Mu-Recursive Functions as Description of While-Loops

We need while-loops. We have shown that not all computable functions can be computed by using only for-loops. Thus, we need to use the construction that we did consider so far – while-loops.

Clarification. Of course, it is clear that while-loops are needed – for efficiency. For example, if we are looking, in an unsorted array $a[i]$ with n elements, for the index d of an element e, we can do it with a for-loop:

```
for(i = 0; i < n; i++)
  {if(a[i] == e){d = i;}}
```

The problem with this code if that once we have found the desired element, further steps are not needed – they are a waste of time. From the viewpoint of efficiency, it is definitely better to use a while-loop:

```
int i = 0;
while(!(a[i] == e))
  {i++;}
```

However, at this point, we are not taking about efficiency, we are interested only in what is computable and what is not. From this viewpoint, the above example does not explain why while-loops are needed: in this example, we *can* use a for-loop instead of a while-loop.

In contrast, the above complex examples do work: there are computable functions (e.g., the Ackermann function) that cannot be computed if we only use for-loops.

How to formalize the notion of a while-loop. In the for-loop, we know how many iterations to make. In the while-loop, with a statement of the type `while(P(n1,...,nk,m))` we do not know a priori how many iterations we make: we stop at the first iteration m at which the property P is not satisfied.

Thus, to describe the while-loop, we need to be able to describe the first value m for which some property (e.g., "not P") is satisfied. This leads us to the following definition.

Definition 1. *By* mu-recursion *(also known as* μ-recursion*), we mean the expression* $\mu m.P(n_1, \ldots, n_k, m)$ *that returns the smallest natural number* m *for which* $P(n_1, \ldots, n_k, m)$ *is true.*

Comments.

- The word "smallest" means the same as "minimum", and μ (pronounced *mu*) is the Greek version of the letter m, the first letter of the word "minimum".
- Instead of m, you can use any other variable, e.g., $\mu a.P(n_1, \ldots, n_k, a)$, etc.

If we add μ-recursion to the list of possible constructions, we get the following definition:

Definition 2. *A function is called* μ-recursive *if it can be obtained from 0,* σ*, and* π_i^k *by using composition, primitive recursion, and* μ*-recursion.*

How to describe μ-recursion in terms of the while-loop. This is straightforward: we start with $m = 0$, and, if the property $P(n_1, \ldots, n_k, m)$ is not satisfied, we increase m by one:

```
int m = 0;
while(!P(n1,...,nk,m))
  {m++;}
```

How to describe a while-loop in terms of μ-recursion. To show how to do this description, let us start with a simple example of a while-loop for computing, for a given number a, the smallest power of two t for which $t \geq a$:

```
int t = 1;
while(t < a)
  {t = 2 * t;}
```

First, we write the same program, but with a for-loop instead of the while-loop. In this case, the resulting program takes the following form:

```
int t = 1;
for(int i = 1; i <= m; i++)
  {t = 2 * t;}
```

This program can be described as a p.r. function $t(i)$:

$$t(0) = 1;$$

$$t(i+1) = 2 * t(i);$$

i.e., as $PR(1, mult(2, \pi_3^3))$. To get the desired result, we need to find the smallest value m for which $!(t(m) < a)$, i.e., the value $\mu m.(!(t(m) < a))$.

So, the result of the above while-loop-using program can be described by substituting this number of iterations m into the expression $t(i)$. Thus, we get the following expression: $t(\mu m.(!(t(m) < a)))$.

Example. The above function $t(i)$ has the following values: $t(0) = 1$, $t(1) = 2$, $t(2) = 4$, $t(3) = 8$, etc. Thus for $a = 3$, the result of applying the above program is $t = 4$; indeed:

- at first, we have $t = 1$; the condition $t < a$ is satisfied, so we go inside the while-loop and replace $t = 1$ with the new value $t = 2 \cdot 1 = 2$;
- for $t = 2$, the condition $t < a$ is still satisfied, so we go inside the while-loop and replace $t = 2$ with the new value $t = 2 \cdot 2 = 4$;
- for $t = 4$, the condition $t < a$ is no longer satisfied, so we get out of the loop and return this value $t = 4$.

Here, the smaller m for which $t(m)$ is not smaller than $a = 3$ is $m = 2$ – and $t(2) = 4$ is exactly what the above program returns.

Important feature of μ-recursive functions: they may run forever. A program that uses only for-loops always stops, since each for-loop stops after a certain number of iterations. In contrast, a while-loop may go on forever – and this often happens to program written by students who are still learning these loops. For example, $\mu m.(0 = 1)$ will never stop: in the corresponding program

```
int m = 0;
while(!(0 == 1))
  {m++;}
```

the condition is always satisfied, so the program will take $m = 0$, $m = 1$, $m = 2$, ..., and never stop.

Some programs containing while-loops stop and produce results for some inputs, and go into an infinite loop for other inputs. For such inputs, we can say that the resulting function is *undefined.*

How to design a program that has a few given values and is undefined for all other inputs? To illustrate how this can be done, let us give an example. Suppose that we want to design a μ-recursive program for computing the following function:

- $f(2) = 4$;
- $f(6) = 5$, and
- $f(n)$ is undefined for all other n.

Let us brainstorm how to describe this function in terms of μ-recursion, i.e., as $\mu m.P(n, m)$ for some property $P(n, m)$.

- to get $f(2) = 4$, we need to make sure that for $n = 2$, the smallest value m for which the property $P(2, m)$ is satisfied is $m = 4$; in particular, this means that we should have $P(2, m)$ false for $m < 4$ and true for $m = 4$;
- to get $f(6) = 5$, we need to make sure that for $n = 6$, the smallest value m for which the property $P(6, m)$ is satisfied is $m = 5$; in particular, this means that we should have $P(6, m)$ false for $m < 5$ and true for $m = 5$;

- to make sure that $f(n)$ is undefined for all other n, we need to make sure that for n which is different from 2 and 6, the property $P(n,m)$ is never satisfied.

Thus, a natural idea is to make sure that the property $P(n,m)$ is only satisfied if either $n=2$ and $m=4$, or if $n=6$ and $m=5$. Thus, we get the following representation of the above function in terms of μ-recursion

$$\mu m.((n=2\,\&\,m=4)\vee(n=6\,\&\,m=5)).$$

The corresponding Java program has the form:

```
int m = 0;
while(!((n == 2 && m == 4) || (n == 6 && m == 5)))
  {m++;}
```

One can easily check:

- that for $n=2$, this program will return $m=4$,
- that for $n=6$, it will return $m=5$, and
- that for all other n, the condition in the while-loop will always be satisfied and thus, the program will never stop.

Subtraction $a \dot{-} b$ as a μ-recursive function. As we have mentioned, often, while-loops make programs more efficient. In particular, this means that the use of μ-recursion can lead to a simpler description of a primitive recursive function. Let us illustrate this on the example of subtraction $a \dot{-} b$.

The description of subtraction as a p.r. function started with a description of the operation "previous" ($\dot{-}\,\dot{-}$) in terms of primitive recursion – i.e., in terms of a for-loop. Then, subtraction $\dot{-}$ was defined in terms of "previous" by using another primitive recursion, i.e., another for-loop. Thus, to describe subtraction $\dot{-}$ in terms of primitive recursion, we needed two embedded for-loops – as compared to, e.g., addition that only requires one for-loop.

It turns out that we can decrease the number of for-loops if we use μ-recursion. The situation would have been easier if we consider a regular subtraction, with possibly negative values. In this case, $a-b=c$ is equivalent to $a=b+c$, so we could describe $a-b$ as $\mu c.(b+c=a)$. This formula still works if $a\ge b$, but if $a<b$ – e.g., for $a=2$ and $b=5$ – it will not work, since for all natural numbers $c\ge 0$, we will have $b+c\ge b>a$ and will never get $b+c=a$.

It turns out that we *can* get the desired description if we replace equality with $\ge$, i.e., if we consider the expression $\mu c.(b+c\ge a)$. Indeed, the desired inequality $b+c\ge a$ is equivalent to $c\ge a-b$.

- if $a\ge b$, i.e., if $a-b\ge 0$, the smallest natural number c for which $c\ge a-b$ is $c=a-b$;

- on the other hand, if $a < b$, then $a - b < 0$ and thus, the inequality $c \geq a - b$ is satisfied already for $c = 0$; so, in this case, the smallest value c satisfying this inequality is 0.

Thus, in both cases, the above μ-recursive formula indeed described the subtraction $a \dot{-} b$.