

# Semantics of Functional Programming

## Lecture I: **PCF** and its Operational Semantics

Chen, Liang-Ting  
lxc@iis.sinica.edu.tw

Formosan Summer School on Logic, Language, and Computation 2014

## 1 Introduction

### The meaning of programs

How can we tell if a program is correct?

1. The meaning of a program is ideally independent of its actual implementation.
2. A rigorous specification of language is essential. Everything must be defined without any ambiguities. No undefined behaviour.
3. A structural approach to semantics. The meaning of a program is built from its parts, so verification is possible.

### Two approaches to be taught

1. **Operational approach:** How values and functions are computed? E.g., for  $\text{add} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$  and numerals  $\underline{2}, \underline{4}$

$$\begin{aligned} \text{add } \underline{2} \ \underline{4} &\rightsquigarrow \text{suc } (\text{add } \underline{1} \ \underline{4}) \\ &\rightsquigarrow \text{suc } \text{suc}(\text{add } \underline{0} \ \underline{4}) \rightsquigarrow \text{suc } \text{suc } \underline{4} \equiv \underline{6} \end{aligned}$$

2. **Denotational approach:** What the values and functions are? The set  $\mathbb{N}_\perp$  of natural numbers with *divergence*  $\perp$  is the denotation of the type  $\text{nat}$ , e.g.,

$$\begin{aligned} \llbracket \text{add } \underline{2} \ \underline{4} \rrbracket &= \llbracket \text{add } \underline{2} \rrbracket \llbracket \underline{4} \rrbracket = (\llbracket \text{add} \rrbracket \llbracket \underline{2} \rrbracket) \llbracket \underline{4} \rrbracket \\ &= (x \mapsto 2 + x) \llbracket \underline{4} \rrbracket = 2 + 4 = 6 \end{aligned}$$

where  $\llbracket \text{add} \rrbracket : \mathbb{N}_\perp \rightarrow (\mathbb{N}_\perp \rightarrow \mathbb{N}_\perp)$ .

## 2 Programming in PCF

### 2.1 Syntax and typing rules for PCF

#### What is PCF?

**PCF** stands for **P**rogramming **C**omputable **F**unctionals,

1. an extension of simply typed lambda calculus with  $\text{nat}$  and general recursion;

2. extremely simple compared to modern programming languages;
3. Turing complete, i.e. every computable function on natural numbers can be defined in **PCF**.

### Syntax of PCF

**Definition 1.** Types in **PCF** are defined by the inference rules

$$\frac{}{\text{nat set}} \quad \frac{\tau_1 \text{ set} \quad \tau_2 \text{ set}}{\tau_1 \rightarrow \tau_2 \text{ set}}$$

or equivalently by the grammar  $\tau := \text{nat} \mid \tau \rightarrow \tau$ .

**Definition 2.** The collection of terms in **PCF** is defined inductively:

$$\begin{aligned} M := & x \mid \lambda x. M \mid M N \mid \text{zero} \mid \text{suc } M \\ & \mid \text{ifz}(M; M; x. M) \mid Yx. M \end{aligned}$$

where  $x$  is a variable.

The operator  $Y$  is called the **fixpoint operator**, or **general recursion**.

### Typing Rules for PCF

A **judgement**  $\Gamma \vdash M : \tau$  denotes that  $M$  has type  $\tau$  under the context  $\Gamma$ . **PCF** consists of

- simply typed lambda calculus (with  $\rightarrow$  only):

$$\frac{}{\Gamma, x : \sigma, \Delta \vdash x : \sigma} \text{ (var)}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x. M : \sigma \rightarrow \tau} \text{ (abs)}$$

$$\frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M N : \tau} \text{ (app)}$$

- the type of natural numbers:

$$\frac{}{\Gamma \vdash \text{zero} : \text{nat}} \text{ (z)}$$

$$\frac{\Gamma \vdash M : \text{nat}}{\Gamma \vdash \text{succ } M : \text{nat}} \text{ (s)}$$

- **if zero** test: it is meant to be the *case analysis* on natural numbers:

$$\frac{\Gamma \vdash M : \text{nat} \quad \Gamma \vdash M_0 : \tau \quad \Gamma, x : \text{nat} \vdash M_1 : \tau}{\Gamma \vdash \text{ifz}(M; M_0; x. M_1) : \tau} \text{ (ifz)}$$

- **general recursion** (to be explained):

$$\frac{\Gamma, x : \sigma \vdash M : \sigma}{\Gamma \vdash Yx. M : \sigma} \text{ (Y)}$$

**Definition 3.** A term  $M$  of type  $\tau$  is called a **program** of type  $\tau$  in **PCF** if it is derivable under an empty context, i.e. the judgement

$$() \vdash M : \tau$$

is derivable where  $()$  denotes the empty context for emphasis.

E.g.  $Yx. \text{succ } x$  and  $\text{ifz}(\text{zero}; \lambda x. \text{zero}; y. \lambda z. y)$  are programs, but  $\lambda y. x y$  or  $\text{succ } (\lambda x. \text{succ } x)$  are not.

### Example: predecessor

The predecessor of natural numbers can be defined as

$$\text{pred} := \lambda x. \text{ifz}(x; \text{zero}; y. y) : \text{nat} \rightarrow \text{nat}$$

with the following typing derivation:

$$\frac{\frac{\frac{}{\Gamma \vdash x : \text{nat}} \quad \frac{}{\Gamma \vdash \text{zero} : \text{nat}} \quad \frac{}{\Gamma, y : \text{nat} \vdash y : \text{nat}}}{\Gamma \vdash \text{ifz}(x; \text{zero}; y. y) : \text{nat}}}{\vdash \lambda x. \text{ifz}(x; \text{zero}; y. y) : \text{nat} \rightarrow \text{nat}}$$

where  $\Gamma := x : \text{nat}$ .

## 2.2 Operational semantics

### One-step reduction

**Definition 4.** A **closed value** denoted **val** is one of the following:

$$\begin{array}{c} \frac{}{\text{zero val}} \\ \frac{M \text{ val}}{\text{succ } M \text{ val}} \\ \frac{}{\lambda x. M \text{ val}} \end{array}$$

A value is meant to be the final result of computation. For example, natural numbers **zero**, **succ zero** and lambda functions  $\lambda x. x$  etc. This formulation also includes ill-typed terms such as  $\text{succ } (\lambda x. M)$ .

### Notation

The notation  $\rightsquigarrow$  is a relation between terms, denoted

$$M \rightsquigarrow M'$$

which means that the term  $M$  reduces to  $M'$  in *one step*.

### Reduction of general recursion and natural numbers

For general recursion, each occurrence of  $Y.M$  reduces to an substitution of  $x$  in  $M$  by itself:

$$\frac{}{Yx. M \rightsquigarrow M[Yx. M/x]} \text{ (}\rightsquigarrow\text{-fix)}$$

For the *eager evaluation*,  $\text{succ } M$  reduces to  $\text{succ } M'$  if  $M$  reduces to  $M'$

$$\frac{M \rightsquigarrow M'}{\text{succ } M \rightsquigarrow \text{succ } M'} \text{ (}\rightsquigarrow\text{-succ)}$$

On the other hand, it is possible to defer the evaluation of natural numbers, and this evaluation is known as the *lazy evaluation*. To do so, we simply remove this reduction rule  $\rightsquigarrow\text{-succ}$  and modify the definition of closed values for **succ** to

$$\frac{}{\text{succ } M \text{ val}}$$

without any assumptions.

### Reduction of ifz

For the if-zero test, the first argument must be reduced to a closed value before branching, but branching can be done before the evaluation on branches:

$$\frac{M \rightsquigarrow M'}{\text{ifz}(M; M_0; x. M_1) \rightsquigarrow \text{ifz}(M'; M_0; x. M_1)} \text{ (}\rightsquigarrow\text{-ifz)}$$

$$\frac{}{\text{ifz}(\text{zero}; M_0; x. M_1) \rightsquigarrow M_0} \text{ (}\rightsquigarrow\text{-ifz}_0\text{)}$$

$$\frac{\text{succ } M \text{ val}}{\text{ifz}(\text{succ } M; M_0; x. M_1) \rightsquigarrow M_1[M/x]} \text{ (}\rightsquigarrow\text{-ifz}_1\text{)}$$

### Reduction for application: call-by-name and call-by-value

In call-by-name evaluation, arguments are substituted directly into the function body. It is a *non-strict* evaluation strategy, because application with non-terminating arguments can be terminating.

$$\frac{M \rightsquigarrow M'}{M N \rightsquigarrow M' N} (\rightsquigarrow\text{-lapp})$$

$$\frac{}{(\lambda x. M) N \rightsquigarrow M[N/x]} (\rightsquigarrow\text{-by-name})$$

In call-by-value evaluation, each argument is evaluated before application, so we replace  $(\rightsquigarrow\text{-by-name})$  by the following two rules. It is a *strict* evaluation strategy, as non-terminating arguments always lead to non-terminating terms.

$$\frac{M \text{ val} \quad N \rightsquigarrow N'}{M N \rightsquigarrow M' N} (\rightsquigarrow\text{-by-value-1})$$

$$\frac{N \text{ val}}{(\lambda x. M) N \rightsquigarrow M[N/x]} (\rightsquigarrow\text{-by-value-2})$$

In the following context, we adopt the *call-by-name* interpretation.

### Divergence

In contrast to dependent type theory, there are divergent **PCF** terms, e.g.

$$Yx.x$$

for any variable  $x : \sigma$ . If we try to evaluate it by one-step reduction, then we will find that this sequence

$$Yx.x \rightsquigarrow x[Yx.x] = Yx.x \rightsquigarrow Yx.x \rightsquigarrow \dots$$

never reaches a closed value.

### Many-step reduction

**Definition 5.** The relation  $\rightsquigarrow^*$  between terms is defined inductively:

$$\frac{}{M \rightsquigarrow^* M}$$

$$\frac{M_1 \rightsquigarrow M_2 \quad M_2 \rightsquigarrow^* M_3}{M_1 \rightsquigarrow^* M_3}$$

Note that  $M \rightsquigarrow^* M'$  if  $M'$  is reachable from  $M$  after finitely many steps of reduction, i.e.  $M = M_0 \rightsquigarrow M_1 \rightsquigarrow \dots M_k = M'$ .

**Proposition 6.** *The relation  $\rightsquigarrow^*$  is reflexive and transitive.*

*Proof.* Easy exercise.  $\square$

### Example: Calculating the factorials

To define the factorials, we are seeking for a function **fact** satisfying

$$\text{fact} : n \mapsto \begin{cases} 0 & \text{if } n = 1 \\ n \times \text{fact}(n') & \text{if } n = n' + 1 \end{cases}$$

and this can be understood as a fixpoint of the functional  $F$  mapping  $f : \mathbb{N} \rightarrow \mathbb{N}$  to  $f' : \mathbb{N} \rightarrow \mathbb{N}$  defined by

$$f' : n \mapsto \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n') & \text{if } n = n' + 1 \end{cases}$$

where  $f'$  does not depend on itself.

Given  $\Gamma := f : \text{nat} \rightarrow \text{nat}$ , we rewrite the definition in **PCF**:

$$\Gamma \vdash \lambda n. \text{ifz}(n; \text{suc zero}; m. n \times (f m)) : \text{nat} \rightarrow \text{nat}$$

and we derive its fixpoint by  $Y$ :

$$\text{fact} := Yf. \lambda n. \text{ifz}(n; \text{suc zero}; m. n \times (f m))$$

where the term **suc zero** represents the natural number 1.

*Example 7.* Let  $\underline{0} := \text{zero}$  and  $\underline{n+1} := \text{suc } \underline{n}$ . We calculate **fact 2**:

$$\begin{aligned} \text{fact } \underline{2} &\rightsquigarrow (\lambda n. \text{ifz}(n; \underline{1}; m. n \times (\text{fact } m))) \underline{2} \\ &\rightsquigarrow \text{ifz}(\underline{2}; \underline{1}; m. \underline{2} \times (\text{fact } m)) \\ &\rightsquigarrow \underline{2} \times (\text{fact } \underline{1}) \\ &\rightsquigarrow \underline{2} \times (\lambda n. \text{ifz}(n; \underline{1}; m. n \times (\text{fact } m))) \underline{1} \\ &\rightsquigarrow \dots \rightsquigarrow \underline{2} \times (\underline{1} \times \underline{1}) \rightsquigarrow^* \underline{2} \end{aligned}$$

where the definition of  $\times : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}$  is left as an exercise.

### In-class exercise

Try to be familiar with **ifz**.

1. Calculate **pred M** for  $M \text{ val}$  to closed values with their derivations: For the base case **zero**:

$$\text{pred zero} \rightsquigarrow^* ?$$

For the inductive case  $M = \text{suc } N$ :

$$\frac{\text{suc } N \text{ val}}{\text{pred } (\text{suc } N) \rightsquigarrow^* ?}$$

2. Define **flip**:  $\text{nat} \rightarrow \text{nat}$  such that **flip zero**  $\rightsquigarrow^*$  **suc zero** and **flip (suc M)**  $\rightsquigarrow^*$  **zero**.

### In-class exercise: fold on natural numbers

fold on natural numbers is defined in Haskell as follows:

```
fold :: (a -> a) -> a -> Integer -> a
fold f e 0 = e
fold f e n = f (fold f e (n - 1))
```

By modifying the definition of **fact**, give the corresponding term of fold in **PCF**.

## 3 Type safety

### Progress Theorem

Every well-typed is either a closed value or a reducible term.

**Theorem 8.** *If  $\vdash M : \tau$  then either  $M$  is a closed value or there exists  $M'$  such that  $M \rightsquigarrow M'$ .*

Note that given a program  $M : \tau$ , its proof produces either a closed value or a **PCF** term  $M'$  with a proof that  $M$  reduces to  $M'$ . Forgetting the proof, the proof itself is indeed a program which asks a **PCF** term, preforms a single reduction and return a term tagged either **done** or **not yet**.

*Proof.* By induction on the derivation of  $\vdash M : \tau$ . For the case that

$$\frac{M : \text{nat}}{\vdash \text{suc } M : \text{nat}}$$

$M$  is either a closed value or a reducible term by induction hypothesis:

1. If  $M$  is a closed value, then  $\text{suc } M$  is also a closed value by definition.
2. Suppose that  $M \rightsquigarrow M'$  for some  $M'$ . Then, by the rule ( $\rightsquigarrow\text{-suc}$ ), we also have  $\text{suc } M \rightsquigarrow \text{suc } M'$ .

Other cases follow similarly.  $\square$

### Substitution Lemma

If a variable  $x : \tau$  in a term  $M$  is substituted by another term  $N$  of the same type, then the type of the resulting term remains.

**Lemma 9.** *If  $\Gamma, x : \sigma \vdash M : \tau$  and  $\Gamma \vdash N : \sigma$ , then  $\Gamma \vdash M[N/x] : \tau$ .*

*Proof.* Induction on the derivation of  $\Gamma, x : \sigma \vdash M : \tau$ . Suppose that  $\Gamma, x : \sigma \vdash M : \tau$  is derived from

$$\frac{}{\Delta, y : \tau, x : \sigma \vdash y : \tau} (\text{var})$$

that is,  $\Gamma = \Delta, y : \tau$  and  $M = y$  for some variable  $y$ . Then, we need to show that  $\Delta, y : \tau \vdash y[N/x] : \tau$ .

1. If  $x = y$ , then  $y[N/x] = N : \sigma$  and  $\sigma = \tau$ .

2. Otherwise,  $x \neq y$ , then  $y[N/x] = y : \tau$ .

Other cases follow similarly.  $\square$

### Preservation Theorem

The one-step evaluation preserves types. This property is also called **Subject Reduction**.

**Theorem 10.** *If  $\vdash M : \tau$  and  $M \rightsquigarrow N$  then  $\vdash N : \tau$ .*

*Proof.* We prove it by induction on the derivation of  $\vdash M : \tau$  and  $M \rightsquigarrow M'$ . For the case that

$$\frac{x : \sigma \vdash M : \sigma}{\vdash Yx. M : \sigma}$$

we do induction on  $\rightsquigarrow$ , but there is exactly one rule applicable:

$$\frac{}{Yx. M \rightsquigarrow M[Yx. M/x]} (\rightsquigarrow\text{-fix})$$

By Substitution Lemma, it follows that  $\vdash M[Yx. M/x] : \sigma$ , and other cases follow similarly.  $\square$

## 4 Big-step semantics

### Call-by-name big-step semantics

Instead of the one-step reduction relation  $\rightsquigarrow$ , we turn to the **big-step** reduction relation  $\Downarrow$ , formulating the notion that a term  $M$  reduce to its final value  $V$ .

$$\frac{}{\text{zero} \Downarrow \text{zero}} (\Downarrow\text{-zero})$$

$$\frac{M \Downarrow V}{\text{suc } M \Downarrow \text{suc } V} (\Downarrow\text{-suc})$$

$$\frac{}{\lambda x. M \Downarrow \lambda x. M} (\Downarrow\text{-lam})$$

$$\frac{M \Downarrow \lambda x. E \quad E[N/x] \Downarrow V}{M N \Downarrow V} (\Downarrow\text{-app})$$

$$\frac{M \Downarrow \text{zero} \quad M_0 \Downarrow V}{\text{ifz}(M; M_0; x. M_1) \Downarrow V} (\Downarrow\text{-ifz}_0)$$

$$\frac{M \Downarrow \text{suc } N \quad M_1[N/x] \Downarrow V}{\text{ifz}(M; M_0; x. M_1) \Downarrow V} (\Downarrow\text{-ifz}_1)$$

$$\frac{M[Yx. M/x] \Downarrow V}{Yx. M \Downarrow V} (\Downarrow\text{-fix})$$

### Closed values

We shall justify the intended meaning: whenever  $M \Downarrow V$ , the term  $V$  is always a closed value:

**Lemma 11.** *For every terms  $M$  and  $V$ , the term  $V$  is a closed value if  $M \Downarrow V$ .*

*Proof.* By induction on the formulation of  $M \Downarrow V$ .  $\square$

Moreover, a closed value reduces to itself:

**Lemma 12.** *If  $V$  is a closed value, then  $V \Downarrow V$ .*

*Proof.* By structural induction on  $V$  **val**. That is, it is sufficient to check that **zero**  $\Downarrow$  **zero**,  $\lambda x. M \Downarrow \lambda x. M$ ; **suc**  $M \Downarrow$  **suc**  $M$  if  $M \Downarrow M$  by induction hypothesis.  $\square$

### Agreement of big-step and one-step semantics

The big-step reduction can be characterised by  $\rightsquigarrow$ :

**Theorem 13.** *For every term  $M$  and  $V$ ,  $M \Downarrow V$  if and only if  $M \rightsquigarrow^* V$  with  $V$  **val**.*

#### Proof sketch.

1. Show that if  $M \Downarrow V$  then  $M \rightsquigarrow^* V$  by induction on  $\Downarrow$  and  $\rightsquigarrow^*$ .
2. By induction on  $\rightsquigarrow$  and  $\Downarrow$ , show that

$$\frac{M \rightsquigarrow N \Downarrow V}{M \Downarrow V}$$

3. By induction on  $\rightsquigarrow^*$ , show that

$$\frac{M \rightsquigarrow^* N \Downarrow V}{M \Downarrow V}$$

In particular, every  $M \rightsquigarrow^* V$  with  $V$  **val**, has  $V \Downarrow V$ , so it follows that  $M \Downarrow V$ .

*Proof.* 1. We show the case ( $\Downarrow$ -fix), which is similar to other cases:

$$\frac{\frac{Yx. M \rightsquigarrow M[Yx. M/x]}{Yx. M \rightsquigarrow^* V} \quad \frac{M[Yx. M/x] \rightsquigarrow^* V}{Yx. M \rightsquigarrow^* V}}{Yx. M \rightsquigarrow^* V}$$

and by assumption  $V$  has no further reduction.

2. We show the case ( $\rightsquigarrow$ -fix), which is similar to other cases. By hypothesis, we have  $Yx. M \rightsquigarrow M[Yx. M/x]$ . If  $M[Yx. M/x] \Downarrow V$ , then by ( $\Downarrow$ -fix) it follows that  $Yx. M \Downarrow V$ .

3. Induction on  $\rightsquigarrow^*$ .  $\square$

By the agreement of big-step and one-step semantics, we easily conclude that the Subject Reduction also holds for big-step semantics:

**Corollary 14** (Subject Reduction for  $\Downarrow$ ). *If  $\vdash M : \tau$  and  $M \Downarrow V$  then  $\vdash V : \tau$ .*

## Exercises

1. Define the following programs in **PCF**.

- (a) Multiplication of natural numbers *Hint*. Define addition first;
- (b) Fibonacci numbers;
- (c) Parity test, i.e. a function determines whether the given argument is an odd or even number. Return **zero** if even, **suc zero** otherwise.

2. Let **bool** be a type with two constructors:

$$\frac{}{\text{true} : \text{bool}}$$

$$\frac{}{\text{false} : \text{bool}}$$

- (a) Provide the typing rule for the conditional construct **if**:

$$\frac{?}{\Gamma \vdash \text{if}(M_0; M_1; M_2) : \tau}$$

- (b) Provide its one-step semantics.

3. Define primitive recursion in **PCF**

$$\text{rec} : \tau \rightarrow (\text{nat} \rightarrow \tau \rightarrow \tau) \rightarrow \text{nat} \rightarrow \tau$$

such that

$$\begin{aligned} \text{rec } e_0 \text{ f zero} & \rightsquigarrow^* e_0 \\ \text{rec } e_0 \text{ f (suc } M) & \rightsquigarrow^* f \text{ M (rec } e_0 \text{ f M)} \end{aligned}$$

respectively

Consider Gödel's **T**, simply typed lambda calculus with natural numbers and *primitive recursion*:

$$\frac{\Gamma \vdash e_0 : \tau \quad \Gamma \vdash M : \text{nat} \quad \Gamma, x : \text{nat}, y : \tau \vdash e_1 : \tau}{\Gamma \vdash \text{rec}(e_0; x. y. e_1; M) : \tau}$$

4. Provide the one-step and big-step reductions for **rec**.

## Reference

*Denotational Semantics* and this lecture are based on the following two books:

1. Thomas Streicher, *Domain-Theoretic Foundations of Functional Programming*, World Scientific, 2006
2. Robert Harper, *Practical Foundations for Programming Languages*, Cambridge University Press, 2012

Their preprints are available on the Internet.