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

Armando B. Matos (armandobcm@yahoo.com)

October 19, 2015

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 characterisation of PR functions.

If a language is total we do not need all the machinery developed by Scott, Stratchey, and others in order to characterise 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. This mathematical expression can be translated into a single-line functional language computation of the form $CR$ where $C$ 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 $\text{inc}$, $\text{dec}$, $\text{zero}$, and $\text{iter}$.

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 or new; it follows easily either from the Loop program representation or from the combinator representation. Although the transformations between the various representations of primitive recursive functions may be seen as “syntactic sugar”, we think that, in particular, the representation of a primitive recursive program (or definition) by a closed expression may be useful (i) to find properties of Loop programs, (ii) to represent a program in terms of its parts, (iii) to represent partial computations, namely when the information about the variables is partial, that is, when the initial value of the variable $x_i$ is the term $S^{n_i}(\bot)$, where $S$ denotes the successor, $n_i$ is a nonnegative integer, and $\bot$ represents an (yet) unknown integer. (These applications are not discussed in this note.)
Definitions of PR functions

<table>
<thead>
<tr>
<th>(Mathematics)</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Standard” definition</td>
<td>program</td>
</tr>
<tr>
<td>Loop program</td>
<td>args</td>
</tr>
<tr>
<td>(Programming)</td>
<td>output</td>
</tr>
<tr>
<td></td>
<td>term</td>
</tr>
<tr>
<td>(Mathematics)</td>
<td>(Programming)</td>
</tr>
</tbody>
</table>

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 program”. 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 $\pi^n_i(x_1, \ldots, x_n) = x_i$ 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.
\[ T : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \times \mathbb{N} \]

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

As instruction sequencing corresponds semantically to composition, we get

\[ \text{Computation of } F \iff T^{(n)}|_2 = \Pi_{0 \leq i < n} E(i) \]

where \( |_2 \) means “take the projection along the second argument \((t)\)”.

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 \texttt{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 whose programs may not halt. In particular:

- The codomain of the semantics function is a set of functions \( \mathbb{N}^n \rightarrow \mathbb{N}^n \.
  We don’t have to deal with integers, the “store”, integer expressions...

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

Note that, as all the programs are total, we do not need a part of the Stratchey
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 emphasise 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.

\[
((\lambda b. b + 1)^{(a)} \cdot (\lambda a. a + 1)^{(b)})^{(n)}
\]

\[
\downarrow
\]

\[
\text{iter3 ((iter1 inc2).(iter2 inc1))}
\]

For reference we rewrite the Haskell term

\[
? \text{iter3 ((iter1 inc2).(iter2 inc1))}
\]

Using the Haskell interpreter we get for instance

\[
? \text{iter3 ((iter1 inc2).(iter2 inc1)) (1,0,10)}
\]

\[
> (4181,6765,10)
\]

What is this particular primitive recursive transformation? That should be clear from the analysis of the answers to the following commands; note that the values of the function are 0, 1, 3, 8... (column marked with “\( \downarrow \)”)

<table>
<thead>
<tr>
<th>Command</th>
<th>Answer</th>
<th>( \downarrow )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 0)}</td>
<td>( 1, 0, 0)</td>
<td></td>
</tr>
<tr>
<td>\text{iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 1)}</td>
<td>( 1, 1, 1)</td>
<td></td>
</tr>
<tr>
<td>\text{iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 2)}</td>
<td>( 2, 3, 2)</td>
<td></td>
</tr>
<tr>
<td>\text{iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 3)}</td>
<td>( 5, 8, 3)</td>
<td></td>
</tr>
<tr>
<td>\text{iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 4)}</td>
<td>(13, 21, 4)</td>
<td></td>
</tr>
<tr>
<td>\text{iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 5)}</td>
<td>(34, 55, 5)</td>
<td></td>
</tr>
<tr>
<td>\text{iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 6)}</td>
<td>( 89, 144, 6)</td>
<td></td>
</tr>
<tr>
<td>\text{iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 7)}</td>
<td>(233, 377, 7)</td>
<td></td>
</tr>
<tr>
<td>\text{iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 8)}</td>
<td>(610, 987, 8)</td>
<td></td>
</tr>
<tr>
<td>\text{iter3 ((iter1 inc2).(iter2 inc1)) (1,0, 9)}</td>
<td>(1597, 2584, 9)</td>
<td></td>
</tr>
<tr>
<td>\text{iter3 ((iter1 inc2).(iter2 inc1)) (1,0,10)}</td>
<td>(4181,6765,10)</td>
<td></td>
</tr>
</tbody>
</table>

5.4 Standard definition

The standard definition of primitive recursive function is Definition 1, page 4. From [Meyer and Ritchie(1967a), Meyer and Ritchie(1967b)] it follows that the
Fixed part:

\[
\begin{align*}
\text{comp~} 0~ \_ & = \text{id} \\
\text{iter~}(n+1)~ tr & = tr.(\text{iter~}n~ tr) \\
\text{inc~} x & = x+1 \\
\text{dec~} 0 & = 0 \\
\text{dec~}(n+1) & = n \\
\text{zero~} \_ & = 0
\end{align*}
\]

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

\[
\begin{align*}
\text{inc1~}(x,y,z) & = (\text{inc~}x,y,z) \\
\text{inc2~}(x,y,z) & = (x,\text{inc~}y,z) \\
\text{inc3~}(x,y,z) & = (x,y,\text{inc~}z) \\
\text{dec1~}(x,y,z) & = (\text{dec~}x,y,z) \\
\text{dec2~}(x,y,z) & = (x,\text{dec~}y,z) \\
\text{dec3~}(x,y,z) & = (x,y,\text{dec~}z) \\
\text{zero1~}(x,y,z) & = (\text{zero~}x,y,z) \\
\text{zero2~}(x,y,z) & = (x,\text{zero~}y,z) \\
\text{zero3~}(x,y,z) & = (x,y,\text{zero~}z) \\
\text{iter1~}tr~(x,y,z) & = \text{iter~}x~tr~(x,y,z) \\
\text{iter2~}tr~(x,y,z) & = \text{iter~}y~tr~(x,y,z) \\
\text{iter3~}tr~(x,y,z) & = \text{iter~}z~tr~(x,y,z)
\end{align*}
\]

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

\[
\begin{align*}
> \text{iter3~}((\text{iter1~}\text{inc2}).(\text{iter2~}\text{inc1}))(1,0,10) & \quad -- \text{function args} \\
(4181,6765,10) & \quad -- \text{answer}
\end{align*}
\]

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.
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 [CWoo (user)(2009)]; 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, \( r \overset{\text{def}}{=} \max(x, y) \).

A Loop program that computes the maximum is

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) ( a \leftarrow 0; \text{for } x(\text{inc } a) );</td>
<td>copy of ( x \rightarrow a )</td>
</tr>
<tr>
<td>(2) ( \text{for } y(\text{dec } a) );</td>
<td>if ( x &gt; y: a &gt; 0 )</td>
</tr>
<tr>
<td>(3) ( r \leftarrow 0; \text{for } y(\text{incr}) );</td>
<td>( r = y ) by default...</td>
</tr>
<tr>
<td>(4) ( \text{for } a(r \leftarrow 0; \text{for } x(\text{incr})) );</td>
<td>if ( y &gt; y ): result is ( r = x )</td>
</tr>
</tbody>
</table>

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

\[
TP = (\lambda r.r + 1)^{(x)} \cdot (\lambda r.0)^{(a)} \cdot (\lambda r.r + 1)^{(y)} \cdot (\lambda r.0)^{(a)} \cdot (\lambda a.a - 1)^{(y)} \cdot (\lambda a.a + 1)^{(z)} \cdot (\lambda a.0)^{(a)}
\]

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: \( r \rightarrow x_0, x \rightarrow x_1, y \rightarrow x_2, a \rightarrow x_3 \).

\[
\text{iter3(iter1 inc0.setZ0)}
\text{.iter2 inc0.setZ0}
\text{.iter2 dec3}
\text{.iter1 inc3.setZ3}
\]

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

Experimenting in the Haskell interpreter:

\[\text{iter3(iter1 inc0.setZ0)}\]
The first integer of the result, 30, is the maximum.

It is not difficult to write the standard definition of \( \text{max}(x, y) \): 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

![Combinator Tree](image)

The set of combinators \( \{S, K\} \) is Turing-complete [Curry and Feys(1958)], [Curry et al.(1972)Curry, Hindley, and Seldin], 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 \( \text{inc} \ i, \text{dec} \ i, i \leftarrow 0 \), and \( \text{iter} \ i \), for every \( i \in \mathbb{N} \).

Also, from [Krishnaswami(2013)], 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?
**Definition 4** An $\exists$-rudimentary logic formula of $L$ is a formula of the form $\exists x : F$ where $F$ is rudimentary. Thus the formula contains only one unbounded quantifier, which is existential. Similarly for $\forall$-rudimentary logic formula.

Examples of languages of arithmetic: well formed arithmetical formulas over

- $L_{PB}$, Presburger arithmetic: $0, ', +$.
- $L_{Pe}$, the arithmetic language used in Peano axiomatic formal system: $0, ', +, \times$.
- $L_{exp}$, exp-arithmetic (or “exponential arithmetic” or “↑-arithmetic”): $0, ', +, \times, \uparrow$.

The operator “$\times$” is sometimes written “$\cdot$” or omitted.

**Definition 5** Term: variables, constants and, when function symbols are present, expressions of the form $f(t_1, \ldots, t_n)$ where $f$ is a function symbol with arity $n$ and $t_1, \ldots, t_n$ are terms.

**Definition 6** Atomic (logic) formula: formulas with the form

- $R(t_1, \ldots, t_n)$ where $F$ is a predicate symbol with arity $n$ and $t_1, \ldots, t_n$ are terms.
- When the identity is present: also $=(t_1, t_2)$ (also written $t_1 = t_2$) where $t_1$ and $t_2$ are terms.

**Definition 7** A (first-order) formula is built up from atomic formulas in a sequence of finitely many steps – called a formation sequence – by applying negation, conjunctions, and quantifications to simpler formulas.

**Example 1** A $\forall$-rudimentary formula of $L_{Pe}$:

$$\forall x \exists y_1 < x \exists y_2 < x \exists y_3 < x \exists y_4 < x : x = y_1 y_1 + y_2 y_2 + y_3 y_3 + y_4 y_4.$$ 

This formula expresses Lagrange’s theorem that says that every natural number is the sum of four squares.

**Example 2** Example of an $\forall$-rudimentary formula of $L_{Pe}$:

$$\forall x (1 < x \implies \exists y < 2 x : (x < y \land \neg (\exists u < y : \exists v < y : y = uv)))$$

This formula expresses Bertrand’s postulate, or Chebyshev’s theorem, that states that there is a prime between any number greater than one and its double.

