THE THEORY AND PRACTICE OF TRANSFORMING CALL-BY-NEED INTO CALL-BY-VALUE

ALAN MYCROFT

ABSTRACT

Call-by-need (which is an equivalent but more efficient implementation of call-by-name for applicative languages) is quite expensive with current hardware and also does not permit full use of the tricks (such as memo functions and recursion removal) associated with the cheaper call-by-value. However the latter mechanism may fail to terminate for perfectly well-defined equations and also invalidates some program transformation schemata.

Here a method is developed which determines lower and upper bounds on the definedness of terms and functions, this being specialised to provide sufficient conditions to change the order and position of evaluation keeping within the restriction of strong equivalence. This technique is also specialised into an algorithm analogous to type-checking for practical use which can also be used to drive a program transformation package aimed at transforming call-by-need into call-by-value at 'compile' time.

We also note that many classical problems can be put in the framework of proving the strong equivalence where weak equivalence is easy to show (for example the Darlington/Burstall fold/unfold program transformation).

Dept. of Computer Science, University of Edinburgh

MOTIVATION

For a purely applicative language (no assignment or GOTO) call-by-need [Wadsworth 71] is a highly desirable parameter passing mechanism, since [Vuillemin 73] it is a safe evaluation mechanism in that it will give the mathematical result whenever the latter is defined and is more efficient than call-by-name.

Basically call-by-need is the same as call-by-name (passing of an expression bound in the calling environment) but with the proviso that the first reference to the parameter causes not only its evaluation but also the replacement of the parameter with the result of the evaluation thus making subsequent accesses much cheaper. It also has the advantage that it corresponds closely to the method a mathematician would use to evaluate an expression. Note that it retains the advantages of call-by-name in that parameters that are not referenced in a particular activation of the function will not be evaluated: this point is very important since evaluating an argument which should not be evaluated may result in the evaluator looping. To summarise, we have that

call-by-value evaluates a parameter exactly once,
call-by-name evaluates a parameter zero or more times,
call-by-need evaluates a parameter at most once.

The main disadvantage of call-by-value is that it may produce undefined values for (mathematically) well-defined expressions, for example consider evaluating

f(1,0) WHERE f(x,y) = IF x=0 THEN 0 ELSE f(x-1,f(x,y))

using call-by-value.

Note that this point is especially relevant to the typical user of a symbolic algebraic manipulation (SAM) system, who is mathematically sophisticated but computationally naive, because he will write similar (but less contrived) recursive definitions and find the system merely moans that time is up!

For the user of a SAM system it is desirable to use call-by-need as the parameter passing mechanism in order that

1. The recursive definitions are as fully defined as possible.
2. The print program may drive the evaluation process so that printing an infinite expression will run out of time when printing it and not during the evaluation prior to printing.

The counter arguments favouring call-by-value are:

1. Call-by-need is clumsy to implement on current architectures (in that each parameter to a function needs to carry a closure around with it). This leads to differences in efficiency which are put by various sources at factors of between 2 and 10; the situation becoming rather worse in a full lazy evaluator.
2. In a call-by-value the system can use memo-functions to avoid recomputation. These will be (semantically) invisible to the user, and

encourage the development of clean "mathematical" rather than "sequential" programs. For example consider:

```
f(n) = IF n<2 THEN 1 ELSE f(n-1)+f(n-2)
                                              (Fibonacci numbers)
```

or

```
C(n,r) = 1  IF r=0 OR r=n
       = C(n-1,r-1) + C(n-1,r) OTHERWISE
                                              (Pascal's triangle)
```

Here evaluation (with $r = n/2$ in the second example) requires in the order of 2^n function calls using the standard implementation. This cost can be made linear in n in exchange for storage by saving the (arguments,result) pairs for previously computed values of f or C. (This technique is called 'memo'ing the function). Unfortunately when using call-by-need, we cannot look at the argument values since to do so causes evaluation effectively at the time of call and hence is equivalent to a call-by-value regime. Thus call-by-value has advantages which extend far beyond current hardware limitations - since exponential costs can rarely be tolerated.

3. Call-by-need does not permit the standard methods of recursion removal to be used, for example:

```
f(x,y) = IF x=0 THEN y ELSE f(x-1,y+1)
```

requires one new closure to be created for y in each recursive call; these all being evaluated 'domino fashion' when y is finally used. For further discussion see [Lang 77].

It is worth noting the great similarity between the optimisations furnished by call-by-need over call-by-name and by using memo-functions. In both cases the effect is to avoid recalculation of known values, and both are optimisations which can convert an exponential cost into a linear one (unlike traditional compiler optimisations to remove common sub-expressions which can only save at most a linear factor in the cost).

Another reason for using the call-by-need parameter passing mechanism is that call-by-value invalidates some program transformation schemata. For example consider the fold/unfold transformation [Darlington+Burstall 77] which replaces a call of a function by its body or vice versa.

The program segment

```
IF e1 THEN e2 ELSE e3                                   ... (1)
```

is equivalent to the segment

```
f(e1,e2,e3) WHERE f(x,y,z) = IF x THEN y ELSE z         ... (2)
```

only if the call-by-need (or name) parameter passing regime is used since the early evaluation of e3 otherwise necessitated by call-by-value in (2) may cause infinite looping. For example compare

```
fact(n) = IF n=0 THEN 1 ELSE n*fact(n-1)
```

with

```
fact(n) = f(n=0, 1, n*fact(n-1))  WHERE f(b,x,y) = IF b THEN x ELSE y
```

the latter being undefined for all n when using call-by-value.

The above arguments suggest that call-by-value is more efficient but call-by-need preferable on aesthetic/definedness considerations. So techniques are

herein developed which allow the system to present a call-by-need interface to the user but which performs a pre-pass on his program annotating those arguments which can validly be passed using call-by-value. Thus the spirit is similar to, and unifies and implements some of the ideas in Schwarz [77].

Note that the technique only provides the information "It is safe to pass certain parameters by value" and is not claimed to detect all such cases. The problem of detecting all such cases is actually not effectively computable, for example consider:

F(x,y) = IF P(x) THEN y ELSE 0

where P(x) is true for all values of x. The argument y will, then, always be evaluated and so could be safely passed by value. This fact is impossible to detect uniformly since, in any sufficiently rich domain, there are tautologies which cannot be detected by any (pre-specified) algorithm (e.g. "The halting problem" for Turing machines). Currently no attempt is made to detect similar tautologies and hence the system "plays safe" and suggests that y is passed by need. In practice (see section on pragmatics) this limitation does not stop most cases of call-by-value being detected.

There is an analogy between the system described here and the "most general type" inferrer used in a language such as ML [Gordon et al 79] which even extends to cover the sort of example above; for example consider the declaration

LET x = IF true THEN 1 ELSE NIL

then the ML type inferrer will produce an error for the type of x whereas in fact it is well (but inelegantly) defined.

In order to be able to change the order of evaluation (e.g. changing call-by-need into call-by-name) without changing the semantics we require referential transparency in the language under study. Applicative languages normally possess this property, with the proviso that error situations (e.g. 1/0) do not result in 'jumpout' action and merely return a special error value to the calling function.

The central stage in the development of the call-by-value detection system is the definition of maps # and b which are semi-decision procedures for termination on recursion equations. The idea is that # will map ALL terminating closed forms onto 1, and SOME non-terminating terms onto 0, and b maps ALL non-terminating terms onto 0. By investigating the effect of # and b with their semi-homomorphic properties on recursion equations we can see the gross structure of the recursion and occurrences of references to arguments without the clutter of detail present in the original equations.

FORMALISM

The formal system in which the theory is developed is that of a scheme, S, of recursion equations together with one standard and two non-standard interpretations.

$S = \{F_i(X_1 \ldots X_{ki}) = U_i;\ 1 \leq i \leq n\}$

where the U_i are (finite) terms defined by the grammar with start symbol T and axioms

- $T ::= X_j$ (individual parameters)
- $T ::= A_j(T_1 \ldots T_{rj})$ (system functions)
- $T ::= F_j(T_1 \ldots T_{kj})$ $1 \leq j \leq n$ (user functions)

We insist that U_i contains no X_r for $r > k_i$. Here all base constructs (including the conditional which is normally regarded as syntax) are considered to be members of the (A_i); note that the A_i are base constants when $r_i = 0$.

An interpretation I, of S, consists of a pair $\langle D,(a_j)\rangle$ where D is a domain and the a_i are continuous functions from $D^r \to D$ where $r = r_i$ is the arity of A_i.

An interpretation I induces for S an interpretation of the function symbols F_i defined in the usual manner as the least fixpoint.

Now let $2 = \{0,1\}$ be the two element Boolean lattice ordered by $0<1$ and use the standard Boolean connectives (we use 0 and 1 to avoid confusion with elements of D).

NOTATION

We use the following notation to simplify expressions:

1. $[P,Q]$ is a partition of a set U if U is the disjoint union of P and Q.
2. Let F be a function with arity k then for a partition $[P,Q]$ of $\{1 \ldots k\}$ we define $F(a/P, b/Q)$ to mean
   $F(x_1 \ldots x_k)$ where $x_i = a$ if i in P
   $= b$ otherwise
3. We will also use $R(x_Q)$ to mean $R(x_i)$ for all i in Q, where R is a predicate.

We next define two more interpretations in terms of $I = \langle D,(a_j)\rangle$ by

$I^{\#} = \langle 2, (a_i^{\#})\rangle$

and

$I^{b} = \langle 2, (a_i^{b})\rangle$

in the following manner: [the definitions are to be seen as monotonic functional extensions of the function

HALT: D -> 2 defined by
HALT(x) = 0 if $x = \bot$
= 1 otherwise]

For all partitions $[P,Q]$ of $\{1,2 \ldots r_i\}$ we define

$a_i^{\#}(0/Q, 1/P) = 0$ if for all $\{x_j$ in D: $1 \leq j \leq r_i\}$ such that $x_Q = \bot$, $x_p \neq \bot$ we have $a_i(x_1 \ldots x_{ri}) = \bot$

$= 1$ otherwise

and

$a_i^{b}(0/Q,1/P) = 0$ if there exists $\{x_j$ in D: $1 \leq j \leq r_i\}$ such that $x_Q = \bot$, $x_p \neq \bot$ and $a_i(x_1 \ldots x_{ri}) = \bot$

$= 1$ otherwise.

Clearly the $(a_i^{\#})$ and (a_i^{b}) are monotonic since we assume the (a_i) are

computable; and, as pointed out by a referee, this implies the conditions $x_p \neq \perp$ are superfluous, but we include them to aid the intuition. For any function g: D^r -> D we will write

$g^{\#}$, g^b: 2^r -> 2

to denote the functions constructed from g by the above technique.

EXAMPLE

Here we use the standard meaning for IF as the 3 argument sequential conditional; and PLUS as the usual (strict) operation on integers:

$IF^{\#}(p, x, y) = p \wedge (x \vee y)$
$IF^{b}(p, x, y) = p \wedge x \wedge y$
$PLUS^{\#}(x, y) = x \wedge y$
$PLUS^{b}(x, y) = x \wedge y$

We can also cope with parallelism, for example

$PIF^{\#}(p, x, y) = (p \vee x) \wedge (p \vee y) \wedge (x \vee y)$

where

PIF(p,x,y) = x if p = TRUE
= y if p = FALSE
= x if x = y
= $\perp$ otherwise.

It is useful to observe that these equations can be read in English to help understanding: the first one (for $IF^{\#}$ and IF^{b}) reads

IF(p,x,y) needs to evaluate both p AND at least one of x OR y.
IF(p,x,y) terminates if p AND x AND y do.

We can further justify this approach of identifying all non-undefined values by noting that any two partially correct evaluation mechanisms (those that give the same result when both terminate) are weakly equivalent (i.e. LUB(E1,E2) exists where the E_i are the results from the two evaluation mechanisms) and hence it is only necessary to discover places where undefinedness can creep in. In passing we note that this point is still relevant in higher order languages since in a well-typed language with flat base domains the universe of discourse is D_n for some n where

D_0 is flat and
$D_{i+1} = D_i + (D_i \to D_i)$

Note (see section on non-discrete domains) that allowing D to be non-discrete might mean that we fail to obtain a very close bound here without more machinery.

$I^{\#}$ and I^b are (non-comparable) interpretations abstracting I in the sense of Cousot+Cousot [77]. However here we use the two non-standard interpretations to "sandwich" the standard interpretation and thus it is important to note that one of the interpretations is "upside-down" relative to the above work.

We naturally define $f^{\#}$ and f^b corresponding to the F_i as the least fixpoints of their defining equations in S under the interpretations $I^{\#}$ and I^b.

Let E, $E^{\#}$, E^b be respectively the denotation functions for terms under I,

$I^{\#}$, I^{b}. Then for all terms e (possibly with free variables) we can associate functions

$$E[\![e]\!] : D^{K} \to D; \quad E^{\#}[\![e]\!], E^{b}[\![e]\!]: 2^{K} \to 2$$

where K is a set containing all the free variables of e.

We have (by an induction left to the reader) that

$$E^{b}[\![e]\!] \leq (E[\![e]\!])^{b} \leq (E[\![e]\!])^{\#} \leq E^{\#}[\![e]\!]$$

for all terms e, the centre inequality reducing to an equality if e has no free variables. The outermost inequalities reduce to equalities if e is of the form $A_i(X_1 \ldots X_{ri})$. Examples like (*) below can be used to show the inequalities may be strict.

This result enables us to deduce that the definition of # and b on the A_i extends to the F_i to give useful information on termination in I. The result is, for all partitions [P,Q] of $\{1,2 \ldots k_i\}$,

$[F_i^{\#}(0/Q, 1/P) = 0$ implies
for all (x_i) such that $x_Q = \bot$ we have
$F_i(x_1 \ldots x_{ki}) = \bot$

and

$F_i^{b}(0/Q, 1/P) = 1$ implies
for all (x_i) such that $x_p \neq \bot$ we have
$F_i(x_1 \ldots x_{ki}) \neq \bot$]

Note that we lose the half of the if and only if of the definition - this is due to the operation of composition rather than recursion, for example take

$$e = [\![\text{IF true THEN } \bot \text{ ELSE } 91]\!] \qquad (*)$$

which gives

$$E^{\#}[e] = 1$$

in spite of the fact that $E[e] = \bot$.

Now these are exactly the two conditions required for the detection of situations where call-by-need may be optimised to call-by-value. The first gives us conditions on a function such that (some of) its formal parameters may uniformly over calls be evaluated before evaluation of the function body and the second gives us conditions on actual parameters which may be evaluated prior to calling uniformly over the head function symbol. We now consider these remarks in more detail with examples:

A condition for the actual parameter e_i associated to formal parameter x_i in a call $F(e_1 \ldots e_k)$ to be safely (i.e. without disturbing the meaning of the call - see Vuillemin [73]) evaluated before calling F is precisely that $F(x_1 \ldots x_k)$ is undefined whenever x_i is. Taking $Q = \{i\}$ in the above equation for $F^{\#}$ gives us a useful sufficiency condition for this to hold.

Similarly, to illustrate the use of F^{b}, suppose we have the following equation:

$$F(x,y) = G(x, y+1) + y$$

Consideration of $F^{\#}$ in the above manner (using the fact that

$+^{\#}(x,y) = x \wedge y$

in the usual interpretation of +) enables us to deduce that y may be passed by value to F. Now this fact means that $y \neq \underline{\perp}$ in the body of F and correspondingly we have that

$E^{b}[y] = 1$

there. Now we use the fact that (giving + its standard meaning)

$+^{b}(x,y) = x \wedge y$

and hence that

$E^{b}[(y+1)] = 1$ since $E^{b}[1] = 1$

This shows that the second parameter to G in the <u>call</u> G(x, y+1) always terminates and hence in implementation terms we may choose to evaluate y+1 prior to the call and give G an <u>evaluated</u> call-by-need thunk rather than the standard unevaluated thunk to be evaluated on its first reference, without disturbing the semantics of the call. A further optimisation still is that, if all calls to G have the above property then we know that $x_2 \neq \underline{\perp}$ in <body> where

$G(x_1,x_2) = body$

and accordingly that x_2 may be classed as a value parameter to G. So, having established this we then have

$E^{\#}[x_2] = E^{b}[x_2] = 1.$

See also the section on transforming programs to use call-by-value below.

To derive solutions for the $f_i^{\#}$ and f_i^{b} which are fixpoints of the systems $\langle S,I^{\#}\rangle$ and $\langle S,I^{b}\rangle$ we develop the following theory: (the aim is not to derive solutions by evaluation but rather by examination of their textual definition by forming

$\lim T^{i}(BOTTOM)$

where T is the functional to be defined below).

Define L by:

$L = (2^{k1} \to 2) \times (s^{k2} \to 2) \times \ldots \times (2^{kn} \to 2)$

The space L has a natural lattice structure defined componentwise by

$(p1, \ldots pn) \leq (q1, \ldots qn)$ if and only if
$(pi \& \sim qi$ is identically zero; $(1 \leq i \leq n))$

Now define T, a transformation on L by

$T: (H_1, \ldots H_n) \to (H_1', \ldots H_n')$

where

$H_i'(x_1, \ldots x_{ki}) = U_i[H_j/F_j;\ 1 \leq j \leq n;\ a^{\#}/A]$

Defining

BOTTOM = (lambda $x_1 \ldots x_{k1}$. 0,
lambda $x_1 \ldots x_{k2}$. 0,
. . . . . .
lambda $x_1 \ldots x_{kn}$. 0)

and TOP = BOTTOM[1/0]

gives the top and bottom of the lattice L, respectively.

The sequence

BOTTOM, T(BOTTOM), T(T(BOTTOM)) ....

gives Kleene's ascending chain (AKC) on the finite lattice L. Hence all these terms are the same from some point onwards with limit value T*(BOTTOM) say. Define T*(TOP) similarly. Now by construction T*(BOTTOM) and T*(TOP) are fixpoints of $\langle S, I^{\#}\rangle$ with all other fixpoints between these two. The fixpoints of $\langle S, I^{b}\rangle$ are similarly defined.

Note now a couple of interesting points;

1. T*(TOP) and T*(BOTTOM) are in general distinct
2. Not all points such that $T^*(BOTTOM) \leq X \leq T^*(TOP)$ are fixpoints of $\langle S, I^{\#}\rangle$

For proof consider

F(x,y,z) = IF x=0 THEN y*z ELSE F(x-1,z,y)

This gives

$f^{\#}(x,y,z) = x \wedge (y \wedge z \vee f^{\#}(x,z,y))$

and hence

$T^*(BOTTOM)\ (x,y,z) = x \wedge y \wedge z$
$T^*(TOP)\ (x,y,z) = x$

also

$H(x,y,z) = x \wedge y$

is between T*(TOP) and T*(BOTTOM) but is not a fixpoint.

The difference between the modes of parameter passing implied by T*(BOTTOM) and T*(TOP) is merely the difference in how the calculation proceeds in the evaluation of F(-1, $\perp$, 0); the first case implying passing (x,y,z) by value and the second just (x). In the call-by-value (for x,y,z) manner F is $\perp$ initially (upon evaluation of the value parameter $\perp$), but in call-by-need (for y,z) $\perp$ is never referenced, however the evaluator loops since the termination condition x=0 is never true. This corresponds to the inductive argument that if F is to terminate then the second argument in the initial call must be evaluated and its evaluation terminate. Thus we see that the fact that T has more than one fixpoint allows the system to be undefined in more than one way, but of course any two undefined values are indistinguishable (except by looking at the internal computation history), and of course the minimal fixpoint of T gives a valid mode of evaluation of parameters. In fact it follows [Vuillemin 73] that any point above the minimal fixpoint defines a mode of evaluation which gives the correct result but there may be differences in the way undefined results are achieved. (I.e. which particular infinite computation the system pursues.)

The existence of points (like H in the above example) which are above the minimal fixpoint (and so define safe evaluation strategies) but which are not themselves fixpoints is now explained:

The fixpoints of T correspond to the "consistent" modes of evaluation in the

following sense:

A mode of evaluation is consistent if it is safe and no argument which is passed by need to a function will inevitably (after a bounded number of further passing by need) be evaluated.

To return to the case of H we can see that it is not a consistent point of T, and so cannot define a sensible mode of evaluation of parameters for F.

It is interesting to note that, in the above example, there the proof that, if F is to terminate, then it must reference its second and third arguments must be based on induction on the computation path. #, however, has the induction 'built in' and so the proof merely consists of case analysis to see how 0 (=1#) can propagate.

PRAGMATICS

For use of the theory above in an algorithm the iteration produced is refined to be both more convenient and more rapidly convergent.

Define

$$Z(H) = Z_n(Z_{n-1}(\ldots (Z_1(H))\ldots))$$

where

$$Z_i(H_1 \ldots H_n) = (H_1 \ldots H_{i-1}, H', H_{i+1} \ldots H_n)$$

where

$$H'(x_1 \ldots x_{ki}) = U_i[H_j/F_j;\ 1 \leq j \leq n;\ a^{\#}/A]$$

Note that Z (like T) is monotonic since it is the result of tupling and composing monotonic functions. We prove that Z* (BOTTOM) = T*(BOTTOM) to show that Z and T give the same result. Z is also more convenient for implementing the iteration as it can be written as n single assignments in a loop rather than the one n-way multiple assignment required by T.

Further improvement in the speed may be effected by the following technique: firstly associate the call-structure graph with the function definitions (the call-structure graph is the directed graph obtained by considering function names as vertices and having an edge from f to g if and only if f contains a call to g in its body). Now partition this graph into its strongly connected components; giving a directed acyclic quotient graph; the strongly connected subgraphs can be analysed by the use of the Z (or T) iteration and the quotient graph is trivial to analyse - we flatten its partial order into a total order and analyse the strongly connected subgraphs according to this order.

A program has been written by the author (in LISP) to implement the above algorithm. A sample run is given below for a simple example and the system has been used on a text-formatter written by Martin Feather in NPL without knowledge of this system. NPL normally has a call-by-value semantics and as a guide to the utility of the system, most paramameters in the paginator were detected as being safely passable by value upon assuming the program should conform to call-by-

need semantics.

TRANSFORMING PROGRAMS TO USE CALL-BY-VALUE

The oustanding cases where the system did not detect call-by-value were due to the following form of recursion in which we test one parameter to give a 'default' value or embark upon a recursive call:

LET mult(x,y) = IF x=0 THEN 0 ELSE mult(x-1,y)+y

the trouble about this case being that it is impossible (without further knowledge) to discover whether the user intended mult(0,$\perp$) to give 0 or $\perp$ - the call-by-need semantics indicate 0 and so y cannot be passed by value without extra knowledge. Of course for any particular call b may be used to detect if the actual parameter terminates and hence optimise the call.

Here I suggest a method by which a program transformation system [Darlington+ Burstall 77] might be driven in order to transform out such non-strict functions by replacing them with strict functions and the basic non-strict conditional function which is well-known to compile and interpret efficiently.

Note that the "ELSE" branch of the above conditional expression satisfies

$E[\![\text{mult}(x-1,y) + y]\!] = x \wedge y$

and hence is strict. So we can replace all calls mult(e1,e2) with

IF e1=0 THEN 0 ELSE mult1(e1,e2)

WHERE mult1(x, y) = mult(x-1, y) + y

and compile all calls to mult1 using call-by-value. But now a priori all calls to mult have the property that the second actual parameter must terminate (if it does not then neither can mult1 by considering b). Hence mult can also be treated as a strict function and compiled appropriately. We can actually do rather better than this by unfolding the call to mult in mult1 and refolding to use the definition of mult1 to get

mult1(x,y) = (IF x-1 = 0 THEN 0 ELSE mult1(x-1,y))+ y

to obtain a strict version of mult to replace the original non-strict version at the expense of doing the test before calling mult. This cost is significantly cheaper than the cost of merely setting up the closure for the second argument for mult.

I call the above technique **rotational refolding of the function mult.** This has an intuitive meaning seen by noting that the infinite tree representation for mult has alternate '+' and 'IF' nodes in its infinite backbone. Then the definition of mult1 is just obtained by taking a different ('+' instead of 'IF') starting point for the folding into finite form. The proof of strong correctness for this type of fold/unfold is much easier than the general case.

```
.R VALARG
*(DEF FACT1 (X)
*     (IF (EQ X 0) 1 (TIMES X (FACT1 (SUB1 X)))))
FACT1
*(DEF FACT2 (X Y)
*     (IF (EQ X 0) Y (FACT2 (SUB1 X) (TIMES X Y))))
FACT2
*(DEF G (X Y Z) (IF X (PLUS Y Z) (DIFFERENCE Y Z)))
G
*(DEF H (X) 3)
H
*(DEF UNDEF (X) (IF (EQ X 0) (UNDEF X) (UNDEF (SUB1 X))))
UNDEF
*(DEF MY-IF (B X Y) (IF B X Y))
MY-IF
*(START) {; see if it all works}
MY-IF : Args (1) may be passed by value
UNDEF : *** totally undefined
H :     *** independent of args
G :     Args (1) (2) (3) may be passed by value
FACT2 : Args (1) (2) may be passed by value
FACT1 : Args (1) may be passed by value
(2 ITERATIONS)
```

NON-DISCRETE DOMAINS

Lazy CONS (see for example Friedman+Wise [76]) gives us some problems since, using the above notation we get

$CONS^{\#}(x,y) = CONS^{b}(x,y) = 1$
$Hd^{\#}(x) = x$
$Hd^{b}(x) = 0$

This unfortunately gives us a very bad bound and we now need to have some knowledge of list structures as our homomorphic image, instead of just $\{0,1\}$, in order to deduce those substructures whose evaluation can be safely moved due to the fact that we no longer have

$x \leq y \iff x=y \text{ OR } x=\perp$.

The author is examining using the notion of regular trees to approximate the limits of the (possibly infinite) Kleene sequences in the obvious image domain

D where $D = 2 + D \times D$

to tackle this problem.

REFERENCES

Berry, G., Bottom-up computation of recursive programs, Research Report 133, IRIA-Laboria, 78150 Le Chesnay, France (1975).

Cousot,P. and Cousot,R., Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, Proc. of Conference on Principles of Programming Languages, Los Angeles (1977).

Darlington, J. and Burstall, R.M., A transformation system for developing recursive programs, JACM, Vol. 24, No. 1 (1977).

Friedman, D.P. and Wise, D.S., CONS should not evaluate its arguments, Proc. of 3rd International Colloquium on Automata, Languages and Programming, Edinburgh (1976).

Gordon, M.J.C., Milner, A.J.R.G. and Wadsworth, C., Edinburgh LCF, Lecture Notes in Computer Science, Springer-Verlag (1979).

Gordon, M.J.C., Milner, A.J.R.G., Morris, L., Newey, M. and Wadsworth, C., A meta-language for interactive proof in LCF, Proc. of 5th ACM STGACT-SIGPLAN Conference on Principles of Programming Languages, Tucson, Arizona (1978).

Lang,B., Threshold evaluation and the semantics of call-by-value, assignment and generic procedures, Proc. of Conference on Principles of Programming Languages, Los Angeles (1977).

Schwarz, J., Using annotations to make recursion equations behave, DAI Research Report No. 43, Dept. of Artificial Intelligence, University of Edinburgh (1977). Also to appear in IEEE Transactions on Software Engineering.

Vuillemin, J., Proof techniques for recursive programs, Ph.D. thesis (chapter 2), published as a paper in 1974 as Correct and optimal implementations of recursion in a simple programming language, Journal of Computer and Systems Sciences, 9, 332-354 (1974).

Wadsworth, C., Semantics and pragmatics of the lambda calculus, Ph.D. thesis, Programming Research Group, Oxford University (1971).

FURTHER WORK

The author is currently pursuing several possible extensions to this work including applying the theory to non-discrete domains (lazy CONS), higher order systems, parallelism, and exploring the general idea of abstract interpretations. It seems that this work can also be applied to programs with parallel base functions in an attempt to determine 'how much' parallelism is necessary for their computation (again I am grateful to an (anonymous) referee) in that the sequential and parallel interpretations are weakly equivalent. See for example [Berry 75], but this is rather beyond the scope of this paper.

ACKNOWLEDGEMENTS

I am grateful for the support of the SRC during this work, and for the helpful advice of Rod Burstall, Robin Milner and David MacQueen at Edinburgh, and also for discussions with Gerard Huet, Jean-Jacques Levy and Bernard Lang at IRIA facilitated by the close links between these institutions. I am indebted to Arthur Norman (Cambridge University) for the original idea of trying to detect call-by-value in one of his "wouldn't it be nice if we had a system that ..." suggestions. Thanks to Eleanor Kerse for speedy typing at short notice.