Occurences in debugger specifications

Yves Bertot

UNITÉ DE RECHERCHE
NRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1350

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

OCCURRENCES IN DEBUGGER SPECIFICATIONS

Yves BERTOT

Décembre 1990

RR_1350

Occurrences in Debugger Specifications

Yves Bertot
INRIA Sophia-Antipolis

Abstract

We describe formal manipulations of programming languages semantics that permit the integration of 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.

Keywords: Programming Languages, Dynamic Semantics, Animation, Occurrences, λ-calculus.

Utilisation d'Occurrences dans la Spécification D'outils de Mise au Point

résumé

Nous décrivons des manipulations formelles de spécifications sémantiques qui permettent d'intégrer l'animation de l'exécution à des interprètes. Nous étudions d'abord le λ-calcul et nous décrivons une implémentation de la notion de résidu. Nous décrivons ensuite des application de ce travail dans le développement d'interprète pour le λ-calcul paresseux et le langage parallèle Occam.

Mots Clés : Langages de Programmation, Sémantique Dynamique, Animation, Occurrences, λ-calcul.

Occurrences in Debugger Specifications

Yves Bertot
INRIA Sophia-Antipolis

We describe formal manipulations of programming languages semantics that permit the integration of 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. Transforming an interpreter into a debugging tool requires two basic facilities. First, one needs to relate the current state of execution with locations in the program; second, one needs to control the pace of execution. These facilities permit program animation and breakpoints.

In this paper, we address the first of these problems: relating execution to locations in the program. This problem is relatively simple in the case of sequential programs, where the semantics uses tree traversals to describe 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 that are the most natural way to designate locations. However, this solution has the drawback of weighing down the formal specification by introducing features that do not really deal with the semantics of the language. The formal manipulations that integrate occurrence computations are shown to be in fact 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.

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:

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

$$\lambda:\quad \frac{M[y\backslash y'] = M' \quad M'[x\backslash T] = M''}{\lambda y.M[x\backslash T] = \lambda y'.M''} \quad (y' \text{ is a new variable})$$

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

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

$$\beta:\quad (\lambda x.M)N \rightarrow M[x\backslash N] \qquad\qquad \xi:\quad \frac{M \rightarrow M'}{\lambda x.M \rightarrow \lambda x.M'}$$

$$\mu:\quad \frac{M \rightarrow M'}{MN \rightarrow M'N} \qquad\qquad \nu:\quad \frac{N \rightarrow N'}{MN \rightarrow MN'}$$

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

3. Introducing Occurrences

The formula $M \rightarrow M'$ 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 $\vdash M \underset{u}{\rightarrow} M'$ where M' is the result of reducing M at occurrence u.

$$m_1(\beta):\quad \vdash (\lambda x.M)N \underset{id}{\rightarrow} M[x\backslash N] \qquad\qquad m_1(\xi):\quad \frac{\vdash M \underset{u}{\rightarrow} M'}{\lambda x.M \underset{u \circ s_2}{\rightarrow} \lambda x.M'}$$

$$m_1(\mu):\quad \frac{\vdash M \underset{u}{\rightarrow} M'}{\vdash MN \underset{u \circ s_1}{\rightarrow} M'N} \qquad\qquad m_1(\nu):\quad \frac{\vdash N \underset{u}{\rightarrow} N'}{\vdash MN \underset{u \circ s_2}{\rightarrow} MN'}$$

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 $\vdash M \underset{v}{\rightarrow} M'$, 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 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 $\vdash u : M \overset{u'}{\rightarrow} M'$. 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:

$$m_2(\beta):\ \vdash u : (\lambda x.M)N \overset{u}{\rightarrow} M[x\backslash N] \qquad m_2(\xi):\ \frac{\vdash s_2 \circ u : M \overset{u'}{\rightarrow} M'}{\vdash u : \lambda x.M \overset{u'}{\rightarrow} \lambda x.M'}$$

$$m_2(\mu):\ \frac{\vdash s_1 \circ u : M \overset{u'}{\rightarrow} M'}{\vdash u : MN \overset{u'}{\rightarrow} M'N} \qquad m_2(\nu):\ \frac{\vdash s_2 \circ u : N \overset{u'}{\rightarrow} N'}{\vdash u : MN \overset{u'}{\rightarrow} MN'}$$

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):

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

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

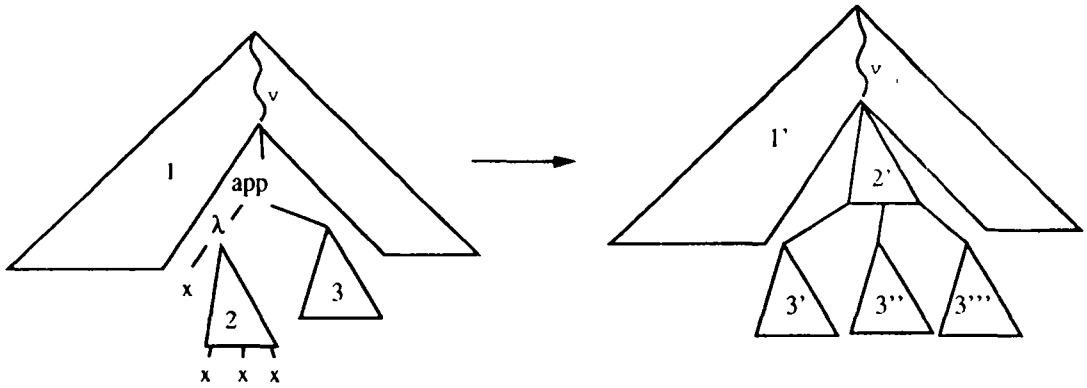

Figure 1. A schematic reduction.

body case. $u \in \mathcal{D}(T_1)$, $u(T_1)$ is not a free occurrence of x in $T_1 \Rightarrow origin_{(M,v)}(u \circ v) = u \circ s_2 \circ s_1 \circ v$.

argument case. $u_2 \in \mathcal{D}(T_2)$, $u_1 \in \mathcal{D}(T_1)$, $u_1(T_1)$ is a free occurrence of the variable x in T_1 $\Rightarrow origin_{(M,v)}(u_2 \circ u_1 \circ v) = u_2 \circ s_2 \circ v$.

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.

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 of 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:

$$(i) \qquad \vdash u : M \underset{v}{\rightarrow} u' : M' \qquad\qquad (ii) \qquad \vdash u_m : M[x \backslash u_t : T] = u' : M'$$

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

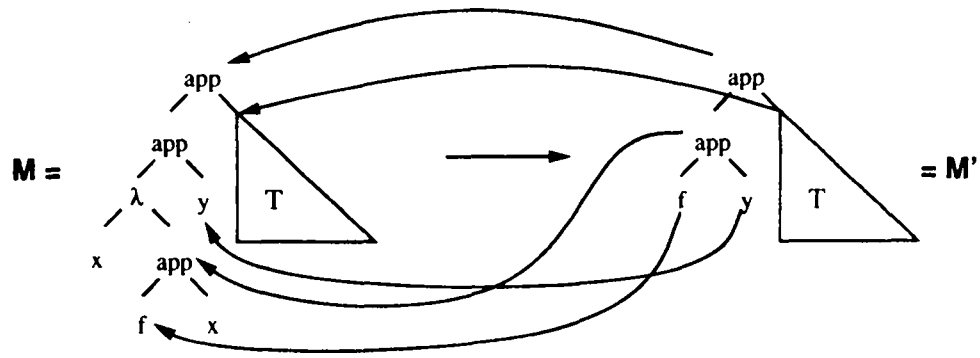

Figure 2. The origin function (curved arrows).

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 $top(t) = value_t(id)$:

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

$$m_3(\mu): \quad \frac{\vdash s_1 \circ u : M \underset{v}{\rightarrow} u' : M'}{\vdash u : MN \underset{vos_1}{\rightarrow} m(u', s_2 \circ u, top(u)) : M'N}$$

$$m_3(\nu): \quad \frac{\vdash s_2 \circ u : N \underset{v}{\rightarrow} u' : N'}{\vdash u : MN \underset{vos_2}{\rightarrow} m(s_1 \circ u, u', top(u)) : MN'}$$

$$m_3(\xi): \quad \frac{\vdash s_2 \circ u : M \underset{v}{\rightarrow} u' : M'}{\vdash u : \lambda x.M \underset{vos_2}{\rightarrow} m(s_1 \circ u, u', top(u)) : \lambda x.M'}$$

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:

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

(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:

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

(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 Λ^* by adding a marked operator λ^*. In the natural semantics specification, three extra rules account for the marked terms:

$$\beta^* : \quad (\lambda^* x.M)N \rightarrow M[x\backslash N] \qquad\qquad \xi^* : \quad \frac{M \rightarrow M'}{\lambda^* x.M \rightarrow \lambda^* x.M'}$$

$$\lambda^* : \quad \frac{M[y\backslash y'] = M' \quad M'[x\backslash T] = M''}{\lambda^* y.M[x\backslash T] = \lambda^* y'.M''} \quad (y' \text{ is a new variable})$$

Given a term M of Λ and a set of occurrences $\mathcal{F}$ designating λ-abstractions in M, we denote $(M, \mathcal{F}) \in \Lambda^*$ the term where the λ operators given by $\mathcal{F}$ are replaced by λ^* 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:
if $(M, \mathcal{F}) \underset{v}{\rightarrow} (M', \mathcal{F}')$ *and* $\vdash u : M \underset{v}{\rightarrow} u' : M'$ *then* $\mathcal{F}' = \{w \in \mathcal{D}(M') \quad | \quad value_{u'}(w) \in \mathcal{F} \circ u\}$.

(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:

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

$$\textit{stop:} \quad \vdash M \underset{\epsilon}{\twoheadrightarrow} M$$

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

$$origin_{(M,\varepsilon)} = Id_{\mathcal{D}(M)} \qquad\qquad origin_{(M,v.S)} = origin_{(M',S)} \circ origin_{(M,v)} \quad \text{if } M \underset{v}{\rightarrow} M'$$

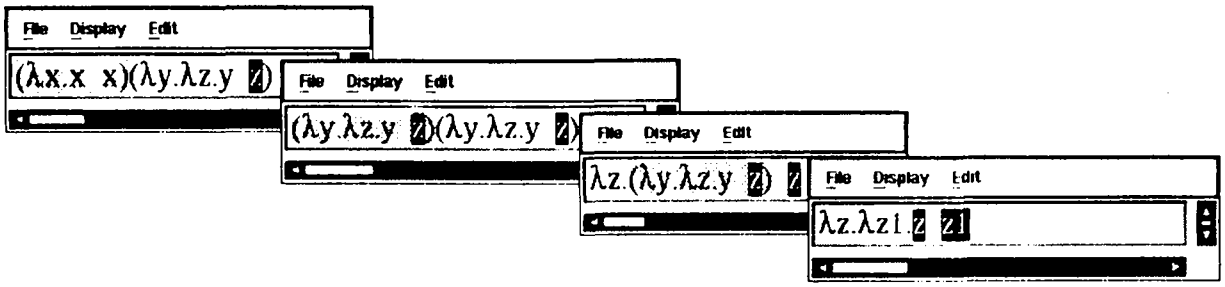

Figure 3. An example automated reduction.

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 $value_t(u) = value_{not}(id)$. 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 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:

if $(M, \mathcal{F}) \underset{S}{\twoheadrightarrow} (M', \mathcal{F}')$ *and* $\vdash id : M \underset{S}{\twoheadrightarrow} u' : M'$ *then* $\mathcal{F}' = \{w \in \mathcal{D}(M') \mid value_{u'}(w) \in \mathcal{F}\}$

(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 Programming 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].

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

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

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:

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

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

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

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:

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

For our problem, this leads to the idea that a formal description of the functional programming language containing occurrence computations is also derivable from the rules of 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:

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

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:

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

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

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

$$m(eval_{app}): \quad \frac{\rho \vdash s_1 \circ u : M \Rightarrow u_\lambda : (\lambda x.P, \rho') \quad \rho'[x\backslash s_2 \circ u : (N, \rho)] \vdash s_2 \circ u_\lambda : P \Rightarrow u' : R}{\rho \vdash u : MN \Rightarrow u' : R}$$

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

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.

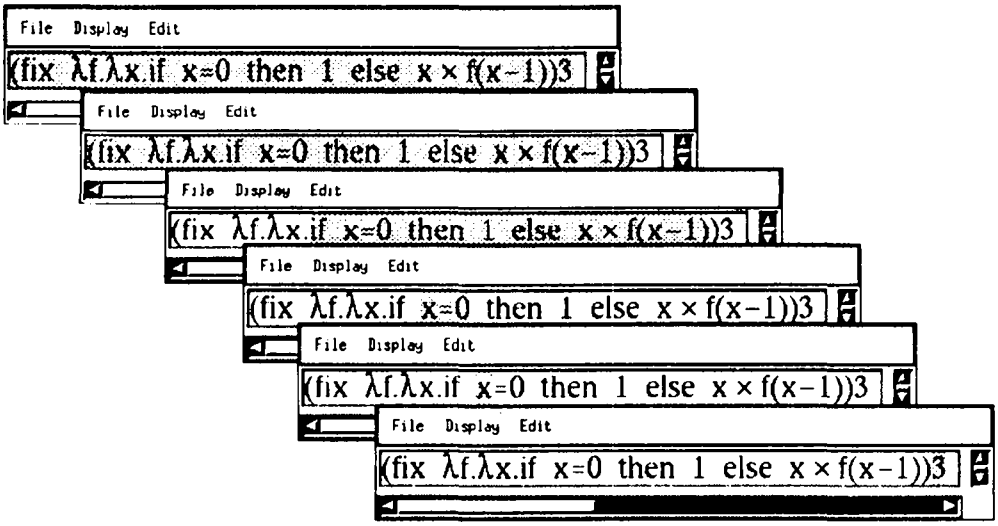

Figure 4. Execution animation of a small functional program.

5.3. *Program Animation*

When compiling the dynamic semantics of our small functional language into Prolog, it is possible to add some procedures that use the computed occurrence to select the subject when a rule is applied. This results in an animation of the program evaluation. 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.

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 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:

$$\frac{\rho \vdash P \rightarrow P', \rho' \quad \rho' \vdash P' \Rightarrow \rho''}{\rho \vdash P \Rightarrow \rho''} \qquad \rho \vdash P \Rightarrow \rho \quad (P \text{ is finished})$$

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

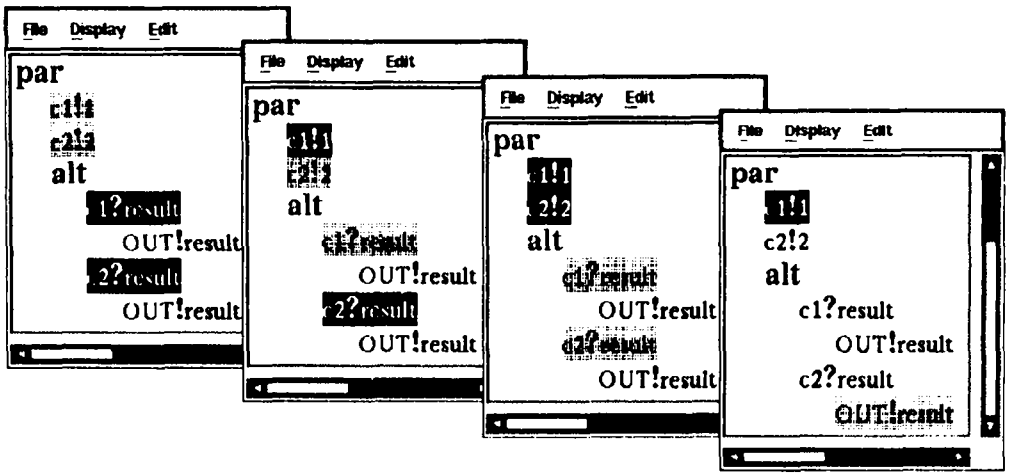

Figure 5. Execution animation of a small parallel program.

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.

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

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

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

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

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 can actually be used in the Centaur system 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 provide the user with tools for guiding the non-determinacy of parallel execution. Using the result of §3, it is possible to specify where the execution must proceed. Second, we can animate the execution of the program in a way that permits to understand the current state of computation. At each step, we can traverse the term that represents the current state of the computation and find two categories of points in this term. 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. The search for these instructions uses a technique that has already been described in [Esterel]. Since the current state of execution is obtained through rewritings, all these elementary instructions are residuals of instructions in the initial program. Since we can compute their origins, we can show these instructions.

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

The usual functionalities of debuggers are animating the execution and setting and stopping at breakpoints. As we showed in our examples, occurrence computations permit to animate executions. The difference between animating and stopping at breakpoints is small since we can modify our debugger to stop when the computed occurrence has a given value, instead of just selecting the corresponding expression. In this respect, the occurrence computations we have presented in this paper are a central component of the development of a complete debugger from a dynamic semantics specification. This motivates the development of a tool that performs automatically the manipulations described in this paper. An important feature of this work is that we use formal machinery to solve a practical problem. We believe this can be extended to the complete development of a generic system for the creation of debuggers.

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.

Imprimé en France
par
.l'Institut National de Recherche en Informatique et en Automatique.

ISSN 0249-6399