Closed form of primitive recursive functions: from imperative programs to mathematical expressions to functional programs

Armando B. Matos (armandobcm@yahoo.com)

November 3, 2014

Abstract

We compare three forms of representing primitive recursive (PR) functions: the "standard" definition, essentially based on primitive recursion, a definition based on a specific register language (whose programs always halt) called LOOP, and the "denotational" definition which uses the methods of denotational semantics applied to the characterization of PR functions.

If a language is total we do not need all the machinery developed by Scott, Stratchey, and others in order to characterize its denotational semantics. For the case of the LOOP language the situation is even easier, because the composition of first order functions and one second order function is sufficient to write a "closed mathematical expression" of an arbitrary primitive recursive definition. The translation of the mathematical expression into a single-line functional language computation CR is straightforward. In the expression CR, C denotes a combinator expression (that represents a PR function) and R the tuple of variables. The set of PR functions correspond exactly to the well formed expressions of combinators allowed in C, namely inc, dec, zero, and iter.

Although not obvious from the standard definition of primitive recursive functions (involving in general several layers of primitive recursive definitions), the possibility of representing the function as a single mathematical expression is not surprising; it follows easily from the LOOP program representation.

1 Introduction

If the functions mapping tuples of integers into tuples of integers (of the same arity) are used as the mathematical objects, and the *composition* of those functions as the mathematical operation, then every primitive recursive function has a closed mathematical form. Only a few such functions are needed, but one of them is a second order function.

The class of primitive recursive functions can be characterized by the LOOP register language [7, 8], (or its equivalent variants). We use the denotational semantics of LOOP to obtain a closed form mathematical expression[1] of *any* PR function. The LOOP program 2 (page 9) and the mathematical expression of the function it defines 3 (page 9) look similar – once the notation is understood. They are however entities from different conceptual worlds – programming and mathematics.

From the closed form expression of a PR function f and its arguments $\overline{x}$ we can also easily define a functional expression from which we can, in a functional language like Haskell [10, 4], evaluate $f(\overline{x})$, without using any program defining the function; that is, the function is completely described by the (user) input *term*. This contrasts with the use of the standard definition of primitive recursive functions which can be directly translated into a Haskell *program*. Figure 1 (page 4) illustrates this somewhat simplistic view.

2 The standard definition of primitive recursive functions

A primitive recursive function (PR) is either basic [11, 5, 9] or defined recursively in terms of other primitive functions; thus, the complete definition involves in general several levels of recursion.

[1]If we accept as a mathematical expression the composition of a function with itself x times, where x is a (mathematical) variable, that is, something like $\lambda x_1 \ldots x_n \cdot (\text{iter}\, f\, x_k)(x_1, \ldots x_n)$, where "iter" is defined by "$\text{iter}\, f\, 0 = \text{identity}$" and "$\text{iter}\, f\,(n{+}1) = f \cdot (\text{iter}\, f\, n)$".

PR definitions

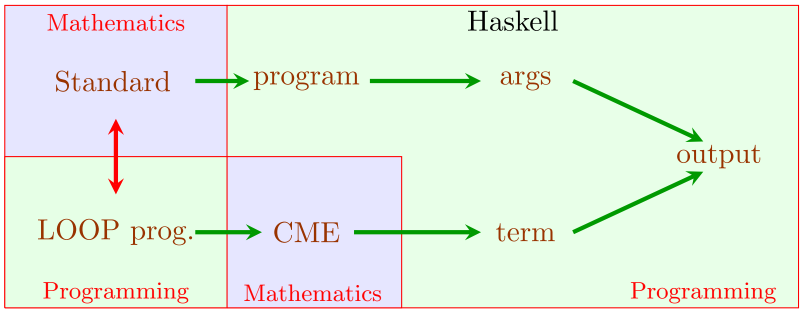


Figure 1: Left: two definitions of primitive recursive functions: (i) [top center and right] a Haskell program is defined from the standard definition of a PR function and runs with the given arguments ("args"); (ii) [bottom center and right] the mathematical expression that represents the PR function, denoted by CME for "Closed Mathematical Expression". It is an abstract representation which is obtained from the semantics of the "LOOP prog.". The "term" node is a Haskell term (including the definition and the values of the arguments) obtained from the CME. It is given by the user as input to the Haskell interpreter.

Definition 1 The class of primitive recursive (PR) functions is the smallest class satisfying

1. [Zero and successor] $0(x) = 0$ and $S(x)$ (successor function) are PR.
2. [Projection] For every $n \geq 1$ and every $0 \leq i \leq n$ the function $\pi_i^n(x_1, \ldots, x_n) = x_i$ is PR.
3. [Composition] For every k, $m \geq 1$, given the k-ary PR function f and the k PR m-ary functions g_i $(1 \leq i \leq m)$, the function

$$h(x_1, \ldots, x_m) = f(g_1(x_1, \ldots, x_m), \ldots, g_k(x_1, \ldots, x_m)$$

is PR.
4. [Primitive recursion] Given the k-ary PR function f and the $k+2$-ary PR function g, the function

$$\begin{cases} h(0, x_1, \ldots, x_k) & = & f(x_1, \ldots, x_k) \\ h(S(y), x_1, \ldots, x_k) & = & g(y, h(y, x_1, \ldots, x_k), x_1, \ldots, x_k) \end{cases}$$

is PR.

3 Register languages

3.1 Instruction `for` and controlled composition

We use a variant of the LOOP language of [7] – a simple, imperative register language. The functions that can be written in this language are *exactly* the primitive recursive functions [7, 8].

Definition 2 LOOP language
A program is a sequence of zero or more instructions. The possible instructions are as follows, where x and x' denote the contents of register `x` respectively before and after the execution of the instruction.

- `inc x`: increment a register by 1, $x' = x + 1$.
- `dec x`: decrement a register by 1, $x' = x \dot{-} 1$.
- `x←0`: set to zero, $x' = 0$.
- `for x`(P): loop instruction, the LOOP program P is executed a number of times which is the initial value of `x`. The program P can modify the variable x.

Initially the arguments of an n-ary function are the contents of the registers $\mathtt{x}_1, \ldots, \mathtt{x}_n$. The output value is the final value of the register $\mathtt{x}_0$.

To distinguish programming variables ("registers") from mathematical ones we denote the former like this `x` and the later like this x.

We give particular attention to the semantics of the instruction `for`.

In order to compose a function with itself, we assume that both the inputs and the outputs (note the plural) of a function (we could call it a "multifunction") are the same, say $\overline{x} \stackrel{\text{def}}{=} (x_1, \ldots, x_n)$.

We denote a function f composed with itself x times by

$$f^{(x)}(\overline{z}) \stackrel{\text{def}}{=} \overbrace{f(f(\ldots f(\overline{z}) \ldots))}^{x\ f\text{'s}}$$

In particular $f^{(0)}(\overline{z}) = \overline{z}$ and $f^{(1)}(\overline{z}) = f(\overline{z})$.

A register program P that uses registers $\overline{x}$ implements a function $T_P : \mathbb{N}^n \to \mathbb{N}^n$. Let the operational semantics of `for` $x(P)$ be: execute x times the program P, where x is the input value corresponding to the register x (there is some ambiguity here). Thus, we can characterize the denotational semantics of the `for` instruction as

$$\frac{P \quad\Longrightarrow\quad T_P}{\texttt{for}\,\texttt{x}(P) \quad\Longrightarrow\quad {T_P}^{(x)}}$$

The LOOP language can be characterized by a denotational semantics based in "transformations" or "multifunctions" (or simply "functions" of tuples into tuples), like T_P above or T in Figure 2 (page 7). This semantics includes projection, *composition* of functions (corresponding to the sequence of two programs), and *controlled composition* of a program, see again Figure 2. It differs from the standard denotational semantics in two aspects: (i) the semantic objects are exclusively functions and (ii) there are no questions about program (or definition) convergence, because all LOOP programs are total.

3.2 Instruction `for`: semantic analysis of an example

The `for` instruction is an iterative instruction of the LOOP language. Imperative languages usually have iterative instructions. We have an *iteration* when the same sequence of instructions (or program) are repeated a certain number of times. Here we assume that number is known before the instruction starts. If follows that, if the inner program never loops indefinitely, that the execution of the `for` instruction also finishes.

Assume that we want to compute the product

$$F \stackrel{\text{def}}{=} E(1) \times E(2) \times \ldots \times E(n)$$

For $1 \leq i \leq n$ it is also assumed that the arithmetical value $E(i)$ can be computed by a LOOP program, say P_E. A program in a LOOP-like language that computes F is then

```
t ← 0; inc t;
i ← 0; inc i;
for n (
    e ← value of E(i);
    t ← t × e;
    inc i;
    )
return t;
```

The variable assignment ("←") and multiplication are easily shown to be implementable in LOOP. By assumption $E(i)$ can be computed by a LOOP program P_E.

Figure 2 (page 7) is a diagram of the previous computation expressed in terms of a composition of equal functions.

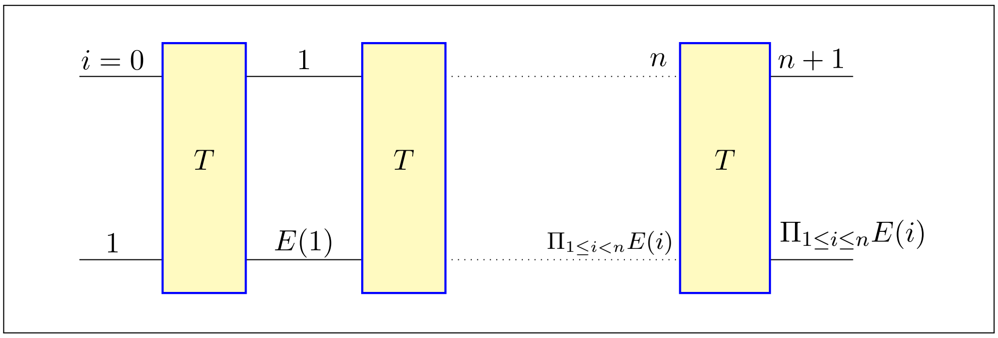


Figure 2: Semantics of a LOOP program equivalent to "`t`$\leftarrow$1; `i`$\leftarrow$1; `for n(e` $\leftarrow$ value of $E($`i`$)$; `t` $\leftarrow$ `t` $\times$ `e`; `inc i`); `return` t". The transformation T given in (1) (page 7) is composed n times with itself, where n is the input value of the register `n`; we have $T^{(n)}(0, 1) = (n, \Pi_{0 \le i < n} E(i))$.

We now look ahead to the semantics of this LOOP program. The transformation $T : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ is

$$T : \begin{cases} i' = i + 1 \\ t' = t \times E(i) \end{cases} \tag{1}$$

As instruction sequencing corresponds semantically to composition, we get

$$\text{Computation of } F \quad \Leftrightarrow \quad T^{(n)}|_2 = \Pi_{0 \le i < n} E(i)$$

where "$|_2$" means "take the projection along the second argument".

4 Denotational semantics of the LOOP language

In the previous section we described the denotational semantics of one instruction of the LOOP language, namely the `for` instruction.

In Figure 3 (page 8) the complete semantics of the LOOP language is described.

This semantics is much simpler than the denotational semantics of languages that may contain constants and variables of several types and of non total languages (whose programs may not halt) because:

- The codomain of the semantics function is a set of functions $\mathbb{N}^n \to \mathbb{N}^n$. We dont't have to deal with integers, the "store", integer expressions...

- A program and its parts is always interpreted as a function of a tuple of n integers into a tuple of n integers.

LOOP program		Transformation $\mathbb{N}^n \to \mathbb{N}^n$
$\texttt{inc}\,\texttt{x}_i$	$\to$	${}^n[\lambda x_i.x_i+1]$
$\texttt{dec}\,\texttt{x}_i$	$\to$	${}^n[\lambda x_i.x_i \dot{-} 1]$
$\texttt{x}_\texttt{i} \leftarrow \texttt{0}$	$\to$	${}^n[\lambda x_i.0]$
$\texttt{for}\,\texttt{x}_i(P)$	$\to$	${}^n[{T_P}^{(x_i)}]$
$P;\,Q$	$\to$	${}^n[T_Q]\cdot{}^n[T_P]$

Figure 3: Semantics of the LOOP language. By ${}^n[T]$ we mean that the transformation T, which only mentions some of the variables $x_1,\ldots,x_n$, is extended to all those variables; for transformation associated with every variable not mentioned by T is the identity. The transformation associated with the programs P and Q are denoted by T_P and T_Q respectively. In the last line the symbol "$\cdot$" denotes the composition of transformations.

– As all the programs are total, we do not need a part of the Stratchey and Scott semantics theory [13, 12, 14]. In particular we do not need concepts like lattices, continuity, least upper bounds...

However we use the mathematical concept of "compose n times a function $T : \mathbb{N}^n \to \mathbb{N}^n$ with itself", denoted by $T^{(n)}$.

Notice that the rules described in Figure 3 allows us to describe the syntax of any LOOP program as a single mathematical expression.

In order to simplify the notation we adopt the following conventions

– (Syntax and semantics) the registers $\texttt{x}_1$, $\texttt{x}_2$, ..., $\texttt{x}_n$ may be denoted by letters like `a`, `b`, `n`, `x`, `y`..., and similarly for mathematical variables.

– (Semantics) the extension to the n variables[2] $(x_1,\ldots,x_n)$ is implicit. Thus we write T instead of ${}^n[T]$.

– (Semantics) $\lambda x_i.x_i{+}1$, $\lambda x_i.x_i \dot{-} 1$, and $\lambda x_i.0$ will be abbreviated by $\lambda_{x_i}[+1]$, $\lambda_{x_i}[\dot{-}1]$, and $\lambda_{x_i}[0]$ respectively.

5 An example: various forms of expressing a primitive recursive function related to the Fibonacci sequence

We express the same PR function in several forms:

– a LOOP program, Section 5.1 (page 9); program 2, page 9;

[2] Or $(x_0,\ldots,x_{n-1})$, when more convenient.

– a mathematical expression, Section 5.2 (page 9); expression (3), page 9;

– a Haskell term, Section 5.3 (page 9); expression (4), page 10;

– using the "standard definition", see Section 5.4 (page 12).

We follow the above order: from the LOOP program, to the closed mathematical expression, the Haskell computation and finally to the standard definition (this last part is not completed).

5.1 LOOP program

The example is the following LOOP program over the registers (a, b, n) with output[3] b

$$P = \boxed{\texttt{for n(for b(inc a); for a(inc b))}} \tag{2}$$

The transformed values of a and b, as a function of n are explained in Section 4.2 of[4] `www.dcc.fc.up.pt/~acm/questionsv.pdf`.

5.2 Closed mathematical expression

The semantics of the program above is[5]

$$T_P = \boxed{(\lambda_b[+1]^{(a)} \cdot \lambda_a[+1]^{(b)})^{(n)}} \tag{3}$$

It is interesting how the denotational semantics of a LOOP program is always a single mathematical expression from which we can compute directly particular transformed tuples such as $T_P(1, 2, 10)$.

Theorem 1 *Any primitive recursive function has a closed mathematical expression involving the composition and the controlled composition of the "increment", "decrement", and "set-to-zero" tuple transformations.*

Note. The "set-to-zero" transformation is redundant, for it can be obtained wth the other transformations. □

5.3 Haskell term

From a closed mathematical expression E of the n-ary primitive recursive function it is easy to define a "prompt command" $\texttt{E}'(\texttt{x}_1, \ldots \texttt{x}_\texttt{n})$ for the computation

[3] Formally, one should use $x_1 \Leftrightarrow a$, $x_2 \Leftrightarrow b$, $x_3 \Leftrightarrow n$, and make at the end an assignment $x_0 \leftarrow b$.

[4] This work deals however with the "reversible LOOP language".

[5] Note that the conventional orders used for sequencing and for composition are opposite.

of $E(x_1, \ldots, x_n)$ applied to n arguments.

First we use an algorithm that, given n, generates a set of fixed definitions, as illustrated in Figure 4 for the case $n = 3$ (page 11). We emphasize that this algorithm is fixed; it is the same for every n-ary primitive recursive function.

Then, from the closed mathematical expression corresponding to the primitive recursive function, we write a Haskell term to obtain the desired computation.

In our example, we rewrite 3 (page 9) as a Haskell term.

$$\left(\lambda_b[+1]^{(a)} \cdot \lambda_a[+1]^{(b)}\right)^{(n)}$$

$$\Downarrow$$

```
iter3 ((iter1 inc2).(iter2 inc1))
```

For reference we rewrite the Haskell term

$$? \ \boxed{\texttt{iter3 ((iter1 inc2).(iter2 inc1))}} \tag{4}$$

Using the Haskell interpreter we get for instance

```
?  iter3 ((iter1 inc2).(iter2 inc1)) (1,0,10)
> (4181,6765,10)
```

What is this particular primitive recursive transformation? That should be clear from the analysis of the answers to the following prompts; note that the values of the function are 0, 1, 3, 8... (second from the right column)

Prompt	Answer
`iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 0)`	`(   1,   0, 0)`
`iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 1)`	`(   1,   1, 1)`
`iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 2)`	`(   2,   3, 2)`
`iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 3)`	`(   5,   8, 3)`
`iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 4)`	`(  13,  21, 4)`
`iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 5)`	`(  34,  55, 5)`
`iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 6)`	`(  89, 144, 6)`
`iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 7)`	`( 233, 377, 7)`
`iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 8)`	`( 610, 987, 8)`
`iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 9)`	`(1597,2584, 9)`
`iter3 ((iter1 inc2).(iter2 inc1)) (1,0,10)`	`(4181,6765,10)`

Fixed part:

```
comp 0      _  = id
iter (n+1) tr = tr.(iter n tr)
inc x = x+1
dec 0     = 0
dec (n+1) = n
zero _ = 0
```

This part depends only on the arity of the functions. In this case $n = 3$.

```
inc1 (x,y,z) = (inc x,y,z)
inc2 (x,y,z) = (x,inc y,z)
inc3 (x,y,z) = (x,y,inc z)

dec1 (x,y,z) = (dec x,y,z)
dec2 (x,y,z) = (x,dec y,z)
dec3 (x,y,z) = (x,y,dec z)

zero1 (x,y,z) = (zero x,y,z)
zero2 (x,y,z) = (x,zero y,z)
zero3 (x,y,z) = (x,y,zero z)

iter1 tr (x,y,z) = iter x tr (x,y,z)
iter2 tr (x,y,z) = iter y tr (x,y,z)
iter3 tr (x,y,z) = iter z tr (x,y,z)
```

Evaluating a primitive recursive function with given arguments. The function is completely described in the command line.

```
> iter3 ((iter1 inc2).(iter2 inc1)) (1,0,10)  -- function args
  (4181,6765,10)                              -- answer
```

Figure 4: The fixed set of Haskell definitions is illustrated for the case $n = 3$. With this fixed algorithm we can (bottom rectangle), evaluate *any* n-ary primitive recursive function.

5.4 Standard definition

The standard definition of primitive recursive function is Definition 1, page 4. From [7, 8] it follows that the program 2, page 9 (or the mathematical expression 3, page 9) define a primitive recursive function. Direct proofs of this fact, that is, proofs using Definition 1, seem to be somewhat more involved, see for instance [3]; when written in detail, they involve a relatively large primitive recursive definitions.

6 Another example: the maximum of two integers

As another example of a PR function written as a closed mathematical formula, we consider the maximum of two integers, $r \stackrel{\text{def}}{=} \max(x, y)$.

A LOOP program that computes the maximum is

	Instruction	Comment
(1)	$\texttt{a} \leftarrow 0$; $\texttt{for}\,\texttt{x}(\texttt{inc}\,\texttt{a})$;	copy of $\texttt{x} \;\rightarrow\; \texttt{a}$
(2)	$\texttt{for}\,\texttt{y}(\texttt{dec}\,\texttt{a})$;	if $\texttt{x} > \texttt{y}$: $\texttt{a} > 0$
(3)	$\texttt{r} \leftarrow 0$; $\texttt{for}\,\texttt{y}(\texttt{inc}\,\texttt{r})$;	$\texttt{r} = \texttt{y}$ by default...
(4)	$\texttt{for}\,\texttt{a}(\texttt{r} \leftarrow 0;\ \texttt{for}\,\texttt{x}(\texttt{inc}\,\texttt{r}))$;	if $\texttt{y} > \texttt{y}$: result is $\texttt{r} = \texttt{x}$

Using the notation previously described, we get the following mathematical expression[6]:

$$T_P \;=\; \overbrace{(\lambda_r[+1]^{(x)} \cdot \lambda_r[0])^{(a)}}^{(4)} \cdot \overbrace{\lambda_r[+1]^{(y)} \cdot \lambda_r[0]}^{(3)} \cdot \overbrace{\lambda_a[\dot{-}1]^{(y)}}^{(2)} \cdot \overbrace{\lambda_a[+1]^{(x)} \cdot \lambda_a[0]}^{(1)}$$

The sub-terms corresponding to the lines (1)–(4) of the LOOP program above are indicated.

In order to write the Haskell term we make the following correspondence of variables: $\texttt{r} \rightarrow \texttt{x}_0$, $\texttt{x} \rightarrow \texttt{x}_1$, $\texttt{y} \rightarrow \texttt{x}_2$, $\texttt{a} \rightarrow \texttt{x}_3$.

```
 iter3(iter1 inc0.setZ0)
.iter2 inc0.setZ0
.iter2 dec3
.iter1 inc3.setZ3
```

or, with shorter mnemonics, `c3(c1 i0.z0).c2 i0.z0.c2 d3.c1.i3.z3`

[6] Recall that, by convention, opposite orders are used to write the composition of two functions and the sequencing of two programs.

Experimenting in the Haskell interpreter:

```
                                                 r x  y  a
>  (c3(c1 i0.z0).c2 i0.z0.c2 d3.c1 i3.z3) (0,20,30,0)
   (30,20,30,0)
```

The first integer of the result, 30, is the maximum.

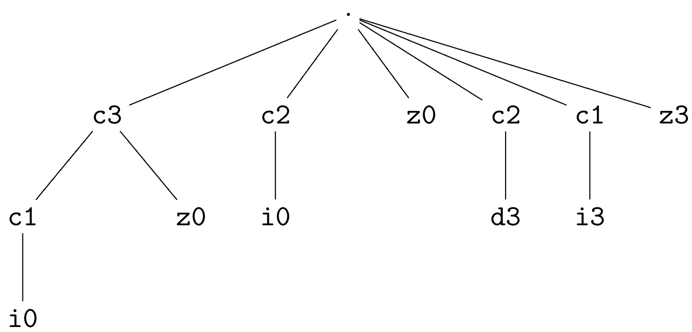
It is not difficult to write the standard definition of $\max(x, y)$: it involves several primitive recursions.

7 Comment

The program part of the Haskell computation, such as

```
c3(c1 i0 z0) c2 i0 z0 c2 d3 c1 i3 z3
```

in the last example (page 13) can be seen as a “combinator expression”. It can corresponds to the tree



The set of combinators $\{S, K\}$ is Turing-complete [1, 2], so that any partial recursive function can be implemented with them. Is there some set of combinators that corresponds exactly to the set of PR functions? Section 6 shows that the answer is yes, if an infinite set of combinators is used, namely `inc`i, `dec`i, `setZ`i, and `iter`i, for every $i \in \mathbb{N}$.

Also, from [6], it seems to follow that a finite set of *typed* combinators is enough to represent all PR functions, but a deeper understanding of those references is needed. In particular we may ask: how can n-ary PR (for every $n \in \mathbb{N}$) functions be represented?

References

[1] Haskell Curry and Robert Feys. *Combinatory Logic*, volume I. North Holland, Amsterdam, 1958.

[2] Haskell Curry, Roger Hindley, and Jonathan Seldin. *Combinatory Logic*, volume II. North Holland, Amsterdam, 1972.

[3] CWoo (user). More examples of primitive recursive functions, 2009. http://www.haskell.org/definition/.

[4] P. Hudak and J. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5), May 1992.

[5] Stephan Cole Kleene. *Introduction to Metamathematics*. North-Holland, 1952. Reprinted by Ishi press, 2009.

[6] Neel Krishnaswami. Combinators for the primitive recursive functions, January 2013. http://cstheory.stackexchange.com/questions/15038.

[7] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. *Proceedings of 22nd National Conference of the ACM*, pages 465–469, 1967.

[8] A. R. Meyer and D. M. Ritchie. Computational complexity and program structure. *IBM Research Report RC 1817*, 1967.

[9] Piergiorgio Odifreddi. *Classical Recursion Theory – The Theory of Functions and Sets of Natural Numbers*. Studies in Logic and the Foundations of Mathematics. Elsevier North Holland, first edition, 1989. Second impression.

[10] Simon Peyton Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[11] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press Cambridge, MA, 1987. Third printing (1992).

[12] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. Technical Report PRG-6, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, 1971.

[13] Dana S. Scott. Outline of a mathematical theory of computation. Technical Report PRG-2, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, 1970.

[14] Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach.* Addison-Wesley Longman Publishing Co., Boston, MA, USA, 1st edition, 1995.