Unfolding Recursive Function Definitions Using the Paradoxical Combinator*

Stephen Fitzpatrick, M. Clint, P. Kilpatrick

1 April 1996

Department of Computer Science, The Queen's University of Belfast,
Belfast BT7 1NN, Northern Ireland.
{S.Fitzpatrick, M.Clint, P.Kilpatrick}@cs.qub.ac.uk

Abstract

Function unfolding is a well-known program transformation technique: it is often used to reduce execution overheads incurred by an implementation's function-calling mechanism and to localise information available in global definitions (localization simplifies further optimizations such as constant propagation and the static evaluation of expressions).

Unfolding function definitions that exhibit either self- or mutual-recursion presents a problem for automated transformation systems: sophisticated control mechanisms may need to be incorporated into the unfolding process to ensure termination. Consequently, many automated transformation systems do not attempt to unfold recursive function definitions. Many of the optimizations that follow from unfolding non-recursive functions can still be performed using techniques such as function cloning and specialization but, because of the separation of function definitions and function uses, these techniques are more complex for an automated system to perform than are the corresponding techniques used for non-recursive functions.

In this paper, the use of the paradoxical combinator, Y, to unfold *all* function definitions, including recursive definitions, is discussed. Full unfolding with the Y combinator is simple for an automated system to perform (requiring only an exhaustive application of a straightforward substitution process). In particular, termination is assured without necessitating a separate control mechanism. After unfolding has been performed, many optimizations can be applied in the same manner as they are applied to unfolded, non-recursive definitions.

1 Introduction

Function unfolding is the process of replacing uses of a function with the function's definition [1]. The potential benefits of function unfolding are well known:

- By reducing the number of function invocations, unfolding can reduce the overheads incurred by an implementation's function invocation mechanism.
- Unfolding can enhance 'locality'. The expressions which occur within a function definition are isolated from the contexts in which the function is used. This isolation is desirable for developing and understanding programs but it can hinder program optimization in at least two ways. First, each of the different contexts in which a function is applied may give rise to a distinct set of optimizations but a single function definition cannot be tailored simultaneously to each of the contexts; function unfolding, however, creates separate instances of the function's definition which can be independently tailored to context.

* Presented at the Durham Transformation Workshop, University of Durham, England,1-2 April 1996.

Second, when optimizing an expression which involves a use of a function, it may be necessary to transfer information from the function's definition to the point at which the function is used (for example, it may be possible to determine from a function's definition that the function is strict in some of its arguments — this information may be useful in preparing an expression that uses the function for parallel evaluation). For an *automated* transformation system, transferring information, though possible, may be cumbersome. Function unfolding textually unites a function's definition and the expressions which use the function, thus increasing the amount of manipulation that can be performed locally.

Other program transformation techniques can be employed to obtain at least some of the benefits that derive from function unfolding. For example, the tailoring of a function definition to each of its uses can be achieved through *function cloning* [2], where a single, original function definition is duplicated to give a set of definitions, each of which can be independently tailored. However, cloning does not improve locality. In general, function unfolding often provides the simplest framework for performing other program transformations.

However, unfolding recursive functions obviously presents a problem for automated transformation systems — many automated systems unfold functions only if they do not exhibit direct (self) recursion [3], utilize a control mechanism that ensures that the number of times that unfolding is performed is finite [4] or unfold only when it can be determined in advance that unfolding will terminate [5]. In each case, the continued presence of global definitions after transformation can inhibit further manipulation.

This paper discusses a method, based on the paradoxical combinator Y, for unfolding all functions in a program, regardless of the pattern of recursion existing within the functions' definitions. Unfolding is complete in that no global definitions are required after transformation: all function definitions are made local to the expressions in which they are used, thus facilitating further manipulation.

1.1 The Paradoxical Combinator, Y

The paradoxical combinator is defined by the identity $Yf \equiv f(Yf)$, where f is a function expression. To illustrate how Y can be used to express recursion, consider a standard definition of the factorial function:

$$\mathsf{fac}{=}\lambda\mathsf{n}{\cdot}\mathsf{if\ (n=0)\ then\ 1\ else\ n*fac(n-1)}\quad .$$

Informally, the instance of fac on the right of the definition can be considered to be a reference to the name introduced on the left of the definition: that is, it is a use of a globally defined name. If the expression fac(n-1) is evaluated, the name fac is replaced by the right side of the definition (λn·if ...). Thus, a set of globally defined names is central to this method of expressing recursion.

To remove the need for globally defined names, the function definition can be rewritten in the form

$$\mathsf{fac}{=}\mathsf{Y}\lambda\mathsf{fac}{\cdot}\lambda\mathsf{n}{\cdot}\mathsf{if\ (n=0)\ then\ 1\ else\ n*fac(n-1)}\quad .$$

Now the instance of fac in the expression fac(n-1) refers to the identifier bound by the abstraction λfac·..., and not to the globally defined name. As with all λ-abstractions, the identifier is arbitrary and can be systematically replaced. For example, the above definition could be rewritten as

$$\mathsf{fac}{=}\mathsf{Y}\lambda\mathsf{f}{\cdot}\lambda\mathsf{n}{\cdot}\mathsf{if\ (n=0)\ then\ 1\ else\ n*f(n-1)}$$

in which fac has been replaced with f. If the expression on the right side is applied to 5, say, it can be evaluated as shown in figure 1. Note that evaluation of the expression never uses the global function name fac.

The function definitions that remain after each definition has been rewritten using Y still define a set of global names corresponding to functions: one function definition may make use of *another* global function. Thus, the rewritten definitions are not fully 'self-contained' — to achieve full self-containment the definitions must be unfolded, as discussed in Section 2.1.

```
fac(5) = (Yλf·λn·if (n=0) then 1 else n*f(n-1)) (5)
       =   by definition of Y
       (λn·if (n=0) then 1
        else n*((Yλf·λn·if (n=0) then 1 else n*f(n-1))(n-1))
       )(5)
       =   β-reduce for n
       if (5=0) then 1
        else 5*((Yλf·λn·if (n=0) then 1 else n*f(n-1))(5-1))
       =   since 5 ≠ 0
       5*((Yλf·λn·if (n=0) then 1 else n*f(n-1))(4))
```

Note that the application (Yλf·...)(4) is fac(4). Evaluation of this expression is performed in the same manner as above, producing an expression equivalent to 4*fac(3), and so on.

Figure 1: Evaluation of fac applied to 5

1.2 Example Optimizations

The need for unfolding recursive functions arose in work on the derivation of efficient Fortran implementations of numerical algorithms from abstract functional specifications [6] and, in particular, from a specification of a *multigrid* algorithm which computes a solution, over a discrete grid, of the equation $Lu = f$ where L is a second order differential operator, u is the grid over which the equation is to be solved and f is an initial grid.

The solution is computed by applying the function solve:

```
fun solve(size:int)(L:(grid*int*int->real), Jc:jacobi, f:grid, bv:(int*int->real), epsilon:real):grid
  = ... iterate(size)(L, Jc, u0, hsqf, epsilon0) ...
```

where size is the grid size, Jc is the Jacobian of L, bv specifies the conditions that hold at the boundary of the grid and epsilon specifies the required accuracy of the solution.

The function solve is not recursive, but it invokes a second function iterate which *is* recursive:

```
fun iterate(size:int)(L:(grid*int*int->real), Jc:jacobi, ui:grid, f:grid, epsilon:real):grid
  = ... iterate(size)(L, Jc, mg(size)(L, Jc, ui, f), f, epsilon) ...
```

An operation is repeated by iterate until a sufficiently accurate approximation to the solution is obtained. The iterate function itself makes use of another recursive function mg which is the kernel of the multigrid method.

```
fun mg(size:int)(L:(grid*int*int->real), Jc:jacobi, u:grid, f:grid):grid
  = ... mg(half_size(size))(L, Jc, constant(half_size(size), 0.0), restrict(grid)) ...
```

The full details of the above functions are not important. What is important is that the shown interfaces seemed to be natural and adhere to the tenets of good programming (e.g. specifying the differential operator and boundary values as parameters permits the main functions to be used for any appropriate second-order differential equation). However, all three of the shown functions are higher-order and may not be efficiently implementable in Fortran.

Now the functions are such that their functional arguments (L, jc and bv) are never changed: they are only passed (unchanged) to other functions and to recursive applications — see Figure 2. Thus, for example, the parameter L in the mg and iterate functions refers to the same function as the parameter L in the solve function.

If mg and iterate could refer directly to the parameters of solve, then passing L as an argument would not be necessary. However, the only names that are visible to all of the functions are global names. Thus, eliminating the parameter L is possible only if the function that calls solve passes a global function as the actual argument for L — see Figure 3.

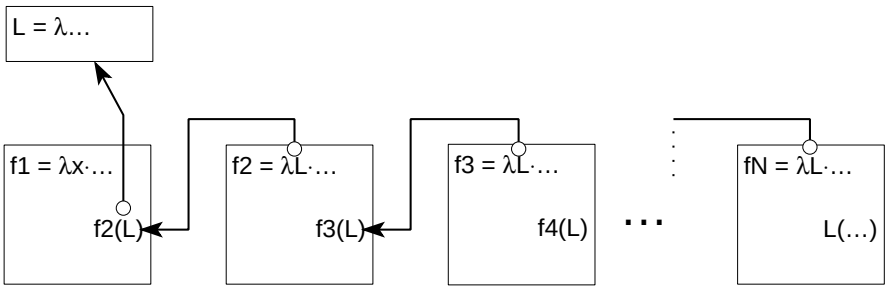

Figure 2: Passing a function parameter unchanged

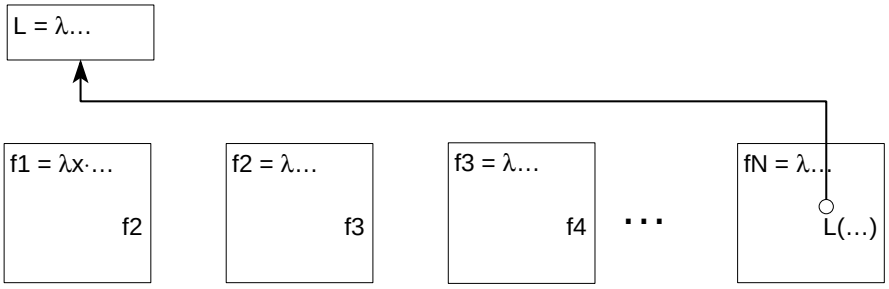

Figure 3: Function parameter eliminated

If the calling function passes, say, a λ-expression as the actual argument, then the expression would need to be, say, λ-lifted into a global function before elimination could proceed. Furthermore, before elimination can be attempted, it must be ensured that each of the functions from which the parameter is to be eliminated is called by only one function (other than itself) since otherwise L may refer to distinct global functions in different invocations; this condition could either be checked directly, or could be ensured by cloning the functions.

If, however, the functions are unfolded then the called functions have access to the parameters of the calling functions[1] and each function use can be independently tailored. Thus, unchanging parameters such as L can be eliminated directly by a simple transformation operating only on local expressions.

Example 1.2-a: *N-fold function application.*

Consider the function nfold, where nfold(n) maps a function f onto f^n.

```
fun compose(f:β → γ, g:α → β):α → γ
  = λx·f(g(x))

fun nfold(n:int):(α → α) → (α → α)
  = if (n=0) then λf·λx·x else λf·compose(f,nfold(n-1)(f))
```

If nfold is fully applied, as in the expression nfold(N)(F)(X), then the application can be converted into a first-order function as follows.

- Unfold nfold and compose, and simplify:

```
(Yλnfold·λn·
  if (n=0)
  then λf·λx·x
  else λf·λx·f(nfold(n-1)(f)(x))
) (N)(F)(X)
```

[1] It may be necessary to perform α-conversion.

- The function expression (λn·. . .) binds only one formal argument (n) but is applied to three actual arguments, so introduce two new formal arguments, say a and b; the body of the function expression (if . . .) is applied to the new arguments.

```
(Yλnfold·λn·λa·λb·
 ( if (n=0)
   then λf·λx·x
   else λf·λx·f(nfold(n-1)(f)(x))
 )(a)(b)
) (N)(F)(X)
```

  Simplify by propagating the application through the conditional expression and evaluating bindings:

```
(Yλnfold·λn·λa·λb·
 ( if (n=0)
   then b
   else a(nfold(n-1)
 )(a)(b))
) (N)(F)(X)
```

- The arguments a and b are passed unchanged to the one recursive application of nfold and so can be eliminated; instances of these arguments, other than those that are arguments to the recursive application, are replaced with the values to which the arguments are initially bound, *viz.* F and X. The resulting expression is first-order.

```
(Yλnfold·λn·
 if (n=0)
 then X
 else F(nfold(n-1))
) (N)
```

Example 1.2-b: *Tree reduction*

Suppose that a Tree is defined to be either (i) a Leaf or (ii) a Node together with a List of Trees:

$$\alpha \text{ Tree} ::= \alpha \text{ Leaf} \mid \alpha \text{ Node} \times (\alpha \text{ Tree}) \text{ List}$$

and assume the following basic constants and functions:

- isLeaf(T:α Tree):boolean — returns $true$ if T is a leaf;
- value(T:α Tree):α — returns the value stored in a Tree that is a Leaf, or the value of the Node of a non-Leaf Tree;
- subtrees(T:α Tree):(α Tree) List — returns a List of the immediate children of T (the children are themselves Trees);
- isNull(L:α List):boolean — returns $true$ if List L is empty;
- hd(L:α List):α and tl(L:α List):α — return the head and tail, respectively, of a (non-empty) List;
- (A:α)::(L:α List) — appends A onto the head of L;
- [] — the empty List.

A reduction operation combines all of the components of a composite object to produce a single value; for example, summing numbers in a List or numbers stored in a Tree. Tree reduction can be defined as follows:

```
Tree_reduce(r:α × α → α, r0:α)(T:α Tree):α
 = if isLeaf(T) then value(T)
    else r(value(T), List_reduce(r, r0)(List_map(Tree_reduce(r, r0))(subtrees(T))))
List_reduce(r:α × α → α, r0:α)(L:α List):α
 = if isNull(L) then r0 else r(hd(L), List_reduce(r, r0)(tl(L)))
List_map(f:α → β)(L:α List):β List
 = if isNull(L) then [] else f(hd(L))::List_map(f)(tl(L))
```

Each of the higher-order functions Tree_reduce, List_reduce and List_map is self-recursive. Execution of the functions gives rise to mutual recursion, since Tree_reduce calls the List_map function which in turn calls the Tree_reduce function (which is bound to the argument f).

Assume that Tree_reduce is used to sum the elements of a tree T: that is, it is required to evaluate the expression Tree_reduce(+,0)(T). Unfolding gives:[2]

```
(Yλ Tree_reduce·λr, r0·λT·
  if isLeaf(T)
  then value(T)
  else r(value(T),
     (Yλ List_reduce·λr, r0·λL·
       if isNull(L) then r0 else r(hd(L), List_reduce(r, r0)(tl(L)))
     )
       (r, r0)
       ((Yλ List_map·λf·λL·if isNull(L) then [] else f(hd(L))::List_map(f)(tl(L)))
         (Tree_reduce(r, r0))
         (subtrees(T))
       )
   )
)(+,0)(T)
```

Now the arguments r and r0 to Tree_reduce are passed unchanged to each recursive application of Tree_reduce, so they may be eliminated as arguments and instances of them may be replaced with the expressions to which they were initially bound (+ and 0).[3] A similar optimization applies for the arguments r and r0 to List_reduce and argument f to List_map.

```
(Yλ Tree_reduce·λT·
  if isLeaf(T)
  then value(T)
  else value(T)+
    (Yλ List_reduce·λL·
      if isNull(L) then 0 else hd(L)+List_reduce(tl(L))
    )(
      (Yλ List_map·λL·if isNull(L) then [] else Tree_reduce(hd(L))::List_map(tl(L)))
      (subtrees(T))
    )
)(T)
```

The resulting expression exhibits the same recursive nature as the original function definitions, but now all functions are first-order functions. The resulting expression could, if required, be rewritten as Tree_reduce′(T) where

[2] The basic List and Tree functions are not unfolded in this example.

[3] The application of the function argument f is converted into infix form when instantiated as +.

```
Tree_reduce′(T)
  = if isLeaf(T) then value(T)
      else value(T)+List_reduce′(List_map′(subtrees(T)))
List_reduce′(L)=if isNull(L) then 0 else hd(L)+List_reduce′(tl(L))
List_map′(L)
  = if isNull(L) then []
      else Tree_reduce′(hd(L))::List_map′(tl(L))
```

Each of the original higher-order functions has been specialized to specific function parameters (*viz.*, + and Tree_reduce).

2 The Unfolding Process

Before discussing the unfolding transformation, some basic notation and terminology is presented below. A function definition is written in the form $name = expression$, where $expression$ is constructed from the standard functions and operators on the basic data types (e.g. integers, reals and booleans) and from 'program-defined' functions. In some contexts, $expression$ (rather than $name = expression$) is said to be the definition of $name$.

The symbol $\doteq$ denotes *syntactic identity* of two expressions: $E_1 \doteq E_2$ is $true$ iff E_1 and E_2 are exactly the same expression.

The symbol $\leftrightarrow$ denotes *semantic equivalence* of two expressions: $E_1 \leftrightarrow E_2$ is $true$ iff E_1 and E_2 evaluate (reduce) to the same value.

The substitution of all free occurrences of an identifier x by an expression y in an expression E is denoted by $E[x \rightarrow y]$. Note that this is a 'once-through' substitution: only instances of x contained in E *before* substitution begins are replaced — any new instances of x introduced by the substitution process (i.e. that are contained in y) are not affected. The simultaneous application of multiple substitutions to an expression is denoted by

$$E[x_1 \rightarrow y_1, x_2 \rightarrow y_2, \ldots]$$

where each x_i is assumed to be unique. (This uniqueness, combined with the stricture that substitution affects only instances of x_i present in an expression before substitution begins, means that the substitution is deterministic.) Multiple, simultaneous substitutions may also be denoted by '$E[x_i \rightarrow y_i], i = 1, 2, \ldots$'.

The application of a transformation T to an expression E is denoted by $T(E)$. The *exhaustive* application of a transformation T to an expression E is denoted by $T^*(E)$ — T is repeatedly applied, initially to E, until application brings about no further change. Thus the sequence of expressions $E, T(E), T(T(E)), \ldots$ is generated until successive expressions are identical. Exhaustive application can be formally defined by:

$$T^*(E) \equiv \text{if } T(E) \doteq E \text{ then } E \text{ else } T^*(T(E)) \ .$$

If a transformation preserves the meaning of an expression, then so does the exhaustive application of the transformation (provided exhaustive application terminates).

2.1 The Transformations: $Inline$ and $Construct$

Assume a set of function definitions

$$f_i = F_i$$

where $i = 1, 2, \ldots, N$ and where each f_i is unique.

Let $Inline$ be the transformation

$$Inline(E) \equiv E[f_j \rightarrow Y \lambda f_j \cdot F_j], \quad j = 1, \ldots, N.$$

Thus, $Inline$ replaces each free occurrence of an identifier corresponding to one of the program-defined functions with the function's definition enclosed in an application of Y. Note that any occurrences of f_j that are free in F_j are bound in $Y\lambda f_j{\cdot}F_j$. The $Inline$ transformation preserves the meaning of the set of function definitions [7, 8].

Form a new set of function definitions

$$g_i = Inline^*(f_i) \qquad (Construct)$$

where $i = 1, 2, \ldots, N$ and where each g_i is a 'new' identifier (that is, distinct from all the other $g_{j\neq i}$ and from all f_j).

The following properties hold:

Equivalence: For each i, $g_i \leftrightarrow f_i$.

Each g_i is initially set equal to f_i and is subject only to a meaning-preserving transformation.

Self-containment: Each g_i is 'self-contained' — that is, no g_i uses any global function.

A use of a global function would occur (on the right side of a definition) as a *free* instance of one of the g_i or one of the f_i. None of the g_i occur on the right side of any of the constructed definitions *before* $Inline$ is applied (since the right side of the definition of g_i is initially the single identifier f_i); furthermore, the $Inline$ transformation does not introduce any uses of any of the g_i. Thus, no g_i can occur in the the final forms of the definitions.

Neither can the final forms contain any free occurrences of any of the f_i. If an expression contains a free occurrence of any of the f_i then application of $Inline$ generates a different expression. But $Construct$ repeatedly applies $Inline$ until no further change occurs. Thus, termination of $Construct$ implies that the transformed expressions contain no free occurrences of any of the f_i.

Termination: The application of $Construct$ to any finite set of functions is guaranteed to terminate.

Consider one of the functions in its initial form: $g_i = f_i$. Application of $Inline$ to this function causes f_i to be replaced with $Y\lambda f_i{\cdot}F_i$. Since F_i may be an arbitrarily complex expression, this substitution can clearly cause the definition of g_i to grow (according to some simple syntactic measure). Since F_i may contain further occurrences of the f_j, further application of $Inline$ may cause further growth. Thus, there is a *prima facie* case *against* termination. However, as discussed below, the number of applications of $Inline$ is bounded by N (the number of the functions f_i).

2.2 Termination

Recall that $Inline$ causes substitution only at *free* occurrences of the f_i, and that there can be no free occurrences of f_i in $Y\lambda f_i{\cdot}E$. Thus, once an instance of f_i has been replaced with $Y\lambda f_i{\cdot}F_i$, no further substitutions for f_i can occur within the replacement.

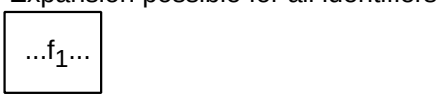

Figure 4: Initial expression

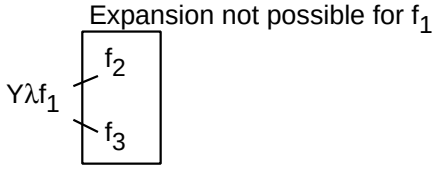

Figure 5: $Inline$ applied once

For example, consider Figure 4. In the expression shown, any of the N f_i are candidates for expansion by $Inline$, since none of them is bound (though it so happens that this particular expression contains only f_1).

In Figure 5, $Inline$ has been applied to replace the free occurrence of f_1 with $Y\lambda f_1{\cdot}F_1$, which contains an instance of f_2 and f_3. These instances are free and so will be expanded when $Inline$ is next applied. What is more important though, is that f_1 *is bound in the replacement and can never occur free in the replacement*, even after further applications of $Inline$. Thus, only $(N-1)$ identifiers are now candidates for further expansion by $Inline$.

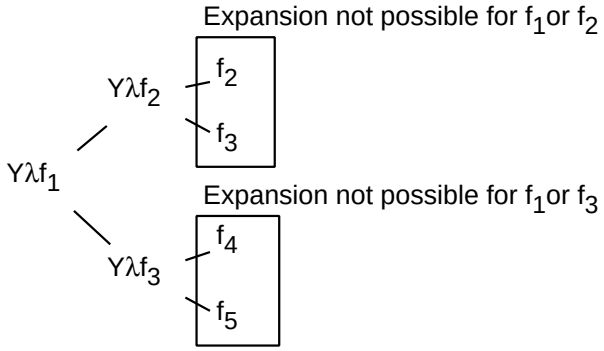

Figure 6: $Inline$ applied twice

In Figure 6, $Inline$ has been applied a second time to replace the instances of f_2 and f_3 with $Y\lambda f_2{\cdot}F_2$ and $Y\lambda f_3{\cdot}F_3$ respectively. F_2 contains an instance of f_2 which is bound and an instance of f_3 which is free. F_3 contains a free instance of f_4 and of f_5.

As before, what is important is that within the expression $Y\lambda f_2{\cdot}F_2$, both f_1 and f_2 are bound (f_1 being bound by the outer λ). Thus, within this expression, the number of identifiers which are candidates for further expansion by $Inline$ has been further reduced to $(N-2)$. Similarly, within the expression $Y\lambda f_3{\cdot}F_3$, both f_1 and f_3 are bound (note though that f_2 is *not* bound) so the number of candidates for expansion here is also $(N-2)$.

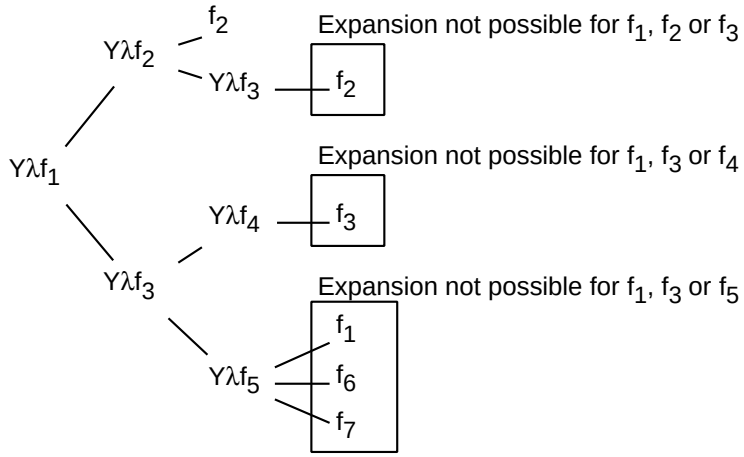

Figure 7: $Inline$ applied three times

In Figure 7, expansion is taken one stage further by applying $Inline$ again. In the expressions introduced at this stage, the number of candidates for further expansion is $(N-3)$ since three identifiers are bound in each expression.

In general, each time $Inline$ is applied, a number of replacements occur. Further expansion can occur only within the replacement expressions. However, the number of candidates for further expansion within

each replacement expression is reduced each time $Inline$ is applied because each application increases the number of identifiers that are bound within the replacements. If the number of functions (N) is finite, then eventually no candidates remain and expansion stops; that is, applying $Inline$ to the expression produces no change, so exhaustive application terminates. More precisely, the number of applications of $Inline$ is bounded by N.

A more thorough treatment of termination is given in the appendix.

In summary, $Construct$ applied to a set of possibly mutually-recursive function definitions produces an equivalent set of function definitions in which all recursion is expressed using Y. No function in the constructed set uses any global function.

3 Unfolding Non-recursive Functions

The $Construct$ transformation can be applied regardless of the pattern of recursion existing within a set of function definitions. For unfolding truly non-recursive functions, however, the paradoxical combinator is not needed. For example, the compose function, defined by

$$\text{compose(f:}\beta \to \gamma\text{, g:}\alpha \to \beta\text{):}\alpha \to \gamma = \lambda\text{x·f(g(x))}$$

would be unfolded as

compose(A, B)

$\longrightarrow$

(Yλcompose·λf, g·λx·f(g(x)))(A, B)

Y is not needed because the function's body does not contain any instances of compose.

The following transformation can be applied after $Construct$ to eliminate unnecessary instances of Y:

$$Y\lambda f{\cdot}E \to E, \quad \text{where } E \text{ does not contain } f.$$

This rule follows from the property of Y that $Y\lambda f{\cdot}E = \lambda f{\cdot}E\ (Y\lambda f{\cdot}E)$: if there are no (free) instances of f in E, then the result of β-reducing the application is E (that is $\lambda f{\cdot}E\ (x) \to E$ if E does not contain f).

This transformation also has the effect of converting some forms of mutual-recursion into self-recursion. Consider the mutually-recursive functions

f(x) = F(g(x))
g(x) = G(f(x))

where F and G indicate computations that do not involve instances of f or g other than those shown. Although these functions are mutually recursive, they can be rewritten so that each exhibits only self-recursion: a single application of classical unfolding is sufficient to produce the self-recursive form [9]:

f(x) = F(G(f(x))
g(x) = G(F(g(x)))

If the functions have been unfolded by $Construct$

f = Yλf·λx·F((Yλg·λx·G(f(x)))(x))
g = Yλg·λx·G((Yλf·λx·F(g(x)))(x))

then a once-through application of the transformation to eliminate unnecessary instances of Y also produces the self-recursive form

f = Yλf·λx·F((λx·G(f(x)))(x))
g = Yλg·λx·G((λx·F(g(x)))(x))

which can be further simplified by β-reducing λ-bindings in which the bound value is the same as the bound identifier (λx·E (x) $\to$ E):

f = Yλf·λx·F(G(f(x)))
g = Yλg·λx·G(F(g(x)))

4 Potential Disadvantages of Unfolding

As with all unfolding techniques, there is the potential for the textual size of a program to grow enormously, which may cause a transformation system to fail due to memory constraints or result in the application time of subsequent transformations becoming prohibitively long.

However, in some cases an algorithm is specified in the form of a set of function definitions together with a single expression which is to be evaluated, in the context of the definitions, to compute the required result. In such circumstances, unfolding can be performed only within this expression, thus reducing the growth of the text of the program. Furthermore, the function definitions can be discarded after unfolding, thereby reducing the size of the final text.

Localising all expressions may increase the execution time of an implementation because of the requirement to repeat evaluation of an expression — for example, an expression may be moved from outside a loop to within its body. Often such deficiencies can be remedied by loop invariant extraction or common sub-expression elimination.

Consider, however, the following 'localization'.

(Yλp·λn·λx·if (n=0) then 1 else x*p(n-1)(x)) (N)(X)

$\longrightarrow$

(Yλp·λn·if (n=0) then 1 else X*p(n-1))(N)

The function computes x^n. In the localized form, X (an arbitrary expression) occurs within the function itself and may be evaluated in each (recursive) invocation; this is undesirable if X is computationally expensive.

Moreover, assuming an eager evaluation scheme, X cannot simply be abstracted from the function in the localized form since there is no way to determine solely from the localized form whether or not X will be evaluated (it will *not* be evaluated if N is zero). If X is to be abstracted, it must at least be established that computational errors cannot arise. There are of course ways to avoid this problem (such as leaving a marker outside the function to indicate the original position of X, to which it can be restored if desired [4]), but it is a problem nonetheless.

5 Conclusions

This paper presents a straightforward transformation, amenable to automated application, for unfolding all functions (including recursive functions) using the paradoxical combinator. Unfolding often enables optimizations and simplifies their implementation as automated transformations.

A Proof of Termination

In this appendix, it is shown that $Construct$ terminates (when applied to a finite set of functions). The proof is organized as follows:

- A grammar is defined for expressions, which provides a basis for proofs by structural induction.
- A condition is stated, which, if satisfied by a transformation T, guarantees that exhaustive application of T terminates.

- It is established, by means of a sequence of lemmas, that $Inline$ satisfies this termination condition and thus that $Construct$ terminates.

A.1 Grammar

Assume the following simple grammar $\mathcal{E}$ for expressions:

$\mathcal{E}$ ::=	C	Constants
|	f_i	Identifiers for program-defined functions
|	V	Identifiers (distinct from f_i)
|	$\lambda v{\cdot}E_1$	Abstraction
|	$Y\lambda f_i{\cdot}E_1$	Fixed point
|	$E_1(E_2)$	Application
|	if E_1 then E_2 else E_3	Conditional

where E_1, E_2, E_3 are expressions and v is an identifier. Note that identifiers for program-defined functions are distinct from other identifiers (only the former can be combined with Y).

In this discussion, it is convenient to use variable subscripts to enumerate all program-defined functions (e.g. in propositions such as $\forall i{\in}\{1,2,\ldots,N\}{\cdot}f_i\ldots$). However, it is assumed that a program uses only literal constants for subscripts; for example, a program *cannot* contain an expression such as f_{i+1}. The use of non-literal subscripts would prevent unfolding, since an arbitrary expression may be dependent upon data extraneous to a program (say, data input during evaluation/execution of a program).

The grammar $\mathcal{E}$ establishes a partial order $<_\mathcal{E}$ on expressions: composite expressions (*viz.* abstractions, fixed points, applications and conditionals) are larger than their component expressions; for example, $E_1(E_2) <_\mathcal{E} E_1$ and $E_1(E_2) <_\mathcal{E} E_2$. This partial order is the basis of proofs by structural induction — to establish that a property P holds for all E:

- it is shown that P holds for the base cases of $\mathcal{E}$ (*viz.* constants and identifiers);
- it is shown that if P holds for all $E' <_\mathcal{E} E$ then P holds for E.

A.2 Exhaustive Application

Definition 1 $T^*(E){\equiv}$ if $T(E) \doteq E$ then E else $T^*(T(E))$

□

The following theorem is assumed without proof.

Theorem 1 *Termination of T^*.*

For a transformation T, if there exists a measure m such that

$$\begin{aligned} \forall E{\cdot}m(E) &\geq 0 \\ \forall E{\cdot}m(E) = 0 &\Rightarrow T(E) \doteq E \\ \forall E{\cdot}m(E) \neq 0 &\Rightarrow m(T(E)) < m(E) \end{aligned}$$

then $\forall E{\cdot}T^*(E)$ terminates.

□

That is, if there is a non-negative measure on expressions which is forced strictly towards zero by a single application of the transformation, and if an expression that has a zero measure is unchanged by the transformation, then exhaustive application of the transformation is guaranteed to terminate.

A.3 Measure on $\mathcal{E}$

To show that exhaustive application of $Inline$ terminates, a measure is required that satisfies the termination conditions of Theorem 1. In this section, a measure is proposed and informally justified; in the following section, it is shown that the measure, in conjunction with $Inline$, satisfies the termination conditions.

As discussed in Section 2.2, $Inline$ terminates because of the reduction in the number of candidates for further expansion resulting from the binding of the identifiers f_i. Thus, the appropriate measure should be related to the number of free identifiers.

Consider the grammar $\mathcal{E}$, on which the measure will be defined by cases. The important components of the grammar, as regards expansion, are:

- The identifiers for program-defined functions, f_i, since expansion occurs only at these identifiers. As discussed previously, the number of times $Inline$ can be applied to an instance of an identifier and its replacements is bounded by the number of identifiers that are free at the point at which the instance occurs. Thus, if b identifiers are bound, the measure associated with a free instance f_i is $(N - b)$, where N is the number of program-defined functions. If f_i is already bound, then its measure is zero since it cannot give rise to expansion.
- Fixed-points ($Y\lambda f_i{\cdot}E_1$), since fixed-points bind the f_i thus preventing them from being expanded. Note that the binding point itself ($Y\lambda f_i$) does not give rise to expansion; expansion can occur only within the body expression (E_1). The binding point does, however, ensure that any measure arising from the body expression is strictly less that it would be if the binding point were not present. Thus, computing the measure for a fixed-point is essentially the same as computing the measure for the body expression, but with the additional binding being taken into account.

Now consider an application: $E{\equiv}E_1(E_2)$. Assume that the measures of E_1 and E_2 are m_1 and m_2 respectively; these measures mean that $Inline$ can be applied at most m_1 times to E_1 (before application ceases to cause change), and at most m_2 times to E_2. Thus, since $Inline$ operates independently on each sub-expression, the maximum number of times it can be applied to the full expression E is the larger of m_1 and m_2, which is consequently the measure of the application. Similar reasoning applies in the case of conditional expressions: the measure of a conditional expression is the maximum of the measures of its component expressions.

Constants and identifiers (other than identifiers corresponding to program-defined functions) do not give rise to expansion, and so their measures are zero. For an abstraction ($E{\equiv}\lambda v{\cdot}E_1$) the binding point itself does not gives rise to expansion, so the measure is the same as that of the body expression (E_1). Note that, unlike fixed-points, the binding of the identifier v is not significant in the computation of the measure since it is assumed that the identifiers for the program-defined functions are distinct from all other identifiers (i.e. v cannot be one of the f_i).

Thus, the required measure on $\mathcal{E}$, M, may be defined as follows.

Definition 2 $M(E){\equiv}\mu(E, \emptyset)$
where $\emptyset$ denotes the empty set and

$$
\begin{array}{lll}
\mu(E,S) ::= & \text{case } E \text{ of} & \\
& C: & 0 \\
& f_i: & \text{if } f_i{\in}S \text{ then } 0 \text{ else } N - |S| \\
& V: & 0 \\
& \lambda v{\cdot}E_1: & \mu(E_1, S) \\
& Y\lambda f_i{\cdot}E_1: & \mu(E_1, S{\cup}\{f_i\}) \\
& E_1(E_2): & \max_{i\in\{1,2\}} \mu(E_i, S) \\
& \text{if } E_1 \text{ then } E_2 \text{ else } E_3: & \max_{i\in\{1,2,3\}} \mu(E_i, S)
\end{array}
$$

where S is a set of identifiers and $|S|$ is the cardinality of S.

Note that the only possible elements of the 'auxiliary' set S are the f_i; that is, S is always a sub-set of $\{f_i | 1 \leq i \leq N\}$. S is used to keep track of which identifiers are bound in each sub-expression of an expression, thus enabling the number of free identifiers to be computed.

Lemma 1 $0 \leq |S| \leq N$. *During the computation of $M(E)$, the size of the auxiliary argument S for the function μ never exceeds N (the number of program-defined functions).*

Since S is always a sub-set of $\{f_i | 1 \leq i \leq N\}$, the size of S is bounded by 0 and N.

□

Lemma 2 $\mu(E, S) \geq 0$.

Consider the following cases of E: abstraction, fixed-point, application and conditional. In each case, the measure of E is the same as the measure of one of its component expressions.

For the other possible cases, the measure is non-negative: for the cases of constant (C) and identifier (V), the measure is 0; for the case of f_i, the measure is either 0 or $N - |S|$, both of which are non-negative since $|S| \leq N$.

Thus, the measure of any expression is non-negative. (Formal proof by structural induction is straightforward.)

□

Lemma 3 $\mu(E, S) \leq N - |S|$.

Proof is by structural induction on $\mathcal{E}$, where the inductive hypothesis is $\mu(E, S) \leq N - |S|$.

Base steps:

For $E \equiv C, V$: $\mu(E, S) = 0 \leq N - |S|$, since $0 \leq |S| \leq N$ (Lemma 1).

$E \equiv f_i$
$\Rightarrow \mu(E, S) =$ if $f_i \in S$ then 0 else $N - |S|$ by definition of μ
$\leq N - |S|$ since $0 \leq |S| \leq N$

Inductive steps:

$E \equiv \lambda v \cdot E_1$
$\Rightarrow \mu(E, S) = \mu(E_1, S)$ by definition of μ
$\leq N - |S|$ by hypothesis, since $E_1 <_{\mathcal{E}} E$

$E \equiv Y \lambda f_i \cdot E_1$
$\Rightarrow \mu(E, S) = \mu(E_1, S \cup \{f_i\})$ by definition of μ
$\leq N - |S \cup \{f_i\}|$ by hypothesis, since $E_1 <_{\mathcal{E}} E$
$\leq N - |S|$

$E \equiv E_1(E_2)$
$\Rightarrow \mu(E, S) = \max_{i \in \{1,2\}} \mu(E_i, S)$ by definition of μ
$\leq N - |S|$ since, for $i \in \{1, 2\}$, $E_i <_{\mathcal{E}} E$
and so $\mu(E_i, S) \leq N - |S|$ by hypothesis

$E \equiv$ if E_1 then E_2 else E_3
$\Rightarrow \mu(E, S) = \max_{i \in \{1,2,3\}} \mu(E_i, S)$ by definition of μ
$\leq N - |S|$ since, for $i \in \{1, 2\}$, $E_i <_{\mathcal{E}} E$
and so $\mu(E_i, S) \leq N - |S|$ by hypothesis

The lemma follows by structural induction.

□

A.4 Unfolding Transformation

To facilitate a demonstration that M and $Inline$ together satisfy the termination conditions, $Inline$ is defined as a function on $\mathcal{E}$.

Definition 3 $Inline(E) \equiv \tau(E, \emptyset)$
where τ is defined by cases on $\mathcal{E}$:[4]

$$
\begin{array}{lll}
\tau(E,S) ::= & \text{case } E \text{ of} & \\
& C: & C \\
& f_i: & \text{if } f_i \in S \text{ then } f_i \text{ else } Y\lambda f_i \cdot F_i \\
& V: & V \\
& \lambda v \cdot E_1: & \lambda v \cdot \tau(E_1, S) \\
& Y\lambda f_i \cdot E_1: & Y\lambda f_i \cdot \tau(E_1, S \cup \{f_i\}) \\
& E_1(E_2): & \tau(E_1,S)(\tau(E_2,S)) \\
& \text{if } E_1 \text{ then } E_2 \text{ else } E_3: & \text{if } \tau(E_1,S) \text{ then } \tau(E_2,S) \text{ else } \tau(E_3,S)
\end{array}
$$

□

Note that, as with μ, S is always a subset of $\{f_i | 1 \leq i \leq N\}$.

Lemma 4 $\mu(E,S) = 0 \Rightarrow \tau(E,S) \doteq E$

Proof is by structural induction over $\mathcal{E}$, where the inductive hypothesis is

$$\phi(E) \equiv \mu(E,S) = 0 \Rightarrow \tau(E,S) \doteq E \ .$$

Base steps:

$\mu(C,S) = 0, \tau(C,S) \doteq C$, and lemma holds for $E \equiv C$.

$\mu(V,S) = 0, \tau(V,S) \doteq V$, and lemma holds for $E \equiv V$.

$E \equiv f_i$
$\Rightarrow \mu(E,S) =$ if $f_i \in S$ then 0 else $N - |S|$ by definition of μ
$\wedge \ \tau(E,S) \doteq$ if $f_i \in S$ then f_i else $Y\lambda f_i \cdot F_i$ by definition of τ

- Case (i): $f_i \in S$.
  $\phi(E) \equiv \mu(E,S) = 0 \Rightarrow \tau(E,S) \doteq E$
  $\equiv 0 = 0 \Rightarrow f_i \doteq f_i$
  $\equiv true$

- Case (ii): $f_i \notin S$.
  The possible elements of set S are the identifiers $f_j, j = 1, \ldots, N$. Since $f_i \notin S$, $|S| < N$. Thus,
  $\phi(E) \equiv N - |S| = 0 \Rightarrow Y\lambda f_i \cdot F_i \doteq f_i$
  $\equiv false \Rightarrow Y\lambda f_i \cdot F_i \doteq f_i$
  $\equiv true$

Inductive steps:

$E \equiv \lambda v \cdot E_1$
$\Rightarrow \phi(E) \equiv \mu(E_1,S) = 0 \Rightarrow \lambda v \cdot \tau(E_1,S) \doteq \lambda v \cdot E_1$ by definition of μ and τ
$\equiv \mu(E_1,S) = 0 \Rightarrow \tau(E_1,S) \doteq E_1$ by matching components of expressions
$\equiv true$ by hypothesis, since $E_1 <_{\mathcal{E}} E$

[4] The conditional expression in the replacement for f_i is evaluated during transformation; the replacement is one or other of the limbs of the conditional expression.

$E{\equiv}Y\lambda f_i{\cdot}E_1$
$\Rightarrow \phi(E) {\equiv} \mu(E_1, S{\cup}\{f_i\}) = 0 \Rightarrow Y\lambda f_i{\cdot}\tau(E_1, S{\cup}\{f_i\}) \doteq Y\lambda f_i{\cdot}E_1$
by definition of μ and τ
$\equiv \mu(E_1, S{\cup}\{f_i\}) = 0 \Rightarrow \tau(E_1, S{\cup}\{f_i\}) \doteq E_1$
by matching components
$\equiv true$ by hypothesis, since $E_1 <_{\mathcal{E}} E$

$E{\equiv}E_1(E_2)$
The argument here is the same as for conditional expressions below, except that only two component expressions are involved.

$E{\equiv}$ if E_1 then E_2 else E_3
$\Rightarrow \phi(E) {\equiv} \max_{i\in\{1,2,3\}} \mu(E_i, S) = 0$
$\Rightarrow$ (if $\tau(E_1, S)$ then $\tau(E_2, S)$ else $\tau(E_3, S)) \doteq$ (if E_1 then E_2 else E_3)
by definition of μ and τ
$\equiv (\forall i{\in}\{1,2,3\}{\cdot}\mu(E_i, S) = 0) \Rightarrow (\forall j{\in}\{1,2,3\}{\cdot}\tau(E_j, S) \doteq E_j$
since μ is non-negative and by matching components
$\equiv \forall j{\in}\{1,2,3\}{\cdot}(\forall i{\in}\{1,2,3\}{\cdot}\mu(E_i, S) = 0) \Rightarrow \tau(E_j, S) \doteq E_j$
But, by hypothesis, $\forall j{\in}\{1,2,3\}{\cdot}\mu(E_j, S) = 0 \Rightarrow \tau(E_j, S) \doteq E_j$.
Therefore $\phi(E){\equiv}true$.

The lemma follows by structural induction.

□

Lemma 5 $\mu(E, S) \neq 0 \Rightarrow \mu(\tau(E, S), S) < \mu(E, S)$

Proof is by structural induction over $\mathcal{E}$, where the inductive hypothesis is

$$\phi(E){\equiv}\mu(E, S) \neq 0 \Rightarrow \mu(\tau(E, S), S) < \mu(E, S) \ .$$

Base steps:
For $E{\equiv}C, V : \mu(E, S) = 0$, so the lemma is trivially true.

$E{\equiv}f_i$
$\Rightarrow \ \mu(E, S) =$ if $f_i{\in}S$ then 0 else $N - |S|$ by definition of μ
$\wedge \ \tau(E, S) \doteq$ if $f_i{\in}S$ then f_i else $Y\lambda f_i{\cdot}F_i$ by definition of τ

- Case (i): $f_i{\in}S \Rightarrow \mu(E, S) = 0$ so $\phi(E)$ is trivially true.
- Case (ii): $f_i \notin S$.
  The possible elements of set S are the identifiers $f_j, j = 1, \ldots, N$. Since $f_i \notin S$, $|S| < N$. Thus,
  $\phi(E) {\equiv} N - |S| \neq 0 \Rightarrow \mu(Y\lambda f_i{\cdot}F_i, S) < N - |S|$
  $\equiv \mu(F_i, S{\cup}\{f_i\}) < N - |S|$ since $|S| < N$ and by definition of μ
  But, by Lemma 3, $\mu(F_i, S{\cup}\{f_i\}) \leq N - |S{\cup}\{f_i\}|$.
  Since $f_i \notin S$, $|S{\cup}\{f_i\}| = |S| + 1$.
  Thus, $\mu(F_i, S{\cup}\{f_i\}) \leq N - |S| - 1 < N - |S|$.
  Therefore $\phi(E){\equiv}true$.

Inductive steps
$E{\equiv}\lambda v{\cdot}E_1$
$\Rightarrow \phi(E) {\equiv} \mu(E_1, S) \neq 0 \Rightarrow \mu(\lambda v{\cdot}\tau(E_1, S), S) < \mu(E_1, S)$ by definition of μ and τ
$\equiv \mu(E_1, S) \neq 0 \Rightarrow \mu(\tau(E_1, S), S) < \mu(E_1, S)$ by definition of μ
$\equiv true$ by hypothesis, since $E_1 <_{\mathcal{E}} E$

$E \equiv Y\lambda f_i \cdot E_1$
$\Rightarrow \phi(E) \equiv \mu(E_1, S \cup \{f_i\}) \neq 0 \Rightarrow \mu(Y\lambda f_i \cdot \tau(E_1, S \cup \{f_i\}), S) < \mu(E_1, S \cup \{f_i\})$
by definition of μ and τ
$\equiv \mu(E_1, S \cup \{f_i\}) \neq 0 \Rightarrow \mu(\tau(E_1, S \cup \{f_i\}), S \cup \{f_i\}) < \mu(E_1, S \cup \{f_i\})$
by definition of μ
$\equiv true$ by hypothesis, since $E_1 <_{\mathcal{E}} E$

$E \equiv E_1(E_2)$
The argument here is the same as for conditional expressions below, except that only two component expressions are involved.

$E \equiv$ if E_1 then E_2 else E_3
$\Rightarrow \mu(E, S) = \max_{i \in \{1,2,3\}} \mu(E_i, S)$ by definition of μ
$\wedge \tau(E, S) \doteq$ if $\tau(E_1, S)$ then $\tau(E_2, S)$ else $\tau(E_3, S)$ by definition of τ
$\Rightarrow \mu(\tau(E, S), S) = \max_{i \in \{1,2,3\}} \mu(\tau(E_i, S), S)$ by definition of μ

There are two cases to be considered: $\mu(E, S) = 0$ and $\mu(E, S) \neq 0$.

- Case (i): $\forall i \in \{1, 2, 3\} \cdot \mu(E_i, S) = 0$. Then, $\mu(E, S) = 0$ and ϕ(E) is trivially true.
- Case (ii): for at least one value of $i \in \{1, 2, 3\}$, $\mu(E_i, S) \neq 0$ and thus $\mu(E, S) \neq 0$.

  Let $R \equiv \{i | 1 \leq i \leq 3 \wedge \mu(E_i, S) \neq 0\}$.
  Then $\mu(E, S) = \max_{i \in R} \mu(E_i, S)$.

  Now, by Lemma 4, $\mu(E_i, S) = 0 \Rightarrow \tau(E_i, S) \doteq E_i$.
  So $\mu(E_i, S) = 0 \Rightarrow \mu(\tau(E_i, S), S) = \mu(E_i, S) = 0$.
  Thus, $\forall i \notin R \cdot \mu(\tau(E_i, S), S) = 0$,
  and so $\mu(\tau(E, S), S) = \max_{i \in \{1,2,3\}} \mu(\tau(E_i, S), S) = \max_{i \in R} \mu(\tau(E_i, S), S)$.

  Now, $\forall i \in R \cdot E_i <_{\mathcal{E}} E$,
  so by hypothesis, $\forall i \in R \cdot \mu(\tau(E_i, S), S) < \mu(E_i, S)$.
  Therefore, $\mu(\tau(E, S), S) = \max_{i \in R} \mu(\tau(E_i, S), S) < \max_{i \in R} \mu(E_i, S) = \mu(E, S)$.
  Thus, $\phi(E) \equiv \mu(E, S) \neq 0 \Rightarrow \mu(\tau(E, S), S) < \mu(E, S) \equiv true$.

The lemma follows by structural induction.

□

A.5 Conclusion

The following properties hold of measure M and transformation $Inline$ (where $M(E) \equiv \mu(E, \emptyset)$, Definition 2):

$$\forall E \cdot M(E) \geq 0$$
$$\forall E \cdot M(E) = 0 \Rightarrow Inline(E) \doteq E$$
$$\forall E \cdot M(E) \neq 0 \Rightarrow M(Inline(E)) < M(E)$$

Proof:

$$M(E) = \mu(E, \emptyset) \geq 0 \quad \text{by Lemma 2}$$

$$M(E) = 0 \Rightarrow Inline(E) \doteq E$$
$$\equiv \mu(E, \emptyset) = 0 \Rightarrow \tau(E, \emptyset) \doteq E$$
$$\equiv true \quad \text{by Lemma 4}$$

$$M(E) \neq 0 \Rightarrow M(Inline(E)) < M(E)$$
$$\equiv \mu(E, \emptyset) \neq 0 \Rightarrow \mu(\tau(E, \emptyset), \emptyset) < \mu(E, \emptyset)$$
$$\equiv true \quad \text{by Lemma 5}$$

Therefore, $Inline$ satisfies the conditions for exhaustive application to terminate (Theorem 1), where the required measure is M. Since transformation $Construct$ is the application of $Inline$ to a finite number of expressions, $Construct$ is also guaranteed to terminate.

References

[1] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[2] K.D. Cooper, M.W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, April 1993.

[3] James M. Boyle and Terence J. Harmer. A practical functional program for the CRAY X-MP. *Journal of Functional Programming*, 2(1):81–126, 1992.

[4] James M. Boyle. Automatic, self-adaptive control of unfold-fold transformations. In E.-R. Olderog, editor, *Proceedings of the IFIP WG2.1-3 Working Conference on Programming Concepts, Methods and Calculi*, San Miniato, Italy, June 94. Elsevier Science B.V. `ftp://info.mcs.anl.gov/pub/tech_reports/reports/P416.ps.Z`.

[5] Björn Lisper. Total unfolding: Theory and applications. *Journal of Functional Programming*, 4(4):479–498, October 1994.

[6] M Clint, Stephen Fitzpatrick, T J Harmer, P L Kilpatrick, and J M Boyle. A family of data-parallel derivations. In Wolfgang Gentzsch and Uwe Harms, editors, *Proceedings of High Performance Computing and Networking, Volume II*, volume 797 of *Lecture Notes in Computer Science*, pages 457–462. Springer-Verlag, April 1994.

[7] J. Vuillemin. *Proof Techniques for Recursive Programs*. PhD thesis, Computer Science Department, Stanford University, Stanford, Calif., 1973.

[8] Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974. ISBN 0-07-039910-7.

[9] Owen Kaser, C. R. Ramakrishnan, and Shaunak Pawagi. On the conversion of indirect to direct recursion. *ACM Letters on Programming Languages and Systems*, 2:151–164, 1993.