A Canonical Calculus of Residuals

Yves Bertot *
INRIA Sophia Antipolis 06561 Valbonne Cedex
France

July 1991

Abstract

We introduce a formal language to describe origin functions, which permit to study the notions of descendance and residuals in reduction systems. Computation on this formal language are defined using a term rewriting system, which we show to be canonical. This work has application in semantics and debugging.

1 Introduction.

We introduce new tools to capture the notions of descendance and residuals that appear regularly in the study of rewriting and reduction systems. These new tools provide an easier encoding of these notions and may have applications in implementing evaluation strategies [1, 5, 8, 13, 14] or debugging algorithms based on rewriting or reduction [3].

Most presentations of descendance and residuals use labeling of terms. For any reduction system, one simply produces a labeled version of the system, together with a procedure to transform a derivation on labeled terms into a derivation on unlabeled terms (called *erasure*) and a procedure to transform a derivation on unlabeled terms in a derivation on labeled terms, given a labeling of the initial term (called *lifting*). Intuitively, a node or a position in the resulting term descends from a node or a position in the initial term if they share the same label in a labeled derivation where the positions in the initial term are labeled with distinct labels. The actual value of the common label actually has no importance, it is only relevant that this label is shared between the original position and the descendant. In this respect the use of labels is rather contrived. You have to use labels, but their value has no importance and you have to label the initial term with distinct labels.

The constraint to use distinct labels is needed because these labels denote positions in the initial term. However, positions in terms are best described using *occurrences*, that is paths inside terms. Instead of labeling the initial

* bertot@sophia.inria.fr

term of a derivation with arbitrary distinct labels, we propose to use the occurrences denoting the positions in that term. When looking at the final term, it is then possible to know the position in the initial term, which a position in the final term comes from, only by looking at the label carried at this position: there is no need to look up in the initial term.

We also propose to separate the labels from the term we want to label. This corresponds to mapping labeled terms to pairs of the form $(M, \mathcal{L})$ where M is an unlabeled term and $\mathcal{L}$ is a function mapping positions in M to labels. If we use occurrences both to denote positions and to label terms, then the function $\mathcal{L}$ maps occurrences to occurrences. Labeling a term with the occurrences denoting each position, as we want to do for initial terms of studied derivations, corresponds simply to constructing a pair $(M, \mathcal{I})$ where M is an unlabeled term and $\mathcal{I}$ is simply the identity function over occurrences.

Separating the labeling function from the term also permits to give an *intensional* description of the labeling function, whereas in-place labeling forces to have an *extensional* description. In a standard labeled term, every position carries its own label and the labeling of the term is thus given extensionally. When implementing labeled terms using the data separation we propose, one can use a formal language to describe labeling functions, with, for instance, a specific element id to denote the identity function over occurrences. The extra cost in memory of labeling the term M with distinct labels is constant, whereas it would be proportional to the size of M with standard labeling. We believe this approach has significant advantages when implementing labeled terms, be it to reason about these terms in a proof system or to use labeling as a basis for program animation or debugging.

A second advantage of our approach for implementing labeled terms is that the language of functions over occurrences can be given once and for all, independently of the actual language of terms under study. Using a standard in-place labeling approach, one would have to duplicate the datatype declaration for the language of terms, providing a new “labeled" operator for every “unlabeled" operator. Thus, if the type of unlabeled terms is given, say, by a declaration of the following kind (in *standard ML* syntax):

```
datatype term = f of term * term
              |
              .
              .
              | g;
```

One would have to write a declaration of the following form:

```
datatype labeled = labeled_f of labeled * labeled * label
                 |
                 .
                 .
                 | labeled_g of label;
```

However, few languages provide the possibility to construct mechanically the second declaration from the first one (*standard ML*, for one, does not provide such a construction). Using our approach, however, one would only have to write the following declaration, where `origin` denotes the datatype representing functions over occurrences.

```
datatype 'a labeled = lab of 'a * origin;
```

And the functions for lifting a term, erasing labels, finding the label at a specific occurrence can be written once and for all independently of the actual language of term one wants to label.

When considering the design of a language for denoting functions over occurrences, we come to another crucial feature of our work. We contend that occurrences can be identified both to projections over terms and to the labeling functions corresponding to such operations.

A projection is a function that maps every term to its subterm at a given occurrence. For instance, the function s_i that maps any term of the form $f(t_1, \ldots, t_i, \ldots, t_n)$ to its child of rank i, t_i, (regardless of the operator f) is a projection. Instead of denoting occurrences using lists of integers, as in [9], we simply write them as compositions of the elementary projections s_i. In this notation, concatenation of lists is simply replaced by standard function composition.

Let us now consider the interaction of projections with labeling. If all the positions in term M are labeled with their occurrence, and N is the subterm at occurrence u, then the node at position given by occurrence v in N is simply labeled with the occurrence $v \circ u$ in our formalism (that would be $u@v$ if we used lists of integers, with @ denoting the *append* operation). Thus, the labeling function $\mu(u)$ naturally associated to the ocurrence u is simply the function mapping any occurrence v to to $v \circ u$. We propose to use this property to identify occurrences with functions over occurrences, the term $s_1 \circ s_2$ being possibly used to represent an occurrence or the function over occurrences $\mu(s_1 \circ s_2)$ naturally associated to this occurrence. In our language of terms, the actual (intended) meaning will be inferred from the context.

The composition operator, $\circ$, appears both as an operator ranging on occurrences, permitting to construct big occurrences from smaller ones (as *append* did for integer lists) and as an operator ranging on functions over occurrences, this time representing usual function composition. We notice however, that the μ function is "contravariant" with respect with composition, that is we shall see that for any two occurrences u and v we have $\mu(v \circ u) = \mu(u) \circ \mu(v)$.

But functions over occurrences obtained through the function μ are very regular and we need other operators in our language to represent very irregular functions over occurrences. Keeping in mind that the initial motivation is to represent labeling of terms, we choose to use operators that permit to follow the recursive structure of terms. The basic idea is as follows: suppose we want to describe the labels carried by a term $M = f(t_1, \ldots, t_n)$. These labels will be completely described once we have given the label u carried at the head of the term and the labelings $f_1, \ldots, f_n$ for the terms $t_1, \ldots, t_n$. Thus, the whole labeling can be described by a term of the form $c([f_1, \ldots, f_n], u)$. The functions $f_1, \ldots f_n$ themselves can be described by terms of the same form. As it appends this operator, c, gives the possibility to describe extensionally the labeling of a term, as it permits to construct a tree isomorphic to the labeled term, responsible for carrying the labels, which need not be occurrences. When using occurrences for labeling however, this notation mixes well with the possibility we have to use occurrences to represent functions over occurrence. For example, we can notice that both the occurrence u and the term $c([u \circ s_1, \ldots, u \circ s_n], u)$ represent the same labeling with occurrences for a term whose head node has less than n children.

The mixing of extensional and intensional description that the operator c and the use of occurrences as functions over occurrences provide is especially suited to the description of residuals in term rewriting systems and the λ-calculus. If we consider a simple β-reduction, say $(\lambda x.xM)N \to NM$, it is obvious that redexes appearing in M after the reduction are residuals of the redexes appearing in M before the reduction, and the application node between the N and the M after the reduction "descends" from the application node between the x and the M. When supposing that the positions in the initial term were labeled with their occurrences, our proposal permits simply to describe the labeling of the result term with the notation $c([s_2, s_2 \circ s_2 \circ s_1], s_2 \circ s_1)$. This description is valid *regardless of the actual content and size of the terms M and N*. Of course, the terms s_2 and $s_2 \circ s_2 \circ s_1$ that appear between the square brackets, [...] represent functions over occurrences, whereas the the term $s_2 \circ s_1$ that appears before the closing parenthesis represents an occurrence. In our opinion, the conciseness of this notation and its polymorphic aspect are strong arguments for its use in implementing labeling and descendance relations.

2 Mathematical Preliminaries.

2.1 Occurrences.

We denote $\mathcal{N}$ the set of natural numbers and $\mathcal{N}_+$ the set $\mathcal{N} - \{0\}$. A *graded alphabet* is a set Σ given with a function $\alpha : \Sigma \to \mathcal{N}$, called *arity* function. The set $T(\Sigma)$ of Σ-*terms* is the least set verifying the following conditions:

1. For every operator $g \in \Sigma$ such that $\alpha(g) = 0$, one has $g \in T(\Sigma)$,
2. For every operator $f \in \Sigma$ such that $\alpha(g) = n > 0$ and every n-tuple $(t_1, \ldots, t_n) \in T(\Sigma) \times \ldots \times T(\Sigma)$, one has $f(t_1, \ldots, t_n) \in T(\Sigma)$.

The operator f is called the *head operator* of the term $f(t_1, \ldots, t_n)$. In the following, Σ-terms are regularly called terms, when the alphabet is obvious

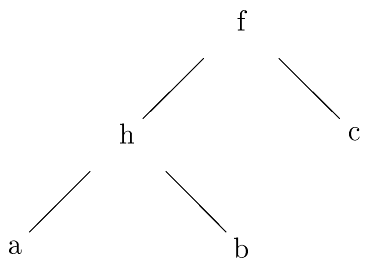

Figure 1: Graphical representation of the tree $f(h(a, b), c)$

from the context.

Finite trees are a suitable graphical representation for terms, as shown in Figure 1. This representation also hints that an alternative presentation of terms is possible, defining first the drawing with all the lines (the tree domain) and then labelling the nodes with operators. This kind of presentation is given in [7, 9].

For any number $i \in \mathcal{N}_+$, let $s_i : T(\Sigma) \rightarrow T(\Sigma)$ be the function such that $s_i(f(t_1, \ldots, t_n)) = t_i$ if $n \geq i$ and $s_i(f(t_1, \ldots, t_n))$ is undefined if $n < i$. We denote $\mathcal{O}$ the set that contains the identity function id of $T(\Sigma)$ and the functions obtained by composing the function s_i. We claim that the elements of $\mathcal{O}$ correctly represent *occurrences*, as they are presented in [7, 9]. The usual presentation is to consider occurrences as strings of natural numbers. Our argument is that the functions s_i represent the elementary strings of one integer and that composing functions represents concatenating strings. We simply denote $u(M)$ the subterm at occurrence u instead of denoting M/u. For this reason, we call *occurrences* the elements of $\mathcal{O}$. We denote $\circ$ the composition operator, with the semantics that $v \circ u(M) = v(u(M))$. For any two occurrences u and v, we note $u \leq v$ if there exists an occurrence w such that $v = w \circ u$, and we say that u is a *prefix* of v (because u is applied first). We concede that this presentation is a bit erroneous in that it gives to much importance to the set $T(\Sigma)$, whereas the usual presentation defines occurrences independently of any language of terms.

Example. *If M is the term $M = f(g(a), h(l(a, b), c), d)$ then the subterm $N = l(a, b)$ appears at the occurrence $s_1 \circ s_2$, which we can also write $N = s_1 \circ s_2(M)$.*

A *tree domain*, D, is a finite subset of $\mathcal{O}$ that verifies the following properties:

1. For any two occurrence u and v such that $u \leq v$, if one has $v \in D$ then one has $u \in \mathcal{D}$,

2. For any two indices i and j such that $i \leq j$ and for any occurrence u, if one has $s_j \circ u \in D$ then one has $s_i \circ u \in D$.

For any term M, we denote $\mathcal{D}(M)$ the set of occurrences that are valid on this term, this set is a tree domain, which we call the *domain* of M.
Example. *If M is the term given in the previous example, then its domain is given by the following equation:*

$$\mathcal{D}(M) = \{id, s_1, s_2, s_3, s_1 \circ s_1, s_1 \circ s_2, s_2 \circ s_2, s_1 \circ s_1 \circ s_2, s_2 \circ s_1 \circ s_2\}.$$

2.2 Reduction Systems.

We consider *reduction systems* in a sense even broader than that of "Combinatory Reduction System" as used by Klop [11]. The class of term manipulations for which origin functions make sense has yet to be precised. In this section, we describe the properties of these manipulation that are fundamental to our study.

Reduction systems are usually studied using an "elementary step" relation, called *reduction*, which we shall denote using a simple arrow $\rightarrow$. We write $M \rightarrow N$ and we say that the term M reduces to N. This notation is not always sufficient for our treatement, as we may need to express *how* M reduces to N. For example, we may consider a term rewriting system containing the following rule r:

$$r: \qquad I(x) \rightarrow x$$

The reduction $I(I(a)) \rightarrow I(a)$ is a legitimate reduction for this system. But there are two ways to go from M to N: the first is to consider that $I(x)$ matches $I(I(a))$, thus performing the rewriting at the top of the term; the other is to consider that $I(x)$ matches $I(a)$, thus performing the reduction on the first subterm of $I(I(a))$. The two reductions are usually considered to be different (and they are for the notion of descendance), and we need extra information to decide which reduction is actually performed. In general, we shall consider that a reduction is given by the amount of information that permits to describe the exact reduction performed between two terms.

Computations in a reduction system are represented by the reflexive transitive closure of the reduction relation, noted $\twoheadrightarrow$, and the elements of this relation are called *derivations*. Here again, we may need extra information to specify exactly a derivation, and we write $M \overset{S}{\twoheadrightarrow} N$, where S is a list whose elements specify exactly all the elementary reductions that compose this derivation. For tree rewriting systems, S is a list of couples (*occurrence, rule*).

Since derivations are elements of a transitive closure, they can be concatenated. If $D = M \overset{S}{\twoheadrightarrow} N$ and $D' = N \overset{S'}{\twoheadrightarrow} P$ are two valid derivations in a reduction system, then $M \overset{S;S'}{\twoheadrightarrow} P$, there exists a derivation that goes from M to P by first performing the steps specified by S, then the steps specified by S', which we denote $D; D'$. For any reduction system $\mathcal{S}$, we denote $Deriv(\mathcal{S})$ the set of all valid derivations in this system.

2.3 Origin Functions.

The intuition in a reduction $M \to N$ is that some parts of the result N descend from parts of M and some parts of N may have been created during the reduction. Origin functions are introduced to describe this notion of descendance.

For any reduction system $\mathcal{S}$, an *origin mapping* $\Omega : Deriv(\mathcal{S}) \to (\mathcal{O} \to \mathcal{O})$ is a function verifying the following constraints:

1. If $D = M \xrightarrow{S} N$ is a derivation in $Deriv(\mathcal{S})$ and $f = \Omega(D)$, then one has $dom(f) \subset \mathcal{D}(N)$ and $codom(f) \in \mathcal{D}(M)$. This constraint is not very important, but it states that the function f expresses the origins of the nodes in N and that these origins are all in M.

2. If D and D' are derivations in $Deriv(\mathcal{S})$ such that $D; D'$ is well defined, then one has $\Omega(D; D') = \Omega(D) \circ \Omega(D')$. This constraint expresses the intuition that the node at occurrence u descends from the node at occurrence w through the derivation $D; D'$ as soon as the node at occurrence u descends from the node at occurrence v by the derivation D' and the node at v descends from the node at w by D.

The images of derivations by Ω are called *origin functions*.

3 Representing Origin Functions.

In this section, we study two methods to represent origin functions. The first method permits to describe a class of regular functions over occurrences, which occur frequently among origin functions. The second method provides a nice way to mix intensional and extensional representations.

3.1 Origins for Projections.

The approach we took makes that occurrences can be perceived as term manipulations. From this point of view, it is natural to associate origin functions to occurrences. If u is an occurrence, when we write $N = u(M)$, this expresses that N is obtained by taking the subterm at occurrence u from M. Let f be the origin function associated to this manipulation. This origin function is very regular. The node at occurrence v in N descends from the node at occurrence v in $u(M)$, that is, the node at occurrence $v \circ u$ in M. If f is the origin function associated to this reduction, we have $f(v) = v \circ u$ for any occurrence $v \in \mathcal{D}(N)$.

We introduce a mapping $\mu : \mathcal{O} \to (\mathcal{O} \to \mathcal{O})$ that summarizes this phenomenon. The mapping μ is defined by the following property:

$$\forall u, v \in \mathcal{O} \quad \mu(u) = f \iff f(v) = v \circ u$$

To simplify notations, we shall note $\mu(u,v)$ for $\mu(u)(v)$.

The mapping μ is a morphism from $\mathcal{O}$ to $(\mathcal{O} \to \mathcal{O})$. Actually, $\mu(id)$ is the identity of $(\mathcal{O} \to \mathcal{O})$. The two spaces have a composition operator, $\circ$. With respect to these operators we have the following property:

$$\forall u,v,w \in \mathcal{O} \quad (\mu(u) \circ \mu(v))(w) = \mu(u)(\mu(v,w)) = w \circ v \circ u = \mu(v \circ u, w)$$

Thus, we have $\mu(u) \circ \mu(v) = \mu(v \circ u)$, μ is *contravariant*. We also have that μ is injective, since $\mu(u, id) = u$.

3.2 General Case.

Not all tree manipulations are as regular as projections, and we want to describe origin functions of arbitrary complexity. However, we can take advantage of the regularity given by the fact that terms are recursively definable.

When specifying the value of a term M, one only needs to provide an operator f and a tuple of terms $(t_1, \dots, t_n)$ and say that $M = f(t_1, \dots, t_n)$. The terms t_1 to t_n may in turn be defined by the same operation, recursively, until one reaches leaves. When providing origins for the nodes in M, one only needs to provide an origin for the head node (at occurrence id) and use origin functions for the subterms t_1 to t_n. These origin functions may in turn be defined by the same operation, recursively.

Denoting $[f_1, \dots, f_n]$ a tuple of functions and $(\mathcal{O} \to \mathcal{O})^*$ the set of tuples of functions over occurrences, we introduce the mapping $c : ((\mathcal{O} \to \mathcal{O})^* \times \mathcal{O}) \to (\mathcal{O} \to \mathcal{O})$ that maps a tuple of functions over occurrences and an occurrence to a function over occurrences defined by the following property:

1. For any tuple $[f_1, \dots, f_n] \in (\mathcal{O} \to \mathcal{O})^*$ and any occurrence u, $c([f_1, \dots, f_n], u)(id) = u$.

2. For any tuple $[f_1, \dots, f_n] \in (\mathcal{O} \to \mathcal{O})^*$, any occurrences u and v, and any index $i \le n$, one has $c([f_1, \dots, f_n], u)(v \circ s_i) = f_i(v)$. This mimics the property that $v \circ s_i(f(M_1, \dots, M_n)) = v(M_i)$.

3. For any tuple $[f_1, \dots, f_n] \in (\mathcal{O} \to \mathcal{O})^*$, any occurrences u and v, and any index $i > n$, one has $c([f_1, \dots, f_n], u)(v \circ s_i)$ is undefined.

Example. *Let us consider the following reduction, taken from the λ-calculus (we use a term representation instead of the usual concrete syntax for λ-terms, to emphasize the tree structure).*

$$@(\lambda(x, @(x, b)), a) \to @(a, b)$$

The origin function for this reduction, f, is defined by the following equations:
$f(id) = s_2 \circ s_1$ $\quad f(s_1) = s_2$ $\quad f(s_2) = s_2 \circ s_2 \circ s_1$
The function f can also be represented by the following notation:

$$f = c([c([], s_2), c([], s_2 \circ s_2 \circ s_1)], s_2 \circ s_1)$$

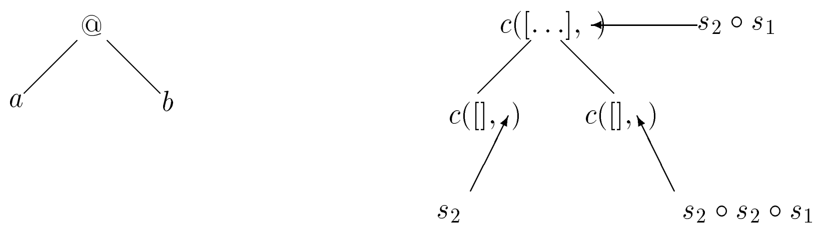

Figure 2: Graphical representation of origin functions.

This representation permits to represent extensively an origin function on a domain, without having to write the elements of the domain. We actually construct a tree isomorphic to the term whose origins are described, labeling each node with the origin for the corresponding occurrence, as is shown in figure 2. Since it is an extensive representation of the origin function, the notation using function c is powerful enough to represent any origin function.

The composition of two functions described using function c is also representable using this function. Actually, one only needs that the function on the right-hand side of the composition operator be represented using c. One obtains the following equality:

$$f \circ c([f_1, \ldots, f_n], u) \quad = \quad c([f \circ f_1, \ldots, f \circ f_n], f(u)) \tag{1}$$

This equality is a direct consequence of the definition of c, but it can be explained intuitively. Let us consider the final term of a manipulation is $P = op(t_1, \ldots, t_n)$, that each of the terms t_i is obtained from a term N by a manipulation whose origin function is f_i, that the origin of the head node in N is u, and that N is obtained from the initial term M by a manipulation whose origin function is f. Then each of the terms t_i is obtained from M by a manipulation whose origin function is $f \circ f_i$ and the origin of the head node of P in M is given by the origin of the node of N at occurrence u, $f(u)$.

The composition of a function represented using c and of a function represented using μ is also easy to describe, this time when the representation using c is on the left-hand side. Actually we have to use the fact that μ is composed of elementary occurrences s_i. One obtains the following equality:

$$c([f_1, \ldots, f_n], u) \circ \mu(v \circ s_i) \quad = \quad f_i \circ \mu(v) \tag{2}$$

Here again, this equality can be described intuitively. If P is obtained from N by taking the subterm at occurrence $v \circ s_i$, if $N = op(t_1, \ldots, t_n)$, and if the j^{th} child of N is obtained from the initial term M by a manipulation whose origin function is f_j, then P is obtained by taking the subterm at occurrence v in the term t_i, whose origins are described by the function f_i.

4 Representing Origins with Trees.

4.1 A Formal Language of Origins.

We introduce the language $Orig$ containing the two sorts $\mathcal{F}$ and $\mathcal{T}$ based on the following abstract syntax, where the index i ranges over $\mathcal{N}_+$:

$$\begin{array}{rcl} \mathcal{F} & = & s_i \mid id \mid \mathcal{F} \circ \mathcal{F} \mid c(\mathcal{T}, \mathcal{F}) \mid nil \\ \mathcal{T} & = & [\mathcal{F}*] \end{array}$$

We call Occ the subset of $Orig$ that contains the trees constructed only with the operators s_i, id, and $\circ$. The operator nil is provided to represent undefined values.

Examples of terms in $Orig$ are $s_1 \circ s_2 \circ id$, $s_1 \circ nil$, and $c([s_1 \circ s_2, nil], c([], id))$.

The terms of Occ can be used to represent occurrences, using the interpretation function $\langle\langle . \rangle\rangle : Occ \to \mathcal{O}$ defined by the following properties:

1. $\langle\langle id \rangle\rangle = id$,
2. $\forall i \in \mathcal{N}_+ \quad \langle\langle s_i \rangle\rangle = s_i$,
3. $\langle\langle u \circ v \rangle\rangle = \langle\langle u \rangle\rangle \circ \langle\langle v \rangle\rangle$.

The terms of $Orig$ can also be used to represent functions over occurrences, using the interpretation function $[\![.]\!] : Orig \to (\mathcal{O} \to \mathcal{O})$ defined by the following properties:

1. $[\![nil]\!] = nil$, where nil is the function that is undefined on any occurrence of $\mathcal{O}$.
2. $[\![id]\!] = 1_{\mathcal{O}}$,
3. $[\![s_i]\!] = \mu(s_i)$,
4. $[\![t_1 \circ t_2]\!] = [\![t_2]\!] \circ [\![t_1]\!]$,
5. $[\![c([t_1, \ldots, t_n], t)]\!] = c([[\![t_1]\!], \ldots, [\![t_n]\!]], [\![t]\!](id))$. Note that this function is undefined on id when $t = nil$.

Notice that the operator $\circ$ in this language is used to represent two operations. One is the composition of occurrences, it corresponds to the concatenation of the paths represented by these occurrences. The other is the composition of functions over occurrences. In the general case, the composition of two origin functions does not correspond to the concatenation of two paths. Remark also that the interpretation of the composition of two terms of $Orig$ by $[\![.]\!]$ corresponds to the composition of the two represented functions, in *reverse* order. This happens because the two interpretations of a composition, as occurrences and as functions over occurrences, are related by the morphism μ, which is contravariant with respect to the operator $\circ$. This property must necessarily appear in one of the interpretations. The reader can check that this permits to have the following property:

$$\forall t \in Occ \quad [\![t]\!] = \mu(\langle\!\langle t \rangle\!\rangle) \tag{3}$$

Second, a tree with the head operator c accepts a tree of the same sort for second child, whereas the mathematical function c we use to describe origin function takes an occurrence as second argument, not an origin function. Here again, this property appears in the interpretation function, that forgets everything about the given origin function, except for the value in id. However, this property gives us the following equation:

$$\forall t \in Occ \quad [\![c([t_1, \ldots, t_n], t)]\!] = c([[\![t_1]\!], \ldots, [\![t_n]\!]], \langle\!\langle t \rangle\!\rangle) \tag{4}$$

Conversely, a tree with the head operator c accept trees of the form $s_i \circ s_j$ as elements of the list it takes as first son, although these trees intuitively denote occurrences, and not functions over occurrences. This property also appears in the interpretation function $[\![\cdot]\!]$, which maps *every* element of *Orig* to a function over occurrences. This property will be used extensively in the following.

4.2 Simplification and Computation.

The main goal of this section is to compute the value of an origin function denoted by an $Orig$ term on a given occurrence, denoted by an Occ term. We notice that this computation is easier for some terms of $Orig$, especially when the given occurrence is id and we describe a canonical rewriting system that permits to transform any term of $Orig$ in one of these terms, denoting the same origin function. We then derive from these remarks a method for computing the value of an origin function on any occurrence.

We say that t is a *head* term, if t verifies one of the following properties:

1. $t \in Occ$ or $t = nil$,
2. $t = c(M, t')$ where $t' \in Occ$ or $t' = nil$.

To compute the value $[\![t]\!](id)$ when t is a head term is easy. In the first case it is denoted by t itself and in the second case it is denoted by t'. This property has a consequence for the computation of the value of an origin function on an occurrence even when this occurrence is not id, since we also have the following result:

$$\forall t \in Orig \quad \forall u \in Occ \quad [\![t]\!](\langle\!\langle u \rangle\!\rangle) = [\![t \circ u]\!](id)$$

We only have to provide a method for transforming the term $t \circ u$ into a head term to enable computing of origins for any occurrence. We achieve this result with the following rewriting system ρ.

$$\begin{array}{lrcl}
\rho_1 : & (t \circ t') \circ t'' & \rightarrow & t \circ (t' \circ t'') \\
\rho_2 : & s_i \circ c([t_1, \ldots, t_i, \ldots, t_n], t) & \rightarrow & t_i \\
\rho_3 : & c(T_1, c(T_2, t)) & \rightarrow & c(T_1, t) \\
\rho_4 : & c([t_1, \ldots, t_n], t) \circ t' & \rightarrow & c([t_1 \circ t', \ldots, t_n \circ t'], t \circ t') \\
\rho_5 : & id \circ t & \rightarrow & t \\
\rho_6 : & nil \circ t & \rightarrow & nil \\
\rho_7 : & t \circ nil & \rightarrow & nil \\
\rho_8 : & c([nil, \ldots, nil], nil) & \rightarrow & nil
\end{array}$$

Actually, rules ρ_2 and ρ_4 are rule schemas, corresponding to every possible combination of i and the length of the tuple between the square brackets $[\ldots]$.

The fact that the represented function is preserved must be proved for every reduction. For rule ρ_1, it comes directly from the associativity of composition, for rule ρ_2 it comes from equation (2), for rule ρ_3 it comes directly from point 5 of the definition of $[\![\ldots]\!]$, for rule ρ_4 it comes from equation (1), for rule ρ_5 it comes directly from the fact that id represents the identity of $\mathcal{O}$, for rules ρ_6 and ρ_7 it comes from the definition of composition, and for rule ρ_8 it comes from the definition of c.

That ρ is terminating is proved by a simple ordering on terms using weight $P : Orig \rightarrow \mathcal{N}$ defined as follows:

- $P(id) = 2$,
- $P(s_i) = 2$,
- $P(t_1 \circ t_2) = P(t_1)^2 P(t_2)$,
- $P(c([t_1, \ldots, t_n], t) = \left(\sum_{1 \leq i \leq n} P(t_i) \right) + P(t)$.
- $P(nil) = 2$.

For any reduction $t \rightarrow t'$ using this rewriting system, one has $P(t') < P(t)$.

All the critical pairs in this rewriting system are confluent. This can be verified automatically using the algorithm of completion of Knuth-Bendix [12]. These properties are enough to prove that this rewriting system is confluent. To complete our study of ρ, we simply need the following property.
Property: *Any term $t \in Orig$ normal for ρ is a head term.*
Proof. First remark that any subterm of a normal term is also normal. We can then prove this property by induction. If $t = t_1 \circ t_2$, then the head operator of t_1 and t_2 cannot be $[\,]$ because of the syntax constraints and it cannot be c or nil because t would not be normal. Then it is necessarily $\circ$, id, or s_i. If the term t_i has a head operator $\circ$ then the induction hypothesis yields $t_i \in Occ$, otherwise t_i is trivially in Occ.

Now, if t is normal and $t = c(M, t')$ then the head operator of t' cannot be $[\,]$ because of the syntax constraints and it cannot be c because t would not be normal for ρ. We can then apply the same reasoning for t' as above and we obtain $t' \in Occ$ or $t' = nil$. $\square$

5 Applications.

In this section, we describe the use of the formal language *Orig* for instrumenting structural semantic descriptions. One example treats in a general way term rewriting systems, whereas the other treats a description of the λ-calculus.

5.1 Linear Term Rewriting Systems.

Term rewriting systems are usually described by rules of the form $\alpha \rightarrow \beta$ where α and β contain variables. The meaning attached to these rules is that any instance of α can be replaced by the corresponding instance of β. To express that a rewriting can be performed in any context, one simply adds rules of the following form, for any operator f and any rank i:

$$\frac{t_i \rightarrow t'_i}{f(t_1, \ldots, t_i, \ldots, t_n) \rightarrow f(t_1, \ldots, t'_i, \ldots, t_n)}$$

Conditional rewriting systems can be obtained by removing some of these rules as in [15], or by adding a predicate P that limits the applicability of the rule, yielding rules of the form:

$$\frac{t_i \rightarrow t'_i \qquad P}{f(t_1, \ldots, t_i, \ldots, t_n) \rightarrow f(t_1, \ldots, t'_i, \ldots, t_n)}$$

As shown by the Centaur experiment [4, 10], these rules can be directly transformed into executable code that allows the computation of derivations. An application of our work is to enhance this code, so that we compute the origin function of a derivation while computing this derivation.

We simply manipulate the rules that describe reductions, replacing uses of the formula $t \rightarrow t'$ (which means t reduces to t') by uses of the formula $t \rightarrow t', w$ (which means t reduces to t' with the origin function w), where the extra parameter w is written in the language *Orig*.

For conditional rules of the form above, the manipulation is systematic, yielding new rules of the following form:

$$\frac{t_i \rightarrow t'_i, w \qquad P}{f(t_1, \ldots, t_i, \ldots, t_n) \rightarrow f(t_1, \ldots, t'_i, \ldots, t_n), c([s_1, \ldots, w \circ s_i, \ldots s_n], id)}$$

The term added in the conclusion, $c([s_1, \ldots, w \circ s_i, \ldots, s_n], id)$ is rather intuitive: the parts s_1, ..., s_j $(j \neq i)$, ..., s_n express that the children that are not affected by the reduction simply come from the corresponding children of the initial term. The part $w \circ s_i$ expresses that t'_i is obtained from t_i by a reduction whose origin function is w and t_i is obtained from the initial term by taking the i^{th} subterm. The part id indicates that the head node of the resulting term comes from the head node of the initial term.

Note that the use of the function μ in the interpretation $[\![.]\!]$ of trees of $Orig$ enables us to express that the origins for the subterm t_j are simply given by the data structure s_j. In this respect, the possibility we have added to use occurrences as origin function is clearly fundamental for this use of origin functions, and it is particularly adapted to structural semantics.

For the axiomatic rules of the form $\alpha \to \beta$, we can also have a systematic treatment, if we restrict our manipulation to linear rewriting systems. Such systems verify the constraint that all the variables in β appear in α and that the variables appearing in α appear only once. For these rules the descendance relation is simple to express: any node that appears under a variable of β in the resulting term descends from the node that appears at the same place relative to the same variable of α in the initial term. This definition is not ambiguous because we are working with linear rewriting systems. This manipulation yields rules like the following one:

$$double(x) \to sum(x,x), c([s_1, s_1], nil)$$

The term added to the reduction, $c([s_1, s_1], nil)$ is simple to understand: both children of the resulting term descend from the only child of the initial term. The head node of the resulting term is created by the reduction, its origin is undefined and represented by nil.

5.2 The λ-calculus.

The λ-calculus is also described using conditional rules, but some of these rules are not based on rewrite formulas. The kernel of the λ-calculus is the β-rule:

$$\beta \qquad (\lambda x.M)N \;\to\; M\{x\backslash N\}$$

The notation $M\{x\backslash N\}$ represents the term M where every free instance of the variable x is replaced by N. This operation must also contain some renaming of the bound variables in M, so as not to catch free variables in N. We compute this operation by introducing an new operator, denoted $[.\backslash.]$ used to represent *substitutions to be done*. The computation is then done using the following conditional rules. (in these rules $FV(M)$ denotes the sets of variables that are free in M):

$$\textit{subst} \qquad x[x\backslash T] \;=\; T$$

$$\textit{diff} \qquad y[x\backslash T] \;=\; y \qquad (y \neq x)$$

$$\textit{app} \qquad \frac{M[x\backslash T] = M' \qquad N[x\backslash T] = N'}{MN[x\backslash T] = M'N'}$$

$$\textit{lambda} \quad \frac{M[y\backslash z] = M' \qquad M'[x\backslash T] = M''}{\lambda y.M[x\backslash T] = \lambda z.M''} \qquad (z \notin FV(M) \cup FV(T))$$

Using this notation, the β-rule must be written as follows:

$$\beta \qquad \frac{M[x\backslash N] = M'}{(\lambda x.M)N \to M'}$$

The language $Orig$ is still adapted to represent origins in the substitution operation. The manipulation yields the following rules:

$$\textit{subst} \qquad x[x\backslash T] \;=\; T, s_3$$

$$\textit{diff} \qquad y[x\backslash T] \;=\; y, s_1 \qquad (y \neq x)$$

$$\textit{app} \qquad \frac{M[x\backslash T] = M', w_1 \qquad N[x\backslash T] = N', w_2}{\begin{array}{c} MN[x\backslash T] = M'N', \\ c([w_1 \circ c([s_1 \circ s_1, s_2, s_3], id), w_2 \circ c([s_2 \circ s_1, s_2, s_3], id)], s_1) \end{array}}$$

$$\textit{lambda} \quad \frac{M[y\backslash z] = M' \qquad M'[x\backslash T] = M'', w}{\begin{array}{c} \lambda y.M[x\backslash T] = \lambda z.M'', \\ c([s_1 \circ s_1, w \circ c([s_2 \circ s_1, s_2, s_3], id)], s_1) \end{array}} \qquad (z \notin FV(M) \cup FV(T))$$

Note how the rule *app* has been manipulated. In the term describing the origin of the result, the functions w_1 and w_2 are composed to functions that describe how $M[x\backslash T]$ and $N[x\backslash T]$ descend from $MN[x\backslash T]$. The same kind of manipulation appears in the rule *lambda*. Note that in this rule the first premise, which only computes a variable renaming, is not used for the origin computation.

This manipulation of the rules for substitution goes along with a manipulation of the rule β that yields the following rule:

$$\frac{M[x\backslash N] = M', w}{(\lambda x.M)N \to M', w \circ c([s_2 \circ s_1, s_1 \circ s_1, s_2], nil)}$$

All these rule manipulations can actually be automatized, as is shown in [2].

6 Conclusion.

In this paper, we have presented a new method for studying descendance and residuals. The original aspects of this method are that it does not use labels, it uses the similarity between concatenation of occurrences and composition of functions, and it uses a canonical rewriting systems to describe residual computations. This approach has two main advantages: first, it permits to produce very compact representation of labeled terms, as it permits to describe labelings *intensionally*; second it permits to describe the process of labeling with very little knowledge of the initial, unlabeled, language of term.

By avoiding the use of labels, this methods saves a good amount of work in that it frees from the task of proving that the labeled reduction system actually describes the same computations as the initial system, through lifting and erasure functions. By separating origin computations from actual reductions of the studied reduction system, it provides a separation of concepts that is profitable in itself. When thinking of mechanically proving properties of reduction systems, it permits to define notions such as "residuals", "developments", or "standardization" independently of the reduction system under study. This has applications in the study of programming languages semantics.

It is also tempting to draw a parallel between conditional rules like those manipulated in this paper and theorems as they are used in proof systems. From this point of view, our work on residuals can also be used at the meta-level of these proof systems, for instance to track the use of hypotheses in a specific proof. This may be useful both for generalizing proofs (removal of unused hypotheses) or debugging proving tactics.

The methods used in this paper permit to describe the simple labelings that are related to descendance. More elaborate labelings have also been developed to study other aspects of reductions [6, 11, 13, 15]. The labels are not simple letters in an alphabet, but rather structured terms, based on extra operators (like underlining in [6, 11, 13]). It is interesting to see how these operators interact with those of our language $Orig$, in order to provide an implementation of these elaborated labelings.

References

[1] H. P. Barendregt, *The Lambda Calculus, Its Syntax and Semantics* North-Holland, 1984.

[2] Y. Bertot, *Une automatisation du Calculs des Résidus en Sémantique Naturelle*, Ph.D Thesis, University of Nice, France, 1991.

[3] Y. Bertot, "Occurrences in Debugger Specifications", *Proceedings of ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, Canada, 1991.

[4] P. Borras et al., "Centaur: the System", *Proceedings of the Third Symposium on Software Development Environments, ACM SIGSOFT'88*, Boston, Massachusetts, 1988.

[5] H. B. Curry, R. Feys, *Combinatory Logic, Vol. I*, North-Holland, 1958.

[6] J. Field, "On Lazyness and Optimality in Lambda Interpreters: Tools for Specification and Analysis", *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, San Francisco, California, 1990.

[7] J. H. Gallier, *Logic For Computer Science*, Harper & Row, 1986.

[8] G. Huet, J.-J. Lévy, "Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems", IRIA Laboria Report 359, 1979.

[9] G. Huet, "Deduction and Computation", *Fundamentals in Artificial Intelligence*, W. Bibel, Ph. Jorrand (editors), Springer Verlag LNCS 232, 1986.

[10] G. Kahn, "Natural Semantics", *Programming of Future Generation Computers*, K. Fuchi, M. Nivat (editors), North-Holland, 1988 (also appears as *INRIA Report no. 601*).

[11] J. W. Klop, *Combinatory Reduction Systems*, Mathematisch Centrum, Amsterdam, 1980.

[12] D. Knuth, P. Bendix, "Simple Word Problems in Universal Algebras", *Computational Problems in Abstract Agebra*, J. Leech (editor), Pergamon, 1970.

[13] J.-J. Lévy, *Réductions Correctes et Optimales dans le Lambda Calcul*, Thèse de Doctorat d'état, University of Paris VII, France, 1978.

[14] J.-J. Lévy, "Optimal Reductions in the Lambda-Calculus", *to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. P. Seldin, J. R. Hindley (editors), Academic Press, 1980.

[15] L. Maranget, "Optimal Derivations in Weak Lambda-Calculi and in Orthogonal Term Rewriting Systems", *Proceedings of the Eightteenth ACM Symposium on Principles of Programming Languages*, Orlando, Florida, 1991, pp. 255–269.