BURT ROSENBERG UNIVERSITY OF MIAMI FALL 1992

Recursion is while-program computable

We show that recursively defined functions are while-program computable via least-upper-bound of partial functions by extension ordering. The purpose is to combine approaches from Kfoury, Moll and Arbib, "A Programming Approach to Computability", hereafter abbreviated KMA, thus introduce the abstract apparatus of posets of functions without diverging the discussion into denotational semantics.

Recursive programs are those which call themselves. Non-recursive programs with subroutine calls can be treated as macros. Therefore we reduce immediately to the case of self-recursion. We let the program refer to itself by a special symbol created and reserved for this purpose. We shall use the symbol R. To transform programs to indices, we need to include R in Table 1 on page 46 of KMF. We will agree that R has decimal representation 54.

Let's take as running example the factorial program,

$$\text{fact}(n) = \begin{cases} 1 & n = 0 \\ n\ \text{fact}(n-1) & n > 0 \end{cases}$$

written as,

```
begin
  if X1=0 then x1:=1
  else begin
    x2:=x1 ; x1:=pred(X1);
    R; {recursion symbol}
    X1:=X2*X1
  end
end
```

There are four problems to contend with,

1. How can we place a function inside itself? This will be resolved by defining a substitution function.

2. Straight textual substitution of the function inside of itself will cause a conflict of variable names. We will define a function which renames variable to avoid the problem.

3. The renaming and substitution must proceed an infinite number of times in order to reach the function. We will approach this problem two ways. First by the notion of a limit function, and second by the definition of a mechanism capable of substituting the function inside itself an arbitrary but finite number of times *on demand.*

4. Assigning semantics to R. Having introduced this symbol, we need to say what happens when while-program execution reaches this symbol. We shall avoid this by removing all references to R before actually calling the result a while-program.

We will restrict our attention, with loss of generality, to unary functions.

Substitution and Renaming

Let the *Recursion Kernel*, abbreviated RK, be the program which defines the recursive function perhaps using the special symbol R. Given a program P, such as RK, its encoding as an integer will be denoted $[P]$. The program encoded by integer i will be denoted P_i. So, for instance,

$$RK = P_{[RK]}.$$

Define a number theoretic function,

$$\begin{array}{rccl} \text{sub}: & \mathbf{N}^2 & \to & \mathbf{N} \\ & (i,j) & \mapsto & \begin{cases} \text{Form program } P \text{ by replacing each } \texttt{R} \text{ in } P_i \\ \text{with } P_j \text{ and return } [P] \end{cases} \end{array}$$

For instance, the following program is sub,

```
begin
  X3:=0 ; {this will be the output program}
  while X1<>0 do  {while there are symbols in the}
  begin           {input program...}
    X4:=head(X1) ;
    if X4<>'R' then X3:=X3||X4 {pass along symbol}
    else X3:=X3||X2 ; {insert Pj}
    X1:=tail(X1)
  end ;
  X1:=X3
end
```

Next we define a function

$$\begin{array}{rccl} \text{ren}: & \mathbf{N}^2 & \to & \mathbf{N} \\ & (i,k) & \mapsto & \begin{cases} \text{Returns } [P] \text{ where P is the program resulting from renaming all variables Xj in } P_i, \\ \text{except X1, to X(j+k-1).} \end{cases} \end{array}$$

That is,

$$\begin{aligned} X1 &\mapsto X1 \\ X2 &\mapsto X(k+1) \\ X3 &\mapsto X(k+2) \\ &\vdots \end{aligned}$$

It would be difficult to give an explicit while-program for this function, however the following description is precise. Scan P_i looking for the symbol X. For each non-X symbol, simply output the symbol unchanged. If X is found, collect into a string α the unbroken string of digits following X and output X followed by the string β derived from α as follows. If α is empty or is the string '1', then $\beta = \alpha$. Else convert α to integer, add $k-1$ to this integer, and β is the string representation of the sum.

As an example, if the factorial function defined above is taken as the recursion kernel,

$$\text{sub}([RK], \text{ren}([RK], 2))$$

would be,

```
begin
  if X1=0 then X1:=1
  else begin
    X2:=X1; X1:= pred(X1);
    begin
      if X1:=0 then X1:=1
      else begin
        X3:=X1; X1:=pred(X1) ;
        R;
        X1:=X3*X1
```

```
      end
    end;
    X1:=X2*X1
  end
end
```

This is not a valid while-program since R is semantically undefined. We correct that problem by simply causing the "call" to R loop indefinitely.

Let $[\bot]$ by the index of some empty function. We intend to remove all references to R by substitution in $[\bot]$. Returning to the factorial example,

$$\begin{aligned} \text{sub}([RK],[\bot]) \quad &\text{yeilds} \quad f_1 = \begin{cases} 1 & n=0 \\ \bot & \text{else,} \end{cases} \\ \text{sub}(\text{sub}([RK],\text{ren}([RK],2)),[\bot]) \quad &\text{yeilds} \quad f_2 = \begin{cases} 1 & n=0,1 \\ \bot & \text{else,} \end{cases} \\ &\vdots \end{aligned}$$

Or in general,

$$f_0 \leq f_1 \leq f_2 \leq \ldots \leq F$$

where $f_0 = \bot$, by convention, and F is the "limit" function of the sequence. It is the recursive function we seek, defined everywhere in the domain by an ever-expanding sequence of functions defined on larger and larger portions of the domain. We need to make the definition of a limit function precise and then show that it is while-program computable.

Interlude on Posets

Recall that a poset $\{S, \geq\}$ is a set S with a reflexive, anti-symmetric, transitive relation $\geq$. Our most important example for this article is the poset of partial functions by extension ordering. We most prove that this is a poset.

Theorem 1 *For any partial function f, $f \geq f$. If f and g are partial functions and $f \geq g$ and $g \geq f$, then $f = g$. If f, g and h are partial functions, and $f \geq g$ and $g \geq h$, then $f \geq h$.*

The proof is an exercise in the definition of extension ordering.

Posets can have two operations, the *meet* and the *join*. The meet of two elements a and b is the greatest lower bound of a and b. That is, it is an

element c such that $c \leq a$, $c \leq b$ and it is the greatest such element. It is not necessary for a meet to exist, that depends on the particular poset. If such a c does exist, one writes $c = a \wedge b$. In the poset of partial functions, given any two partial functions, their meet always exists, see the related problem on the midterm. However, the least upper bound of two functions is generally not definable. For instance, if f and g are two distinct total functions then the existence of any h such that $h \geq f$ and $h \geq g$ would force $f = g$, contradicting our assumption. However, if $f \leq g$ then the least upper bound does exist and is equal to g. Proof: $f \leq g$ and $g \leq g$ so g is an upper bound, but if h is also an upper bound then $h \geq g$. So g is the least upper bound. Then g is called the *join* of f and g and is written $g = f \vee g$.

Not only are finite joins definable if the two functions are compatible in extension ordering, so are infinite joins.

Theorem 2 *Let $\mathcal{F} = \{ f_1, f_2, f_3, \dots \}$ be a perhaps infinite family of partial functions such that for any integers i and j for which $i \leq j$ then $f_i \leq f_j$. Then a unique least upper bound of the family exists and is written,*

$$\bigvee_i f_i = F$$

PROOF: We define a function F and show that it is the least upper bound of the family $\mathcal{F}$.

$$F(n) = \begin{cases} f_i(n) & f_i(n) \text{ exists for some } i \\ \perp & \text{for no } i \text{ is } f_i(n) \text{ defined.} \end{cases}$$

The function F is well-defined: either $f_i(n)$ is defined for some i or for no i. If it is defined for more than one i, the value at n agrees. It is necessary to show that for all i, $F \geq f_i$. Suppose $f_i(n)$ is defined. Then from the definition of F, $F(n)$ is defined and $F(n) = f_i(n)$. So $F \geq f_i$ for any i. It is also necessary to show that if $G \geq f_i$ for all i, then $G \geq F$. Suppose $F(n)$ is defined. Then $f_i(n)$ is defined for some i. Because $G \geq f_i$, $G(n)$ is defined as well. And $G(n) = f_i(n) = F(n)$. So $G \geq F$. □

Examples

The integers with the usual ordering is a poset. A finite meet always exists,

$$a \wedge b = min(a, b),$$

as does a finite join,

$$a \vee b = max(a, b).$$

However an infinite join does not always exist even if it is of a family $\{ a_1, a_2, \dots \}$ for which $i \leq j$ implies $a_i \leq a_j$. However, the set of integers with a new element called infinity does always have infinite joins,

$$\bigvee_i a_i = \begin{cases} \max_i\{a_i\} & \text{if there is a finite such maximum} \\ \infty & \text{if the } a_i \text{ are unbounded} \end{cases}$$

Infinite joins do not always exist for the poset of the rationals with usual size ordering. A sequence of rationals can be made to converge on an irrational, say pi or the square root of 2. However, a bounded set of reals does always have a least upper bound and this is an important property of the reals. For instance, a circle can be approximated by a series of inscribed polygons, each approximating the circle more closely than the previous. The areas of these polygons form an increasing, bounded sequence $a_1, a_2, \dots$ whose least upper bound is said to be the area of the circle.

In the same way, if $f_0 \leq f_2 \leq f_2 \leq \dots$ is the sequence of functions approximating the recursive function F, we want to show that F is the least upper bound of the f_i. Thinking about the meaning of recursion, a recursive program is defined at input n if and only if after a finite number of calls to itself the processing of input n reaches a level where the calculation can be performed with no further recursion. That is, if and only if there exists and i for which $f_i(n)$ is defined. And this is exactly the definition if $F = \vee_i f_i$.

Therefore, to prove that recursion is no more powerful than while-programs we need to show that given a *certain* sequence of functions computable by while-programs, their join is also computable by while-programs. It is not enough just that this sequence is ascending, in the sense $i \leq j$ implies $f_i \leq f_j$. Because any function is the least upper bound of a set of functions with finite but increasing domain, and any function of finite domain is computable. We shall show that under the hypothesis that the sequence of functions is effectively enumerable, then the join is computable.

Main Theorem and Application

In this section the technique of dovetailed computation is used to prove the following theorem. The close of this section will argue its relevance for the major result of this article.

Theorem 3 *Let $\mathcal{F} = \{ f_1, f_2, \dots \}$ be an effectively enumerable sequence of computable functions such that $i \leq j$ implies $f_i \leq f_j$. Then $\vee_i f_i$ is computable.*

PROOF: Recall from KMA that there exists a universal while-program for unary functions,

$$\Phi(x, y) = f_x(y).$$

The proof was constructive, demonstrating a while-program which calculated for any pair of integers x, y the output of program P_x on input y. The code can be modified to form a new function $\Phi(x, y, z)$ such that on input triple (x, y, z) the code:

- simulates $P_x(y)$ for z steps;
- if the simulated program halts on or before z steps, a special *haltflag* is set true, else the flag is false;
- in either case the program halts and the pair (X1, *haltflag*) is considered the output.

Because the family $\mathcal{F}$ is effectively enumerable, let $ee : \mathbf{N} \to \mathbf{N}$ be a total function for which $f_i = P_{ee(i)}$. Our aim is to use dovetail through all pairs of integers, (i, j), running the i-th program for j steps. To to this, let $\tau : \mathbf{N}^2 \to \mathbf{N}$ be a computable bijection and $\pi_1, \pi_2 : \mathbf{N} \to \mathbf{N}$ two functions "inverting" τ in the sense that,

$$i = \tau(\pi_1(i), \pi_2(i)).$$

Then we shall run all programs for all number steps using the following program,

```
begin
  i:=0;
  haltflag:=false;
  while not haltflag do begin
    j:=ee(pi1(i));
    N:=pi2(i);
    Phi(j,X1,N);
    i:=succ(i);
  end
end
```

If there is any program P_j which halts on $x1$, it does so in N steps. So for the appropriate i, haltflag will be set true and X1 will have the proper output for having computed $P_j(x1)$. □

Theorem 4 *Recursive programs are computable by while-programs.*

PROOF: It only remains to show that the sequence of functions whose limit defines the recursive function is effectively enumerable. The enumerating program is,

```
begin
  if X1=0 then X1:=#B
  else begin
    ktot:=k;
    r:=#RK;
    while x1>1 do
    begin
      r:=sub(r,ren(#RK,ktot));
      ktot:=ktot+k;
      X1:=pred(X1)
    end;
    X1:=sub(r,#B)
  end
end
```

Where `#B` is $[\perp]$, `#RK` is $[RK]$ and k is the largest integer such that Xk appears in the recursive kernel RK. □