Recursion

Alex Kavvos

Reading: PFPL, §19

1 Termination for the simply-typed λ-calculus

The simply-typed λ-calculus (STLC) has a property that is very unusual for a programming language.

Theorem 1 (Termination)**.** For every $\vdash e : \tau$ there exists a v val such that $e \longmapsto^* v$.

This may be proven using the technique of **logical relations**; see e.g. here.

In other words, every program written in the STLC terminates with a value. However, we intuitively know that any realistic programming language allows **infinite loops**. This theorem says that it is impossible to write a term with infinite behaviour in the STLC, so there is room to increase its expressivity.

2 Recursion and fixed points

We want to add **general recursion** to the STLC; this will enable the writing of recursive programs, as in Haskell.

Consider the following recursive definition of the factorial function:

$$\mathsf{fact}(n) = \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * \mathsf{fact}(n-1)$$

First we use (informal) λ-notation to abstract away the argument:

$$\mathsf{fact} = \lambda n.\ \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * \mathsf{fact}(n-1)$$

Then we use λ-notation again to abstract away the **recursive call**:

$$\mathsf{fact} = \underbrace{(\lambda f.\ \lambda n.\ \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * f(n-1))}_{F}(\mathsf{fact})$$

This is an equation of the form $\mathsf{fact} = F(\mathsf{fact})$, which is to say that fact is a **fixed point** of the higher-order function given by $F(f) \stackrel{\text{def}}{=} \lambda n.\ \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } n * f(n-1)$. The types here are

$$\mathsf{fact} : \mathbb{N} \rightharpoonup \mathbb{N} \qquad\qquad F : (\mathbb{N} \rightharpoonup \mathbb{N}) \to (\mathbb{N} \rightharpoonup \mathbb{N})$$

Therefore one way to add recursion to a programming language is to include a construct that computes the fixed point of any function $F : \sigma \to \sigma$. If we have fixed points at all types then we have them for $\mathbb{N} \to \mathbb{N}$ as well.

Curiously, this may be achieved within Haskell itself.

```
fix :: (a -> a) -> a
fix f = f (fix f)

h :: (Integer -> Integer) -> (Integer -> Integer)
h f n = if n == 0 then 1 else n * f (n-1)

fact :: Integer -> Integer
fact = fix h
```

3 PCF

PCF (= Programming Computable Functions) = (some version of) the STLC + fixed points. Syntax chart:

types	τ	$::=$	Nat	natural numbers
			$\tau_1 \rightharpoonup \tau_2$	(partial) function type
pre-terms	e	$::=$	x	variables
			zero	zero
			$\mathsf{succ}(e)$	successor
			$\mathsf{ifz}(e; e_0; x.\, e_1)$	zero test
			$\lambda x : \tau.\, e$	abstraction
			$e_1(e_2)$	application
			$\mathsf{fix}(x : \tau.\, e)$	fixed point

The **statics** of PCF are given by the following typing rules.

$$\textsc{Var}\ \frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \textsc{Zero}\ \frac{}{\Gamma \vdash \mathsf{zero} : \mathsf{Nat}} \qquad \textsc{Succ}\ \frac{\Gamma \vdash e : \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}(e) : \mathsf{Nat}}$$

$$\textsc{Lam}\ \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x : \sigma.\, e : \sigma \rightharpoonup \tau} \qquad \textsc{App}\ \frac{\Gamma \vdash e_1 : \sigma \rightharpoonup \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1(e_2) : \tau}$$

$$\textsc{IfZero}\ \frac{\Gamma \vdash e : \mathsf{Nat} \qquad \Gamma \vdash e_0 : \tau \qquad \Gamma, x : \mathsf{Nat} \vdash e_1 : \tau}{\Gamma \vdash \mathsf{ifz}(e; e_0; x.\, e_1) : \tau} \qquad \textsc{Fix}\ \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}(x : \tau.\, e) : \tau}$$

What has been removed: products, sums (can be added back at will). What has been replaced: numbers and strings (by natural numbers, with an "if zero" test). What has been added: fixed points. The **dynamics** are

$$\textsc{Val-Zero}\ \frac{}{\mathsf{zero}\ \mathsf{val}} \qquad \textsc{Val-Succ}\ \frac{e\ \mathsf{val}}{\mathsf{succ}(e)\ \mathsf{val}} \qquad \textsc{Val-Lam}\ \frac{}{\lambda x : \tau.\, e\ \mathsf{val}} \qquad \textsc{D-Succ}\ \frac{e \longmapsto e'}{\mathsf{succ}(e) \longmapsto \mathsf{succ}(e')}$$

$$\textsc{D-App-1}\ \frac{e_1 \longmapsto e_1'}{e_1(e_2) \longmapsto e_1'(e_2)} \qquad \textsc{D-Beta}\ \frac{}{(\lambda x : \tau.\, e_1)(e_2) \longmapsto e_1[e_2/x]}$$

$$\textsc{D-Fix}\ \frac{}{\mathsf{fix}(x : \tau.\, e) \longmapsto e[\mathsf{fix}(x : \tau.\, e)/x]} \qquad \textsc{D-Ifz-1}\ \frac{e \longmapsto e'}{\mathsf{ifz}(e; e_0; x.\, e_1) \longmapsto \mathsf{ifz}(e'; e_0; x.\, e_1)}$$

$$\textsc{D-Ifz-Zero}\ \frac{}{\mathsf{ifz}(\mathsf{zero}; e_0; x.\, e_1) \longmapsto e_0} \qquad \textsc{D-Ifz-Succ}\ \frac{\mathsf{succ}(e)\ \mathsf{val}}{\mathsf{ifz}(\mathsf{succ}(e); e_0; x.\, e_1) \longmapsto e_1[e/x]}$$

For example, the following terms are well-typed.

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

We have the following transition sequences.

$$\begin{aligned}
&\mathsf{pred}(\mathsf{zero}) \longmapsto \mathsf{ifz}(\mathsf{zero}; \mathsf{zero}; x.\, x) \longmapsto \mathsf{zero} \\
&\mathsf{pred}(\mathsf{succ}(\mathsf{zero})) \longmapsto \mathsf{ifz}(\mathsf{succ}(\mathsf{zero}); \mathsf{zero}; x.\, x) \longmapsto \mathsf{zero} \\
&\mathsf{pred}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))) \longmapsto \mathsf{ifz}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero})); \mathsf{zero}; x.\, x) \longmapsto \mathsf{succ}(\mathsf{zero}) \\
&\mathsf{pred}(\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero})))) \longmapsto \mathsf{ifz}(\mathsf{succ}(\mathsf{succ}(\mathsf{succ}(\mathsf{zero}))); \mathsf{zero}; x.\, x) \longmapsto \mathsf{succ}(\mathsf{succ}(\mathsf{zero})) \\
&\quad\vdots
\end{aligned}$$