say that we rename enough both substitutes and metaterms appearing in the rewrite rule in the usual way. See [15] for more details.

5.2 CRSs with Directors

Overview. To lift the work done on TRSs to CRSs, we have to take care of two new features:

- the abstraction $[x]t$;
- the distinction between variables and metavariables.

The first one is easily dealt with: we simply have to modify the conditions required on directors such that in an annotated term $\sigma_{[x]t}(x) = \emptyset$ while $\sigma_t(x)$ may be different from $\emptyset$ (recall that we work under Barendregt's convention).

The second point is more subtle. First notice that only directors on variables are needed to represent terms. Terms with metavariables only make sense in rewrite rules, so the best choice is to annotate metavariables in a way that makes the substitution uniform when a rule is applied to a term. Metaterms of the form $Z(t_1, \ldots, t_n)$ are thus considered unary and for instance $Z(x, y)$ is annotated by $\{Z \mapsto\downarrow, x \mapsto\downarrow, y \mapsto\downarrow\}$. Notice that directors for metaterms mix information about ground variables and metavariables; however, after instantiation, there will be only information about ground variables.

Terms. An annotated metaterm is a pair (t, σ) where t is a metaterm and $\sigma : \mathcal{MT} \to (\mathcal{V} \amalg \mathcal{MV}) \to \mathcal{L}$ is called a director. Again, we write $\sigma_t(x)$ instead of $\sigma(t)(x)$. We say that σ_t has rank n if for every variable or metavariable α, $\sigma_t(\alpha) \in \mathcal{L}_n$. The director σ has to satisfy the following requirements (α represents any variable or metavariable):

- Variables:
  - σ_x is of rank 1
  - $\sigma_x(x) =\downarrow$
  - $\sigma_x(\alpha) = \emptyset$ for $\alpha \neq x$
- Abstractions:
  - $\sigma_{[x]t}$ is of rank 1
  - $\sigma_{[x]t}(x) = \emptyset$
  - $\sigma_t(\alpha) \neq \emptyset \Rightarrow \sigma_{[x]t}(\alpha) =\downarrow$ for $\alpha \neq x$
- Functional symbols:
  - $\sigma_{f(t_1,\ldots,t_n)}$ is of rank n
  - $\forall i, 1 \leq i \leq n, (\sigma_{t_i}(\alpha) \neq \emptyset \Rightarrow i \in \sigma_{f(t_1,\ldots,t_n)}(\alpha))$
- Metavariables:
  - $\sigma_{Z(t_1,\ldots,t_n)}$ is of rank 1 (notice that it is not of rank n)
  - $\sigma_{Z(t_1,\ldots,t_n)}(Z) =\downarrow$
  - $(\exists i, 1 \leq i \leq n, \sigma_{t_i}(\alpha) \neq \emptyset) \Rightarrow \sigma_{Z(t_1,\ldots,t_n)}(\alpha) =\downarrow$

– $(\forall i, 1 \le i \le n, \sigma_{t_i}(\alpha) = \emptyset) \Rightarrow \sigma_{Z(t_1,\dots,t_n)}(\alpha) = \emptyset$ for $\alpha \neq Z$.

The very last condition ensures that e.g. $\sigma_{Z(x)}(y) = \emptyset$ whereas we could have $\sigma_{f(x)}(y) =\downarrow$ and y being erased somewhere below (arriving at x in that case).

Compilation and readback are similar to Section 3 and omitted.

Reduction relation. Rewrite rules are of the form $(l, \sigma) \to (r, \sigma')$. The reduction relation is defined as in the previous section with the difference that substitution is extended to annotated terms by $(t, \sigma)\varsigma = (t\varsigma, \sigma')$ where:

$$\sigma'_x = \sigma_{x\varsigma} = \sigma_x, \text{hence } \sigma'_x(\alpha) = \begin{cases} \downarrow & \text{if } \alpha = x, \\ \emptyset & \text{otherwise.} \end{cases}$$

$$\sigma'_{[x]t} = \sigma_{([x]t)\varsigma} = \sigma_{[x](t\varsigma)}, \text{hence } \sigma'_{[x]t}(\alpha) = \begin{cases} \downarrow & \text{if } \exists \beta, \sigma_t(\beta) \neq \emptyset \wedge \sigma_{\varsigma(\beta)}(\alpha) \neq \emptyset, \\ \emptyset & \text{otherwise.} \end{cases}$$

Note that we disallow variable capture by implicit safeness conditions, hence $\sigma_{\varsigma(\alpha)}(x) = \emptyset$ for every α. Also note that the existential quantifier is a lot more operational than it seems, since the search is bounded both by the size of the domain of σ_t (i.e. the number of free variables of t) and the size of the support of ς.

$$\sigma'_{f(t_1,\dots,t_n)} = \sigma_{f(t_1,\dots,t_n)\varsigma} = \sigma_{f(t_1\varsigma,\dots,t_n\varsigma)},$$
$$\text{hence } \sigma'_{f(t_1,\dots,t_n)}(\alpha) = \bigcup_{\sigma_{\varsigma(\beta)}(\alpha) \neq \emptyset} \sigma_{f(t_1,\dots,t_n)}(\beta).$$

Again, the union is of a very finite kind so that computation is indeed easy.

For metavariables, there are three cases depending on the substitution ς. If $\varsigma(Z) = Z$, then:

$$\sigma'_{Z(t_1,\dots,t_n)} = \sigma_{Z(t_1,\dots,t_n)\varsigma} = \sigma_{Z(t_1\varsigma,\dots,t_n\varsigma)}$$
$$\text{hence } \sigma'_{Z(t_1,\dots,t_n)}(\alpha) = \begin{cases} \downarrow & \text{if } \alpha = Z \text{ or if } \exists i, \beta, \sigma_{t_i}(\beta) \neq \emptyset \wedge \sigma_{\varsigma(\beta)}(\alpha) \neq \emptyset, \\ \emptyset & \text{otherwise.} \end{cases}$$

If $\varsigma(Z)$ is a (meta)-projection, i.e. if $\varsigma(Z) = \underline{\lambda}(x_1 \dots x_n).x_j$ for some j, then $Z(t_1, \dots, t_n) = t_j$ and:

$$\sigma'_{Z(t_1,\dots,t_n)}(\alpha) = \begin{cases} \emptyset & \text{if } \alpha = x_k \text{ for some } k, \\ \sigma_{t_j}(\alpha) & \text{otherwise.} \end{cases}$$

Finally, in the general case, $\varsigma(Z) = \underline{\lambda}(x_1 \dots x_n).t$ (provided the two previous cases do not apply), so that $Z(t_1, \dots, t_n) = t\{x_1 \leftarrow t_1, \dots, x_n \leftarrow t_n\}$.

$$\sigma'_{Z(t_1,\dots,t_n)}(\alpha) = \begin{cases} \emptyset & \text{if } \alpha = x_k \text{ for some } k, \\ \sigma_t(\alpha) \;\cup \displaystyle\bigcup_{\substack{1 \le j \le n, \\ \sigma_{t_j}(\alpha) \neq \emptyset}} \sigma_t(x_j) & \text{otherwise.} \end{cases}$$

It is clear that only local knowledge of the term and substitution is needed to compute σ', and the number of elementary computations required is bounded independently of the size of the term (for the computation of a single director).

Properties. We have defined the result of substitution on directors, hence we have also defined reduction on terms with directors. By construction, the new directors correspond to the old ones, hence the information about free variables is maintained by reduction. This can be formalised in the following way.

Proposition 6 Substitution as defined above is correct with respect to the substitution on (meta)terms without directors (i.e. $\mathbf{t}\varsigma = [\![(\!|\mathbf{t}|\!)\varsigma]\!]$ with the notations of Section 3).

Proof It is easy to check that the intermediate steps in the derivations above are correct.

Definition 7 σ is said to be strongly correct for t if $x \in \mathsf{fv}(t') \Leftrightarrow \sigma_{t'}(x) \neq \emptyset$ for every subterm t' of t.

Lemma 8

- If σ is strongly correct for t, then σ' (as given above) is strongly correct for $t\varsigma$.
- If σ is strongly correct for t and $t \to u$, then σ' is strongly correct for u, where σ' is defined as above with the substitution ς corresponding to the rewrite $t \to u$.

Proof Easy consequences of Proposition 6.

Theorem 9 (Correctness) If $\mathbf{t} \to \mathbf{u}$ on annotated terms, then $(\!|\mathbf{t}|\!) \to (\!|\mathbf{u}|\!)$ on standard terms.

$$\begin{tikzcd}
\mathbf{t} \arrow[r] \arrow[d, dotted, "(\!|\cdot|\!)"'] & \mathbf{u} \arrow[d, dotted, "(\!|\cdot|\!)"] \\
t \arrow[r, dashed] & u
\end{tikzcd}$$

Proof First recall that $(\!|\cdot|\!)$ only erases information, so this is immediate if the rewrite rules do not have conditions about free variables. If they do, then it follows from Lemma 8.

Theorem 10 (Completeness) If $t \to u$ on standard terms, then there exists an annotated term $\mathbf{u}$ such that $[\![t]\!] \to \mathbf{u}$ on annotated terms and $(\!|\mathbf{u}|\!) = u$.

$$\begin{tikzcd}
t \arrow[r] \arrow[d, dotted, "{[\![\cdot]\!]}"'] & u \\
\mathbf{t} \arrow[r, dashed] & \mathbf{u} \arrow[u, dotted, "(\!|\cdot|\!)"']
\end{tikzcd}$$

Proof Again a consequence of Lemma 8.

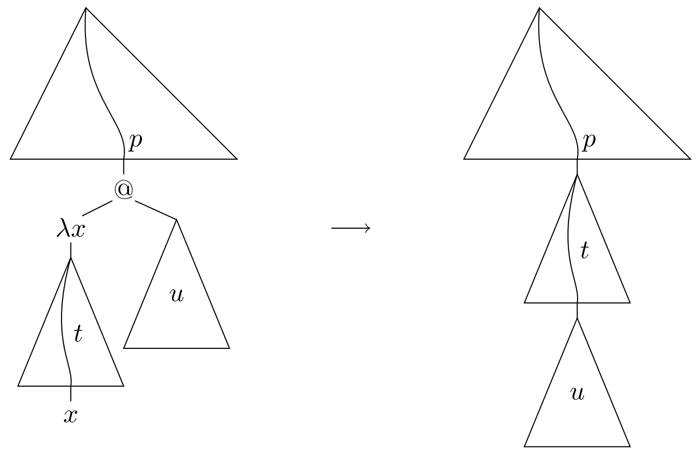

Figure 2: A typical β-reduction

Locality. Contrary to the case of TRSs, we cannot guarantee in this case that only few directors need to be updated. This is due to the mechanism of substitution implicit in CRSs, and this property is thus recovered in the subset of CRSs with explicit substitutions (ESCRSs, see below).

We may see the situation more precisely on a typical example. The λ-calculus is an instance of a CRS (see [15]), with symbols λ (of arity 1) and @ (of arity 2), and only rewrite rule (β) given by $@(\lambda([x]Z(x)), Y) \to Z(Y)$. A typical reduction is shown on Figure 2. All directors on the path from the root of the term to u (in the term to the right of the figure) should be modified to take into account the free variables of u. This induces a cost linear in the height of the context plus the height of t, and is not restricted to the case of erasing (as opposed to the situation with TRSs).

It is thus not realistic to use directors with general CRSs, but this is neither a surprise nor a disappointment: directors are intended to give information to direct substitutions, so they are only meaningful in a context where the substitutions are explicit. However, although without any practical use, it is nice from a theoretical point of view to be able to define directors in the very general context of CRSs.

Conditional CRSs. Again, we may add explicit conditions in the rewrite rules about the free variables of subterms, without any trouble and so that directors allow to resolve these conditions in constant time. Note that some conditions are already implicit in CRSs (e.g. in $[x]Z$, we know that x is not free in any term substituted for Z; but in $[x]Z(x)$, we cannot be sure that $x \in \mathsf{fv}(t)$ if Z is substituted by t). Maybe this is to be taken as a hint that a better framework should be designed to integrate in a more homogeneous way rewriting and directors.

5.3 ESCRSs with Directors

ESCRSs. A reduction step in a CRS is not a good unit of computational complexity, because of the implicit notion of substitution which is allowed arbitrarily deep in a term. Moreover, we do not have good properties of locality, which makes it unrealistic to maintain directors during reduction. We thus need a better framework for our purpose, namely CRSs with explicit substitutions. The pleasant feature of CRSs in this respect is that, following Bloo and Rose [3], so-called CRSs with explicit substitutions (or ESCRSs) are a subclass of CRSs. More precisely, a CRS is an ESCRS if all metavariable applications in the right hand side of every rewrite rule occur in the form $Z(x_1, \ldots, x_n)$ such that $Z(x_1, \ldots, x_n)$ also occurs in the left hand side of that same rule. ESCRSs thus avoid using the strong tool of substitution of CRSs and only local knowledge of the term is needed at each step. Moreover, there is a systematic way to *explicify* (borrowing the formulation of [3]) a CRS, that is to give an ESCRS with satisfactory properties of simulation, preservation of confluence and preservation of strong normalisation (with some restrictions) (see [3] for details).

Explicitation of the reduction relation. ESCRSs do not use all the power of CRSs, thus we may simplify the definition of the reduction relation for this subclass. In fact, we do not use the λ-calculus-like substitution, hence we do not need substitutes at all. We say that substitutions associate a term to a metavariable application of the form $Z(x_1, \ldots, x_n)$ and we no longer have to deal with cases of the form $Z(t_1, \ldots, t_n)$:

$$\varsigma\,(Z(x_1, \ldots, x_n)) = t$$

and ς is homomorphically extended to metaterms (with the previous restriction) in the usual way.

Of course, care should still be taken to avoid name clashes, but only with respect to abstractions and no longer to substitutes (the closedness condition ensures that the variables appearing in a metavariable application are bound by an abstraction).

Reduction in ESCRSs with directors. The substitution is now simpler because metaterms of the form $Z(t_1, \ldots, t_n)$ do not exist any more. There is only one case here. Assume $\varsigma(Z(x_1, \ldots, x_n)) = t$, then, very simply:

$$\sigma'_{Z(x_1,\ldots,x_n)} = \sigma_{Z(x_1,\ldots,x_n)\varsigma} = \sigma_t.$$

Locality. Since the mechanism of substitution of CRSs is forbidden in ESCRSs, we are in a case very similar to that of TRSs concerning locality: directors need to be recomputed only in the image of the right hand side of the rule and, in case of erasing, on the path from the root of the term to the position of the rewrite. Again, this last part may be omitted at the price of performing some extra (explicit) substitutions later.

6 Lambda-Calculus with Explicit Substitutions

Generalities. We may now present in this formalism director strings for the λ-calculus (with explicit substitutions) as described in [17]. Motivation for this is of course to improve the efficiency of evaluators or compilers of functional languages, which are based on the λ-calculus. The work done in [17] also includes the presentation of a particular strategy of reduction that takes advantage of directors and proves very efficient on experimental results.

First, since in pure λ-calculus functional symbols are only unary or binary, we use the following abbreviations, which give a better intuition of directors:

- unary symbols (variable and abstraction): $- \equiv \emptyset_1, \downarrow \equiv \{1\}_1 \in \mathcal{L}_1$
- binary symbols (application and substitution):
  $- \equiv \emptyset_2, \curvearrowleft \equiv \{1\}_2, \curvearrowright \equiv \{2\}_2, \curvearrowleft\!\!\curvearrowright \equiv \{1,2\}_2 \in \mathcal{L}_2$

For instance, the intuition behind a director $\curvearrowleft$ is: the corresponding variable occurs only in the left subterm of a binary construct; or equivalently: a substitution for this variable should be propagated only to the left.

In this context, following the notations of [17], elements of $\mathcal{L}$ are called *directors* and ordered lists of elements of $\mathcal{L}$ are called *director strings* and are used instead of functions from $\mathcal{V}$ to $\mathcal{L}$. This can be thought of as using a variant of de Bruijn indices [7] instead of names thus avoiding problems of capture due to λ-binders. This is only a variant because in [17] a new variable introduced by a λ-binder appears last (and not first) in the strings; this is reminiscent of Crégut's reversed de Bruijn indexing [5].

Notice that the use of lists also makes a difference for the erasing process, because when a variable is erased, it does not have any corresponding director in the strings of the subterms. This can be understood in the following way: we extend directors to $\mathcal{L} \cup \bot$ and impose the following condition on all terms: $\sigma_{f(t_1,\ldots,t_n)}(x) \in \{\emptyset, \bot\} \Rightarrow \forall i.\sigma_{t_i}(x) = \bot$. Then, when writing directors as lists, we simply erase every occurrence of $\bot$.

This present work may be seen as a way to quotient director strings of [17] by the order of the elements of the lists. The rewrite rules given in [17] are then just a particular case of ESCRS with directors as in Section 5.

Remark 11 Note that from an implementation point of view, the use of names and functions here corresponds to an implementation in terms of arrays, instead of lists.

The system. As a typical example, we will develop a bit more the example of the λ-calculus. We first make explicit the underlying conditional ESCRS. The functional symbols (with their arity) are: λ^1 (abstraction), $@^2$ (application) and Σ^2 (explicit substitution). The rewrite rules are as follows.

$$\begin{array}{lrcl}
(b) & @(\lambda([x]Z(x)), Y) & \to & \Sigma([x]Z(x), Y) \\
(v) & \Sigma([x]x, Y) & \to & Y \\
(a_1) & \Sigma([x](@(Z_1(x), Z_2)), Y) & \to & @(\Sigma([x]Z_1(x), Y), Z_2) \\
 & & & \text{if } x \in \mathsf{fv}(Z_1(x)) \\
(a_2) & \Sigma([x](@(Z_1, Z_2(x))), Y) & \to & @(Z_1, \Sigma([x]Z_2(x), Y)) \\
 & & & \text{if } x \in \mathsf{fv}(Z_2(x)) \\
(a_3) & \Sigma([x](@(Z_1(x), Z_2(x))), Y) & \to & @(\Sigma([x]Z_1(x), Y), \Sigma([x]Z_2(x), Y)) \\
 & & & \text{if } x \in \mathsf{fv}(Z_1(x)) \cap \mathsf{fv}(Z_2(x)) \\
(l) & \Sigma([x]\lambda([y]Z(x,y)), Y) & \to & \lambda([y]\Sigma([x]Z(x,y), Y)) \\
 & & & \text{if } x \in \mathsf{fv}(Z(x,y)) \\
(c) & \Sigma([x]\Sigma([y]Z(y), Y(x)), X) & \to & \Sigma([y]Z(y), \Sigma([x]Y(x), X)) \\
 & & & \text{if } x \in \mathsf{fv}(Y(x)) \\
(e) & \Sigma([x]Z, Y) & \to & Z
\end{array}$$

As an illustration, we may give the reduction relation induced by the rule (a_1) in a fully explicit way, as in [17]. From now one, we allow the usual readable notation for this ESCRS. The rule (a_1) now looks like:

$$(a_1) \quad (Z_1(x)\ Z_2)[x/Y] \to Z_1(x)[x/Y]\ Z_2 \qquad \text{if } x \in \mathsf{fv}(Z_1(x))$$

And with directors:

$$\begin{aligned}
(a_1') \quad & (((Z_1(x))^{\{Z_1, x:\downarrow\}}\ Z_2^{\{Z_2:\downarrow\}})^{\{Z_1, x:\curvearrowleft; Z_2:\curvearrowright\}}[x/Y^{\{Y:\downarrow\}}])^{\{Z_1, Z_2:\curvearrowleft; Y:\curvearrowright\}} \\
& \to (((Z_1(x))^{\{Z_1, x:\downarrow\}}[x/Y^{\{Y:\downarrow\}}])^{\{Z_1:\curvearrowleft; Y:\curvearrowright\}}\ Z_2^{\{Z_2:\downarrow\}})^{\{Z_1, Y:\curvearrowleft; Z_2:\curvearrowright\}}
\end{aligned}$$

Note that the condition $x \in \mathsf{fv}(Z_1(x))$ is enforced by the directors for x (two occurrences).

Now consider an annotated term $\mathbf{t} = (t, \sigma)$ where $t = (u\ v)[x/w]$ and u, v, w are terms (not metaterms). Assume that $\sigma_u(x) \neq \emptyset$ and $\sigma_v(x) = \emptyset$. The term $\mathbf{t}$ then rewrites using (a_1') (in the annotated terms) with the substitution $\varsigma = \{Z_1 \mapsto \underline{\lambda} x.u, Z_2 \mapsto v, Y \mapsto w\}$ to (t', σ') with $t' = (u[x/w])\ v$ and σ' is as follows (where r is the right hand side of (a_1') and $\alpha \neq x$):

$$\begin{aligned}
\sigma'_{t'}(\alpha) &= \bigcup_{\sigma_{\varsigma(\beta)}(\alpha) \neq \emptyset} \sigma_r(\beta) = \bigcup_{\sigma_u(\alpha) \neq \emptyset} \sigma_r(Z_1) \cup \bigcup_{\sigma_v(\alpha) \neq \emptyset} \sigma_r(Z_2) \cup \bigcup_{\sigma_w(\alpha) \neq \emptyset} \sigma_r(Y) \\
&= \bigcup_{\sigma_u(\alpha) \neq \emptyset \vee \sigma_w(\alpha) \neq \emptyset} \curvearrowleft \cup \bigcup_{\sigma_v(\alpha) \neq \emptyset} \curvearrowright \\
&= \begin{cases}
- & \text{if } \alpha \notin \mathsf{fv}(u) \wedge \alpha \notin \mathsf{fv}(w) \wedge \alpha \in \mathsf{fv}(v), \\
\curvearrowleft & \text{if } (\alpha \in \mathsf{fv}(u) \vee \alpha \in \mathsf{fv}(w)) \wedge \alpha \notin \mathsf{fv}(v), \\
\curvearrowright & \text{if } (\alpha \notin \mathsf{fv}(u) \wedge \alpha \notin \mathsf{fv}(w)) \wedge \alpha \in \mathsf{fv}(v), \\
\curlywedgedownarrow & \text{if } (\alpha \in \mathsf{fv}(u) \vee \alpha \in \mathsf{fv}(w)) \wedge \alpha \in \mathsf{fv}(v).
\end{cases}
\end{aligned}$$

This is exactly the reduction of [17]. The very good news is that we are now allowed to be implicit about this reduction relation since we can generate it “automatically”, which was not the case in [17].

7 Calculus of Inductive Constructions

The previous section gives the basis for an efficient implementation of functional languages. We now illustrate how the same approach could be used for the implementation of functional proof assistants like Coq [20]. Coq is a proof assistant based on the calculus of inductive constructions [4], which is a rich type theory. However, from an implementation point of view, we only have to deal with type-erased terms (as in [12], in a different context), which are nothing more than λ-terms with a case construct:

$$t ::= x \mid \lambda x.t \mid t_1\, t_2 \mid C(\vec{t}) \mid \texttt{case}\ t\ \texttt{of}\ (C_i(\vec{x_i}) \to a_i)_{i \in I}$$

Every constructor $C(\cdot)$ is considered an n-ary symbol of the right arity. Every branch in the `case` construct should have the same variables, so the `case` is actually considered as a binary symbol. A `case` branch construct of the form $(C_i(\vec{x_i}) \to \cdot)$ acts as a binder i.e. $\mathsf{fv}(C_i(\vec{x_i}) \to a_i) = \mathsf{fv}(a_i) \setminus \vec{x_i}$. Reduction is defined in the intuitive way. Unfortunately, this system is not exactly an instance of a CRS. However, it fits very well in the abstract framework of Section 3 and there is no technical difficulty to apply directly the previously exposed techniques. This may again be a hint that this work should also be valid in a more general framework. However, we would have to do all the work again from the beginning to be formal about this case. This is clearly out of the scope of this paper, especially since there is no technical difficulty.

Moreover, we exposed in [17] an efficient way to reduce annotated terms to normal form and to compare them for β-equality using a variant of the closed reduction strategy. The two preliminary steps necessary to use our work in an actual implementation of a theorem proving tool are thus almost completed.

8 Conclusion

The director strings of [17] proved very useful to express in a more concrete way, closer to implementation, efficient strategies of the λ-calculus such as those of [10], but were somewhat *ad hoc* for that particular case. We have provided here a generalisation of this work that extends the interest of these techniques to higher-order rewriting, with potential applications in the implementation of functional languages and proof assistants. While doing so, we also hope to have brought a better understanding to the concepts of directors and free variables, in particular it is now clear why the information can be maintained locally in a framework with explicit substitutions. Explicit substitutions are the right framework for director strings, and director strings are the right implementation technique for explicit substitutions systems (of course depending on the motivations).

Acknowledgements

The author is grateful to Maribel Fernández, Paul-André Melliès and the anonymous referees for their comments.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[3] R. Bloo and K. H. Rose. Combinatory reduction systems with explicit substitution that preserve strong normalisation. In *Proceedings of Rewriting Techniques and Applications (RTA'96)*, volume 1103 of *Lecture Notes in Computer Science*, pages 169–183, 1996.

[4] Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[5] P. Crégut. An abstract machine for lambda-terms normalization. In *Lisp and Functional Programming 1990*, pages 333–340. ACM Press, 1990.

[6] R. David and B. Guillaume. A λ-calculus with explicit weakening and explicit substitution. *Mathematical Structures in Computer Science*, 11(1):169–206, 2001.

[7] N. G. de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–392, 1972.

[8] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 243–320. MIT Press, 1990.

[9] M. Fernández and I. Mackie. Director strings and explicit substitutions. WEST-APP'01, Utrecht, 2001.

[10] M. Fernández, I. Mackie, and F-R. Sinot. Closed reduction: Explicit substitutions without alpha-conversion. *Mathematical Structures in Computer Science*, 2005. to appear.

[11] M. Fernández, I. Mackie, and F-R. Sinot. Lambda-calculus with director strings. *Applicable Algebra in Engineering, Communication and Computing*, 2005. to appear.

[12] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proceedings of ICFP'02, Pittsburgh, Pennsylvania, USA*, 2002.

[13] J.R. Kennaway and M.R. Sleep. Director strings as combinators. *ACM Transactions on Programming Languages and Systems*, 10(4):602–626, 1988.

[14] J. W. Klop. Combinatory reduction systems. Mathematical Centre Tracts 127, Centre for Mathematics and Computer Science, Amsterdam, 1980. PhD thesis.

[15] J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1-2):279–308, December 1993.

[16] P. Lescanne and J. Rouyer-Degli. The calculus of explicit substitutions lambda-upsilon. Technical Report RR-2222, INRIA, 1995.

[17] F.-R. Sinot, M. Fernández, and I. Mackie. Efficient reductions with director strings. In R. Nieuwenhuis, editor, *Proceedings of Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[18] R. Statman. The typed λ-calculus is not elementary recursive. *Theoretical Computer Science*, 9:73–81, 1979.

[19] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[20] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, 2004. `http://coq.inria.fr`.