

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 $e \mapsto^* v$.

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:

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

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

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

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

$$\text{fact} = \underbrace{(\lambda f. \lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * f(n - 1))}_F(\text{fact})$$

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

$$\text{fact} : \mathbb{N} \rightarrow \mathbb{N} \qquad F : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$$

Therefore one way to add recursion to a programming language is to include a construct that computes the fixed point of any function $F : \sigma \rightarrow \sigma$. If we have fixed points at all types then we have them for $\mathbb{N} \rightarrow \mathbb{N}$ 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	$\tau ::=$	Nat	natural numbers
		$\tau_1 \rightarrow \tau_2$	(partial) function type
pre-terms	$e ::=$	x	variables
		zero	zero
		succ(e)	successor
		ifz($e; e_0; x. e_1$)	zero test
		$\lambda x : \tau. e$	abstraction
		$e_1(e_2)$	application
		fix($x : \tau. e$)	fixed point

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

VAR	ZERO	Succ
$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}$	$\frac{}{\Gamma \vdash \text{zero} : \text{Nat}}$	$\frac{\Gamma \vdash e : \text{Nat}}{\Gamma \vdash \text{succ}(e) : \text{Nat}}$
LAM	APP	
$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x : \sigma. e : \sigma \rightarrow \tau}$	$\frac{\Gamma \vdash e_1 : \sigma \rightarrow \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1(e_2) : \tau}$	
IFZERO	FIX	
$\frac{\Gamma \vdash e : \text{Nat} \quad \Gamma \vdash e_0 : \tau \quad \Gamma, x : \text{Nat} \vdash e_1 : \tau}{\Gamma \vdash \text{ifz}(e; e_0; x. e_1) : \tau}$	$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \text{fix}(x : \tau. e) : \tau}$	

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

VAL-ZERO	VAL-SUCC	VAL-LAM	D-SUCC
$\frac{}{\text{zero val}}$	$\frac{e \text{ val}}{\text{succ}(e) \text{ val}}$	$\frac{}{\lambda x : \tau. e \text{ val}}$	$\frac{e \mapsto e'}{\text{succ}(e) \mapsto \text{succ}(e')}$
D-APP-1	D-BETA		
$\frac{e_1 \mapsto e'_1}{e_1(e_2) \mapsto e'_1(e_2)}$	$\frac{}{(\lambda x : \tau. e_1)(e_2) \mapsto e_1[e_2/x]}$		
D-FIX	D-IFZ-1		
$\frac{}{\text{fix}(x : \tau. e) \mapsto e[\text{fix}(x : \tau. e)/x]}$	$\frac{e \mapsto e'}{\text{ifz}(e; e_0; x. e_1) \mapsto \text{ifz}(e'; e_0; x. e_1)}$		
D-IFZ-ZERO	D-IFZ-SUCC		
$\frac{}{\text{ifz}(\text{zero}; e_0; x. e_1) \mapsto e_0}$	$\frac{\text{succ}(e) \text{ val}}{\text{ifz}(\text{succ}(e); e_0; x. e_1) \mapsto e_1[e/x]}$		

For example, the following terms are well-typed.

$$\begin{aligned} \vdash \text{pred} &\stackrel{\text{def}}{=} \lambda n : \text{Nat}. \text{ifz}(n; \text{zero}; x. x) : \text{Nat} \rightarrow \text{Nat} \\ \vdash \text{fix}(n : \text{Nat}. \text{succ}(n)) &: \text{Nat} \end{aligned}$$

We have the following transition sequences.

$$\begin{aligned} \text{pred}(\text{zero}) &\mapsto \text{ifz}(\text{zero}; \text{zero}; x. x) \mapsto \text{zero} \\ \text{pred}(\text{succ}(\text{zero})) &\mapsto \text{ifz}(\text{succ}(\text{zero}); \text{zero}; x. x) \mapsto \text{zero} \\ \text{pred}(\text{succ}(\text{succ}(\text{zero}))) &\mapsto \text{ifz}(\text{succ}(\text{succ}(\text{zero})); \text{zero}; x. x) \mapsto \text{succ}(\text{zero}) \\ \text{pred}(\text{succ}(\text{succ}(\text{succ}(\text{zero})))) &\mapsto \text{ifz}(\text{succ}(\text{succ}(\text{succ}(\text{zero}))); \text{zero}; x. x) \mapsto \text{succ}(\text{succ}(\text{zero})) \\ &\vdots \end{aligned}$$