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

October 26, 2008

1 Are all functions computable?

It is easy to confuse functions in mathematics and programs in computing science. They seem to be very similar objects. If we have a (mathematical) function f from $\mathbf{N}$ to $\mathbf{N}$ then we can write $f(i)$ which expresses the result of applying f to the natural number i. In programming we also have something we call "functions" which produces a result when we apply it to an input of the right type. An immediate observation is that many "functions" in programming languages are not deterministic in the sense that equal input yields equal output. There are many kind of strange side-effects which can cause this behavior. But even if we only consider so called pure functions there is a clear distinction.

It is clear that we cannot assume that these concepts are identical just because we use the same name for them. When we in mathematics say that we have a function $f \in \mathbf{N} \to \mathbf{N}$ then we mean that f is a subset of $\mathbf{N} \times \mathbf{N}$ such that if $(a, b) \in f$ and $(a, c) \in f$ then $b = c$. [1] A program on the other hand is something which when given an input can be executed on a computer to yield a result. There is no reason to believe that this concept should coincide with the function concept in mathematics.

1.1 Functions are different from programs: the diagonalization argument

There is a simple argument showing that the notions of program and function do not coincide. We just count how many programs taking a natural number to a natural number there are and compare it with how many elements there are in the set $\mathbf{N} \to \mathbf{N}$. We know that these sets are infinite, so we must use the standard

[1] We can also use the definition that f is a binary relation such that if $a\ f\ b$ and $a\ f\ c$ then $b = c$.

mathematical method to compare the cardinality of two sets: The set A has at least as high cardinality as the set B if there is a surjective [2] function from A to B. We will usually compare the cardinality of a set with the cardinality of $\mathbf{N}$. We will say that a set is *countable* if it does not have more elements than the set $\mathbf{N}$:

Definition 1 (countable) *The set A is countable if there is a surjective function $f \in \mathbf{N} \to A$.*

The following two facts says that there are more functions then programs:

Fact 1 *The set of programs is countable.*

This is an immediate fact for anyone who has written a program. Programs are stored in the computer as a text string, each character having its ascii code. We can look at this string as a binary number. So each program can be seen as a number, hence the set of all programs can be seen as a subset of all natural numbers. Hence it must be countable.

The other way of being convinced of this is to use the mathematical fact that the set of all strings is countable if the alphabet is finite.

Fact 2 *The set $\mathbf{N} \to \mathbf{N}$ is not countable*

If it were, we could find an enumeration $f_1, f_2, \ldots$ of all its elements. But then the function defined by

$$\mathbf{diag}(i) = f_i(i) + 1$$

cannot be in the set, since if it were, we must have that $\mathbf{diag} = f_j$, for some number j. But this is impossible, since $\mathbf{diag}(j) \neq f_j(j)$.

So, a simple diagonalization argument shows that there are more functions in $\mathbf{N} \to \mathbf{N}$ then programs taking a natural number to a natural number. We will call a function in $\mathbf{N} \to \mathbf{N}$ $\mathcal{L}$-computable if there is a program in the programming language $\mathcal{L}$ computing it:

[2] That a function $f \in A \to B$ is *surjective* means that each element in B is the image of some element in A.

Definition 2 ($\mathcal{L}$-computable) *A function* $\mathtt{f} \in \mathbf{N} \to \mathbf{N}$ *is* $\mathcal{L}$-computable *if there is a program* $\mathtt{P}$ *in* $\mathcal{L}$ *taking a natural number to a natural number such that when* $\mathtt{P}$ *is applied to* $\mathtt{n}$ *the result is* $\mathtt{f}(\mathtt{n})$

This definition needs refinement [3] but this will be postponed to later when we will be more precise. We have here choosen to work with the set **N**, but we will also use **Bool**, the set of boolean values.

1.2 A nontrivial constant function

It is an open mathematical problem whether Goldbach's conjecture is true.

Conjecture 1 (Goldbach) *Every even number greater than 2 can be written as a sum of two prime numbers*

We can then ask ourselves if the constant function $\mathtt{g} \in \mathbf{N} \to \mathbf{N}$ defined by

$$\mathtt{g}\ \mathtt{n} = \begin{cases} 1 & \text{if Goldbach's conjecture is true,} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

is computable? It seems that it is not possible to write a program behaving like $\mathtt{g}$ without knowing that the conjecture holds. If we reason like a classical mathematician we conclude that $\mathtt{g}$ is indeed computable. If Goldbach's conjecture is true we choose a program which always outputs 1 and if it is false we choose a program which always outputs 0. So the function is computable but it is not possible to write a program for it. For a constructive mathematician this reasoning does not hold and $\mathtt{g}$ is not computable (until we prove the conjecture).

[3] A program in the language $\mathcal{L}$ does not take natural numbers as input and output, instead there is some way of representing them.

1.3 A program which cannot be written

Suppose that a person comes up to you and says that he has written a program **halts** taking a boolean value to a boolean value such that

$$\textbf{halts}\ x = \begin{cases} \textbf{true} & \text{if the computation of } x \text{ terminates,} \\ \textbf{false} & \text{otherwise} \end{cases}$$

Do you believe him? You just write down this program:

$$T\ x = \ \text{if}\ \ \textbf{halts}\ x\ \ \text{then}\ \ \textbf{loop}\ \ \text{else}\ \ \textbf{true}$$

The program **loop** is a program which never terminates. We now see that the program T reverses the termination behaviour of its argument, i.e. that the program $T\ a$ terminates if and only if a does not terminate. And this holds for all boolean values a! In particular for the boolean value s defined recursively by

$$s = T\ s$$

Does s terminate? From the definition of s it terminates if $T\ s$ terminates. But this terminates if the argument s does not terminate. So s terminates if s does not terminate.

So s does not terminate? But if this is so, then $T\ s$ does not terminate (from the definition of s). And $T\ s$ does not terminate if s terminates. So s does not terminate if s terminates.

We have a contradiction and we can conclude that the program **halts** does not exist.

1.4 It is impossible to have a model for computation which captures all terminating programs

We have noticed that there are more functions in the set $\mathbf{N} \to \mathbf{N}$ then programs from natural numbers to natural numbers. A natural idea is then to try to restrict our interest to the subset of $\mathbf{N} \to \mathbf{N}$ which only contains the computable functions. But this leads to problems.

Let us identify the set of strings (denoting programs) and the set **N** as we did earlier. It seems obvious that all reasonable models of computations should allow us to to write a program P (for parser) which when given a string as input can decide whether the string is a syntactically correct program. This is obvious since we want it to be mechanically decidable whether a program is correctly formed or not. We should not require any intelligence to see whether a program is wellformed. But if we have such a program, then we can write a program G (for generate) which for an input i outputs the ith program p_i. We generate first all natural numbers $0, 1, \ldots$ and (looking at a number as the ascii code of a string) remove (using P) all syntactically erroneous strings. We can then output the last program after having constructed $i + 1$ programs.

So we will have a way to mechanically enumerate *all* programs in the model, we first execute the program G with input 1 obtaining a string representing the program p_1, then with input 2 obtaining a string representing the program p_2, and so on. We will use the notation f_i for the function which the program p_i computes. But now we can use the diagonalization argument to construct a computable function in $\mathbf{N} \to \mathbf{N}$ which is not in the enumeration. The following function

$$d(i) = f_i(i) + 1$$

is definitely computable. To compute the function for the value i we just generate the ith program (using G), give i as input to that program and add 1. All these operations should be computable. Hence, d is a computable function and it is not in the enumeration $\{f_i\}$ of all computable functions.

We can escape this contradiction if we take a subset of all *partial* functions from **N** to **N**. Then in the reasoning above each function f_i can be partial and the definition of the function d does not necessarily give a function which is different from all functions in the enumeration (if the ith function is undefined for the argument i, then d becomes undefined for that argument).

The computational intuition behind an undefined function value is that the corresponding program does not terminate for that value.

1.5 The halting problem

We can try to recreate the contradiction above in the following way. Consider the function $\mathtt{dd}$ defined by

$$\mathtt{dd}(\mathtt{i}) = \begin{cases} \text{undefined} & \text{if } \mathtt{f_i(i)} \text{ is defined,} \\ 0 & \text{otherwise} \end{cases}$$

It is clear that the function $\mathtt{dd}$ is not in the enumeration $\{\mathtt{f_i}\}$, but is it computable? It would be if we could solve the halting problem, namely construct a program $\mathtt{H}$ which when given two [4] inputs $(\mathtt{i}, \mathtt{j})$ outputs 1 if the program $\mathtt{p_i}$ terminates for the input $\mathtt{j}$ and otherwise outputs 0.

If we use the notation $\langle \mathtt{P}, \mathtt{i} \rangle$ for the result of giving the input $\mathtt{i}$ to the program $\mathtt{P}$, then we could define the program $\mathtt{DD}$ which computes $\mathtt{dd}$ by

$$\langle \mathtt{DD}, \mathtt{i} \rangle = \text{ if } \langle \mathtt{H}, (\mathtt{i}, \mathtt{i}) \rangle \text{ then } \textbf{loop} \text{ else } 1$$

We are a bit informal here, but it seems clear that to test whether a number is equal to 0 should be computable. We use the notation **loop** for a program which never terminates.

In summary, if the halting problem is computable, then we are able to recreate the contradiction. The only conclusion is that that the halting problem is not computable. Notice the similarity between this argument and the previous example of a function which cannot be written.

1.6 Is it always possible to write a self-interpreter for a programming language?

A self-interpreter for a language $\mathcal{L}$ is a program e which when given a program (string) $\mathtt{p}$ and its input $\mathtt{n}$ computes the result of executing $\mathtt{p}$ on $\mathtt{n}$, i.e.

$$\langle e, (\mathtt{p}, \mathtt{n}) \rangle = \langle \mathtt{p}, \mathtt{n} \rangle$$

[4] We can represent two inputs by concatenation with a new symbol in between the inputs.

We will here show that it is impossible to write a self-interpreter for a language in which all programs terminate.

Suppose that it is possible to write the self-interpreter e. Then we can define the program d in the following way:

For a given input n apply first the program G on it. The result is p_n, the n'th program in the enumeration of all programs. This program is then interpreted by the self-interpreter e together with the input n, obtaining the same result as applying p_n to the input n. Then we add 1 to the result.

It is clear that the program d is not in the enumeration $\{p_k\}$, since if it were equal to the program p_n it would yield the same result for all inputs, in particular for the input n. But d is constructed in such a way that

$$\langle d, n \rangle = \langle p_n, n \rangle + 1 \neq \langle p_n, n \rangle$$

We have a contradiction and must conclude that it is impossible to write a self-interpreter for a language in which all programs terminates.