Models of Computation

Bengt Nordström,
Department of Computing Science,
Chalmers and University of Göteborg,
Göteborg, Sweden

January 17, 2014

Contents

1 The primitive recursive functions

1.1 Intuitive syntax and semantics . . . .

1.2 Some programming examples . . . .

1.2.1 The predecessor function in PRF . . . .

1.2.2 The factorial function. . . . .

1.3 A more precise syntax . . . .

1.4 Operational semantics . . . .

1.5 Denotational semantics . . . .

1.6 Termination of primitive recursive functions . . . .

2 Fast growing functions and big numbers

3 The set RF **of all partial recursive functions**

4 Historical remarks

1 The primitive recursive functions

1.1 Intuitive syntax and semantics

In informal mathematical notation we often define the addition function in the following way:

$$\begin{aligned} 0 + n &= n \\ m' + n &= (m + n)' \end{aligned}$$

We have used the notation m' for the successor of the number m. In a functional language we can use a similar definition:

$$\mathrm{add}\ 0\ n = n$$
$$\mathrm{add}\ (s\ m)\ n = s\ (\mathrm{add}\ m\ n)$$

We know that this is a meaningful definition since the addition function for the argument $s\ n$ is defined using the value of the function for the argument n. This kind of recursion is well defined since n is smaller than $s\ n$. The recursion scheme is called primitive recursion.

In the simple case that the function being defined has only one argument the scheme looks like:

$$f(0) = g$$
$$f(y + 1) = h(y, f(y))$$

where g is a natural number and h is a given primitive recursive function of two arguments. We notice that in order to define what a primitive recursive function of one argument is, we have to know what a primitive recursive function of two arguments is. We therefore have to generalize and define what a primitive function of $n + 1$ arguments (for all n) is:

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$$
$$f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(y, x_1, \ldots, x_n))$$

where the functions g and h are given primitive recursive functions (of n and $n+2$ arguments, respectively.

This class of functions is an early example of a *computation model*, a mathematical model of a computing device (a programming language or a computer). We will give the model by giving a precise description of the syntax and semantics of the primitive recursive functions.

Let PRF_n express the set of all primitive recursive functions of arity n (i.e. with n arguments). We assume that $n \in \mathrm{N}$, i.e. we allow the number of arguments to be 0. The intutition is that each program $p \in \mathrm{PRF}_n$ denotes a primitive recursive function f in the set $\mathrm{N}^n \to \mathrm{N}$. We will construct the class by using the five simple program forming operations showed in figure 1. The simple programs

$$\begin{aligned}
z &\in \mathrm{PRF}_0 \\
s &\in \mathrm{PRF}_1 \\
p^n_i &\in \mathrm{PRF}_{n+1} \ \text{ if } i \leq n \\
g\{f_1, \ldots, f_m\} &\in \mathrm{PRF}_n \ \text{ if } g \in \mathrm{PRF}_m, f_i \in \mathrm{PRF}_n, 1 \leq i \leq m \\
\mathrm{rec}(g, h) &\in \mathrm{PRF}_{n+1} \ \text{ if } \ g \in \mathrm{PRF}_n, h \in \mathrm{PRF}_{n+2}
\end{aligned}$$

Figure 1: Informal syntax

are z, s och p^n_i and the composite programs are $f\{g_1, \ldots g_n\}$ and rec(f, g), where $f, g, g_1, \ldots g_n$ are programs.

- The program z (which takes no argument) computes always to 0.
- The program s stands for the successor function, it is computed by adding 1 to its only argument.
- The program p^n_i takes n arguments and computes to its i-th argument.
- The program $g\{f_1, \ldots, f_n\}$ is a generalization of the usual functional composition $g \circ f$.
- Finally, the program rec(g, h) expresses the scheme for primitive recursion.

The intuitive semantics is shown in figure 7. We are using the notation $\bar{x}$ for the n arguments $x_1, \ldots, x_n$. We notice that the semantics is given by telling what value the programs will output when we apply them to their arguments (which in this case always is a list of natural numbers).

Notice that there are no variables and no function application in this model. Instead, projections and compositions are used. It takes some time to get used to write programs without variables, we will show some examples in the following section. It can be skipped in the first reading of this chapter.

$$
\begin{aligned}
z[] &= 0 \\
s[x] &= x+1 \\
p_i^n[x_1,\dots,x_n] &= x_i \\
g\{f_1,\dots,f_m\}[\bar{x}] &= g[f_1[\bar{x}],\dots,f_m[\bar{x}]] \\
\mathrm{rec}(g,h)[\bar{x},0] &= g[\bar{x}] \\
\mathrm{rec}(g,h)[\bar{x},y+1] &= h[\bar{x},y,\mathrm{rec}(g,h)[y,\bar{x}]]
\end{aligned}
$$

Figure 2: Informal semantics

1.2 Some programming examples

1.2.1 The predecessor function in PRF

The predecessor function pred is defined by the following equations:

$$
\begin{aligned}
\mathrm{pred}[0] &= 0 \\
\mathrm{pred}[n+1] &= n
\end{aligned}
$$

How can we express this in PRF? It seems natural to guess that the general shape of pred is

$$\mathrm{pred} =_{\mathrm{def}} \mathrm{rec}(g,h)$$

where g and h are placeholders for unknown programs. We know that the arity of pred is 1, this means that g must have arity 0 and h arity 2. From the first equation for pred it follows that we can define

$$g =_{\mathrm{def}} \mathrm{z}$$

From the second clause it follows that $\mathrm{pred}[n+1] = h[n, p(n)]$, which is equal to n if use a projection function to pick the first argument:

$$h =_{\mathrm{def}} p_1^2$$

Hence we can define

$$\mathrm{pred} =_{\mathrm{def}} \mathrm{rec}(\mathrm{z}, p_1^2)$$

1.2.2 The factorial function.

The factorial function is the function $\mathrm{fac}[n] = 1 * 2 * \cdots n$, or if we put it on primitive recursive form:

$$\begin{aligned}\mathrm{fac}[0] &= 1\\ \mathrm{fac}[n+1] &= (n+1) * \mathrm{fac}[n]\end{aligned}$$

This uses the primitive recursion scheme, so we can try with

$$\mathrm{fac} =_{\mathrm{def}} \mathrm{rec}(g, h)$$

where $g \in \mathrm{PRF}_0$ and $h \in \mathrm{PRF}_2$. We must have that $g[] = 1$, which is satisfied if

$$g =_{\mathrm{def}} \mathrm{s}\{z\}.$$

We know that the following holds for the program h:

$$\begin{aligned}\mathrm{fac}[n+1] &= \mathrm{rec}(g, h)[n+1]\\ &= h[n, \mathrm{fac}[n]]\\ &= (n+1) * \mathrm{fac}[n]\end{aligned}$$

Hence, we want to construct a program h in PRF_2 such that

$$h[n, \mathrm{fac}[n]] = (n+1) * \mathrm{fac}[n]$$

This is fulfilled if h satisfies $h[n, m] = \mathrm{mul}[n+1, m]$, where mul is a program for multiplication (this can also be expressed in PRF).

Let us try to define h as a composition

$$h = \mathrm{mul}\{e_1, e_2\}$$

for some (yet unknown) programs e_1 and e_2. We know that the following must hold:

$$\begin{aligned}\mathrm{mul}\{e_1, e_2\}[n, m] &= \mathrm{mul}[e_1[n, m], e_2[n, m]]\\ &= \mathrm{mul}[n+1, m].\end{aligned}$$

This holds if

$$e_1[n, m] = n + 1$$

$$e_2[n, m] = m.$$

which is satisfied if e_2 is a projection

$$e_2 =_{\text{def}} p^2_2$$

and e_1 is a composition:

$$e_1 =_{\text{def}} s\{p^2_1\}$$

since $s\{p^2_1\}[n, m] = s[p^2_1[n, m]] = n + 1$

To conclude, we can define the factorial function by

$$\begin{aligned} \text{fac} =_{\text{def}} \text{rec}(&s\{z\},\\ &\text{mul}\{s\{p^2_1\}, p^2_2\}) \end{aligned}$$

A more experienced person can write this immediately from the defining equations for the function. It is even possible to write a compiler which translates the defining equations to a program in PRF.

1.3 A more precise syntax

The syntax and semantics which was given before were not very precise. We have to remove the three dots which we have in the description of the syntax and semantics. It is always a sign of imprecision to have the dots, different people interpret them in different ways.

Let us first define the set A^m of vectors of length m by the following inductive definition:

$$\text{nil} \in A^0$$

$$as.a \in A^{n+1} \;\text{ if }\; a \in A \text{ and } \; as \in A^n$$

Notice that we are using so called snoc-lists, the list $a_1, a_2, \dots, a_n$ is analyzed as $(a_1, a_2, \dots), a_n$.

Now, we can give a more precise definition (in figure 3) of the abstract syntax of PRF$_n$.

$$\begin{aligned}
z &\in \mathrm{PRF}_0 \\
s &\in \mathrm{PRF}_1 \\
\mathrm{proj}(n,i) &\in \mathrm{PRF}_n \;\text{ if }\; 1 \le i \le n \\
\mathrm{comp}(g,\bar{f}) &\in \mathrm{PRF}_n \;\text{ if }\; g \in \mathrm{PRF}_m, \bar{f} \in (\mathrm{PRF}_n)^m \\
\mathrm{rec}(g,h) &\in \mathrm{PRF}_{n+1} \;\text{ if }\; g \in \mathrm{PRF}_n, h \in \mathrm{PRF}_{n+2}
\end{aligned}$$

Figure 3: Abstract syntax of PRF

1.4 Operational semantics

We will give an inductive definition of the computation relation $p \longrightarrow q$, the program p computes to the value q. We have to decide what kind of things are computed and what a value is. When we gave the intuitive semantics of what a program is we expressed this by saying what it means to apply a program to its input. So, the thing which is computed is an object in PRF$_n$ together with its input. What is then a value? An obvious choice is that we let the values be a natural number. Hence, in the computation relation $p \longrightarrow q$, p is always a program together with its input and q is always a natural number.

Let us define the set PRFI, of programs together with its input, by the following inductive definition (with only one clause):

$$p[\bar{y}] \in \mathrm{PRFI} \;\text{ if }\; p \in \mathrm{PRF}_n \text{ and } \bar{y} \in \mathrm{N}^n$$

We can also express it in the following way:

$$\frac{p \in \mathrm{PRF}_n \qquad \bar{y} \in \mathrm{N}^n}{p[\bar{y}] \in \mathrm{PRFI}}$$

We will interpret the definition as that the set PRF has one binary constructor **.**[**.**] whose arguments are a primitive recursive program and a list of numbers.

Now, we can give the operational semantics. We will define the computation relation $p \longrightarrow q$ over the sets PRF and N as an inductive definition in figure 8. In order to explain the semantics, it is necessary to define another computation relation $\mathrm{ps} \Longrightarrow \mathrm{ns}$, which expresses that a vector ps of primitive recursive functions applied to a common input-vector is computed to a vector of natural numbers. This is done in the obvious way, each primitive recursive function in the list is applied to the same input list. The function $\mathrm{th}(n, i, \bar{y})$ is equal to the i:th element

$$\frac{}{z[] \longrightarrow 0} \qquad \frac{}{s[\mathrm{nil}.n] \longrightarrow n+1}$$

$$\frac{\mathrm{th}(n, i, \bar{y}) = v}{\mathrm{proj}(n, i)[\bar{y}] \longrightarrow v} \qquad \frac{\bar{f}[\bar{y}] \Longrightarrow \bar{n} \qquad g[\bar{n}] \longrightarrow v}{\mathrm{comp}(g, \bar{f})[\bar{y}] \longrightarrow v}$$

$$\frac{g[\bar{y}] \longrightarrow v}{\mathrm{rec}(g, h)[\bar{y}.0] \longrightarrow v} \qquad \frac{\mathrm{rec}(g, h)[\bar{y}.n] \longrightarrow i \qquad h[\bar{y}.i.n] \longrightarrow v}{\mathrm{rec}(g, h)[\bar{y}.(n+1)] \longrightarrow v}$$

where the relation $\Longrightarrow$ is defined by

$$\frac{}{\mathrm{nil}[\bar{y}] \Longrightarrow \mathrm{nil}} \qquad \frac{f[\bar{y}] \longrightarrow k \qquad \bar{f}[\bar{y}] \Longrightarrow ks}{(\bar{f}.f)[\bar{y}] \Longrightarrow ks.k}$$

Figure 4: Operational semantics

of $\bar{y}$, if $\bar{y} \in A^n$ and is defined for $1 \leq i \leq n$.

1.5 Denotational semantics

As an alternative way of defining the semantics for a computation model, we can give the denotational semantics of the programs in it. This is a function which maps an arbitrary program to its "meaning", a mathematical object. The idea is that you understand a program when you understand what mathematical object it denotes.

In this case, we will give a function

$$[\![p]\!] \in N^n \to N \quad \text{if} \quad p \in \mathrm{PRF}_n$$

by structural recursion over the abstract syntax of p. This is done in figure 5.

$$\begin{aligned}
[\![z]\!](\mathrm{nil}) &= 0\\
[\![s]\!](\mathrm{nil}.j) &= j + 1\\
[\![p^n_i]\!](\bar{y}) &= \mathrm{th}(n, i, \bar{y})\\
[\![g\{\bar{f}\}]\!](\bar{y}) &= [\![g]\!]([\![\bar{f}]\!]^*(\bar{y}))\\
[\![\mathrm{rec}(g, h)]\!](\bar{y}.0) &= [\![g]\!](\bar{y})\\
[\![\mathrm{rec}(g, h)]\!](\bar{y}.(y + 1)) &= [\![h]\!](\bar{y}.y.([\![\mathrm{rec}(g, h)]\!](y.\bar{y})))
\end{aligned}$$

where the semantical function $[\![\bar{f}]\!]^* \in (\mathrm{PRF}_n)^m \to N^m$ is defined by

$$\begin{aligned}
[\![\mathrm{nil}]\!]^*(t) &= \mathrm{nil}\\
[\![\bar{f}.f]\!]^*(t) &= ([\![\bar{f}]\!]^*(t)).([\![f]\!](t))
\end{aligned}$$

Figure 5: Denotational semantics

1.6 Termination of primitive recursive functions

Now, when we have a precise descripton of the set PRF we can formulate and prove that all programs in PRF terminate.

We want to show that all programs terminate for all their inputs:

$$\forall i \in N.\forall p \in \mathrm{PRF}_i.\mathrm{Term}(p)$$

where the predicate Term is defined by

$$\mathrm{Term}(p) \equiv \forall \bar{y} \in N^i.\exists m \in N.p[\bar{y}] \longrightarrow m$$

The proof is by induction over the abstract syntax, we get one case for each clause in the inductive defintion of the set PRF_i:

- We want to prove that Term(z). But the program $z[]$ always terminates according to the first clause in the operational semantics.
- We want to prove that Term(s). This is true according to the second clause in the operational semantics.
- We want to prove Term(proj(n, i)), for $n \in N, i \leq n$. This follows from the third clause.
- We want to prove Term(comp$(g, \bar{f})$), if g terminates and all programs in $\bar{f}$ terminates. We compute the program comp$(g, \bar{f})[\bar{y}]$ by first computing the program $\bar{f}[\bar{y}]$. According to the induction hypothesis this terminates with a value $\bar{n}$, $\bar{n} \in N^m$. Finally, we compute the program $g[\bar{n}]$, which also terminates (according to the induction assumption).
- We want to prove Term(rec(g, h)) if g terminates and h terminates. We know that rec$(g, h) \in$ PRF$_{n+1}$, $g \in$ PRF$_n$ and $h \in$ PRF$_{n+2}$. We want to show that rec$(g, h)[\bar{y}.m]$ always terminates (we can ignore the case when the input list is empty since rec$(g, h) \in$ PRF$_{n+1}$.) We will show this by induction over the natural number m.

  We have two cases:

  - The base case, when $m = 0$. The program rec$(g, h)[\bar{y}.0]$ terminates with the value of $g[\bar{y}]$, according to one of the clauses in the operational semantics.
  - The induction step. Suppose that rec$(g, h)[\bar{y}.m]$ terminates with the value i. The program rec$(g, h)[\bar{y}.(m + 1)]$ then terminates with the value of the program $h(\bar{y}.m.i)$. This value always exists, since h is a program which always terminates (according to the induction assumption).

2 Fast growing functions and big numbers

Since all primitive recursive functions terminate we can use a diagonalization argument to show that there exists a computable total function which is not primitive recursive[1]. There is, however, a more concrete example.

We get multiplication by iterating the addition function:

$$m * n = \underbrace{m + \cdots + m}_{\text{n terms}}$$

We use the following primitive recursive definition of multiplication:

$$\begin{aligned} \text{mul}(m, 0) &= 0 \\ \text{mul}(m, s(n)) &= \text{add}(m, \text{mul}(m, n)) \end{aligned}$$

Similarly, we get exponentiation by iterating multiplication:

$$m \uparrow n = \underbrace{m * \cdots * m}_{\text{n terms}}$$

which corresponds to the following primitive recursive definition:

$$\begin{aligned} \uparrow(m, 0) &= 1 \\ \uparrow(m, s(n)) &= \text{mul}(m, \uparrow(m, n)) \end{aligned}$$

We can continue this process, we get the tower-operation by iterating exponentiation:

$$m \uparrow\uparrow n = \underbrace{m \uparrow \cdots \uparrow m}_{\text{n terms}}$$

which corresponds to the following primitive recursive definition:

$$\begin{aligned} \uparrow\uparrow(m, 0) &= 1 \\ \uparrow\uparrow(m, s(n)) &= \uparrow(m, \uparrow\uparrow(m, n)) \end{aligned}$$

We continue to define the $\uparrow\uparrow\uparrow$-operation by iterating the $\uparrow\uparrow$ operation.

[1]Exercise: Explain this in detail!

Notice that these operations are increasing very fast, for instance already the number $3 \uparrow\uparrow 3$ is around one million times the number of Swedish inhabitants:

$$\begin{aligned} 3 \uparrow\uparrow 3 &= 3 \uparrow 3 \uparrow 3 \\ &= 3 \uparrow 27 \\ &= 7\,625\,597\,484\,987 \end{aligned}$$

The number denoted by $3 \uparrow\uparrow\uparrow 3$ becomes difficult to grasp:

$$\begin{aligned} 3 \uparrow\uparrow\uparrow 3 &= 3 \uparrow\uparrow 3 \uparrow\uparrow 3 \\ &= 3 \uparrow\uparrow 7625597484987 \\ &= 3 \uparrow 3 \cdots 3 \uparrow 3 \quad \text{(with } 7\,625\,597\,484\,987 \text{ terms)} \\ &= 3^{3^{3^{3^{3^{3^{3^{\cdots}}}}}}} \qquad \text{(with } 7\,625\,597\,484\,987 \text{ exponentiations)} \end{aligned}$$

Notice that already $3^{3^{3^3}}$ is much bigger than the number of atoms in the universe (since $3^{3^{3^3}} = 3^{7\,625\,597\,484\,987} > 10^{10^{12}} = 10^{1\,000\,000\,000\,000} \ggg 10^{70}$).

Now, if $3 \uparrow\uparrow\uparrow 3$ is difficult to grasp, what about $3 \uparrow\uparrow\uparrow\uparrow 3$?

$$\begin{aligned} 3 \uparrow\uparrow\uparrow\uparrow 3 &= 3 \uparrow\uparrow\uparrow (3 \uparrow\uparrow\uparrow 3) \\ &= 3 \uparrow\uparrow \cdots \uparrow\uparrow 3 \quad \text{with } 3^{3^{3^{3^{3^{3^{3^{\cdots}}}}}}} \quad \text{terms} \end{aligned}$$

where we have $7\,625\,597\,484\,987$ exponentiations in the number of terms.

So if we had 4 (instead of $7\,625\,597\,484\,987$) number of exponentiations in the number of terms we could not even write down the expression in the form $3 \uparrow\uparrow \cdots \uparrow\uparrow 3$ if we used one term for each atom in the universe. And we know that already $3 \uparrow\uparrow 3 \uparrow\uparrow 3$ (which is $3 \uparrow\uparrow\uparrow 3$) was enormous! But – as we all know– most numbers are much bigger than $3 \uparrow\uparrow\uparrow\uparrow 3$.

We can continue this process and define a whole series of operations $\uparrow, \uparrow\uparrow, \uparrow\uparrow\uparrow$ $, \uparrow\uparrow\uparrow\uparrow, \ldots$. We can now introduce an arbitrary number of operations, one for each natural number n. We let for instance $\uparrow^5$ stand for $\uparrow\uparrow\uparrow\uparrow\uparrow$, so for each k we use the notation $\uparrow^k$ for the operation with k arrows. Notice that $\uparrow^k$ is *not* a function applied to the argument k ! It is only a schematic notation for k repetitions of the symbol $\uparrow$.

We have that

$$m \uparrow^{k+1} n = \underbrace{m \uparrow^{k} \cdots \uparrow^{k} m}_{\text{n terms}}$$

It is clear that all these operations are primitive recursive functions. If we have a definition of the $\uparrow^{k}$-operation then we can express the $\uparrow^{k+1}$-operation by primitive recursion:

$$\begin{aligned}\uparrow^{k+1}(m, 0) &= 1\\ \uparrow^{k+1}(m, s(n)) &= \uparrow^{k}(m, \uparrow^{k+1}(m, n))\end{aligned}$$

Notice here that we have a number of operations $\uparrow^1, \uparrow^2, \uparrow^3, \uparrow^4, \uparrow^5, \uparrow^6, \ldots$ which all have a uniform definition. One operation is defined from the previous operation by using primitive recursion. What happens if we try to look at k as an argument to the k-th operation? So we will try to look at $\uparrow^k$ as a function $\uparrow$ applied to the number k yielding the k-th operation. Let us consider the ternary function $\uparrow$ which is defined such that $\uparrow(k, m, n)$ is equal to the value of $\uparrow^k(m, n)$, for each k, m and n:

$$\begin{aligned}\uparrow(0, m, n) &= \text{mul}(m, n)\\ \uparrow(k+1, m, 0) &= 1\\ \uparrow(k+1, m, s(n)) &= \uparrow(k, m, \uparrow(k+1, m, n))\end{aligned}$$

Now something happens. This function is not primitive recursive! A version of this function is called Ackermann's function after the person who defined it around 70 years ago. It is possible to show that the ternary $\uparrow$-function grows faster than any primitive recursive function. On the other hand it is clear that the function is computable: If we want to compute $\uparrow(k, m, n)$ we first compute the value of the argument k and then construct the operation $\uparrow^k$. This construction process is computable, we can use the method above. Then we just compute $\uparrow^k(m, n)$, which is primitive recursive and hence computable.

3 The set RF of all partial recursive functions

If we want to extend PRF to the class of all recursive functions we extend it with an operator min which expresses linear search. We define a new class of functions RF_n, the set of all recursive functions of arity n by extending the inductive definition of the abstract syntax of PRF with one clause:

$$\begin{aligned}
z &\in \mathrm{RF}_0 \\
s &\in \mathrm{RF}_1 \\
\mathrm{proj}(n, i) &\in \mathrm{RF}_n \ \text{ if } \ 1 \le i \le n \\
\mathrm{comp}(g, \bar{f}) &\in \mathrm{RF}_n \ \text{ if } \ g \in \mathrm{RF}_m, \bar{f} \in (\mathrm{RF}_n)^m \\
\mathrm{rec}(g, h) &\in \mathrm{RF}_{n+1} \ \text{ if } \ g \in \mathrm{RF}_n, h \in \mathrm{RF}_{n+2} \\
\mathrm{min}(f) &\in \mathrm{RF}_n \ \text{ if } \ f \in \mathrm{RF}_{n+1}
\end{aligned}$$

Figure 6: Abstract syntax of RF

The informal semantics of the min-function is that $\mathrm{min}(f)$ computes the minimal number k for which $f[\bar{x}.k] = 0$.

$$\mathrm{min}(f)[\bar{x}] = min\{k \in \mathrm{N} \mid f[\bar{x}.k] = 0\}$$

Figure 7: Informal semantics

Intuitively, the function application $\mathrm{min}(f)[\bar{x}]$ is computed by linear search. We first compute $f[\bar{x}.0]$. If the result is 0, the value of the function application is 0. Otherwise, we continue to compute $f[\bar{x}.1]$. If this result is 0 we return 1. If it is nonzero, we continue to increase the last argument until we reach a function value which is 0. This computation does not have to terminate, since there is not necessarily a value of the last argument for which the function is 0. Another cause for nontermination is that the computation of f applied to some value does not terminate. So, the informal definition is not completely correct. We can be sure that $\mathrm{min}(f)[\bar{x}$ computes the least k for which $f[\bar{x}.k] = 0$ only in the case there is such a k and that f terminates for all arguments less than k. This can be achieved if we for instance require that f is primitive recursive. But this is not the approach we take here. Instead, we will define the computation using a linear search, or more precisely by adding two clauses to the operational semantics of PRF:

$$\frac{f[\bar{y}.0] \longrightarrow 0}{\mathrm{min}(f)[\bar{y}] \longrightarrow 0} \qquad \frac{\mathrm{min}(\mathrm{shift} f)[\bar{y}] \longrightarrow i}{\mathrm{min}(f)[\bar{y}] \longrightarrow i+1}$$

Figure 8: Additional rules for the operational semantics of RF

In the rules above, the function $\mathrm{shift}(f)$ is defined by

$$(\mathrm{shift}\ f)[\bar{y}.a] = f[\bar{y}.(a+1)]$$

It is clear that min is computable:
In a functional language, we could define the min-function by:

```
min f ys  =  g f 0
   where g f i = if (f ys.i) = 0 then i else g f (i+1)
```

and in an imperative language:

```
min f ys  =
   i = -1;
    repeat i = i+1 until f ys.i = 0
   return i
```

4 Historical remarks

The first to write the ordinary primitive recursive definitions of addition and multiplication was probably Hermann Grassmann [3] . It was later rediscovered by Dedekind [2]. The class of primitive recursive functions were known by Hilbert [4] in 1926. At that time his student Wilhelm Ackermann had defined the ternary

$\uparrow$-function and showed that it is not primitive recursive. This result was not published until 1928 [1].

The founder of the theory of primitive recursive functions was Rózsa Péter [6], who also coined the term “primitive recursive”. She simplified Ackermann’s formulation (together with Raphael Robinson) to a function of two arguments:

$$\begin{aligned} A(0, y) &= y + 1 \\ A(x + 1, 0) &= A(x, 1) \\ A(x + 1, y + 1) &= A(x, A(x + 1, y)) \end{aligned}$$

which is now the traditional formulation in modern textbooks. This is formally simpler than the original, but the connection with the arithmetical operations is lost.

The notation $\uparrow$ originates from Knuth in 1976 [5].

References

[1] W. Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematical Annals*, 99:118–133, 1928.

[2] Richard Dedekind. *Was sind und was sollen die Zahlen?* F. Vieweg, Braunschweig, 1888. Translated by W.W. Beman and W. Ewald in Ewald (1996): 787–832.

[3] Hermann Grassmann. *Lehrbuch der Mathematik für höhere Lehranstalten*. Enslin, 1861.

[4] David Hilbert. ‘Über das unendliche’. *Mathematische Annalen*, 95:161–90, 1926. Translated by Stefan Bauer-Mengelberg and Dagfinn Føllesdal in van Heijenoort (1967): 367–92.

[5] D.E. Knuth. *Selected Papers in Computer Science*, chapter Mathematics and computer science: coping with finiteness. Cambridge University Press, 1996. also published in Science 194, 1235–1242.

[6] Rozsa Peter. *Recursive Functions.* Academic Press, 1967.

[7] Jean van Heijenoort. *From Frege to Gödel: A source book in mathematical logic 1879–1931.* Harvard University Press, Cambridge MA, 1967.