Recursion II

Alex Kavvos

Reading: PFPL, §19

1 Programming in PCF

PCF is the first realistic language we have seen in this course: it is a purely functional language with recursion and natural numbers. To demonstrate its expressivity we show how to write the following functions in it.

```
data Nat = Zero | Succ Nat

pred :: Nat -> Nat
pred Zero = Zero
pred (Succ n) = n

plus :: Nat -> Nat -> Nat
plus n Zero = n
plus n (Succ x) = Succ (plus n x)

times :: Nat -> Nat -> Nat
times = ???

fact :: Nat -> Nat
fact Zero = Succ Zero
fact (Succ n) = times (Succ n) (fact n)
```

In the preceding lecture we saw that `pred` can be translated as

$$\vdash \mathsf{pred} \stackrel{\text{def}}{=} \lambda n : \mathsf{Nat}.\, \mathsf{ifz}(n; \mathsf{zero}; x.\, x) : \mathsf{Nat} \rightharpoonup \mathsf{Nat}$$

We can write `plus` as follows.

$$\vdash \mathsf{plus} \stackrel{\text{def}}{=} \mathsf{fix}(f : \mathsf{Nat} \rightharpoonup \mathsf{Nat} \rightharpoonup \mathsf{Nat}.\, \lambda n : \mathsf{Nat}.\, \lambda m : \mathsf{Nat}.\, \mathsf{ifz}(m; n; x.\, \mathsf{succ}(f(n)(x)))) : \mathsf{Nat} \rightharpoonup \mathsf{Nat} \rightharpoonup \mathsf{Nat}$$

Finally, assuming we have defined a term $\vdash \mathsf{times} : \mathsf{Nat} \rightharpoonup \mathsf{Nat} \rightharpoonup \mathsf{Nat}$, we can define the factorial function by

$$\vdash \mathsf{fact} \stackrel{\text{def}}{=} \mathsf{fix}(f : \mathsf{Nat} \rightharpoonup \mathsf{Nat}.\, \lambda n : \mathsf{Nat}.\, \mathsf{ifz}(n; \mathsf{succ}(\mathsf{zero}); x.\, \mathsf{times}(n)(f(x)))) : \mathsf{Nat} \rightharpoonup \mathsf{Nat}$$

2 Turing-completeness

The examples given above demonstrate that there exist PCF terms corresponding to many Haskell functions that can be defined on the type `Nat` using pattern matching and recursion. But how far does this go?

Definition 1. A partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is **PCF-definable** iff there exists a PCF term $\vdash e : \mathsf{Nat} \rightharpoonup \mathsf{Nat}$ with

$$f(x) \simeq y \iff e(\mathsf{succ}^x(\mathsf{zero})) \longmapsto^* \mathsf{succ}^y(\mathsf{zero})$$

We then have the following theorem.

Theorem 2 (Turing-completeness of PCF). A function is PCF-definable if and only if it is **partial recursive**.

This means that PCF can compute every function $\mathbb{N} \rightharpoonup \mathbb{N}$ that is believed to be computable by a digital computer (or Turing machine, or register machine, or RAM machine, or ...) according to the **Church-Turing thesis**.

3 Sequentiality

The Turing-completeness of PCF implies that it is as powerful a language as we can have at type $\mathbb{N} \rightharpoonup \mathbb{N}$. Nevertheless, in the study of programming languages we care about more than just the computability of partial functions of natural numbers.

Gordon Plotkin established the following theorem about PCF in 1977.

Theorem 3. There is no PCF term $\vdash \mathsf{por} : \mathsf{Nat} \rightharpoonup \mathsf{Nat} \rightharpoonup \mathsf{Nat}$ that satisfies the following three criteria.

1. If $e_1 \longmapsto^* \mathsf{zero}$ then $\mathsf{por}(e_1)(e_2) \longmapsto^* \mathsf{zero}$.
2. If $e_2 \longmapsto^* \mathsf{zero}$ then $\mathsf{por}(e_1)(e_2) \longmapsto^* \mathsf{zero}$.
3. If $e_1 \longmapsto^* \mathsf{succ}(\mathsf{zero})$ and $e_2 \longmapsto^* \mathsf{succ}(\mathsf{zero})$ then $\mathsf{por}(e_1)(e_2) \longmapsto^* \mathsf{succ}(\mathsf{zero})$.

Thinking of zero as true and $\mathsf{succ}(\mathsf{zero})$ as false (like in Unix!), the term por can be thought of as encoding a **parallel or** function. This amounts to evaluating the expression $A \vee B$ by spawning two threads that will evaluate A and B concurrently. These two expressions might be very complicated—in fact, they might contain infinite loops! If one of the two spawned threads returns true, so does the evaluation of $A \vee B$. Otherwise, the computation continues until both A and B evaluate to false.

If we represent an infinite loop by the symbol $\bot$, the parallel or function has the following 'truth table.'

$\vee$	**tt**	**ff**	$\bot$
tt	**tt**	**tt**	**tt**
ff	**tt**	**ff**	$\bot$
$\bot$	**tt**	$\bot$	$\bot$

Even though PCF can compute every numerical function that a digital computer can, it cannot compute this very simple 'parallel' function! It is often said that PCF embodies the notion of **sequential functional computation**.

The classical theory of computability exclusively discusses the computability of partial functions on natural numbers through some model of computation, e.g. Turing machines. In those terms all known models of computation have equivalent expressivity (the Church-Turing thesis). However, classical computability cannot capture this kind of higher-order expressivity. This is sometimes called the '**Church-Turing anti-thesis**.'