The Inf function in the system F

René David

The Inf function in the system F

R David (*)
Laboratoire de Mathématiques
Université de Chambéry
73376 Le Bourget du Lac Cedex
email : david@univ-savoie.fr

Abstract. We give a λ term of type Nat, Nat->Nat in the system F that computes the minimum of 2 Church numerals in time O(inf.log(inf)). This refutes a conjecture of the "λ folklore".

I Introduction

It is known (see [11]) that the representation of the integers by the Church numerals in the second order lambda calculus (the Girard-Reynolds system F) has - as far as efficiency is concerned - the drawback that the predecessor cannot be computed (even in the pure lambda calculus) in constant time. Though this is not a serious problem for the predecessor itself (nobody will use the unary notation for the integers on a computer and in binary notation it is quite normal to compute the predecessor in time the length of its notation) this becomes a real problem if the predecessor operation has to be iterated for example to compute the difference or the minimum of 2 integers.
B Maurey has given a term Inf = λnλm ((n F λx n) (m F λx m)) where F = λfλg (g f) that computes the Inf function in time O(inf) but JL.Krivine ([6,13]) has shown that this term cannot be typed of type Nat, Nat -> Nat in the system F where Nat is ∀x ((x->x)->(x->x)).
There is a term (see below) of type Nat, Nat -> Nat that computes the Inf function in time O(inf^2) and it was usually thought that this was the best that can be done, because there would be no way to "alternate" the decrementation of 2 arguments in a typed context. We show that this is not the case and give here a lambda term of type Nat, Nat -> Nat that computes the Inf function in time O(inf.log(inf)) .
I guess that it could be shown (I have not checked it) that this term can be typed in Krivine's system AF2 (the second order functional arithmetic which is - essentially - a first order extension of the system F) of type : ∀x∀y(Nat(x), Nat(y)->Nat(inf(x,y))) where the function

(*) This work has been partially supported by URA 753 (Logic group in Paris 7) and the LIP at the ENS-Lyon

symbol inf is defined by the usual equations : inf(x,0)=0 ; inf(0,sy)=0 ; inf(sx,sy)=s inf(x,y). This is not at all a trivial exercice since the following facts (to mention only a few of them) are used in the proof but their proof have no algorithmic content in the term itself so the typing has -in some way- to take care of this :

- the transitivity of < . It is used in : if $n>2^k$ and $m\leq 2^k$ then we know that $n>m$.
- $(k+2)(k+3)/2 < 2^k+8$. It is used to prove that inf(n,m) iterations are enough to find the minimum.
- the algorithm given to compute the predecessor of an integer in binary notation really computes the predecessor.
- and so on...

I conjecture that there is no typed term computing the Inf function in time O(inf).

In [12](also see [10]) M.Parigot introduces the type system TTR (recursive type theory), the main reason for that was to give a typed representation of the integers with a typed predecessor working in constant time. TTR is an extension of AF2 where inductive definitions for types are allowed . For exemple Nat_TTR is there defined by :

$$\mathrm{Nat\_TTR}(x)=\mu N\ \forall X\ (\forall y(N(y) \to X(s(y))),\ X(0) \to X(x))$$

that is we mean :

$$\mathrm{Nat\_TTR}(x)\Leftrightarrow\forall X\ (\forall y(\mathrm{Nat\_TTR}(y) \to X(s(y))),\ X(0) \to X(x))$$

and we do not give any algorithmic content to <=> .

The representation of the integers in this system is then : zero = $\lambda f\lambda x\ x$, the successor succ=$\lambda n\lambda f\lambda x\ (f\ n)$, the predeccesor pred=$\lambda n\ (n\ \mathrm{Id}\ \mathrm{zero})$ where Id=$\lambda x\ x$.

There is a (typed and linear time) transformation between the AF2 representation and the TTR representation.

One way is trivial.

$\lambda n\ (n\ \mathrm{succ}\ \mathrm{zero}) : \forall x\ (\mathrm{Nat\_AF2}(x)\to \mathrm{Nat\_TTR}(x))$ where

$$\mathrm{Nat\_AF2}(x) = \forall X\ (\forall z\ (X(z) \to X(Sz)),\ X(0) \to X(x))$$

The other way is more tricky and uses the technic of storage operators (see[9,10]). It is -essentially- proved in [10] (p 28) that $\lambda\nu\ (\nu\ \rho\ \tau\ \rho)$ where :

$\tau= \lambda d\lambda f\ (f\ \mathrm{zero})$ $\quad\rho= \lambda y\lambda z\ (G\ (y\ z\ \tau\ z))$ $\quad G= \lambda x\lambda y\ (x\ \lambda z\ (y\ (s\ z)))$

can be typed of type $\forall x\ (\mathrm{Nat\_TTR}(x) \to \mathrm{Nat\_AF2}(x))$ and transforms, in linear time, the TTR representation of n to its AF2 representation.

Since the term given by Maurey can be typed - in TTR - with type $\forall x\forall y(\mathrm{Nat\_AF2}(x) \to \mathrm{Nat\_AF2}(y) \to \mathrm{Bool}(\inf(x,y)))$ it is easy to find a term of type $\forall x\ \forall y\ (\mathrm{Nat\_TTR}(x) \to \mathrm{Nat\_TTR}(y) \to \mathrm{Bool}(\inf(x,y)))$ that computes the inf in time O(inf) .

II Basic notations

The notations are standard (see [1], [8]). I adopt the following usual abbreviations:

$(a\ b_1\ b_2\ ...b_n)$ for $(...((a\ b_1)\ b_2)...b_n)$

$A_1,A_2,...,A_n$ ->B for $(A_1\text{->}(A_2\text{->}...(A_n\text{->}B)...)$

≈ is the β equivalence

nf(t) is the normal form of t .

hdnf(t) is the head normal form of t .

t ->_h t' : t reduces to t' by some steps of head reduction .

time(t) = the number of β reductions to go (by left reduction) from t to its normal form.

hdtime(t) = the number of β reductions to go (by head reduction) from t to its head normal form.

Main types

Nat = ∀x ((x –> x) –> (x –> x))

Bool = ∀x (x –> (x –> x))

List = ∀x ((Bool, x –> x) –> (x –> x))

Nat × Nat = ∀y ((Nat –> Nat –> y)->y)

Some constructors on these types

s = the successor = λnλfλx (f (n f x)) : Nat –> Nat

zero (also called false, nil) = λfλx x : Nat (also of type Bool, List)

true = λxλy x : Bool

not = λaλxλy (a y x) : Bool->Bool

cons = the concatenation on List = λbλlλfλx (f b (l f x)) : Bool, List –> List

Abbreviations

[n] =λfλx (f (f ...(f x)...))

$[a_0,...a_k]$ = λfλx (f a_0 (f ...(f a_k x)...))

{n} = (s (s ...(s zero)...)

$\{a_0,...a_k\}$ = (cons a_0 (cons ... (cons a_k nil)...))

Storage operators

The role of the storage operators is to force - during a head reduction - a call by value . For details on the computation, type and time see [9,10]

Nstore = λn (n H δ) : $\forall$o(Nat* $\to$ $\neg\neg$Nat)
where Nat* = $\forall$x (($\neg$x $\to$ $\neg$x) $\to$ ($\neg$x $\to$ $\neg$x)) and $\neg$x = x $\to$ o
δ= λf (f zero) and H= λxλy (x λz (y (s z)))

Nstore is a storage operator for Nat, that is (Nstore t_n g) reduces - by head reduction- to (g {n}) in time O(time(t_n)) if g is a variable and $t_n \approx [n]$
So time ((Nstore t_n G)) = O(time (t_n)) + time ((G {n}))

Bstore = λb (b λf (f true) λf (f false)) : $\forall$o(Bool* $\to$ $\neg\neg$Bool)
where Bool* = $\forall$x ($\neg$x $\to$ ($\neg$x $\to$ $\neg$x))
Bstore is a storage operator for Bool , that is (Bstore b g) reduces - by head reduction- to (g true) (resp (g false)) in time O(time(b)) if b $\approx$ true (resp false) and g is a variable

Lstore = λl (l H δ) : $\forall$o(List* $\to$ $\neg\neg$List)
where List* = $\forall$x ((Bool*, $\neg$x $\to$ $\neg$x) $\to$ ($\neg$x $\to$ $\neg$x))
H= λa (Bstore a λbλrλf (r λz (f (cons b z)))) and δ = λf (f nil)
Lstore is a storage operator for List, that is (Lstore l g) reduces - by head reduction- to (g $\{a_0 ...a_k\}$) in time O(time(l)) if g is a variable and l $\approx$ $[a_0,...a_k]$

III The inf term

Before giving *good_inf* I remind here *easy_inf* the " usual " term for the function : n, m $\to$ if n<m then n else m ; *easy_inf* is such that :
time ((*easy_inf* [n] [m])) = O(((inf(n,m))^2) (see [2])

easy_inf = λnλm (n A λp zero m) : Nat,Nat $\to$ Nat
where A = λuλm (m H <zero,zero> false) : (Nat $\to$ Nat) $\to$ (Nat $\to$ Nat)
H = λc <(s (c true)),(s (u (c true)))> : Nat $\times$ Nat $\to$ Nat $\times$ Nat
and <a,b> is λf (f a b)

It is more convenient to define first *inf* (= the function : n, m $\to$ if n<m then true else false) and then

good_inf (= the function : n, m $\to$ if n<m then n else m)
Nat,Nat $\to$ Nat
$\lambda n \lambda m$ (*inf* n m n m)

The two basic tricks of the algorithm are the following :

1) compare n and m in the following way : (this is the same idea as in [4])
Iterate the following function (with initial arguments (n, m, 0, 0) and local arguments (n', m', k', p'))
if m'=0 then answer false else if n'=0 then answer true else :
if $n'>2^{k'}$ and $m'>2^{k'}$ then iterate with arguments (n', m', (k'+1), k')
that is : compare n' and m' with $2^{k'+1}$, and remember that $n'>2^{k'}$ and $m'>2^{k'}$ else
if $n>2^{k'}$ and $m\leq 2^{k'}$ then answer false else
if $n\leq 2^{k'}$ and $m>2^{k'}$ then answer true else
if $n\leq 2^{k'}$ and $m\leq 2^{k'}$ then iterate with arguments ($n'-2^{p'}$, $m'-2^{p'}$, 0, 0)
that is : compare $n'-2^{p'}$ and $m'-2^{p'}$ where p' is the largest integer such that n' , m' > $2^{p'}$

2) compute $n-2^k$ or compare n to 2^k in the following way : iterate n times the decrementation of 1 starting from 2^k ; n is used as the iterator whereas 2^k - and its predecessors - are written in binary notation (the higher order bit being - on the opposite to the usual notation - on the right, that is at the end of the list of length k) . It is convenient to assume that the useless "0" bits of high order at the right of the representation l of an integer are kept, i.e the length of l and (pred l) are the same .
The main point is : since we are making head reductions, we donot have to compute entirely n - 2^k (see the proof of lemma 4) and so, even if n is much larger than 2^k, the time to compare n with 2^k is $O(k\ 2^k)$.

The next lemma is crucial and used without mention in almost all the other lemmas .

Lemma 0

Let u, v, $v_1,\ldots,v_n$ be λ terms and u' = hdnf(u). Then :
hdtime((u v_1 ... v_n))= hdtime(u) + hdtime((u' v_1 ...v_n))
hdtime(u[v/x]) = hdtime(u) + hdtime(u'[v/x]) .
proof : Easy , by induction on hdtime(u) . see [9,10] .

I now introduce - in the following lemmas - some sub-terms of the λ term *inf* and give their properties .

Lemma 1

let *pred* = λl (l G D false) : List – > List

where D = λb nil : Bool -> List , G = λaλyλb (b (cons a (y true)) (cons (not a) (y a))) : Bool, Bool->List, Bool -> List

then

1) if nf([a_0, ..., a_k]) ≠ [false, .., false] then (pred {a_0, ..., a_k}) ≈ [b_0, ..., b_k] where [b_0, ..., b_k] is the binary representation of the predecessor of the integer whose binary representation is [a_0, ..., a_k]

2) if the a_i are true or false then : time ((pred {a_0, ..., a_k})) = O(k)

proof : easy .

Lemma 2

let *test_list* = λlλnλm (l B true n m) : List,Nat,Nat – > Nat

where B=λbλr (b false r) then if n, m are variables :

1) (test_list [a_0, ..., a_k] n m) ≈ if nf([a_0, ..., a_k]) ≠ [false, .., false] then n else m

2) if the a_i are true or false then time ((test_list {a_0, ..., a_k} n m)) = O(k)

proof : easy .

Lemma 3

let *list* = λk (k cons_0 [true]) : Nat->List

where cons_0 = λlλfλx (f false (l f x)) then :

1) (list [k]) ≈ [false, ..., false, true]

2) time((list {k})) = O(k)

proof : easy .

Lemma 4

Let *next* = λgλl (test_list l (s (g l)) (Lstore (pred l) g)) :

(List – > Nat) – > (List – > Nat)

Let *Dif* =λnλk (n next λx zero (list k)) : Nat,Nat – > Nat

Let *Test* = λnλkλaλb ((Dif n k) λx a b) : Nat,Nat,Bool,Bool – > Bool

then

1) (Dif [n] [k]) ≈ [n-2^k] and (Test [n] [k] a b) ≈ if n>2^k then a else b

2) if a and b are variables then time ((Test {n} {k} a b))= O(k 2^k)

3) if 2^k<n≤$2^{(k+1)}$ then time((dif {n} {k})) = O(k.2^k)

proof :

1) is easy to see .

2) It follows from the properties of Lstore and the previous lemmas that if g is a variable, the a_i are true or false and l = $\{a_0, ..., a_k\}$ represents - in binary - a non zero integer p then hdnf (next g l) = (g $\{b_0, ..., b_k\}$) where $[b_0, ..., b_k]$ represents p-1 and hdtime ((next g l))= O(k) .
Thus let u=({n} next λx zero (list {k})) and
v= ({n} next λx zero (list {k}) λx a b) ;
- If n ≤ 2^k then u->$_h$ (λx zero l') for some l' and so u ≈ zero , v ≈ b and time(v) = O(k .2^k)
- If n > 2^k then u->$_h$ (next G {false, ..., false}) in time O(k.2^k) , with G= ($next^{n-2^k}$ λx zero) and so v->$_h$ (s (G {false, ..., false}) λx a b) ->$_h$ a (this last reduction is in 4 steps !); and time ((Test {n} {k} a b))= O(k 2^k) . This proves 2) .
3) Finally it is easy to see that (($next^p$ λx zero) {false, ..., false}) reduces to [p] in time O(p k) . This proves 3) .

Lemma 5

Let n, m, p be integers such that 2^p < n, m ≤ 2^{p+1}, g is a variable and u = (Nstore (Dif {m} {p}) (Nstore (Dif {n} {p}) g)) , then
hdnf(u) = (g {m-2^p} {n-2^p}) and hdtime(u) = O(p 2^p)
proof : This follows easily from the lemma 4 and the properties of Nstore .

Lemma 6

Let *Iteration* = λgλnλmλkλp (m λx (n λx (Test n k (Test m k (g n m (s k) k) false) (Test m k true Iter)) true) false) : (Nat,Nat,Nat,Nat –> Bool) –> (Nat,Nat,Nat,Nat –> Bool)
where Iter = ((Nstore (Dif m p) (Nstore (Dif n p) g)) zero zero)

Let n, m, k, p be integers, g a variable and u be the head normal form of (Iteration g {n} {m} {k} {p}) then :
1) - if m=0 then u= false else
- if n=0 then u= true else
- if n>2^k and m>2^k then u= (g {n} {m}, {k+1}, {k}) else
- if n>2^k and m≤2^k then u= false else
- if n≤2^k and m>2^k then u= true else
- if n≤2^k and m≤2^k then u= (g {n-2^p} {m-2^p} zero zero)
2) hdtime((Iteration g {n} {m} {k} {p})) = O(k 2^k)
proof : This follows from the lemma 5 .

Definition

Let *inf* = λnλm ((s^8 n) Iteration Init n m zero zero) : Nat,Nat –> Bool
where *Init* = λnλmλpλq true : (Nat,Nat,Nat,Nat –> Bool)

Theorem

For every natural numbers n and m :

1) (inf [n] [m]) ≈ [inf(n,m)]

2) time((inf [n] [m])) = O(inf(n,m) . log(inf(n,m)))

Proof : We show that at most inf(n,m) +8 iterations are enough to find the minimum . It is then clear that the roles of n and m are - in fact - symetric; assume then that $n \le m$ and let k be such that $2^k < n \le 2^{(k+1)}$. Note that Init - the initialisation of the iteration - will then never be used and so any thing - of the good type - would in fact do .

- If $m > 2^{(k+1)}$: the algorithm find the minimum in k+2 iterations and the computation time is $O\left(\sum_{i=1}^{k+2} i\, 2^i\right) = O(k\, 2^k) = O(\text{inf Log(inf)})$.

- If $m \le 2^{(k+1)}$: after k+2 iterations the head normal form is (iteration^r Init $\{n-2^k\}$ $\{m-2^k\}$ zero zero) for some r . By repeating the argument (since $n-2^k \le 2^k$) it is then clear that the maximum number of iterations to find the minimum is : (k+2) +(k+1)+...+1 = (k+2)(k+3)/2 which is easily seen to be less than 2^k+8 , and that the computation time is at most :

$$\sum_{i=1}^{k+1} \sum_{j=1}^{i+1} O(j.2^j) = O(k.2^k) = O(\text{inf log(inf)}) .$$

The complete term

The following term has been tested on computers . The experiences made show that the computation time (number of β left reductions) is less than 300 inf log(inf) .

s = λnλfλx (f (n f x))

zero =λfλx x

nil= λfλx x

false = λfλx x

true= λxλy x

cons =λbλlλfλx (f b (l f x))

d1= λf (f zero)

H1= λxλy (x λz (y (s z)))

Nstore = λn (n H1 d1)

Bstore =λb (b λf (f true) λf (f false))

d2 = λf (f nil)

H2= λa (Bstore a λbλrλf (r λz (f (cons b z))))

Lstore =λl (l H2 d2)

B=λbλr (b false r)

test_list =λlλnλm (l B true n m)

cons_0 = λlλfλx (f false (l f x))

list =λk (k cons_0 (cons true nil))

not = λaλxλy (a y x)

G = λaλyλb (b (cons a (y true)) (cons (not a) (y a)))

D = λb nil

pred= λl (l G D false)

next =λgλl (test_list l (s (g l)) (Lstore (pred l) g))

Dif =λnλk (n next λx zero (list k))

Test=λnλkλaλb (n next λx zero (list k) λx a b)

Init =λnλmλpλq true

Iteration =λgλnλmλkλp (m λx (n λx (Test n k (Test m k (g n m (s k) k) false) (Test m k true ((Nstore (Dif m p) (Nstore (Dif n p) g)) zero zero))) true) false) inf =λnλm (s (s (s (s (s (s (s (s n))))))) Iteration Init n m zero zero)

good_inf =λnλm (inf n m n m)

IV a term in TTR

Proposition 1

There is a term of type ∀x ∀y (Nat_TTR(x), Nat_TTR(y) -> Bool(inf(x,y)) that computes the inf function in time O(inf)
where Bool(b) is the TTR (or AF2 - it's the same !) type for the booleans i.e Bool(b) := ∀X(X(true), X(false) -> X(b)) and inf is specified by : inf(0,y)=true inf(Sx,0)=false inf(Sx,Sy)=inf(x,y) .

proof : this follows easily from the linear time transformation from TTR to AF2 mentionned in the introduction and the next lemma .

Lemma

The term λnλm ((n F1 λx true) (m F2 λx false)) where
F1=F2=λfλg (g f) has in TTR the type :
∀x ∀y (Nat_AF2(x), Nat_AF2(y) -> Bool(inf(x,y))

proof : This typing is - essentially - due to JL Krivine (see [6]).
Let U be such that :
U(x) <=> ∀y(∀z(U(z)->Bool(inf(Sz,y)))->Bool(inf(x,y)))

Fact 1 : |— F1 : ∀x(U(x)->U(Sx))
proof : f:U(x), g: ∀z(U(z)->Bool(inf(Sz,y))) |— (g f) :Bool(inf(Sx,y) .
So f:U(x) |— λg (g f) : U(Sx) .

Fact 2 : |— λx true : U(0)
proof : |— true : Bool(true) = Bool(inf(o,y))

Fact 3 : n:Nat(x) |— (n F1 λx true) : U(x)

Fact 4 : |— F2 : ∀y(∀x(U(x)->Bool(inf(Sx,y))) -> ∀x(U(x)-> Bool(inf(Sx,Sy)))
proof : f: ∀x(U(x)->Bool(inf(Sx,y))),
g:U(x) {<=> ∀y(∀z(U(z)->Bool(inf(sz,y)))->Bool(inf(x,y)))} |— (g f) : Bool(inf(x,y)) and Bool(inf(Sx,Sy))=Bool(inf(x,y))

Fact 5 : $\vdash \lambda x \text{ false} : \forall x(U(x) \to Bool(inf(Sx,0)))$

Fact 6 : $m{:}Nat(y) \vdash (m \text{ F2 } \lambda x \text{ false}) : \forall x(U(x) \to Bool(inf(Sx,y))) = \forall z(U(z) \to Bool(inf(Sz,y)))$

Fact 7 : $n{:}\ Nat(x),\ m{:}Nat(y) \vdash ((n \text{ F1 } \lambda x \text{ true})\ (m \text{ F2 } \lambda x \text{ false})) : Bool(inf(x,y))$
proof : by fact 3 {and $U(x) \Leftrightarrow \forall y(\forall z(U(z) \to Bool(inf(sz,y))) \to Bool(inf(x,y)))$ } and fact 6.

Bibliographie

[1] H.Barendregt The lambda calculus ; North holland (1984)
[2] L.Colson Représentation intentionnelle d'algorithmes dans les systèmes fonctionnels. Thèse (université de Paris VII (1991))
[3] " About intensional behaviour of primitive recursive algorithms LNCS 372 (1989)
[4] R.David a primitive recursive algorithm for the inf function CRAS T317 série 1 1993 p 899-902
[5] " About the asymptotic behaviour of primitive recursive algorithms (in preparation)
[6] JL.Krivine un algorithme non typable dans le system F ; CRAS T 304 1987
[7] JL.Krivine& M.Parigot Programming with proofs; in SCT 87 (1987), Wendish-Rietz; in EIK 26 (1990) pp 146-167
[8] JL.Krivine lambda calcul , types et modèles; Masson (1990)
[9] " Opérateurs de mise en mémoire et traduction de Gödel; Arch.for Math. Logic 30 (1990) pp 241-267
[10] K.Nour Opérateurs de mise en mémoire en lambda calcul pur et typé Thése (Université de Chambéry (1993))
[11] M.Parigot On representation of data in lambda calculus. to appear in LNCS
[12] " Recursive programming with proofs; in TCS 94 (1992) pp 335-356
[13] P.Roziere un résultat de non typabilité dans Fω ; CRAS T 309 1989