A Variadic Extension of Curry's Fixed-Point Combinator

Mayer Goldberg (gmayer@cs.bgu.ac.il)*
Department of Computer Science
Ben Gurion University, Beer Sheva 84105, Israel

ABSTRACT

We present a systematic construction of a variadic applicative-order multiple fixed-point combinator in Scheme. The resulting Scheme procedure is a variadic extension of the n-ary version of Curry's fixed-point combinator. It can be used to create mutually-recursive procedures, and expand arbitrary `letrec`-expressions.

Keywords: Fixed points, fixed-point combinators, applicative order, lambda-calculus, Scheme, variadic functions

1. INTRODUCTION

Since the early days of Scheme programming, defining and using various fixed-point combinators have been classical programming exercises (for example, *Structure and Interpretation of Computer Programs* [1, Section 4.1.7, Page 393], and *The Little LISPer* [6, Chapter 9, Page 171]): Fixed-point combinators are used to replace recursion and circularity in procedures and data structures, with self application.

Replacing *mutual recursion* with self-application is done in one of two ways: (A) We can reduce mutually-recursive functions to simple recursive functions, and use a singular fixed-point combinator to replace singular recursion with self-application. Examples of this approach can be found in Bekič's theorem for the elimination of simultaneous recursion [3, Page 39], and in Landin's classical work on the mechanical evaluation of expressions [7]. (B) We can use a set of *multiple fixed-point combinators.* This approach is taken in a particularly beautiful construction due to Smullyan [2, Pages 334-335]. When replacing recursion among $n \geq 1$ recursive functions, a different set of multiple fixed-point combinators needs to be used for each n, each set containing progressively more complex expressions.

In Scheme, however, we can do better. Scheme provides syntax for writing *variadic procedures* (i.e., procedures that take arbitrarily many arguments), so that upon application, an identifier is bound to a list of these arguments. Scheme also provides two procedures that are particularly suitable for use in combination with variadic procedures:

- The `apply` procedure, which takes a procedure and a list, and applies the procedure to the elements of the list, as if it were called directly with the elements of the list as its arguments. For example: `(apply + '(1 2 3))` returns the same result as `(+ 1 2 3)`.

- The `map` procedure, which, in its simplest form, takes a procedure and a list of arguments, and applies the procedure to each one of these arguments, returning a list of the results. For example, `(map list '(1 2 3))` returns the list `((1) (2) (3))`.

By using variadic procedures, `apply`, and `map`, we can define a single Scheme procedure that can be used to define any number of mutually-recursive procedures. This way, we would not have to specify that number in advance.

The construction of variadic multiple fixed-point combinators is not immediate. We are only aware of one published solution — in Queinnec's book *LISP In Small Pieces* [8]. In Section 5 we compare our construction and the one found in Queinnec's book [8, Pages 457–458].

This work presents a variadic multiple fixed-point combinator that is a natural extension of Curry's fixed-point combinator. Our construction uses only as many Scheme-specific idioms as needed for working with variadic procedures (namely, `apply` and `map`), and is thus faithful both to the spirit of Scheme, in which it is written, as well as to the λ-calculus whence it comes.

The rest of this paper is organized as follows. We first review standard material about fixed-point combinators for singularly recursive procedures (Section 2), and then how to extend fixed-point combinators to handling mutual recursion among n procedures (Section 3). We then present our applicative-order variadic multiple fixed-point combinator (Section 4), and compares it with Queinnec's solution (Section 5). Section 6 concludes.

*This work was carried out while visiting Basic Research in Computer Science (`www.brics.dk`),. funded by the Danish National Research Foundation.

Permission to make digital or hard copies, to republish, to post on servers or to redistribute to lists all or part of this work is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To otherwise copy or redistribute requires prior specific permission.
Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.
Copyright 2002 Mayer Goldberg.

2. SINGULAR FIXED-POINT COMBINATORS

Fixed-point combinators are used in the λ-calculus to solve fixed-point equations. Given a term M, we are looking for a term x (in fact, the "smallest" such x in a lattice-theoretic sense) that satisfies the equation $Mx = x$ (the fixed-point equation), where equality is taken to be the equivalence relation induced by the one-step $\beta\eta$-relation. A *fixed-point combinator* is a term that takes any term M as an argument and returns the fixed point of M. If Φ is a fixed-point combinator, and M is some term, then $x = \Phi M$ is the fixed point of M, and satisfies $Mx = x$. Substituting the definition of x into the fixed point equation, we see that a fixed-point combinator is a term Φ such that for any term M, $\Phi M = M(\Phi M)$.

There exist infinitely-many different fixed-point combinators, though some are particularly well-known. The best known fixed-point combinator is due to Haskell B. Curry [4, Page 178]:

$$Y_{\text{Curry}} \equiv \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))$$

Encoding literally the above in Scheme would not work: Under Scheme's *applicative order* the application of

```
(lambda (f)
  ((lambda (x) (f (x x)))
   (lambda (x) (f (x x)))))
```

to *any* argument will diverge, because the application `(x x)` will evaluate before `f` is applied to it, resulting in an infinite loop. The solution is to replace `(x x)` with an expression that is both equivalent, and in which the evaluation of the given appilcation is delayed, namely, with a `lambda`-expression: If `(x x)` should evaluate to a one-argument procedure, then we can replace it with `(lambda (arg) ((x x) arg))`; If to a two-argument procedure, then we can replace it with `(lambda (arg1 arg2) ((x x) arg1 arg2))`, etc.[1] Not wanting to commit, however, to the arity of `(x x)`, we will use a variadic version of the η-expansion, i.e., wrap `(x x)` with `(lambda args (apply ··· args))`, giving:

```
(define Ycurry
 (lambda (f)
   ((lambda (x)
      (f (lambda args (apply (x x) args))))     (1)
    (lambda (x)
      (f (lambda args (apply (x x) args)))))))
```

Fixed-point combinators are used in programming languages in order to define recursive procedures [6, 7]. The trick is to define the recursive procedure as the solution to some fixed-point equation, and then use a fixed-point combinator to solve this equation. For example, the Scheme procedure that computes the *factorial* function satisfies the following recurrence relation

```
fact  ≡  (lambda (n)
           (if (zero? n) 1
               (* n (fact (- n 1)))))
```

can be rewritten as the solution of the following fixed-point equation:

```
fact  =  ((lambda (fact)
            (lambda (n)
              (if (zero? n) 1
                (* n (fact (- n 1)))))) fact)
```

and can be solved using, e.g., Curry's fixed-point combinator (Expr. (1)):

```
(define fact
  (Ycurry
   (lambda (fact)
     (lambda (n)
       (if (zero? n) 1
           (* n (fact (- n 1))))))))
```

3. MULTIPLE FIXED-POINT COMBINATORS

Just as recursive functions are solutions to fixed-point equations, which can be solved using fixed-point combinators, so are *mutually recursive functions* the solutions to *multiple fixed-point equations*, which can be solved using *multiple fixed-point combinators*:

A set of n multiple fixed points is defined as follows: Given the terms $M_1, \ldots, M_n$, we want to find terms $x_1, \ldots, x_n$ (the set of fixed points), such that $x_i = M_i x_1 \cdots x_n$, for $i = 1, \ldots, n$ (a system of n multiple fixed-point equations).

Extending the notion of singular fixed-point combinators, n multiple fixed-point combinators are terms $\Phi_1^n, \ldots, \Phi_n^n$, such that for any $M_1, \ldots, M_n$, we can let $x_i = \Phi_i^n M_1 \cdots M_n$, for $i = 1, \ldots, n$, and $\{x_i\}_{i=1}^n$ are the multiple fixed points that solve the given system of equations. Substituting the definitions of $\{x_i\}_{i=1}^n$ into the system of multiple fixed-point equations, we arrive at the concise statement that multiple fixed-point combinators are terms that $\Phi_1^n, \ldots, \Phi_n^n$, such that for any $M_1, \ldots, M_n$, we have

$$\begin{aligned}(\Phi_i^n M_1 \cdots M_n) &= \\ &M_i(\Phi_1^n M_1 \cdots M_n) \cdots (\Phi_n^n M_1 \cdots M_n)\end{aligned}$$

for all $i = 1, \ldots, n$.

Curry's fixed-point combinator can be extended to a set of n multiple fixed-point combinators for solving a system of n multiple fixed-point equations. The i-th such extension, ${Y_{\text{Curry}}}_i^n$, is given by

$$\begin{aligned}{Y_{\text{Curry}}}_i^n \equiv \lambda f_1 \cdots f_n.((\lambda x_1 \cdots x_n.f_i(&x_1 x_1 \cdots x_n) \\ &\cdots \\ &(x_n x_1 \cdots x_n)) \\ (\lambda x_1 \cdots x_n.f_1(&x_1 x_1 \cdots x_n) \\ &\cdots \\ &(x_n x_1 \cdots x_n)) \\ \cdots \\ (\lambda x_1 \cdots x_n.f_n(&x_1 x_1 \cdots x_n) \\ &\cdots \\ &(x_n x_1 \cdots x_n)))\end{aligned}$$

[1]This transformation is known colloquially as "η-expansion." The *η-reduction* consists of replacing $(\lambda\nu.M\nu)$ with M when ν does not occur free in M. The point of the η expansion in Scheme is that the body of procedures evaluate at application time rather than at closure-creation time, and so the η-expansion is used to *delay* evaluation.

Encoded in Scheme, the *applicative-order* version of Curry's multiple fixed-point combinator, $Y_{\text{Curry}_i^n}$, is given by:

```
(define Ycurryin
  (lambda (f1 ... fn)
    ((lambda (x1 ... xn)
       (fi (lambda args
             (apply (x1 x1 ... xn) args))
           ...
           (lambda args
             (apply (xn x1 ... xn) args))))
     (lambda (x1 ... xn)
       (f1 (lambda args
             (apply (x1 x1 ... xn) args))     (2)
           ...
           (lambda args
             (apply (xn x1 ... xn) args))))
     ...
     (lambda (x1 ... xn)
       (fn (lambda args
             (apply (x1 x1 ... xn) args))
           ...
           (lambda args
             (apply (xn x1 ... xn) args)))))))
```

Obviously, this is an *abbreviated meta-notation*, and for any specific n, the ellipsis ('$\cdots$') would need to be replaced with the corresponding Scheme expressions (so that `fi` refers to one of an actual list of parameters). In fact, one way to describe the aim of this paper is that we would like to avoid this meta-linguistic shorthand, and construct a Scheme procedure that takes arbitrarily-many arguments and returns a list of their multiple fixed points.

Mutually recursive function definitions can be rewritten as solutions to multiple fixed-point equations. For example, the Scheme procedures that compute the predicates *even*, *odd* satisfy the following mutual recurrence relation:

```
even? ≡ (lambda (n)
          (if (zero? n) #t
            (odd?  (- n 1))))
 odd? ≡ (lambda (n)
          (if (zero? n) #f
            (even? (- n 1))))
```

can be rewritten as the solutions of the following system of multiple fixed-point equations:

```
even? = ((lambda (even?  odd?)
           (lambda (n)
             (if (zero? n) #t
               (odd? (- n 1))))) even? odd?)
 odd? = ((lambda (even?  odd?)
           (lambda (n)
             (if (zero? n) #f
               (even? (- n 1))))) even? odd?)
```

and can be solved using Curry's multiple fixed-point combinators, where $n = 2$, and $i = 1, 2$:

```
(define E
  (lambda (even? odd?)
    (lambda (n)
      (if (zero? n) #t
          (odd? (- n 1))))))
(define O
  (lambda (even? odd?)                              (3)
    (lambda (n)
      (if (zero? n) #f
          (even? (- n 1))))))
(define even? (Ycurry12 E O))
(define odd? (Ycurry22 E O))
```

The aim of the next section is to show how we can construct a *variadic* version of Curry's multiple fixed-point combinator that can be used to define any number of recursive mutually-recursive procedures.

4. A VARIADIC MULTIPLE FIXED-POINT COMBINATOR

Expr. (Expr. (2)) specifies `Ycurryin` for any i, n, such that $1 \le i \le n$. For different choices of i, n, we would get a different procedure, and as n grows, each procedure gets progressively larger and more complex. This could be a real problem, for example, if we were to use multiple fixed-point combinators to expand `letrec`-expressions: We would need many different multiple fixed-point combinators, for many different values of n, even in a moderately-large program. We could, of course, hide the multiple fixed-point combinators through the use of a macro, but we couldn't hide the code bloat that would follow from the creation of a large number of these ever-growing "recursion-makers."

We address this issue by constructing a variadic multiple fixed-point combinator in Scheme. Variadic procedures, used together with the builtin procedures `apply` and `map`, form the basis for our programming idioms for working with meta-linguistic ellipsis in Scheme.

Throughout the rest of the section we are going to employ the following conventions, or rules, for converting "meta-linguistic Scheme" into actual Scheme:

Argv Lists of arguments will be written in Scheme as a single variable named in the *plural*. For example, `x1` $\cdots$ `xn` and `f1` $\cdots$ `fn` will be written as `xs` and `fs` respectively.

AbsArgv An abstraction over a list of arguments will be written in Scheme using a variadic lambda. For example: `(lambda (f1` $\cdots$ `fn)` *M*`)` will be written as `(lambda fs` *M*`)`.

AppArgv An application of a procedure to a list of arguments will be written as an application of the Scheme procedure `apply` to the procedure and the variable denoting the list of arguments. For example: The expression `(xi x1` $\cdots$ `xn)` will be written as `(apply xi xs)`.

IndAbsArgv A list of expressions that is indexed by some variable (e.g., `(x`*i* `x1` $\cdots$ `xn)`, which is indexed by `x`*i* $=$ `x1` $\cdots$ `xn`) will be written in the following way: We shall consider a "representative member" indexed by some variable, and then abstract over that variable and map the resulting procedure over the indexing set, using the `map` procedure. For example, the list of terms `(apply x1 xs)` $\cdots$ `(apply xn xs)` will be obtained by considering the "representative member" `(apply xi xs)`, abstracting over `xi`, and mapping the resulting procedure over `xs`, giving

```
(map (lambda (xi) (apply xi xs)) xs)
```

Recall $Y_{\text{Curry}}{}_i^n$, the variadic extension of Curry's fixed-point combinator, in Scheme (Expr. (2)):

```
(lambda (f1 ··· fn)
 ((lambda (x1 ··· xn)
   (fi (lambda args
        (apply (x1 x1 ··· xn) args))
       ...
       (lambda args
        (apply (xn x1 ··· xn) args))))
  (lambda (x1 ··· xn)
   (f1 (lambda args
        (apply (x1 x1 ··· xn) args))
       ...
       (lambda args
        (apply (xn x1 ··· xn) args))))
  ...
  (lambda (x1 ··· xn)
   (fi (lambda args
        (apply (x1 x1 ··· xn) args))
       ...
       (lambda args
        (apply (xi x1 ··· xn) args))
       ...
       (lambda args
        (apply (xn x1 ··· xn) args))))
  ...
  (lambda (x1 ··· xn)
   (fn (lambda args
        (apply (x1 x1 ··· xn) args))
       ...
       (lambda args
        (apply (xn x1 ··· xn) args)))))))
```

The various representative sub-expressions we will consider are enclosed in nested frames.

Starting with the innermost frame, for any `xi` = `x1` ... `xn`, the application `(xi x1 ··· xn)` is written, using the AppArgv rule, as

```
(apply xi xs)                                   (4)
```

Moving outward, towards the next enclosing frame, we apply to Expr. (4) the variadic version of the η-expansion in order to make sure that our fixed points reduce properly under applicative order:

```
(lambda args                                    (5)
 (apply (apply xi xs) args))
```

Note that the above expression is indexed by `xi` (that is, `xi` is a free variable that ranges over a list) in Expr. (5), and we need to obtain the list of such expressions for each `xi` = `x1` $\cdots$ `xn`. Using the IndAbsArgv rule, we abstract `xi` over Expr. (5) and map the resulting procedure over the list `xs`. The list of applications is therefore given by

```
(map (lambda (xi)                               (6)
      (lambda args
       (apply (apply xi xs) args))) xs)
```

Moving outward towards the next enclosing frame, we see that Expr. (6) forms the list of arguments to `fi` (which is also a free variable that ranges over a list). Using the AppArgv rule, the application is written out using `apply`:

```
(apply fi                                       (7)
 (map (lambda (xi)
       (lambda args
        (apply
         (apply xi xs) args))) xs))
```

Moving outward, towards the next enclosing frame, we see that Expr. (7) is the body of an abstraction over `x1` $\cdots$ `xn`. Using the AbsArgv rule, we encode this abstraction using a variadic lambda with the parameter `xs`:

```
(lambda xs                                      (8)
 (apply fi
  (map (lambda (xi)
        (lambda args
         (apply
          (apply xi xs) args))) xs)))
```

Moving outward, towards the next enclosing frame, we see that Expr. (8) is indexed by `fi` (that is, `fi` is a free variable that ranges over a list) in Expr. (8), and we need to obtain the list of such expressions for each `xi` = `x1` ... `xn`. Using the IndAbsArgv rule, we abstract `fi` over Expr. (8) and map the resulting procedure over the list `fs`. The list of applications is therefore given by

```
(map (lambda (fi)                                    (9)
      (lambda xs
       (apply fi
        (map (lambda (xi)
              (lambda args
               (apply (apply xi xs) args)))
             xs))))
     fs)
```

The above expression corresponds to the list `x1` $\cdots$ `xn`. The next step is to compute the list of multiple fixed points: For any particular $i \in \{1, \ldots, n\}$, the i-th fixed-point combinator $Y_{\text{Curry}}{}_i^n$ is given by `(xi x1 ··· xn)`, which, in Scheme would be written as `(apply xi xs)`. Using the IndAbsArgv rule, to obtain the list of all such terms, for each `xi` = `x1` ... `xn`, we abstract `xi` over Expr. (9) and map the resulting procedure over the list `xs`. This is the second time we have referred to the list `x1` $\cdots$ `xn` in this step, so rather than compute it twice, we bind its value to the identifier `xs`, using a `let`-expression, the body of which will be:

```
(map (lambda (xi)
      (apply xi xs)) xs)
```

Using the AbsArgv rule, we now abstract over `f1 ··· fn` using a variadic `lambda`, and define the procedure `curry-fps` that takes any number of procedures and returns a list of their multiple fixed points:

```
(define curry-fps
  (lambda fs
    (let ((xs
           (map
            (lambda (fi)
              (lambda xs
                (apply fi
                 (map                              (10)
                  (lambda (xi)
                    (lambda args
                      (apply (apply xi xs) args)))
                      xs))))
            fs)))
      (map (lambda (xi)
             (apply xi xs)) xs))))
```

On the other hand, if we are only interested in $Y_{\text{Curry}_1^n}$, for example, for the purpose of macro-expanding `letrec`-expressions without using side-effects, then we can simplify the body of the `let`-expression in Expr. (10) so that we just compute the first fixed point:

```
(define curry-fps-1n
  (lambda fs
    (let ((xs
           (map
            (lambda (fi)
              (lambda xs
                (apply fi
                 (map (lambda (xi)                 (11)
                        (lambda args
                          (apply
                           (apply xi xs) args)))
                      xs))))
            fs)))
      (apply (car xs) xs))))
```

For example, consider the general `letrec`-expression, where $M_1, \ldots, M_n$ denote the definitions of the procedures `f1`, ..., `fn` respectively, and $Expr_1, \ldots, Expr_m$ denote the expressions in the body of the letrec:

```
(letrec ((f1 M1)
               ⋮
         (fn Mn))
  Expr1 ··· Exprm)
```

Using `curry-fps-1n`, the above expression can be rewritten, without side effects, using the *fresh variable* `body`, as follows:

```
(curry-fps-1n
 (lambda (body f1 ··· fn) Expr1 ··· Exprm)
 (lambda (body f1 ··· fn) M1)
                          ⋮
 (lambda (body f1 ··· fn) Mn))
```

5. RELATED WORK

In his book *LISP In Small Pieces* [8, Pages 457–458], Queinnec exhibits the Scheme procedure `NfixN2`, that is a variadic, applicative-order multiple fixed-point combinator. The `NfixN2` procedure, along with a help procedure are given below:

```
(define NfixN2
  (let ((d
         (lambda (w)
           (lambda (f*)
             (map
              (lambda (f)
                (apply f
                 (map
                  (lambda (i)
                    (lambda a
                      (apply
                       (list-ref ((w w) f*)(12)
                                 i)
                       a)))
                  (iota 0 (length f*)))))
              f*)))))
    (d d)))

(define iota
  (lambda (start end)
    (if (< start end)
        (cons start (iota (+ 1 start) end))
        '())))
```

While Queinnec's construction certainly works, it strikes us as unnatural in the context of Scheme:

The name of the `iota` help procedure comes from the programming language APL, where, given an integer argument n, the monatic *iota* function returns the vector of integer in the range $1, \ldots, n$. The above implementation of *iota* in Scheme takes two integers *start* and *end*, and returns the *list* of integers in the range of $start, \ldots, end - 1$. A common programming idiom in APL is to de-reference a vector `v` by another vector `w` of indecies into `v`, to obtain a permuted sub-vector of `v` that is the same size as `w`. We note the use of this idiom in the procedure `NfixN2`, where the list `(iota 0 (length f*))` is used as a list of indecies for extracting elements from the list returned by `((w w) f*)`, and is used to construct a new list in the expression `(map (lambda (i) ···) (iota 0 (length f*)))`. In fact, the procedure `NfixN2` could be coded directly and concisely into DyalogAPL [5], a dialect of APL2 that supports closures and higher-order functions.

Furthermore, the construction of the variadic fixed-point combinator is not a natural extension of one of the familiar singular fixed-point combinators, e.g., Curry's or Turing's fixed-point combinators. This is probably due to the use of APL idioms in the code.

We feel that from a pedagogical point of view, it would be better to exhibit a variadic fixed-point combinator that is a natural extension of one of the familiar singular fixed-point combinators, in a way that would be both systematic and native to Scheme.

The natural idioms for working with variadic procedures are `apply` and `map`. By sticking to these procedures and declining the temptations to use other procedures and other metaphors, we obtained a construction for multiple fixed-point combinator that corresponds rather faithfully to the extension of Curry's fixed-point combinator to n multiple fixed-point equations.

An additional benefit is efficiency: While fixed-point combinators in Scheme are not normally evaluated by their efficiency (but rather by applicative order call-by-value), and while this remark is not intended to be a sales pitch for super-fast, super-efficient fixed-point combinators, it should not be surprising that when we adhere to programming metaphors that are natural for a given language, we are often rewarded by better execution times. The `NfixN2` (Expr. (12)) procedure presented in Section 5 uses a variant of the *iota* function in APL in order to generate indices, repeatedly, with each recursive call. But generating indices is only half the problem: The individual functions are then accessed using `list-ref`, which traversing a list of functions at linear time. This suggests that the average time to access a function will increase as the number of mutually recursive functions grows. Empirical evidence suggests that this is indeed the case.

We ran two kinds of tests: Firstly, we tried to see how the two variadic fixed-point combinators would perform on a set of mutually-recursive functions, as the input grew. Secondly, we tried to see how the two variadic fixed-point combinators would perform when the number of mutually-recursive functions was increased.

Below, we tabulated the CPU time (in milliseconds, under Petite Chez Scheme running on a dual UltraSPARC-II with 1G RAM) for evaluating the mutually-recursive `even?` and `odd?` procedures for various input values (given by N):

N	`curry-fps`	`NfixN2`
10^3	10	30
10^4	70	250
10^5	760	2,430
10^6	7,500	24,650
10^7	74,810	242,570

In order to test execution speeds as the number of mutually-recursive procedures changed, we created a Scheme procedure that given an integer argument `n` creates n mutually-recursive procedures. We started with the standard definition of *Ackermann's function*:

```
(lambda (a b)
  (cond ((zero? a) (+ b 1))
        ((zero? b) (ack (- a 1) 1))
        (else (ack (- a 1)
                   (ack a (- b 1))))))
```

We duplicated this definition n times, numbering each instance sequentially. Then, for each call to `ack` in the body of each of the instances of `ack`, we randomly select one of the numbered instances. We then use the same n mutually-recursive procedures to time the computation of *Ackermann*(3, 5) using both variadic multiple fixed-point combinators.

The table below lists the CPU time (again, milliseconds, under Petite Chez Scheme running on a dual UltraSPARC-II with 1G RAM) for evaluating mutually-recursive versions of Ackermann's function, as the number of mutually-recursive procedures (given by N) varies.

N	`curry-fps`	`NfixN2`
1	320	570
2	390	1,160
3	390	1,910
10	770	14,280
20	1,240	51,670
30	1,770	112,050
100	5,640	1,199,370
200	12,650	5,020,170
300	18,970	> 12 hours
1000	99,929	—

It is clear that using either `NfixN2` or `curry-fps` to define a large set of mutually-recursive procedures will result in performance penalties that are proportional to the number of procedures, however it is also clear that using the built-in support for working with lists of arguments (i.e., `apply`, `map` and variadic procedures) is superior to picking individual elements explicitly.

6. CONCLUSION

We presented a systematic construction for an applicative-order variadic multiple fixed-point combinator in Scheme. Starting out with Curry's singular fixed-point combinator, we considered the extension to multiple fixed-point equations, parameterized by the number of equations and the index of the fixed point. Using variadic `lambda`-expressions, and the elementary procedures for working with lists of arguments (`apply`, `map`), we were able to formulate four rules, or conventions, for converting Scheme expressions written with meta-linguistic ellipses ('$\cdots$') into actual Scheme code. This enabled us to define a multiple fixed-point combinator without having to specify the number of equations. The value returned by this procedure is the list of all the fixed points.

This variadic multiple fixed-point combinator directly corresponds to the multiple fixed-point extension of Curry's singular fixed-point combinator.

7. ACKNOWLEDGMENTS

This work was supported in part by the Danish Research Academy. The author is grateful to BRICS[2] for hosting him and for providing a stimulating environment. Thanks are also due to Olivier Danvy for his invaluable suggestions and encouragement, and to the referees, whose comments and suggestions on an earlier version of the manuscript did much to improved the paper.

APPENDIX

A. SAMPLE RUN

[2]Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

We encode the procedures `even?` and `odd?` in Scheme (Exprs. (3)) using the variadic `curry-fps` procedure (defined in Expr. (10)) for computing the list of multiple fixed points.

```
;;; defining the Even functional:
> (define E
    (lambda (even? odd?)
      (lambda (n)
        (if (zero? n) #t ; return Boolean True
            (odd? (- n 1))))))
;;; Defining the Odd functional:
> (define O
    (lambda (even? odd?)
      (lambda (n)
        (if (zero? n) #f ; return Boolean False
            (even? (- n 1))))))
;;; Finding the list of fixed points:
> (define list-even?-odd? (curry-fps E O))
> (define even? (car list-even?-odd?))
> (define odd? (cadr list-even?-odd?))
> (even? 6)
#t
> (odd? 4)
#f
```

B. REFERENCES

[1] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, McGraw-Hill Book Company, Second edition, 1996.

[2] Hendrik P. Barendregt. Functional Programming and the λ-calculus. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 7, pages 323 – 363. MIT Press, Cambridge, Massachusetts, 1990.

[3] Hans Bekič. *Programming Languages and Their Definition*. Number 177 in Lecture Notes in Computer Science. Springer-Verlag, 1984.

[4] Haskell B. Curry, Robert Feys, and William Craig. *Combinatory Logic*, volume I. North-Holland Publishing Company, 1958.

[5] Dyadic Systems, Limited. Dyalog APL. `http://www.dyadic.com/`.

[6] Daniel P. Friedman and Matthias Felleisen. *The Little LISPer*. Science Research Associates, Inc, 1986.

[7] Peter J. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6:308–320, 1964.

[8] Christian Queinnec. *LISP In Small Pieces*. Cambridge University Press, 1996.