



An Analysis of Bohm's theorem

G rard Huet

► **To cite this version:**

G rard Huet. An Analysis of Bohm's theorem. [Research Report] RR-2008, INRIA. 1993. inria-00074664

HAL Id: inria-00074664

<https://hal.inria.fr/inria-00074664>

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destin e au d p t et   la diffusion de documents scientifiques de niveau recherche, publi s ou non,  manant des  tablissements d'enseignement et de recherche fran ais ou  trangers, des laboratoires publics ou priv s.



INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

*An Analysis
of Böhm's Theorem*

Gérard HUET

N° 2008
Août 1993

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

*R*apport
de recherche

1993

Une analyse du théorème de Böhm

Gérard Huet

En hommage à Corrado Böhm

Résumé

Nous présentons dans cet article une analyse détaillée du théorème de Böhm, un résultat essentiel de séparabilité en λ -calcul pur. Le théorème est expliqué de manière complètement constructive en tant que développement algorithmique dans le langage de programmation fonctionnel ML.

An analysis of Böhm's theorem

Gérard Huet

In honor of Corrado Böhm

Abstract

We present in this article a detailed analysis of Böhm's theorem, an essential separability result in pure λ -calculus. The theorem is explained completely constructively as an algorithmic development in the functional programming language ML.

1 Introduction

We analyse in this article an important result of λ -calculus, the separability theorem, first proved by Corrado Böhm in 1968[2]. The descriptive formalism used is not the usual meta-language of Mathematics, but an actual programming language of the ML family, more specifically CAML V3.1, developed in Projet Formel at INRIA Rocquencourt[7]. This presents the advantages of being more rigorous, more constructive, and of allowing better understanding by the reader who may interactively execute all definitions and examples. This article is adapted from the notes for a graduate course taught at Université Paris VII[9].

This paper may be considered an implementation of the following suggestion, quoted from Corrado Böhm's original paper[2]: “*Il metodo di dimostrazione per il teorema 1 e' costruttivo, percio' suggerisce di per se' un algoritmo per la costruzione delle formule.*”

1.1 λ -calculus: Abstract Syntax

We give here our notations for pure λ -calculus. In the usual presentation of λ -expressions, there are three constructors: variables such as x , applications of an expression e_1 to an expression e_2 , written $(e_1 e_2)$, and finally functional abstraction of an expression ex over a variable x considered as its formal parameter, traditionally written $\lambda x \cdot ex$, but for which we shall use the notation $[x]ex$. This notation stands for the algorithm which computes the value of expression ex , given an input argument x .

We shall use this notation for our *concrete syntax*, for instance to present examples. But we avoid the difficulties of renaming, such as α -conversion, by considering an *abstract syntax* of terms, where variables are not represented as actual identifier strings, but rather as *reference indexes* identifying canonically their binding abstractor, in the spirit of de Bruijn's representation[5]:

```
#type term =
#   Ref of int          (* variables as reference depth *)
#   | Abs of term       (* abstraction [x]M *)
#   | App of term * term (* application (M N) *);;
Type term is defined
```

Let us comment on the three constructors of type `term`. `App(M,N)` represents the application of function `M` to argument `N`. If `M` is an expression containing a free variable x , then the function which associates `M` to its formal argument x , noted $[x]M$ in concrete syntax, is represented abstractly as `Abs(M)`. Finally, a `Ref(n)` construct designates an occurrence of the variable declared in the n th `Abs` binder above it.

Every term `M` is meaningful in a minimum context depth `min_context_depth(M)`:

```
#let min_context_depth M = min_rec 0 M
#   where rec min_rec n = function
#     Ref(m)   -> m-n+1
#     | Abs(M)  -> min_rec (n+1) M
#     | App(M,N) -> max (min_rec n M) (min_rec n N);;
Value min_context_depth is <fun> : term -> int
```

We say that term M is *closed* if its minimum context depth is 0. We shall also call *combinator* a closed term. The *closure* of a term M is the closed term `closure(M)` defined as follows.

```
#let closure M = iterate abstract (min_context_depth M) M
#
#           where abstract M = Abs(M);;
Value closure is <fun> : term -> term
```

We assume that a parser of closed λ -expressions has been defined. This permits the translation of a concrete syntax expression, enclosed between quotation marks `<<` and `>>`, into an abstract syntax ML value of type `term`:

```
#<<[x](x [y](x y))>>;;
(Abs (App ((Ref 0),(Abs (App ((Ref 1),(Ref 0))))))) : term
```

We also assume an `ERROR` routine which raises a qualified exception.

```
#let ERROR message = raise failure(message);;
Value ERROR is <fun> : string -> 'a
```

Let us give as examples a few standard combinators needed in the following:

```
#let I = <<[x]x>>                (* Identity *)
#and K = <<[x,y]x>>              (* Constant generator *)
#and S = <<[x,y,z](x z (y z))>>  (* Shonfinkel's composition *)
#and A = <<[f,x](f x)>>          (* Application *)
#and Y = <<[f]([x](f (x x)) [x](f (x x)))>>  (* Curry's fixpoint *)
#and T = <<([x,f](f (x x f)) [x,f](f (x x f)))>>  (* Turing's fixpoint *)
#and True = <<[x,y]x>>          (* True and first projection *)
#and False = <<[x,y]y>>        (* False and second projection *)
#and Pair = <<[x,y,p](p x y)>>  (* Pairing and conditional *)
#and Omega = <<([x](x x) [x](x x))>>;;      (* Undefined *)
Value I is (Abs (Ref 0)) : term
Value K is (Abs (Abs (Ref 1))) : term
Value S is ... : term
...
```

1.2 Substitution

λ -calculus is a calculus of substitutions. The fundamental computation primitive consists in replacing a position of some redex `App(Abs(M),N)` by the result of substituting N to the first free variable of M , defined as `(subst N M)` as follows. This operation comprises two steps. We need to recursively explore M in order to find occurrences of the substituted variable. Then, for each such occurrence, we need to copy N , suitably adjusted so that its own free variables are correctly bound. This last task is performed by function `(lift n)`, which recomputes references of global variables across n extra levels of abstraction.

```
#let lift n = lift_rec 0
#where rec lift_rec k = lift_k
```

```

#where rec lift_k = function
#   Ref(i)   -> if i<k then Ref(i)      (* bound variables are invariant *)
#             else Ref(n+i)           (* free variables are relocated by n *)
# | Abs(M)   -> Abs(lift_rec (k+1) M)
# | App(M,N) -> App(lift_k M, lift_k N);;
Value lift is <fun> : int -> term -> term

```

For instance:

```

#lift 1 (Abs(App(Ref 0,Ref 1))));;
(Abs (App ((Ref 0),(Ref 2)))) : term

```

Here is the inverse of (lift 1):

```

#exception OCCURS_FREE;;
Exception OCCURS_FREE is defined

#let unlift1 = unlift_check 0
#   where rec unlift_check n = function
#     Ref(i)   -> if i=n then raise OCCURS_FREE
#               else if i<n then Ref(i) else Ref(i-1)
# | Abs(M)   -> Abs(unlift_check (n+1) M)
# | App(M,N) -> App(unlift_check n M, unlift_check n N);;
Value unlift1 is <fun> : term -> term

```

Thus $(\text{unlift1 } (\text{lift } 1 \ M)) = M$ for every term M .

We now give the substitution function. We substitute N for $\text{Ref}(0)$ in M by $(\text{subst } N \ M)$. The typical case is the β -redex $([x]M \ N)$, which reduces to $(\text{subst } N \ M)$.

```

#let subst N = subst_N 0
#where rec subst_N n = function
#   Ref(k)   -> if k=n then lift n N      (* substituted variable *)
#               else if k<n then Ref(k)  (* bound variables *)
#               else Ref(pred(k))       (* free variables *)
# | Abs(M)   -> Abs(subst_N (n+1) M)
# | App(M,M') -> App(subst_N n M, subst_N n M');;
Value subst is <fun> : term -> term -> term

```

For instance, we get:

```

#subst (Abs(Ref 0)) (Abs(App(Ref 0,Ref 1))));;
(Abs (App ((Ref 0),(Abs (Ref 0))))) : term

```

Now that substitution problems are understood, we shall generally deal in our examples with concrete syntax. Thus, from now on, we ask ML to pretty-print values of type `term` with a printer which prints such values as λ -expressions. The above example yields now:

```

#let (Abs M) = <<[x,y](y x)>> in subst I M;;
[x0](x0 [x1]x1) : term

```

1.3 β -reduction

The main computation rule of λ -calculus, called β -reduction, consists in replacing a *redex*, i.e. a subexpression of the form $([x]M N)$, by $\text{subst } N M$. Thus one step of β -reduction of term M at redex position u is obtained by `(beta_reduce M u)`.

We need first to define the datatype of positions. Positions are lists of the successive directions along the path leading to the position. The directions are F which indicates that we traverse an abstraction, and $A(L)$ (resp. $A(R)$) which indicates that we traverse an application to the left (resp. to the right).

```
#type sibling = L | R
#and direction = F | A of sibling
#and position == direction list;;
Type sibling is defined
Type direction is defined
Type position is defined
```

The subterm of term M at position u is computed as `subterm(M,u)`:

```
#let rec subterm = function
#   (e, [])          -> e
# | (Abs(e), F::u)   -> subterm(e,u)
# | (App(e,_), A(L)::u) -> subterm(e,u)
# | (App(_,e), A(R)::u) -> subterm(e,u)
# | _                -> ERROR "Position not in domain";;
Value subterm is <fun> : term * position -> term
```

Similarly, the result of replacing the subterm at position u of term M by term N is `replace N (M,u)`:

```
#let replace N = rep_rec
#where rec rep_rec = function
#   (_, [])          -> N
# | (Abs(e), F::u)   -> Abs(rep_rec (e,u))
# | (App(l,r), A(L)::u) -> App(rep_rec (l,u),r)
# | (App(l,r), A(R)::u) -> App(l,rep_rec (r,u))
# | _                -> ERROR "Position not in domain";;
Value replace is <fun> : term -> term * position -> term
```

We are now ready to define the result of reducing the β -redex at position u in term M :

```
#let beta_reduce M u = match subterm(M,u) with
#   App(Abs(body),arg) -> replace (subst arg body) (M,u)
# | _                  -> ERROR "Position not a beta redex";;
Value reduce is <fun> : term -> position -> term
```

1.4 Head normal form

A *head normal form* is any term M of the form $[x_1, \dots, x_n](x M_1 \dots M_p)$, with $n, p \geq 0$. Note that if M reduces to N by β -reduction, then N is of the form $[x_1, \dots, x_n](x N_1 \dots N_p)$. Thus n , x , and p are invariant by β -reduction.

Remark. Every term is either a head normal form, or of the form: $[x_1, \dots, x_n](R M_1 \dots M_p)$, with R a redex. In this last case R is called the *head redex* of the term.

Now we reduce a λ -term to its head normal form, when it has one, with the function `hnf` below. It uses the *normal order* of evaluation, which reduces redexes in the leftmost-outermost order.

```
#let rec hnf = function
#   Ref(n)      -> Ref(n)
# | Abs(M)      -> Abs(hnf M)
# | App(M1,M2) -> match hnf(M1) with
#   Abs(N) -> hnf(subst M2 N)
#   | h     -> App(h,M2);;
Value hnf is <fun> : term -> term
```

For instance:

```
#hnf<<([x][y](x (y x)) [u](u u) [v][w]v)>>;
[x0](x0 x0) : term
```

We say that a λ -term M is *defined* iff `hnf(M)` terminates. Since `hnf` uses the normal order of evaluation, if $(M N)$ is defined, so is M . Similarly, if $[x]M$ is defined, so is M . Furthermore, when M is defined, the normal reduction issued from M has an initial sequence which reduces M to $N = \text{hnf}(M)$, and N is the first head normal form in this reduction sequence.

1.5 Normal form

A *normal form* is a term which is not reducible. The *normalisable* terms are those for which the normal reduction procedure `nf` below terminates. This results from the standardisation theorem.

```
#let rec nf = function
#   Ref(n)      -> Ref(n)
# | Abs(M)      -> Abs(nf M)
# | App(M1,M2) -> match hnf(M1) with
#   Abs(N) -> nf(subst M2 N)
#   | h     -> App(nf h,nf M2);;
Value nf is <fun> : term -> term
```

1.6 Extensionality: η -conversion

The rule of η -conversion is a syntactic version of the property of extensionality. It is consistent with the interpretation of λ -terms as denoting functions. η -conversion is explained as the congruence closure of a second reduction rule, traditionally called η -reduction.

An η -redex is a subterm of the form $[x](N\ x)$, with x not appearing free in N . η -reduction consists in replacing this subterm by N . In abstract form, we replace $\text{Abs}(\text{App}(\text{lift } 1\ N, \text{Ref } 0))$ by N . The symmetric relation is called η -expansion. Finally, η -conversion is the congruence closure of η -reduction.

```
#let eta_reduce M u = match subterm(M,u) with
#   Abs(App(N,Ref(0))) -> (try replace (unlift1 N) (M,u)
#                           with OCCURS_FREE -> ERROR "Position not an eta redex")
# | _                    -> ERROR "Position not an eta redex";;
Value eta_reduce is <fun> : term -> position -> term
```

1.7 Böhm trees

We shall now see how to compute progressively successive approximations of a term, in a potentially infinite partial structure called the *Böhm tree* of the λ -term, which represents the limit of all β -reductions issued from a given term. It consists of layers of approximations, each approximation corresponding to a head-normal form.

First of all we shall profit of the additional structure of head normal forms to represent variables in a better way. Every variable is represented by a double key $\text{Index}(k,i)$ where k addresses upwards the hnf layer where the variable is declared, while i indexes in the corresponding list of bound variables in a left-to-right fashion. Thus in $[x_1,x_2,\dots,x_n]x_i(\dots)$ the head variable x_i is represented as $\text{Index}(0,i)$. Note that this representation of head variables is invariant by head η -expansion.

```
#type var = Index of int * int;;
Type var is defined
```

We now define the type of Böhm trees approximants.

```
#type bohm = Hnf of int * var * bohm list
#           | Future of term;;
Type bohm is defined
```

There are two kinds of nodes in a Böhm tree approximant: **Hnf** nodes, where an hnf approximation has been computed, and **Future** nodes, containing a λ -term waiting to be examined. When this term is undefined the tree cannot be grown further in this direction. Thus $\text{Future}(M)$ plays the role of a syntactic “bottom” meaning “not yet defined”. Any term may be transformed into a Böhm tree approximant by “delaying” it with the constructor **Future**:

```
#let delay M = Future(M);;
Value delay is <fun> : term -> bohm
```

We now compute one level of approximation, evaluating a (defined) term into an **Hnf** form. The extra argument **display** of function **approx** contains a stack of natural numbers intended to hold the successive arities along a Böhm tree branch. The auxiliary function **lookup** translates a **Ref** index m into an **Index** var, using a **display**.

```

#let lookup m display = look (0,m) display
# where rec look(k,j) = function
#   []      -> ERROR "Variable not in scope of display"
# | n::rest -> if j>=n then look (k+1,j-n) rest
#           else Index(k,n-j);;
Value lookup is <fun> : int -> int list -> var

#let approx display M = apx (0,[]) M
# (* n counts the outside abstractions, args stacks the applicands *)
# where rec apx (n,args) = function
#   Ref(m)  -> Hnf(n,lookup m (n::display),map delay args)
# | Abs(M)  -> (match args with
#             []      -> apx (n+1,[]) M
#             | N::rest -> apx (n,rest) (subst N M))
# | App(M,N) -> apx (n,N::args) M;;
Value approx is <fun> : int list -> term -> bohm

#let approximate = approx [];;
Value approximate is <fun> : term -> bohm

```

We get a tree in Hnf form by function evaluate, given a display:

```

#let evaluate display = function
#   Future(M) -> approx display M
# | h         -> h;;
Value evaluate is <fun> : int list -> bohm -> bohm

```

Examples. Let us consider the Loop combinator, defined as the application of combinator Y to the term $[x,y](y\ x)$; The symbol $\hat{\ }$ below is *anti-quotation*: it allows a previously defined term, bound to an ML identifier xxx, to be inserted in a concrete expression as $\hat{\ }xxx$.

```

#let Loop = <<(^Y [e][x](x e))>>;;
Value Loop is
  ([x0]([x1](x0 (x1 x1)) [x1](x0 (x1 x1))) [x0,x1](x1 x0)) : term

#let BT1 = approximate Loop;;
Value BT1 is
  Hnf(1,(Index (0,1)),
    [Future [x0]([x1,x2](x2 x1) (x0 x0)) [x0]([x1,x2](x2 x1) (x0 x0))]) : bohm

#let BT2 = let (Hnf (n,_,[(Future M)])) = BT1 in approx [n] M
#in BT1 = BT2;;
true : bool

```

Here Loop has the infinite Böhm tree solution of $B=[x](x\ B)$. Similarly, the combinators Y and T admit the same infinite Böhm tree $[x](x\ B)$, with B solution of the equation $B=(x\ B)$.

Now, let us consider the combinator J:

```
#let J = <<(^Y [u,x,y](x (u y)))>>;
Value J is
  ([x0]([x1](x0 (x1 x1)) [x1](x0 (x1 x1))) [x0,x1,x2](x1 (x0 x2))) : term
```

Let us look at the approximants of J:

```
#let BT1 = approximate J;;
Value BT1 is (Hnf (2,(Index (0,1)), [(Future
  ([x0]([x1,x2,x3](x2 (x1 x3)) (x0 x0))
  [x0]([x1,x2,x3](x2 (x1 x3)) (x0 x0)) u0)])) : bohm

#let BT2 = let (Hnf (2,_,[(Future M])) = BT1 in approx [2] M;;
Value BT2 is (Hnf (1,(Index (1,2)), [(Future
  ([x0]([x1,x2,x3](x2 (x1 x3)) (x0 x0))
  [x0]([x1,x2,x3](x2 (x1 x3)) (x0 x0)) u0)])) : bohm

#let BT3 = let (Hnf (1,_,[(Future M])) = BT2 in approx [1;2] M
#in BT3=BT2;;
true : bool
```

The combinator J has thus the infinite Böhm tree:
 $[x_1, x_2](x_1 [x_3](x_2 [x_4](x_3 \dots [x_n](x_{n-1} \dots)))$). It may be considered as an infinite η -expansion of combinator I.

2 Separability

We now have all the conceptual tools to study separability.

2.1 Böhm's Theorem

Definition. We say that λ -terms M and N are *separable* iff there exist combinators C_1, \dots, C_n such that $\text{nf}((\text{closure } M) C_1 \dots C_n) = \text{True}$ and $\text{nf}((\text{closure } N) C_1 \dots C_n) = \text{False}$.

The aim of this paper is to prove:

Böhm's Theorem. Any two normal forms are either η -convertible or separable.

We shall now develop additional notions which are needed for the proof of this theorem.

2.2 Accessibility in Böhm trees

Definition. A *path* of length s is a sequence of positive integers: $P = \text{Path}[k_1; \dots k_s]$.

```
#type path = Path of int list;;
Type path is defined
```

Getting the i -th component of path P :

```
#let get i (Path P) = nth P i;;
Value get is <fun> : int -> path -> int
```

A path denotes a position in a Böhm tree. At depth i , it indicates that we access its k_i 's son, i.e. $(\text{get } b_i \ k_i)$, or if this is not possible that we effect the necessary η -expansions.

Definition. Let B be a Böhm tree in Hnf form, and P be a path. We say that P is *accessible in B towards B' modulo η -conversion* iff $(\text{access_tree_top } B \ P)=(B', b, \text{stack})$, for some b and stack , with access_tree_top defined below. First we discuss a more general procedure access_tree .

The function access_tree collects occurrences of the first bound variable x along the path, with its arity p , in argument $\text{stack}:\text{arities}$. The boolean b indicates whether the head variable of B' is x or not. We allow η -expansions of the head normal forms represented by the layers of approximations, and thus a Böhm tree may be “stretched to the right” arbitrarily by η -expansions in order to accomodate a given path.

```
#type arity = Absent
#           | Present of int
#and arities == arity list;;
Type arity is defined
Type arities is defined
```

Definition. *Relevant* variables refer to the outermost bound ones in the tree.

```
#let relevant (i,level) = i=Index(level,1);;
Value relevant is <fun> : var * int -> bool
```

The auxiliary procedure subeta below finds the k th son of a tree node of arity n in the list l of its p sons, possibly using η -expansion when $k>p$.

```
#let subeta l k n p display = if k<=p then evaluate display (nth l k)
#                             else Hnf(0,Index(1,n+k-p),[]);;
Value subeta is <fun> : bohm list -> int -> int -> int -> int list -> bohm
```

We are now ready to give the procedure that accesses a tree in Hnf form along a path.

```
#let rec access_tree (display,level) stack (Hnf(n,i,args) as h) =
# let b = relevant(i,level) in function
#   [] -> h,b,rev(stack)
# | k::P -> let p = list_length args
#           and display' = n::display
#           in let B'= subeta args k n p display'
#               and inspect = if b then Present(p) else Absent
#               in access_tree (display',level+1) (inspect::stack) P B';;
Value access_tree is <fun> :
  int list * int -> arities -> bohm -> int list -> bohm * bool * arities
```

```
#let access_tree_top = access_tree ([],0) [];;
Value access_tree_top is <fun> :
  bohm -> int list -> bohm * bool * arities
```

Definition. Let M be a closed term, B a Böhm tree. We say that path P is *accessible* in term M towards B modulo η -conversion iff $(\text{access } M \ P)=(B,b,\text{stack})$ for some b and stack , where:

```
#let access M (Path P) = access_tree_top (approximate M) P;;
Value access is <fun> : term -> path -> bohms * bool * arities
```

Examples.

```
#let M = <<[x](x [y](x y))>>;;
Value M is [x0](x0 [x1](x0 x1)) : term
```

```
#access M (Path[]);;
((Hnf (1,(Index (0,1)),[(Future [x0](u0 x0))])),true,[]) : bohms * bool * arities
```

```
#access M (Path[1]);;
((Hnf (1,(Index (1,1)),[(Future u0)])),true,[(Present 1)]) : bohms * bool * arities
```

```
#access M (Path[1;1]);;
((Hnf (0,(Index (1,1)),[])),false, [(Present 1); (Present 1)]) :
bohms * bool * arities
```

```
#access M (Path[1;2]);;
((Hnf (0,(Index (1,2)),[])),false, [(Present 1); (Present 1)]) :
bohms * bool * arities
```

Definition. The *shape* of a Böhm tree in Hnf form is the triple consisting of its arity, its head variable, and the number of its immediate subtrees:

```
#let shape = function
#   Hnf(n,i,b) -> (n,i,list_length b)
#   | _       -> ERROR "Not in Head Normal Form";;
Value shape is <fun> : bohms -> int * var * int
```

Definition. Two Böhm trees in Hnf form are said to be *similar* if they have the same shape, up to η -conversion:

```
#let similar (B,B') =
#   let (n,i,p)   = shape B
#   and (n',i',p') = shape B'
#   in i=i' & p+n'=p'+n;;
Value similar is <fun> : bohms * bohms -> bool
```

Intuitively, this means that the two trees are defined, and that the corresponding top-level approximations may be made similar by η -conversion, in the sense that $\text{Hnf}(n,i,1)$ and $\text{Hnf}(n,i,1')$ are similar when $|1|=|1'|$: same binding prefix, same head variable, same number of immediate subtrees.

Definition. Let M and M' be two closed terms, and P be a path such that $(\text{access } M \ P) = (B, _, _)$ and $(\text{access } M' \ P) = (B', _, _)$. We say that P *distinguishes* M and M' iff B and B' are not similar.

Theorem. Distinguishability entails separability. If two (closed) terms are distinguishable by some path, they are separable.

The proof of this theorem will be given as correctness of the Böhm-out algorithm below, which exhibits the context which separates the two terms.

2.3 A Böhm-out toolkit

In this section we give a few parametric combinators needed in the following.

```
#let Pair_ n = iterate Abs (n+1) (applist (range n))
#   where rec applist = function
#         []   -> Ref(0)
#         | n::l -> App(applist l, Ref(n));;
Value Pair_ is <fun> : int -> term

#Pair_ 3 = <<[x1,x2,x3,x4](x4 x1 x2 x3)>>;;
true : bool

#Pair = Pair_ 2;;
true : bool

#let Pi k n = if n>=k then iterate Abs n (Ref(n-k)) else ERROR "Pi";;
Value Pi is <fun> : int -> int -> term

#Pi 5 7;;
[x0,x1,x2,x3,x4,x5,x6]x4 : term

#(I = Pi 1 1) & (True = Pi 1 2) & (False = Pi 2 2);;
true : bool

#let K_ n = Pi 1 (n+1);;
Value K_ is <fun> : int -> term

#K_ 5;;
[x0,x1,x2,x3,x4,x5]x0 : term

#K = K_ 1;;
true : bool
```

The next function generates the combinator which, applied to any n arguments, evaluates to x .

```
#let Cst x n = nf(App(K_ n,x));;
Value Cst is <fun> : term -> int -> term
```

```

#Cst K 3;;
[x0,x1,x2,x3,x4]x3 : term

#let ff = Cst False
#and tt = Cst True
#and ii = Cst I;;
Value ff is <fun> : int -> term
Value tt is <fun> : int -> term
Value ii is <fun> : int -> term

```

Finally, the `di` generator builds lists of combinator `I`:

```

#let rec di = function 0 -> [] | n -> I::di(n-1);;
Value di is <fun> : int -> term list

```

Thus `di 3 = [I; I; I]`.

2.4 Semi-separability

We first start with an exercise, in order to understand the use of the combinators above as generalised projections.

Consider the closed head normal form $M=[x_1,x_2, \dots, x_n](x_i M_1 \dots M_p)$.

Fact. There exist terms C_1, C_2, \dots, C_n such that $\text{nf}(M C_1 C_2 \dots C_n) = I$.

Proof. The list `[C1;C2;...;Cn]` is computed by `semi_sep(M)` below.

```

#let semi_sep M = let (n,Index(0,i),p) = shape(approximate M)
#                 in identity(n,i-1,p)
#                 where identity(n,i,p) = di(i)@(ii(p)::di(n-i));;
Value semi_sep is <fun> : term -> term list

```

The next function *projects* a term `M` along a context `[N1; ...; Nn]`, by applying `M` successively to `N1, ..., Nn` and normalising:

```

#let project M context = nf(it_list (fun P Q -> App(P,Q)) M context);;
Value project is <fun> : term -> term list -> term

```

For instance:

```

#project Y (semi_sep Y);;
[x0]x0 : term

```

Remark. In the usual terminology, we would say that defined terms are *solvable*. The reverse is immediate. This characterisation of definedness by semi-separability was first remarked by Wadsworth.

2.5 Separating nonsimilar approximations

We first examine the base case of the theorem, when we deal with two nonsimilar approximations. There are two cases, dealt with by `sep1` and `sep2` below.

First, `sep1` separates two closed head normal forms with distinct head variables:

$[x_1, x_2, \dots, x_n](x_i M_1 \dots M_p)$ and
 $[x_1, x_2, \dots, x_{n'}](x_{i'} N_1 \dots N_{p'})$, with $i \neq i'$.

```
#let sep1(i,i',p,p',n,n') = (* Assumes i<>i' *)
# let sep_base1 i i' j k f g n = di(i-1)@(f(j)::di(i'-i-1)@(g(k)::di(n-i')))
# in if n>=n' then let k=p'+n-n'
#           in if i<i' then sep_base1 i i' p k tt ff n
#           else sep_base1 i' i k p ff tt n
#           else let k=p+n'-n
#           in if i<i' then sep_base1 i i' k p' tt ff n'
#           else sep_base1 i' i p' k ff tt n';;
Value sep1 is <fun> : int * int * int * int * int * int -> term list
```

This may be checked by case analysis, as an exercise in β -reduction. Similarly, `sep2` separates two closed head normal forms:

$[x_1, x_2, \dots, x_n](x_1 M_1 \dots M_p)$ and
 $[x_1, x_2, \dots, x_{n'}](x_1 N_1 \dots N_{p'})$ with $p + n' \neq p' + n$.

```
#let sep2(p,p',n,n') = (* Assumes p+n'<>p'+n *)
# let sep_base2 d k m f b = ii(k)::di(m-1)@(f(d)::di(d-1)@[b])
# and d = p-n-(p'-n')
# in if n>=n' then if d>0 then sep_base2 d p n tt False
#           else sep_base2 (-d) (p'+n-n') n ff True
#           else if d>0 then sep_base2 d (p+n'-n) n' tt False
#           else sep_base2 (-d) p' n' ff True;;
Value sep2 is <fun> : int * int * int * int -> term list
```

Remark that more generally the two closed head normal forms:

$[x_1, x_2, \dots, x_n](x_i M_1 \dots M_p)$ and
 $[x_1, x_2, \dots, x_{n'}](x_{i'} N_1 \dots N_{p'})$ with $p + n' \neq p' + n$, may be separated by the sequence $di(i-1)@sep2(p,p',n,n')$.

2.6 The Böhm-out algorithm

Let us first give the main idea. Let M and N be two closed terms, given with a path P leading to two nonsimilar positions in their respective Böhm trees. The Böhm-out algorithm computes as $(\text{separate } (M,N) P)$ a list of combinators C_1, C_2, \dots, C_n which separate M and N .

We want to “bring to the top” the difference between the two terms by successive applications, until the resulting terms have nonsimilar Böhm trees. When this condition is achieved, we apply algorithms `sep1` and `sep2` above. When the two terms have similar Böhm trees, we consider the first bound variable, say x , in their head normal forms. We examine all occurrences of x as head variable along path P . There are three cases.

First, if there is no such occurrence, we get rid of x simply by applying the two terms to any term, say $C1=I$. If there is exactly one such occurrence in path P , we substitute to x the combinator $C1=Pi\ k\ p$, for appropriate k and p . When there are several such occurrences, we linearise by substituting to x a pairing combinator $C1=Pair_maxp$, with $maxp$ the maximum number of descendants of nodes with head variable x in path P . We thus replace every such occurrence of x by a distinct x_i locally bound at its level.

We have to be careful with other possible relevant occurrences of x : as heads of the nonsimilar subtrees at position P in the Böhm trees of M and N . When this happens, we do the same as in the case of multiple occurrences, in order to preserve the property of the two subtrees to be nonsimilar, while replacing the global head x by a new local z . This completes the analysis of the different cases. In each case, we call ourselves recursively. The path P stays the same, except in the single-occurrence case, where it is shortened by skipping its i -th element, using the auxiliary function `skip`, which removes the i -th level of path P :

```
#let skip i (Path P) = Path(coll_rec 1 P)
# where rec coll_rec lev = function
#   []      -> ERROR "Out of range skip"
# | dir::P -> if i=lev then P
#           else dir::coll_rec (lev+1) P;;
Value skip is <fun> : int -> path -> path
```

The various cases of the analysis above form the constructors of type `item`:

```
#type item = None
#           | Once of int*int (* level,p *)
#           | Several of int (* maxp *);;
Type item is defined
```

The next procedure collects occurrences of the first variable x as head variable along path P :

```
#let analyse (n,n',p,p',b,b') = anal 1
#where rec anal lev = function
#   ([],[ ]) -> if b then if b' then Several(max p p')
#                 else Several(if n'>n then p+n'-n else p)
#   else if b' then Several(if n>n' then p'+n-n' else p')
#   else None
# | (Present(p)::st1,Present(p')::st2) -> let pmax = max p p'
#   in (match (anal (lev+1) (st1,st2)) with
#       Several(maxp) -> Several(max pmax maxp)
#       | Once(_,pi)   -> Several(max pmax pi)
#       | None         -> Once(lev,pmax))
# | (Absent::st1,Absent::st2) -> anal (lev+1) (st1,st2)
# | _ -> ERROR "Non similarity along path";;
Value analyse is <fun> :
  int * int * int * int * bool * bool -> arities * arities -> item
```

We are now prepared to give the Böhm-out algorithm. We assume the Böhm trees of closed terms M and N are accessible by path P , minimal leading to nonsimilar trees:

```

#let rec separate (M,N) P =
# let (B, b, st) = access M P
# and (B',b',st') = access N P
# in let (n, v, p) = shape(B)
#     and (n',v',p') = shape(B')
#     in if P=Path[] then (* The Bohm trees of M and N are not similar *)
#         let (i,i') = match (v,v') with
#             Index(0,i),Index(0,i') -> (i,i')
#             | _ -> ERROR "Non closed term"
#         in if i<>i' then sep1(i,i',p,p',n,n')
#             else if p+n'<>p'+n then di(i-1)@sep2(p,p',n,n')
#             else ERROR "Similar trees"
#         else match (analyse (n,n',p,p',b,b') (st,st')) with
#             None -> build_context I (* Any term would do *) (M,N) P
#             | Once(m,p) -> (* head var appears just once on path *)
#                 let k = get m P
#                 in build_context (Pi k p) (M,N) (skip m P)
#             | Several(maxp) -> build_context (Pair_maxp) (M,N) P
#and build_context C (M,N) P = C::separate (App(M,C),App(N,C)) P;;
Value separate is <fun> : term * term -> path -> term list
Value build_context is <fun> : term -> term * term -> path -> term list

```

Fact. Well-foundedness of separate. I.e., the evaluation of `separate (M,N) P` always terminates.

Proof. By induction on `triple(P)=(s,schi,n1)`, where `s=list_length(P)`, `schi` is the number of variables bound at first level which are head variables at several levels, and `n1` is the number of first level variables.

Remark. We use pairing operators to rename multiple occurrences of the first variable along the path, as collected by `analyse`. Actually this renaming is not always necessary; when `analyse` considers a set of occurrences `(lev,p)` in the path with same `k` at every level `lev`, we could directly apply `(Pi k p)`, and get a shorter context. This renaming is necessary when the variable is *schizophrenic*, i.e. the path traverses two levels with `x` head variable, but distinct `k`'s. We call *schizophreny* of the situation the integer `schi`.

```

#let separates context (M,N) =
# (project M context = True) & (project N context = False);;
Value separates is <fun> : term list -> term * term -> bool

```

Examples.

```

#let context = separate(I,Pair) (Path []);;
Value context is [[x0,x1,x2,x3]x2; [x0]x0; [x0,x1,x2,x3]x3] : term list

#separates context (I,Pair);;
true : bool

```

```

#let M = <<[u](u ^I (u ^I ^I))>>
#and N = <<[u](u ^I (u u ^I))>>
#and P = Path[2;1];;
Value M is [x0](x0 [x1]x1 (x0 [x1]x1 [x1]x1)) : term
Value N is [x0](x0 [x1]x1 (x0 x0 [x1]x1)) : term
Value P is (Path [2; 1]) : path

#let context = separate (M,N) P;;
Value context is
  [[x0,x1,x2](x2 x0 x1); [x0,x1]x1; [x0,x1]x0;
   [x0,x1,x2,x3]x2; [x0]x0; [x0,x1,x2,x3]x3] : term list

#separates context (M,N);;
true : bool

#let M = <<[u,v](v [w](u w))>>
#and N = <<[u,v](v [w](u u))>>
#and P = Path[1;1]
#in separate (M,N) P;;
[[x0,x1](x1 x0); [x0]x0; [x0,x1,x2](x2 x0 x1); [x0]x0;
 [x0]x0; [x0,x1,x2,x3]x3; [x0,x1,x2,x3]x2] : term list

#let M = <<[u](u u)>>
#and N = <<[u](u [v,w](w v))>>
#and P = Path[1]
#in separate (M,N) P;;
[[x0,x1,x2](x2 x0 x1); [x0]x0; [x0,x1]x0; [x0]x0;
 [x0,x1,x2,x3]x3; [x0,x1,x2,x3]x2] : term list

```

2.7 The separability theorem

We now prove the Separability Theorem above:

Separability Theorem. Distinguishable terms are separable.

Proof. Let us assume that M and N are two closed terms distinguished by a path P . We proceed by induction on $\text{triple}(P)=(s, \text{sch}_i, n_1)$. When $s=0$, we use the correctness of sep_1 and sep_2 , a mere exercise in β -reduction. When $s>0$, we reason by cases on the number of times the outermost bound variable x_1 in the head normal forms of M and N occurs as head variable along path P . If it occurs just once at level m with arity p , and P goes through its k -th subtree, then $(P_i \ k \ p)$, applied to M and N , respectively, will propagate by successive approximations into their respective Böhm trees until level m , which it will collapse in such a way that $(\text{skip } m \ P)$, of length $s-1$, is a distinguishing path for them. The result follows by induction. If it occurs several times, with max_p its maximum arity along P , we linearise x_1 by substituting $C_1=\text{Pair}_-(\text{max}_p)$ to it. We check that P distinguishes $(M \ C_1)$ and $(N \ C_1)$, with decreased schizopreny sch_i-1 . Finally, when it does not occur, we just eliminate x_1 by substituting I to it. We check that P distinguishes $(M \ I)$ and

(N I), with same schizophrenia and $n1-1$ outermost bound variables. As remarked above, we treat the case where $x1$ appears as head variable of the tree accessed by P in M or N in the same way as a multiple occurrence (see the base case of `analyse` above). This simplifies the treatment, since it defers the substitution to these head variables to the end of the Böhm-out process, with P empty. This may create a separating context longer than necessary, but as remarked above we do not care here about minimizing the number of C_i 's.

2.8 Searching for a separating path

In this section we shall attempt to construct a path separating two λ -terms. This is not decidable in general, and thus we can only hope for a semi-decision algorithm. In particular, we have to choose a sub-tree at every level in such a way that we avoid undefined ones. We shall ignore this problem here and thus look in the Böhm trees of the two terms in a leftmost-outermost manner.

Searching for a path separating two trees. `search_path` searches in a depth-first, left-to-right manner.

```
#let rec search_path ((Hnf(n,_,l),Hnf(n',_,l')) as trees) (dis,dis') path =
#  if similar(trees) then
#    let p = list_length l and d = n::dis
#    and p' = list_length l' and d' = n'::dis'
#    in let check k = let h = subeta l k n p d
#                    and h' = subeta l' k n' p' d'
#                    in search_path (h,h') (d,d') (k::path)
#    in try try_find check (range (max p p'))
#      with failure _ -> ERROR "Not separable"
#  else rev(path);;
Value search_path is <fun> :
  bohms * bohms -> int list * int list -> int list -> int list
```

Note. Similarity does not depend on the display. Further we do not need to keep track of η -expansions. This is an essential property of our representation of variables with constructor `Index`.

Finding a path separating two terms.

```
#let separ_path (M,N) =
#  Path(search_path (approximate M, approximate N) ([],[]) []);;
Value separ_path is <fun> : term * term -> path
```

This procedure may loop because it looks depth-first, like for instance in `separ_path(<<[x](x ^Omega x)>>, <<[x](x ^Omega ^I)>>)`, or because the two terms are not separable, like for instance in `separ_path(Y,T)`. For certain non-separable pairs of terms, it actually fails, as for instance:

```
#separ_path(I,A);;
```

```
Evaluation Failed: failure "Not separable"
```

Exercises.

1. Program a breadth-first version `search_path_breadth` of algorithm `search_path`, which will return a path separating two trees if it can be found in a breadth-first search left-to-right. Thus `search_path_breadth` ought to succeed on the approximations of terms $\ll[x,y](x \text{ } ^Y x)\gg$ and $\ll[x,y](x \text{ } ^Y y)\gg$.
2. The preceding procedure will still fail to find a path separating $\ll[x,y](x \text{ } ^{\Omega} x)\gg$ and $\ll[x,y](x \text{ } ^{\Omega} y)\gg$, because it avoids looping on infinite branches, but still loops trying to find an approximation to undefined terms. Write a semi-decision procedure `separ_path` which will return a path separating two terms if there exists one at all. This procedure must dove-tail single-steps β -reductions from the two terms, without attempting to compute head-normal forms in one atomic step. It will still loop on pairs such as (Y,Y) , of course.

Remark. Combinators which are η -convertible, such as A and I , are not separable. Combinators which have the same Böhm tree, such as Y and T , are not separable either. Note also that the combinator I is not separable from J given above, even though their Böhm trees are different, and they are not η -convertible.

2.9 Left Böhm separator

Putting everything together, we get a left Böhm separator as:

```
#let Bohm (M,N) = separate (M,N) (separ_path (M,N));;
Value Bohm is <fun> : term * term -> term list
```

For instance:

```
#Bohm(I,Pair);;
[[x0,x1,x2,x3]x2; [x0]x0; [x0,x1,x2,x3]x3] : term list

#Bohm( $\ll[s,z]z\gg$  (* Church(0) *), $\ll[s,z](s z)\gg$  (* Church(1) *));;
[[x0,x1,x2]x2; [x0,x1]x0] : term list

#Bohm(I,Y);;
[[x0,x1]x1; [x0,x1,x2]x2; [x0,x1]x0] : term list

#Bohm( $\ll[x](x x)\gg$ , $\ll[x,y](x y y)\gg$ );;
[[x0,x1,x2](x2 x0 x1); [x0,x1,x2,x3](x3 x0 x1 x2); [x0,x1]x0; [x0]x0;
 [x0]x0; [x0,x1,x2,x3,x4]x3; [x0,x1,x2,x3,x4]x4] : term list

#Bohm( $\ll[u](u \text{ } ^I (u \text{ } ^I \text{ } ^I))\gg$ , $\ll[u](u \text{ } ^I (u u \text{ } ^I))\gg$ );;
[[x0,x1,x2](x2 x0 x1); [x0,x1]x1; [x0,x1]x0; [x0,x1,x2,x3]x2; [x0]x0;
 [x0,x1,x2,x3]x3] : term list

#Bohm( $\ll[u,v](v [w](u w))\gg$ , $\ll[u,v](v [w](u u))\gg$ );;
[[x0,x1](x1 x0); [x0]x0; [x0,x1,x2](x2 x0 x1); [x0]x0; [x0]x0;
 [x0,x1,x2,x3]x3; [x0,x1,x2,x3]x2] : term list
```

```
#Bohm(<<[u](u (u u u) u)>>, <<[u](u (u u [u,v,w](w u v)) u)>>);;
[[x0,x1,x2,x3](x3 x0 x1 x2); [x0]x0; [x0,x1,x2]x0; [x0]x0; [x0,x1,x2]x1;
[x0]x0; [x0]x0; [x0,x1,x2,x3,x4]x4; [x0,x1,x2,x3,x4]x3] : term list
```

2.10 Application to normal forms

Proposition. If M and N are two closed normal forms which are not η -convertible, they are distinguishable.

Proof. By induction on the maximum height of the (finite) Böhm trees of M and N .

Let $M=[x_1, x_2, \dots, x_n](x_i M_1 \dots M_p)$ and $N=[x_1, x_2, \dots, x_{n'}](x_{i'} N_1 \dots N_{p'})$. If M and N are not similar, the empty path distinguishes them. Otherwise, we must have $i=i'$, and $p+n'=p'+n$. Assume for instance $n \geq n'$: $n=n'+d$. The term N is η -convertible to $[x_1, x_2, \dots, x_n](x_i N_1 \dots N_p)$, taking $N(p'+j)=u(n'+j)$ for $1 \leq j \leq d$. Since M and N are not η -convertible, there must exist $1 \leq k \leq p$ such that M_k is not η -convertible to N_k ; by induction hypothesis these two terms are distinguishable by some path P , and thus M and N are distinguishable by path $k:P$.

Corollary 1. Böhm's theorem.

Proof. Remark that two terms are η -convertible if and only if their closure is. Apply the Separability Theorem.

Corollary 2. If M and N are two closed normal forms which are not η -convertible, `separ_path(M,N)` succeeds and returns a path which distinguishes them.

Proof. `separ_path(M,N)` terminates with the shortest path in lexicographic ordering which distinguishes M and N .

Corollary 3. If M and N are two closed normal forms which are not η -convertible, `Bohm(M,N)` terminates with a context C such that `(project M C)=True` and `(project N C)=False`. When M and N are η -convertible, `Bohm(M,N)` terminates with failure "Not separable".

Many extensions of Böhm's theorem have been studied, for instance for the simultaneous separation of n terms[8, 6, 3]. The separability problem is a special case of solving equations in λ -calculus[11, 4, 10]. The notion of separability we have defined here is not the most general; it should rather be called *separability without memory*. If we rather define separability of M and N as the existence of a combinator X such that $(X M)$ reduces to `True` and $(X N)$ reduces to `False`, this allows X to be of the form $[z](z C_1 \dots C_n)$, with the possibility of z occurring in the C_i 's as a *memory* of the separated term.

3 Discussion

The importance of Böhm's theorem is that it gives a very strong requirement for a structure to be a model of λ -calculus. Two normalisable terms are identifiable in a non-trivial model only if they are $\beta\eta$ -convertible. This fixes roughly the three degrees of freedom of a model with respect to the completely syntactical one of Böhm trees:

- It may be extensional (i.e. verify η , possibly identify I and J) or not

- It may identify more or less the undefined terms
- It may be more or less rich in non-definable points.

In this paper, we have given a treatment of Böhm's theorem which attempted to be as constructive as possible. This was obtained by splitting the result into two phases. The Böhm-out technique is first explained, as a total function taking as argument a distinguishing path in the two Böhm trees: computing a distinguishing path is only a semi-computable algorithm. We gave here a version which computes such a path when it can be found in a left-to-right fashion, a condition which suffices to the application to Böhm's theorem.

Our notion of Böhm trees has the advantage of keeping all the possible information in the **Future** parts, which hold ordinary terms. This presents several advantages. First of all, the usual substitution and head normal form algorithms, for λ -terms, suffice, there is no need to reduce Böhm trees. Then, this structure is well-suited to define a variety of models, where unsolvable subparts may be more or less identified, by suitable congruences (such as β , or $\beta\eta$ -conversion) defined on terms.

Several minor details may be pointed out. First, the **Index** representation of variables, invariant by η -expansion. Then, the Böhm-out technique uses only one operation on Böhm trees, equivalent to substitution by application of a term to the head variable. The bound variables on the discriminating path are eliminated in a left-to-right fashion. Finally, we remark that our formalisation has not increased the size of the descriptive text. Indeed, the Böhm-out technique is completely described in less than a page of ML code (algorithms `sep1`, `sep2`, and `separate`).

We consider such a constructive development as *pre-formal*. We leave as a challenge to mechanised proof systems the completely formal development (i.e. machine-checked proof) of Böhm's theorem.

References

- [1] H. Barendregt. *The Lambda-Calculus: Its Syntax and Semantics*. North-Holland (1980).
- [2] C. Böhm. *Alcune proprietà delle forme $\beta - \eta$ -normali nel $\lambda - K$ -calcolo*. Pubblicazioni dell'Istituto per le Applicazioni del Calcolo N. 696, Roma, 1968.
- [3] C. Böhm, M. Dezani-Ciancaglini, P. Peretti and S. Ronchi della Rocca. *A discrimination algorithm inside $\lambda - \beta$ -Calculus*. *Theoretical Computer Science* **8** (1979), 271–291.
- [4] C. Böhm, A. Piperno and E. Tronci. *Solving equations in Lambda-Calculus*. In *Logic Colloquium'88*, Eds Ferro, Bonotto, Valentini and Zanardo, North-Holland (1989).
- [5] N.G. de Bruijn. *Lambda-Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem*. *Indag. Math.* **34,5** (1972), 381–392.
- [6] M. Coppo, M. Dezani-Ciancaglini and S. Ronchi della Rocca. *(Semi-)separability of finite sets of terms in Scott's D_∞ models of the λ -calculus*. In *Proc. 5th ICALP*, Eds G. Ausiello and C. Böhm, LNCS **62** (1978), 142–164.
- [7] G. Cousineau and G. Huet. *The CAML Primer*. Rapport Technique 122, INRIA, Sept. 1990.



Unité de Recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399

