Occurrences in Debugger Specifications

Yves Bertot
INRIA Sophia-Antipolis
2004, Route des Lucioles – b.P. 109
06561 VAlbonne Cedex (France)
bertot@mirsa.inria.fr

Abstract

We describe formal manipulations of programming language semantics that permit execution animation for interpreters. We first study the use of occurrences in the λ-calculus and we describe an implementation of the notion of *residuals*. We then describe applications in the development of interpreters for the lazy λ-calculus and the parallel language Occam.

1. Introduction

Formal descriptions of programming language semantics have already been shown to yield executable specifications of interpreters for these languages [Mini-ML], [Esterel]. However, while the obtained interpreters have the clear advantage of being "certified" implementations, they lack a nice user interface for the very reason that the definition only deals with semantic values. An interpreter can be tranformed into a debugging tool by adding tracing, profiling, or control of execution functionalities. For us, an execution trace is a list of basic instruction calls that describes a history of execution, a profile is a list of statistics of instruction usages, and execution control is the capability to stop program execution at any time to observe the current state of computation.

These three functionalities can easily be improved with the help of the basic facility that we call *subject tracking* in this paper and that permits to relate execution to locations in the program. For example, an execution trace message of the form "instruction I is called with arguments L" can be improved into a message of the form "instruction I is called at position P with arguments L" where P describes a location in the program. This is especially useful if instruction I is used in several places in the program. A profiler without subject tracking actually seems useless, since it is only capable of giving the time spent in basic instructions, without indicating where these basic instructions appear in the program, it is thus unable to show the parts of the program where most of the time is spent. This possible with the help of subject tracking. Also, a bare execution controller can only provide functionalities like "execute one instruction and then stop" or "execute until the execution reaches a function call and then stop", whereas it can not provide functionalities like "execute until the execution reaches location P and then stop" where P is the location of a specific instruction or function call in the program, unless subject tracking is provided.

in : *Programming Language Design and Implementation,* Toronto, Canada, June, 1991.

The problem of relating the execution state to locations in the program is relatively simple in the case of sequential programs, where the semantics uses tree traversals to describe the execution. It is noticeably harder in the case of concurrent programs where the semantics uses tree rewritings to describe execution. As a result of tree rewritings, a part of the original program may appear at several instances in the term that represents the current state of execution. Our problem then corresponds to relating subparts of the rewritten term to the original program.

A solution to this problem is to integrate the relation to the original program within the semantic definition, using occurrences which are the most natural way to designate locations. However, this solution has the drawback of weighing down the work of the designer who produces the formal specification by asking him to deal with features that are not really related to the semantics of the language. The formal manipulations that integrate occurrence computations are shown, in fact, to be systematic and automatizable. Given an implementation of these manipulations, it is then possible to derive an interpreter, which provides program animation, from the dynamic semantics of a programming language and the designer can be freed from this task.

We base our study on the λ-calculus, both because it serves as a foundation for a wide class of programming languages and because it is the simplest "programming language" whose semantics is described using a rewriting system. On the formal side of our study, we provide a natural implementation to the notion of residuals. On the pragmatic side, we show how to use these results in interpreters for the lazy λ-calculus [Lazy] and a subset of the parallel language Occam [Occam].

2. The Basic Material

2.1. *Occurrences*

Programs are trees. For designating locations in a tree we use *occurrences* as described in [TRS], for example. For every natural number k, we consider the function s_k which maps any tree, $op(t_1, \ldots, t_k, \ldots, t_n)$, to its child of rank k, i.e, t_k. We call an *occurrence* any function obtained by composing an arbitrary number of times the functions s_k. The composition operation is denoted $\circ$, we denote its neutral element id, and $\mathcal{O}$ is the set of all the occurrences. Also, we use the notation $u \leq v$ when v and u are in $\mathcal{O}$ and there exists $u' \in \mathcal{O}$ such that $v = u' \circ u$. For example, if we consider the tree $T = f(g(1,2),3)$, we have $s_2(T) = 3$, $s_1(T) = g(1,2)$, and $s_2 \circ s_1(T) = 2$.

For any tree T we call its *domain*, denoted $\mathcal{D}(T)$, the set of occurrences that are valid on this tree. For example, the domain of the tree $T = f(g(1,2),3)$ is the set $\mathcal{D}(T) = \{id, s_1, s_2, s_1 \circ s_1, s_2 \circ s_1\}$.

2.2. *The λ-calculus*

We first formalize the process of *substitution.* Substituting a term N for a variable x in a term M is represented by the formula $M[x\backslash N]$ and is axiomatized using the following inference rules:

$$subst\colon\quad x[x\backslash T] = T$$

$$diff\colon\quad y[x\backslash T] = y \qquad (x \neq y)$$

$$\lambda\colon\quad \frac{M[y\backslash y'] = M' \quad M'[x\backslash T] = M''}{\lambda y.M[x\backslash T] = \lambda y'.M''}$$

(y' is a new variable)

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

The relation of β-reduction $M \to M'$ is axiomatized with the following rules:

$$\beta\colon\quad (\lambda x.M)N \to M[x\backslash N]$$

$$\xi\colon\quad \frac{M \to M'}{\lambda x.M \to \lambda x.M'}$$

$$\mu\colon\quad \frac{M \to M'}{MN \to M'N}$$

$$\nu\colon\quad \frac{N \to N'}{MN \to MN'}$$

The compilation of this specification into Prolog code [Typol] yields a λ-term evaluator.

3. Introducing Occurrences

The formula $M \to M'$ means that there exists an occurrence u such that $u(M) = (\lambda x.T_1)T_2$, $u(M') = T_1[x\backslash T_2]$, and M is equal to M' otherwise. We manipulate the specification to also compute u.

3.1. *Computing the Reduction Occurrence*

We axiomatize the formula $\vdash M \underset{u}{\to} M'$ where M' is the result of reducing M at occurrence u.

$$m_1(\beta)\colon\quad \vdash (\lambda x.M)N \underset{id}{\to} M[x\backslash N]$$

$$m_1(\xi)\colon\quad \frac{\vdash M \underset{u}{\to} M'}{\lambda x.M \underset{u\,\circ\, s_2}{\to} \lambda x.M'}$$

$$m_1(\mu)\colon\quad \frac{\vdash M \underset{u}{\to} M'}{\vdash MN \underset{u\,\circ\, s_1}{\to} M'N}$$

$$m_1(\nu)\colon\quad \frac{\vdash N \underset{u}{\to} N'}{\vdash MN \underset{u\,\circ\, s_2}{\to} MN'}$$

The modification for the rule β is simple. When this rule is applied, its subject is exactly the reduced redex. For the three other rules, a single principle appears. If T is the subterm of the rule's subject S at the occurrence v (expressed by the equation $T = v(S)$) and if the premise expresses that the reduced redex, R, is found in T at the occurrence u (expressed by the equation $R = u(T)$), then this reduced redex is found in the rule's subject at the occurrence $u \circ v$ (obtained from the equations $R = u(v(S)) = u \circ v(S)$). In rule μ, for example, v is equal to s_1.

Assume $\vdash M \underset{v}{\to} M'$, the correctness of v is given by the existence of a term $(\lambda x.T_1)T_2$ such that $v(M) = (\lambda x.T_1)T_2$ and $v(M') = T_1[x\backslash T_2]$ (Proof by induction on the length of v, omitted here).

This manipulation permits an enhancement of our λ-term evaluator, so that we can specify in advance the reduction to perform, by giving the corresponding occurrence.

3.2. *Subject Tracking*

The method we present in this section is an alternative to the previous one. Here, we privilege the subject, S_E, of the formula proved at the bottom of

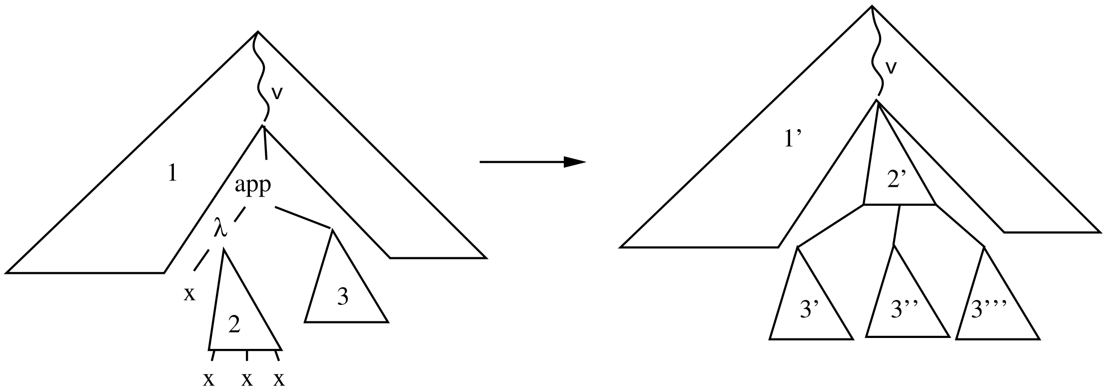

Figure 1. A schematic reduction.

the proof, the *endformula* in Gentzen's terminology [Gentzen]. To the subject of every formula we associate its occurrence in S_E. We axiomatize the formula $\vdash u : M \stackrel{u'}{\rightarrow} M'$. The parameters M and M' are the same as in $\vdash M \rightarrow M'$. The parameters u and u' are such that $M = u(S_E)$ and the reduced redex is $u'(S_E)$. The obtained rules are as follows:

$$m_2(\beta):\quad \vdash u : (\lambda x.M)N \stackrel{u}{\rightarrow} M[x\backslash N]$$

$$m_2(\xi):\quad \frac{\vdash s_2 \circ u : M \stackrel{u'}{\rightarrow} M'}{\vdash u : \lambda x.M \stackrel{u'}{\rightarrow} \lambda x.M'}$$

$$m_2(\mu):\quad \frac{\vdash s_1 \circ u : M \stackrel{u'}{\rightarrow} M'}{\vdash u : MN \stackrel{u'}{\rightarrow} M'N}$$

$$m_2(\nu):\quad \frac{\vdash s_2 \circ u : N \stackrel{u'}{\rightarrow} N'}{\vdash u : MN \stackrel{u'}{\rightarrow} MN'}$$

This manipulation is also systematic and very similar to manipulation m_1. The correctness of the computed occurrence is stated by the following equivalence (Proof by induction on the length of u, omitted):

$$m_2 \vdash v : M \stackrel{u \circ v}{\rightarrow} M' \quad \Leftrightarrow \quad m_1 \vdash M \underset{u}{\rightarrow} M'$$

This equivalence simply means that if M is found in the reference term S_E at the occurrence v and the reduction is done in S_E at occurrence $u \circ v$, then this reduction is actually done in M at occurrence u.

In this new method, the occurrences we use are addresses relative to the reference S_E, instead of being addresses relative to the subject of the corresponding formula.

4. Residual Tracking

In the reduction shown in figure 1, the terms that were in areas 1, 2, and 3 reappear in the areas $1'$, $2'$, and $3'$, $3''$, and $3'''$. We wish to replace this geometric intuition by a systematic analytical treatment.

Let us consider a β-reduction $\vdash M \underset{v}{\rightarrow} M'$, where $v(M) = (\lambda x.T_1)T_2$. The movements of subterms between M and M' are given by a function $origin_{(M,v)} : \mathcal{D}(M') \rightarrow \mathcal{D}(M)$ with the following properties:
disjoint case:

$$u \in \mathcal{D}(M), \quad v \not\leq u \quad \Rightarrow origin_{(M,v)}(u) = u$$

body case: If $u \in \mathcal{D}(T_1)$ and $u(T_1)$ is not a free occurrence of x in T_1 then one has:

$$origin_{(M,v)}(u \circ v) = u \circ s_2 \circ s_1 \circ v$$

argument case: If $u_2 \in \mathcal{D}(T_2)$, $u_1 \in \mathcal{D}(T_1)$, and $u_1(T_1)$ is a free occurrence of the variable x in T_1 then one has:

$$origin_{(M,v)}(u_2 \circ u_1 \circ v) = u_2 \circ s_2 \circ v$$

Remark that the three cases are exclusive, even when $T_1 = x$. We wish now to compute the function *origin*, with the help of a proper data structure.

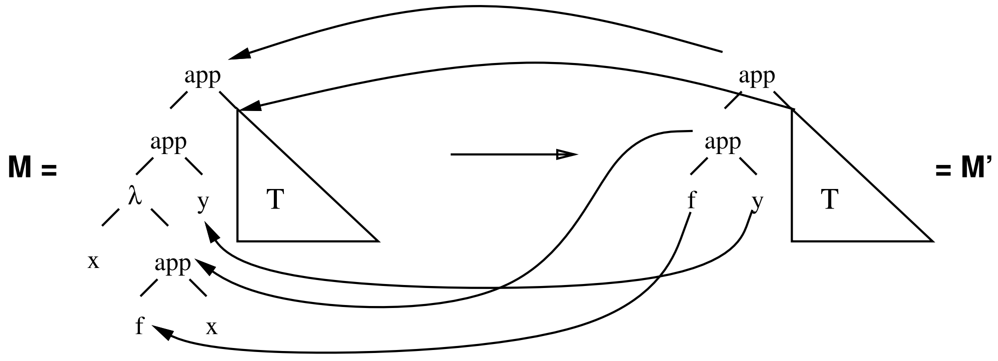

Figure 2. The origin function (curved arrows).

4.1. *Extended Occurrences*

For a given λ-term T we want to encode the values of a function $f : \mathcal{D}(T) \rightarrow S$, for some set of values S. We use a tree of values T' isomorphic to T that carries at each node the value of f. The tree T' is in the set of the trees of values constructed from the elements of S as atomic operators and an operator m of arity 3 (because both λ-abstraction and application are operators of arity 2). Thus, T' has the following properties:

If $u(T) = MN$ or $u(T) = \lambda x.M$ then T' must verify $s_3 \circ u(T') = f(u)$.

If $u(T) = x$ then T' must verify $u(T') = f(u)$.

To any tree of values t is associated a function $value_t$ such that $value_t(u) = v_m$ if $u(t) = m(t_1, t_2, v_m)$ or $u(t) = v_m \in \mathcal{O}$. We want to have $value_t = f$.

In the case of origin functions, the values are themselves occurrences. We consider trees of values as an extension of occurrences and call them *extended occurrences*. Also, we encode the function f such that for all v, $f(v) = v \circ u$ by the occurrence u. To express this abbreviation, the definition of the *value* function associated to a tree t is slightly modified with a new equation: $value_t(u) = v \circ v_m$ if $u = v \circ u'$ and $u'(t) = v_m \in \mathcal{O}$.

In the example given in figure 2, the function $origin_{(M,s_1)}$ is given by the curved arrows that go from M' to M. The reader can check that it is encoded by the extended occurrence $m(m(u_2, s_2 \circ s_1, u_1), s_2, id)$, where $u_1 = s_2 \circ s_1 \circ s_1$ and $u_2 = s_1 \circ u_1$.

4.2. *Computing Extended Occurrences*

We axiomatize formulae of the following two forms:

$$(i) \qquad \vdash u : M \underset{v}{\rightarrow} u' : M'$$

$$(ii) \qquad \vdash u_m : M[x \backslash u_t : T] = u' : M'$$

Formula (i) means the same as $\vdash M \underset{v}{\rightarrow} M'$ for the parameters M, M', and v. For the parameters u and u', we use a reference term S_E as in §3.2; u is the occurrence describing the position of M in S_E ($M = u(S_E)$); u' gives the value of the origin function for the subexpressions of M' as coming from sub-expressions of S_E. The parameter v is an address relative to the subject M. As far as formula (ii) is concerned, it means the same as $M[x \backslash T] = M'$ for the parameters M, x, T, and M'. For the parameters u_m, u_t, and u' it expresses $M = u_m(S_E)$, $T = u_t(S_E)$, and u' is the extended occurrence describing the origin function for M'. The specification uses a function *top* such that $top(t) = value_t(id)$:

$$m_3(\beta) \quad \frac{\vdash s_2 \circ s_1 \circ u : M[x \backslash s_2 \circ u : N] = u' : M'}{\vdash u : (\lambda x.M)N \underset{id}{\rightarrow} u' : M'}$$

$$m_3(\mu): \quad \frac{\vdash s_1 \circ u : M \underset{v}{\rightarrow} u' : M'}{\vdash u : MN \underset{v \circ s_1}{\rightarrow} m(u', s_2 \circ u, top(u)) : M'N}$$

$$m_3(\nu): \quad \frac{\vdash s_2 \circ u : N \underset{v}{\rightarrow} u' : N'}{\vdash u : MN \underset{v \circ s_2}{\rightarrow} m(s_1 \circ u, u', top(u)) : MN'}$$

$$m_3(\xi): \quad \frac{\vdash s_2 \circ u : M \underset{v}{\rightarrow} u' : M'}{\vdash u : \lambda x.M \underset{v \circ s_2}{\rightarrow} m(s_1 \circ u, u', top(u)) : \lambda x.M'}$$

Here again, the manipulation is systematic. The principle uses a notion of *input* and *output* parameters, very close to that introduced in [Typol Ag]. For every formula of the form $\vdash M \rightarrow M'$ we consider the parameter M to be *inherited* and the parameter M' to be *synthesized*. For each rule, the inherited parameters of the conclusion and the synthesized parameters of the premises are *input* parameters. Conversely, the synthesized parameters in the conclusion and the inherited parameters in the premises are *output* parameters. A distinct logical variable is associated to each input parameter and the extended occurrences for the output parameters are then constructed to express that these parameters are made of parts of the input parameters.

We use the same principle to axiomatize the formula (ii) from the rules that describe substitution. In $\vdash M[x\backslash T] = M'$, the inherited parameters are M and T and the synthesized parameter is M'. The principle is not exactly followed for the rule λ to express that the origins are not altered by α-conversion:

$$m_3(\lambda): \quad \frac{\vdash M[y\backslash y'] = M' \qquad \vdash s_2 \circ u : M'[x\backslash u_t : T] = u'' : M''}{\vdash u : \lambda y.M[x\backslash u_t : T] = m(s_1 \circ u, u'', top(u)) : \lambda y'.M''}$$

(y' is a new variable)

We do not give the other rules, for the sake of conciseness. The correctness of the computed data structure is stated as follows:

$$\vdash u : M \overset{v\circ u}{\rightarrow} u' : M'$$
$$\Downarrow$$
$$\forall w \in \mathcal{D}(M') \quad value_{u'}(w) = origin_{(M,v)}(w) \circ u$$

(Proof by induction on the size of the proof tree, omitted.)

4.3. *Residuals*

The usual notion in λ-calculus for tracking motion of subterms is that of *residuals*. A formal description, taken from [Barendregt], extends the language Λ of λ-terms into a language $\Lambda^\star$ by adding a marked operator $\lambda^\star$. In the natural semantics specification, three extra rules account for the marked terms:

$$\beta^\star: \quad (\lambda^\star x.M)N \rightarrow M[x\backslash N]$$

$$\xi^\star: \quad \frac{M \rightarrow M'}{\lambda^\star x.M \rightarrow \lambda^\star x.M'}$$

$$\lambda^\star: \quad \frac{M[y\backslash y'] = M' \quad M'[x\backslash T] = M''}{\lambda^\star y.M[x\backslash T] = \lambda^\star y'.M''}$$

(y' is a new variable)

Given a term M of Λ and a set of occurrences $\mathcal{F}$ designating λ-abstractions in M, we denote $(M, \mathcal{F}) \in \Lambda^\star$ the term where the λ operators given by $\mathcal{F}$ are replaced by $\lambda^\star$ operators.

Given a λ-term M and a family of occurrences $\mathcal{F}$ designating redexes in M, a redex occurrence v, and a λ-term M' such that $M \underset{v}{\rightarrow} M'$, the set of *residuals* of $\mathcal{F}$ through the reduction at occurrence v is the set $\mathcal{F}'$ such that $(M, s_1 \circ \mathcal{F}) \underset{v}{\rightarrow} (M', s_1 \circ \mathcal{F}')$.

The origin function permits to handle residuals in the following sense:

$$(M, \mathcal{F}) \underset{v}{\rightarrow} (M', \mathcal{F}') \wedge \ \vdash u : M \underset{v}{\rightarrow} u' : M'$$
$$\Downarrow$$
$$\mathcal{F}' = \{w \in \mathcal{D}(M') \mid value_{u'}(w) \in \mathcal{F} \circ u\}$$

(Proof by induction on the size of the proof tree, omitted).

4.4. *Composing Reductions*

We now study the transitive closure of β-reduction, represented by formulae of the form $\vdash M \underset{S}{\twoheadrightarrow} M'$, where S is a sequence of occurrences. We use the notation $S_1 \cdot S_2$ for the concatenation of two sequences, the notation $v \cdot S$ for the sequence whose first element is the occurrence v and the rest is the sequence S, and the notation ε for the empty sequence. The transitive closure is axiomatized using the following two rules:

$$continue: \quad \frac{\vdash M \underset{v}{\rightarrow} M' \qquad \vdash M' \underset{T}{\twoheadrightarrow} M''}{\vdash M \underset{v \cdot T}{\twoheadrightarrow} M''}$$

$$stop: \quad \vdash M \underset{\varepsilon}{\twoheadrightarrow} M$$

To a derivation $M \underset{S}{\twoheadrightarrow} M'$ we associate an origin function defined by composing the origin function of all the elementary steps:

$$origin_{(M,\varepsilon)} = Id_{\mathcal{D}(M)}$$

$$origin_{(M,v.S)} = origin_{(M',S)} \circ origin_{(M,v)}$$

(if $M \underset{v}{\rightarrow} M'$)

We use the composition of occurrences to represent the composition of origin functions. This leads to extra equations, which correspond to the associativity of function composition (u, v_1, and v_2 are extended occurrences, v_m is a pure occurrence):

$$s_1 \circ m(v_1, v_2, v_m) = v_1 \qquad s_2 \circ m(v_1, v_2, v_m) = v_2$$
$$m(v_1, v_2, v_m) \circ v = m(v_1 \circ v, v_2 \circ v, top(v_m \circ v))$$

where $top(t) = value_t(id)$. Thus, for any pure occurrence u and any extended occurrence t, we have $value_t(u) = value_{u \circ t}(id)$. In an actual implementation, these equations must be used as soon as possible to reduce the size of the extended occurrences. Without simplification, the data structure can take on a size which is proportional to the length of the derivation; with simplification the size can be kept proportional to the size of the resulting λ-term.

We apply m_3 to the rules *continue* and *stop*, with the specification that in $\vdash M \twoheadrightarrow M'$, M is inherited

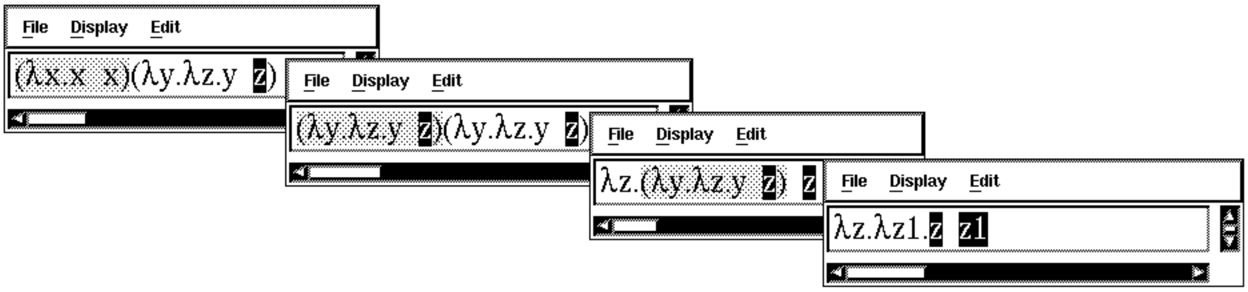

Figure 3. An example automated reduction.

and M' is synthesized. In the rules we obtain, extended occurrences are truly used as generalization of pure occurrences. A proof tree for $\vdash u : M \underset{S}{\twoheadrightarrow} u' : M'$ may contain formulae of the form $\vdash v : N \underset{w}{\rightarrow} v' : N'$ where w is a pure occurrence but both v and v' are extended occurrences.

The correspondence with residuals is still valid, as stated by the following property:

$$(M, \mathcal{F}) \underset{S}{\twoheadrightarrow} (M', \mathcal{F}') \wedge \ \vdash id : M \underset{S}{\twoheadrightarrow} u' : M'$$
$$\Downarrow$$
$$\mathcal{F}' = \{w \in \mathcal{D}(M') \quad | \quad value_{u'}(w) \in \mathcal{F}\}$$

(Proof by induction on the length of S, omitted).

Given a set $\mathcal{F}$ of redex occurrences in a term M, a *development* is a derivation where every elementary step reduces a redex that is a residual of one of the redexes given by $\mathcal{F}$. A *complete* development is obtained when the resulting term contains no residual of the redexes given by $\mathcal{F}$ any more. Using the results of this section, we can have our evaluator perform complete developments for us.

4.5. *Computer Implementation*

The specification obtained through m_3 is used in an actual implementation within the Centaur system [Centaur]. In this implementation, we can construct λ-terms using a syntax directed editor, select redexes, and compute reductions, developments, and residuals from these λ-terms and selections.

The four windows given in figure 3 describe successive reductions of a λ-term. In the three first windows the reduced sub-expression is shown in gray. We also selected a sub-expression of the first term in reverse video and all its residuals through the successive reductions have been computed and shown in reverse video.

5. Application to Functional Languages

Functional programming languages are derived from the λ-calculus. Basically, one adds data structures with specific operators and rewrite rules for numerals, booleans, pairs, etc. and one restricts the reduction relation by choosing an evaluation strategy. Here, we take as an example the lazy λ-calculus, where one only considers closed terms, one does not reduce under a λ-abstraction, and the parameter of a function call is not evaluated at binding time, but only when needed. Following this strategy, evaluating a λ-abstraction does nothing and evaluating an application begins by evaluating the function part until a λ-abstraction is obtained, then performs the β-reduction, and continues with the result, as stated in the following rules taken from [Lazy].

$$ev_\lambda : \quad \vdash \lambda x.M \twoheadrightarrow \lambda x.M$$

ev_{app} :
$$\frac{\vdash M \twoheadrightarrow \lambda x.P \quad \vdash P[x\backslash N] = P' \quad \vdash P' \twoheadrightarrow R}{\vdash MN \twoheadrightarrow R}$$

The reader can check that these rules actually describe a subset of the derivation relation described in §4.4.

As shown in [Abstract], substitution is a high level operation and can be replaced by an *environment*. This corresponds to delaying the substitution mechanism until it is actually needed, i. e., when one looks at a variable, it should be substituted. An environment is a list of substitutions (pairs of a variable and a value). When using environment, the correct value of a λ-term, E, can be represented by the pair (E, ρ), called a *closure*, of E and the environment ρ that gives the value of the free variables in E. The evaluation with an environment is given by the formula $\rho \vdash E \Rightarrow R$, where E evaluates to R and ρ gives the values of the free variables in E and R:

$$subst_\rho : \quad \frac{\rho' \vdash N \Rightarrow R}{\rho[x\backslash(N, \rho')] \vdash x \Rightarrow R}$$

$$diff_\rho : \quad \frac{\rho \vdash x \Rightarrow R}{\rho[y\backslash N] \vdash x \Rightarrow R} \quad (x \neq y)$$

$$eval_\lambda : \quad \rho \vdash \lambda x.M \Rightarrow (\lambda x.M, \rho)$$

$eval_{app}$:
$$\frac{\rho \vdash M \Rightarrow (\lambda x.P, \rho') \quad \rho'[x\backslash(N, \rho)] \vdash P \Rightarrow R}{\rho \vdash MN \Rightarrow R}$$

Also, a *fixpoint* operator is usually added, with the semantics described by $fixM = M(fixM)$. This fixpoint permits the definition of recursive functions. The evaluation rule for this operator is as follows:

$$eval_{fix}: \quad \rho \vdash fix\lambda f.P \Rightarrow (P, \rho[f\backslash(fix\lambda f.P, \rho)])$$

5.1. *Subject Tracking*

What should be understood from the previous section is that a formal description of a functional programming language using environments as in [Mini-ML] is actually derivable from the basic rules of the λ-calculus. Actually, if we replace every occurrence of $fixM$ by YM where $Y = \lambda f.(\lambda x.xx)(\lambda x.f(xx)))$ we can establish the following correspondence between formulae:

$$\emptyset \vdash M \Rightarrow R \quad \Rightarrow \quad \vdash M \twoheadrightarrow R$$

We can derive occurrence computations for the functional programming language from the ones we have for the λ-calculus. We first obtain two rules $m(ev_\lambda)$ and $m(ev_{app})$ that correspond to ev_λ and ev_{app}. For conciseness we only give the latter:

$$m(ev_{app}): \quad \frac{\begin{array}{c}\vdash s_1 \circ u : M \twoheadrightarrow u_l : \lambda x.P \\ s_2 \circ u_l : P[x\backslash s_2 \circ u : N] = v : P' \\ \vdash v : P' \twoheadrightarrow u' : R\end{array}}{\vdash u : MN \twoheadrightarrow u' : R}$$

Already from this rule, we see that the environment will no longer be a list of pairs, but a list of triples $[x\backslash u_n : N]$, where the extra field corresponds to the parameter u_n added by manipulation m_3 in the formula $u : M[x\backslash u_n : N] = u' : M'$ that used to describe substitution with subject tracking. For closures, the interpretation expresses that the closure (N, ρ) corresponds to the term N where the substitutions given by ρ have to be applied. We consider that such a structure has the same origin as its first child N. We can now axiomatize the formula $\rho \vdash u : M \Rightarrow u' : R$ using the following rules:

$$m(subst_\rho): \quad \frac{\rho' \vdash u_n : N \Rightarrow u' : R}{\rho[x\backslash u_n : (N, \rho')] \vdash u : x \Rightarrow u' : R}$$

$$m(diff_\rho): \quad \frac{\rho \vdash u : x \Rightarrow u' : R}{\rho[y\backslash u_v : V] \vdash u : x \Rightarrow u' : R} \quad (x \neq y)$$

$$m(eval_\lambda): \quad \rho \vdash u : \lambda x.M \Rightarrow u : (\lambda x.M, \rho)$$

$m(eval_{app})$:

$$\frac{\begin{array}{c}\rho \vdash s_1 \circ u : M \Rightarrow u_\lambda : (\lambda x.P, \rho') \\ \rho'[x\backslash s_2 \circ u : (N, \rho)] \vdash s_2 \circ u_\lambda : P \Rightarrow u' : R\end{array}}{\rho \vdash u : MN \Rightarrow u' : R}$$

$m(eval_{fix})$:

$$\rho \vdash u : fix\lambda f.P \Rightarrow s_2 \circ s_1 \circ u : (P, \rho[f\backslash u : (fix\lambda f.P, \rho)])$$

5.2. *Automatic Manipulation of the Rules*

When presenting manipulation m_3, we used a specification of *inherited* and *synthesized* expression in the logical formulae that appeared in the rules. In formula $\rho \vdash M \Rightarrow R$ we see that the expression ρ is not exactly treated like an inherited expression. The difference is that only one field of the pairs contained in the environment is considered as an input parameter and the corresponding extra parameter is added in the pair itself. If we want to automatize the manipulation m we must add a possibility to specify this feature. Thus, the specification will not only contain *inherited* and *synthesized* expressions, but also expressions whose mode is represented by a structure. For example, environments receive the specification $(unspecified, inherited)\star$, thus expressing that they are lists of terms of arity 2, where only the second child should be treated as inherited. This also expresses that the manipulation will increase their arity to 3.

Another feature we need to specify is the behavior of some constructors like the closure constructor. If an occurrence is attached to closure of the form (M, ρ), we want to express that this occurrence applies directly to the first field of the closure, M, as described by the following equation:

$$u\big((M, \rho)\big) = u(M)$$

Let us take for example the rule $eval_{app}$:

$$\frac{\rho \vdash M \Rightarrow (\lambda x.P, \rho') \quad \rho'[x\backslash(N, \rho)] \vdash P \Rightarrow R}{\rho \vdash MN \Rightarrow R}$$

The mode repartition for the formula $\rho \vdash M \Rightarrow R$ is that ρ has the mode $(unspecified, inherited)\star$, M has the mode *inherited* and R has the mode *synthesized*. Thus, the input expressions are MN in the conclusion and $\lambda x.P$ and R in the premises; the output expressions are R in the conclusion and M, N, and P in the premises. The computation of occurrences takes place as in §3.2 and yields the rule $m(eval)$ given above.

5.3. *Program Animation*

When compiling an inference system into a Prolog program, logic formulae are represented by predicates and inference rules are represented by clauses, where the conclusion of the rule is represented by the clause head and the premises are represented by procedure calls (predicates) in the clause tail. For example, the logic formula $\rho \vdash u : N \Rightarrow u' : R$ can be represented by the prolog predicate $eval(\rho, u, N, u', R)$,

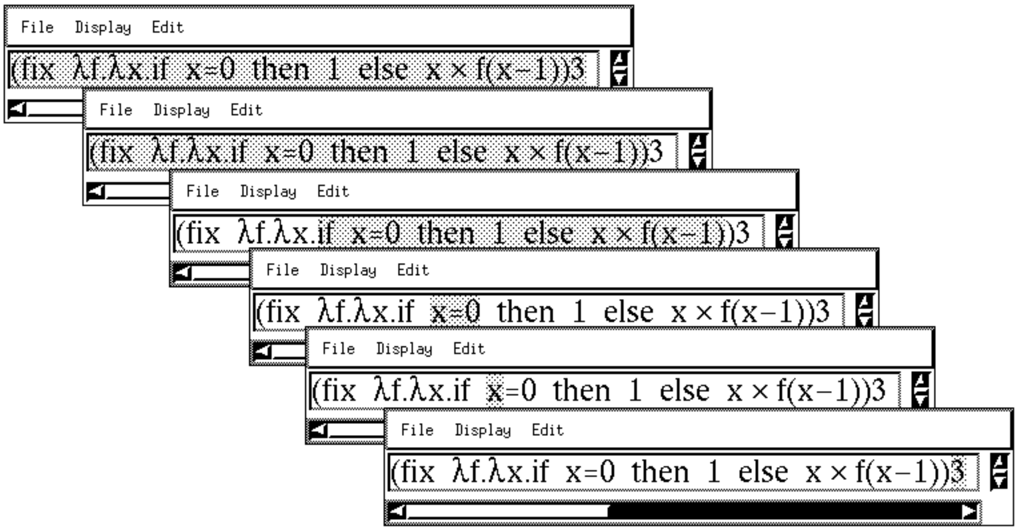

Figure 4. Execution animation of a small functional program.

and the rule $m(subst_\rho)$ is translated into the following clause:

$$eval(cons(\rho, triple(x, u_n, clos(N, \rho'))), u, x, u', R) :- \\ \quad eval(\rho', u_n, N, u', R).$$

We can also add predicates in the clause, whose effect is not to change the result of the execution, but to control or observe it. Thus, the actual translation of $m(subst_\rho)$ may be the following clause:

$$eval(cons(\rho, triple(x, u_n, clos(N, \rho'))), u, x, u', R) :- \\ \quad control(subst_\rho, eval(cons(\rho, triple\ \ldots\ , R)), \\ \quad eval(\rho', u_n, N, u', R).$$

In this clause, we have added a call to the procedure *control*, that receives two arguments. The first argument tells which rule is currently applied, the second argument tells the question this rule attempts to solve. We use this second parameter as a simplified representation of the current state of execution.

The predicate *control* itself can be defined by a clause of the following form:

$control(rule, question) :-$
< animation part >
< user interaction part >
< tracing part > .

In the animation part, we can use the occurrences in the program to change the aspect of the corresponding part of the program, either by underlining or changing the display colors. For example, we know that if the applied rule is $m(subst_\rho)$ then the question is a predicate of operator *eval* whose second parameter, u, indicates the origin of the evaluated expression (a variable in this case).

In the user interaction part, we can perform some execution control. For example, the interpreter may maintain a list of program locations which corresponds to breakpoints. We only have to compare the origin of the evaluated expression with the occurrences of breakpoints. If there is a match, then we can interrupt the execution and let the user observe the current state of execution, represented by the question, with appropriate tools. If there is no match, the execution continues without any telling the user.

In the tracing part, we can use the information given in the question to perform side effects. In the case of a real trace, these side effects are outputting the relevant information in a file (for example to express that a variable is being evaluated); this information can use the occurrences that appear in the question (to express which instance of the variable is evaluated). In the case of a profiler, the origins that appear in the question are important to describe the part of the program that is currently executed.

For example, figure 4 gives successive snapshots of a window where the execution of a little program is animated. At each step, the evaluator shows in the program the expression that it is going to evaluate. Notice that the expression 3 is evaluated only when one needs to evaluate the variable x as is normal with lazy evaluation.

6. Application to Parallel Languages

The technology presented so far can also be adapted to parallel programming languages. We take for example a subset of the language Occam [Occam], where we keep only the parallel construct **par**, the sequence construct **seq** and the selection construct **alt**. Processes can communicate through *channels* using output commands (*channel*!*value*) and input

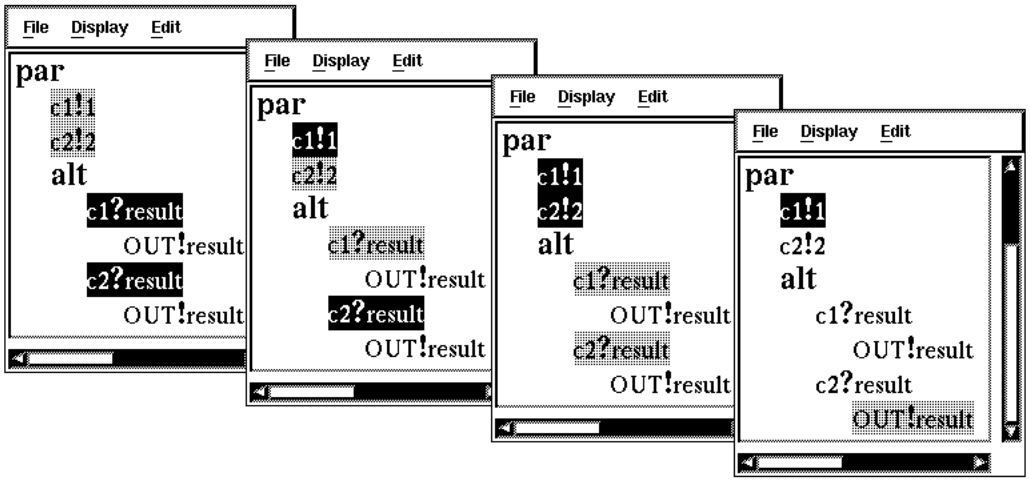

Figure 5. Execution animation of a small parallel program.

commands (*channel?input*).

The semantics of this language is specified using the transitive closure of a one-step reduction, as we already saw for the λ-calculus. This semantics also uses an environment to store the values of variables and the state of communication channels. We axiomatize the formula $\rho \vdash P \Rightarrow \rho'$ which expresses that the execution of the program P transforms the environment ρ into the environment ρ' and the formula $\rho \vdash P \rightarrow P', \rho'$ which expresses that a single step reduction of the program P in the environment ρ yields a new program P' and an environment ρ'. The rules for complete execution are the following ones:

$$\textit{step:} \quad \frac{\rho \vdash P \rightarrow P', \rho' \quad \rho' \vdash P' \Rightarrow \rho''}{\rho \vdash P \Rightarrow \rho''}$$

$$\textit{end:} \quad \rho \vdash P \Rightarrow \rho \qquad (P \text{ is finished})$$

The rules for one step execution are given below. To implement the rendez-vous implied by the channel communication, we introduce two auxiliary constructs, **sync** and **nothing**. The semantics of these constructs is given by *termination rules* which we do not give here for the sake of simplicity. These rules are also used for the semantics of the sequence. They simply express that there is nothing left to do in a given process, and that no output expression is waiting for a rendez-vous. For environments, the notation $\rho[c] = i$ expresses that the environment ρ contains the value i for the channel c, $\rho \leftarrow [c, i]$ denotes the environment that contains the same values as ρ except that the channel c has the value i, and $\rho \leftarrow (v, i)$ has the same meaning for a variable v.

$$\frac{\rho \vdash P_1 \rightarrow P_1', \rho'}{\rho \vdash \mathbf{par} P_1 P_2 \rightarrow \mathbf{par} P_1' P_2, \rho'}$$

$$\frac{\rho \vdash P_2 \rightarrow P_2', \rho'}{\rho \vdash \mathbf{par} P_1 P_2 \rightarrow \mathbf{par} P_1 P_2', \rho'}$$

$$\frac{\rho \vdash P_1 \rightarrow P_1', \rho'}{\rho \vdash \mathbf{seq} P_1 P_2 \rightarrow \mathbf{seq} P_1' P_2, \rho'}$$

$$\frac{\rho \vdash P_2 \rightarrow P_2', \rho'}{\rho \vdash \mathbf{seq} P_1 P_2 \rightarrow P_2', \rho'} \quad (P_1 \text{ has terminated in } \rho)$$

$$\rho \vdash c!i \rightarrow \mathbf{sync}(c), \{\rho \leftarrow [c, i]\}$$

$$\frac{\rho[c] = i}{\rho \vdash c?v \rightarrow \mathbf{nothing}, \rho \leftarrow [c, \emptyset](v, i)}$$

$$\frac{\rho \vdash C_1?v_1 \rightarrow \mathbf{nothing}, \rho'}{\rho \vdash \mathbf{alt}\ C_1?v_1\ P_1\ C_2?v_2\ P_2 \rightarrow P_1, \rho'}$$

$$\frac{\rho \vdash C_2?v_2 \rightarrow \mathbf{nothing}, \rho'}{\rho \vdash \mathbf{alt}\ C_1?v_1\ P_1\ C_2?v_2\ P_2 \rightarrow P_2, \rho'}$$

The manipulation m_3 can be directly applied to this semantics. One only has to take care of the existence of constructs of different arity.

6.1. *Visualizing Parallel Execution*

The semantics given above is actually used to produce an interpreter for this subset of Occam. The results of this paper permit to enhance this interpreter in two directions. First, we can animate the execution by showing all the elementary instructions that can be executed at each step of execution. Second, we can enable the user to guide the parallel execution by stating which branch has to be chosen, when a non-deterministic choice appears.

To perform these enhancements, we first have to apply the manipulations m et m_1 to the rules for one step execution, and the manipulation m to the rules for complete execution. The rule *Step*$'$ obtained after these manipulations has the following form :

$$\frac{\rho \vdash u : P \xrightarrow{v} u' : P', \rho' \quad \rho' \vdash u' : P' \Rightarrow \rho''}{\rho \vdash u : P \Rightarrow \rho''}$$

When translating this rule into a Prolog clause, we can add a procedure call, as we did for the rules describing the functional programming language. The translation of the rule *step* thus becomes the following prolog clause:

$$
\begin{aligned}
&exec(\rho, u, P, \rho') :- \\
&\quad control(step, exec(\rho, u, P, \rho'), v), \\
&\quad one\_step(\rho, u, P, v, u', P'), \\
&\quad exec(\rho', u', P', \rho'').
\end{aligned}
$$

The animation part can rely on an observation program that can be deduced from the language's dynamic semantics by a method we already describe in [Esterel]. This observation program permits to show two categories of locations in the executed program. The first category contains the redexes, that is, the elementary commands that can be executed next. The second category contains the elementary instructions that are waiting for a rendez-vous. We can then enable the user to choose among the occurrences of the first category the occurrence v where it wants the execution to be performed.

The windows given in figure 5 are successive snapshots of a window where the execution of a small program is animated. At each step, the instructions that are executable are shown in grey and the instructions that are waiting for a rendez-vous are shown in reverse-video. Notice that our implementation of synchronized communication states that an output command is waiting for a rendez-vous only after it has written in a channel and as long as the value is not read by an input command.

7. Discussion

We have shown that occurrence computations were a central component of debugging tools, we have described a formal treatement of these computations for the λ-calculus, and we have shown how this work can be applied to a small functional language and a small subset of the parallel language Occam.

An important feature of this work is that we use formal machinery to solve a practical problem. We believe this can be used to develop a general methodology for the implementation of debuggers and possibly to create a system that would permit the direct creation of such debugging tools from a language's formal definition. The implementation of debuggers has until now been done in an *ad hoc* manner.

Further work can be done in two very different directions. On the formal side, it is important to automatize the manipulations we have described in this paper and to give a precise meaning to the computed origin functions. We can also try to address more complicated labelling schemes, such as the ones used in [Optimal] or [Weak]. Such labelling schemes also have practical applications, as they permit to describe the history of parts of the execution state.

On the practical side, the full range of applications for occurrence computations has yet to be explored, and real-size debuggers, tracers, or profilers have to be implemented using these techniques, so that we can find other parts that could be shared from one tool to the other, or from one language to the other.

Acknowledgements

I am very grateful to A. Hirschowitz, G. Kahn, and J.J. Lévy for their numerous discussions and suggestions on early drafts of this paper.

References

[Abstract] J. Hannan, D. Miller, "From Operational Semantics to Abstract Machines: Preliminary Results", *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, June 1990, pp. 323-332.

[Barendregt] H. Barendregt, *The Lambda Calculus*, Studies in Logic, North Holland (1984).

[Centaur] P. Borras et al., "CENTAUR: the system", *Proceedings of the ACM SIGSOFT'88: Third Symposium on Software Development Environments*, Boston, Massachusetts, November 1988. (Also appears as *INRIA Research Report no. 777.*)

[Esterel] Y. Bertot, "Implementation of an Interpreter for a Parallel Language in Centaur", *Proceedings of the European Symposium On Programming*, Copenhagen, Denmark, May 1990, pp. 57–69. (Also appears as *INRIA Research Report no. 1076.*)

[Gentzen] M. Szabo, *The Collected Papers of Gerhard Gentzen*, Studies in Logic, North Holland, (1969).

[Lazy] S. Abramsky, "The Lazy Lambda Calculus", *Declarative Programming*, D. Turner (Editor), Addison Wesley (1988).

[Mini-ML] D. Clément, J. Despeyroux, T. Despeyroux, G. Kahn, "A Simple Applicative Language: Mini-ML", *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, Cambridge, Massachusetts, August 1986.

[Occam] INMOS Limited, *OCCAM Programming Manual*, Prentice Hall International Series in Computer Science (1984).

[TRS] G. Boudol, "Computational semantics of term rewriting systems", *Algebraic Methods in Semantics*, M. Nivat, J. Reynolds (Editors), Cambridge University Press (1985).

[Typol] T. Despeyroux, "Executable Specification of Static Semantics" *Proceedings of the International Symposium on Semantics of Data Types*, June 1984, Sophia-Antipolis, France, Springer-Verlag LNCS 173.

[Typol Ag] I. Attali, P. Franchi-Zannettacci, "Unification-free Execution of TYPOL Programs by Semantic Attributes Evaluation", *Proceedings of the Fifth International Conference Symposium on Logic Programming*, Seattle, The MIT Press, August 1988.

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