

# Semantics of Functional Programming

## **PCF** and its Operational Semantics

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

Institute of Information Science, Academia Sinica

Formosan Summer School on Logic, Language, and  
Computation 2014

# Why semantics?

*The hitch is that defining a language a posteriori, i.e. after its design has been frozen by the existence of implementations and uses, can hardly improve it. To create a good programming language, semantics must be used a priori, as a design tool that embodies and extends the intuitive notion of uniformity.*

*— John C. Reynolds*

- Implementation-led design.
- *C++ The International Standard*, 1338 pp., 2012.  
Note that the committee consists of 200+ people.
- 1900+ language issues!

**Revision 89, 2014-05-27:** Issues [1351](#), [1356](#), [1465](#), [1590](#), [1639](#), [1708](#), and [1810](#) were returned to "review" status for further discussion. Restored [issue 1397](#) to "ready"; it had incorrectly been moved back to "drafting" because of a misunderstood comment. Added new issues [1866](#), [1867](#), [1868](#), [1869](#), [1870](#), [1871](#), [1872](#), [1873](#), [1874](#), [1875](#), [1876](#), [1877](#), [1878](#), [1879](#), [1880](#), [1881](#), [1882](#), [1883](#), [1884](#), [1885](#), [1886](#), [1887](#), [1888](#), [1889](#), [1890](#), [1891](#), [1892](#), [1893](#), [1894](#), [1895](#), [1896](#), [1897](#), [1898](#), [1899](#), [1900](#), [1901](#), [1902](#), [1903](#), [1904](#), [1905](#), [1906](#), [1907](#), [1908](#), [1909](#), [1910](#), [1911](#), [1912](#), [1913](#), [1914](#), [1915](#), [1916](#), [1917](#), [1918](#), [1919](#), [1920](#), [1921](#), [1922](#), [1923](#), [1924](#), [1925](#), [1926](#), [1927](#), [1928](#), [1929](#), [1930](#), and [1931](#).

# Standard ML

- Semantics-led design.
- R. Milner, M. Tofte, R. Harper, and D. MacQueen *The Definition of **Standard ML** (Revised)*, 128 pp., 1997.
- Standard ML is a safe, modular, strict, functional, polymorphic programming language . . . and a formal definition with a proof of soundness.

# Overview

In this lecture, we will present simply typed lambda calculus in a different manner, where terms and typing rules are introduced separately. In this approach, terms might not be well-typed at all.

Then, we discuss its computational meaning by **one-step reduction** and define many-step reduction. Later we introduce the concept of **type safety**.

Finally, we extend simply typed lambda calculus with natural numbers and general recursion. This extension is called **PCF**, *Programming Computable Functional*. We formalise new features by what we have learnt later.

# The approach *à la* Curry

We introduce a different approach to simply typed lambda calculus where terms and typing rules are introduced separately.

$$\frac{x \text{ var}}{x \text{ term}}$$

$$\frac{x \text{ var} \quad M \text{ term}}{\lambda x. M \text{ term}}$$

$$\frac{M \text{ term} \quad N \text{ term}}{M N \text{ term}}$$

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

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

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

# The existence of ill-typed terms

In contrast the approach *à la* Church where every term is introduced with a type, there are ill-typed terms in the approach *à la* Curry:

## Example 1

$(\lambda x. x x)$  is a term if  $x$  is a variable, because

$$\frac{x \text{ var} \quad \frac{\frac{x \text{ var}}{x \text{ term}} \quad \frac{x \text{ var}}{x \text{ term}}}{x x \text{ term}}}{\lambda x. x x \text{ term}}$$

However,  $(\lambda x. x x)$  cannot be assigned a type unless  $\sigma \rightarrow \sigma = \sigma$ .

# Reduction

**One-step reduction** relation  $\rightsquigarrow$  between terms is introduced to describe the flow of computation from a term to another term in a single step, regardless of types. We introduce two rules for applications:

$$\frac{M \rightsquigarrow M'}{M N \rightsquigarrow M' N} (\rightsquigarrow\text{-lapp})$$
$$\frac{}{(\lambda x. M) N \rightsquigarrow M[N/x]} (\rightsquigarrow\text{-app})$$

## Example 2

$(\lambda x. \lambda y. x) M N$  can be reduced to  $M$  by the following derivation

$$\frac{\frac{}{(\lambda x. \lambda y. x) M \rightsquigarrow (\lambda y. M)} (\rightsquigarrow\text{-app})}{((\lambda x. \lambda y. x) M) N \rightsquigarrow (\lambda y. M) N} (\rightsquigarrow\text{-lapp})$$



# Many-step reduction

As we will mostly discuss a sequence of reductions, it is convenient to define another relation  $\rightsquigarrow^*$  so that  $M \rightsquigarrow^* N$  means  $M$  reduces to  $N$  in finitely many steps.

## Definition 3

The many-step reduction relation  $\rightsquigarrow^*$  is defined inductively by

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

## Proposition 4 (Reflexivity of $\rightsquigarrow^*$ )

*For every term  $M$ ,  $M \rightsquigarrow^* M$ .*

For example, one has

$$(\lambda x. \lambda y. x) M N \rightsquigarrow^* (\lambda y. M) N$$

by the derivation

$$\frac{\frac{(\lambda x. \lambda y. x) M \rightsquigarrow (\lambda y. M)}{((\lambda x. \lambda y. x) M) N \rightsquigarrow (\lambda y. M) N} \quad (\lambda y. M) N \rightsquigarrow^* (\lambda y. M) N}{(\lambda x. y. x) M N \rightsquigarrow^* (\lambda y. M) N}$$

**Exercise.** Evaluate the following terms (formally or informally).

1  $(\lambda x. x) y$

2  $(\lambda x. x x) (\lambda x. x x)$

3  $(\lambda x. \lambda y. \lambda z. y) M_0 M_1 M_2$

## Induction on derivation

Every instance of  $M \rightsquigarrow^* N$  must be constructed by one of the cases, so we can analyse its structure case by case.

### Proposition 5 (Transitivity of $\rightsquigarrow^*$ )

*For every three terms  $M_1$ ,  $M_2$ , and  $M_3$ , if  $M_1 \rightsquigarrow^* M_2$  and  $M_2 \rightsquigarrow^* M_3$ , then  $M_1 \rightsquigarrow^* M_3$ .*

Given derivations of  $M_1 \rightsquigarrow^* M_2$  and  $M_2 \rightsquigarrow^* M_3$ , we do case analysis on the derivation of  $M_1 \rightsquigarrow^* M_2$ . Also, we can assume that the premise satisfy this property, that is, the induction hypothesis.

## Proof.

- 1 For  $\overline{M_1 \rightsquigarrow^* M_1}$ , it unifies  $M_2$  to  $M_1$ , so the given derivation  $M_2 \rightsquigarrow^* M_3$  is just the goal derivation as  $M_1 = M_2$ .
- 2 For  $\frac{M_1 \rightsquigarrow M \quad M \rightsquigarrow^* M_2}{M_1 \rightsquigarrow^* M_2}$ , we infer that  $M \rightsquigarrow^* M_3$  by induction hypothesis, so we derive the goal

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



Similarly, we can do induction on the formation of terms, typing rules, and any other inductive definitions.

**Exercise.** Show that if  $M \rightsquigarrow^* M'$  then  $M N \rightsquigarrow^* M' N$  for any term  $N$  by induction on the derivation of  $M \rightsquigarrow^* M'$ .

## Reductions on ill-typed terms

Reductions can be applied to ill-typed terms and sometimes it reduces to a well-typed closed term!

$$(\lambda x. \lambda y. x) (\lambda x. x) (\lambda x. x x) \rightsquigarrow^* (\lambda x. x)$$

On the other hand, the reduction of ill-typed terms may be reduced to itself:

$$\begin{aligned} (\lambda x. x x) (\lambda x. x x) &\rightsquigarrow (x x)[(\lambda x. x x)/x] \\ &= (\lambda x. x x) (\lambda x. x x) \rightsquigarrow \dots \end{aligned}$$

## Type safety: well-typed programs don't go wrong

In contrast to ill-typed terms, well-typed closed terms have some nice properties. First, every well-typed closed term can be reduced further or it is a **value**.

### Theorem 6 (Progress Theorem)

*If  $\vdash M : \tau$ , then either  $M \rightsquigarrow M'$  for some  $M'$  or  $M = \lambda x. M'$ .*

To show this property, we do structural induction on the derivation of  $\vdash M : \tau$  and either produce a derivation of  $M \rightsquigarrow M'$  or show that  $M = \lambda x. M'$ .

## Proof.

- 1 The context of  $\overline{\Gamma, x : \sigma, \Delta \vdash x : \sigma}$ , is non-empty, so it does not satisfy the assumption.
- 2 For that case  $\frac{x : \sigma \vdash M : \tau}{\vdash \lambda x. M : \sigma \rightarrow \tau}$  (abs), we have already given a term in this form  $\lambda x. M$ .
- 3 For  $\frac{\vdash M : \sigma \rightarrow \tau \quad \vdash N : \sigma}{\vdash M N : \tau}$  (app), by introduction hypothesis either  $M \rightsquigarrow M'$  for some  $M'$  or  $M = \lambda x. M'$ . For the former case, we apply ( $\rightsquigarrow$ -lapp):

$$\frac{M \rightsquigarrow M'}{M N \rightsquigarrow M' N}$$

For the latter case, we apply ( $\rightsquigarrow$ -app)

$$\frac{}{(\lambda x. M') N \rightsquigarrow M'[N/x]}$$



Moreover, the type of a well-typed closed term is always preserved by reductions:

### Theorem 7 (Preservation Theorem)

*If  $\vdash M : \tau$  and  $M \rightsquigarrow M'$ , then  $\vdash M' : \tau$ .*

However, to show this property, we need the following lemma saying that types are preserved by substitution.

### Lemma 8 (Substitution Lemma)

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



By the introduction on the derivation of  $\vdash M : \tau$  and  $M \rightsquigarrow M'$  at the same time.

## Proof of Preservation Theorem.

- 1  $\vdash M : \tau$  cannot be constructed by (*var*), since the context is empty.
- 2 For  $\frac{x : \sigma \vdash M : \tau}{\vdash \lambda x. M : \sigma \rightarrow \tau}$ , there is no reduction rule for  $\lambda x. M$ , so a derivation  $(\lambda x. M) \rightsquigarrow M'$  cannot exist.
- 3 For  $\frac{\vdash M : \sigma \rightarrow \tau \quad \vdash N : \sigma}{\vdash M N : \tau}$ , we do induction on the derivation of  $M N \rightsquigarrow M'$ .



# Summary

To define a language, we specify following sets of rules

**Syntax** type, term, and typing rules.

**Semantics** reduction rules.

In particular, well-typed closed terms share type safety:

**Progress Theorem** for every well-typed closed term, it either can be reduced further or is a value;

**Preservation Theorem** for every well-typed closed term, its type is preserved by reduction.

Next, we add some features to simply typed lambda calculus and type safety remains.

# Introduction to PCF

**PCF**, which stands for **P**rogramming **C**omputable **F**unctionals, is a functional programming language and it consists of

- 1 simply typed lambda calculus,
- 2 natural numbers, and
- 3 general recursion (to be explained).

We will introduce the latter two features step by step.

It has two rules of type formation:

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

Still, 'set' is a synonyms of 'type'.

# Term formation, typing, and reduction for natural numbers

Every natural number is either zero or a successor of some natural number.

$$\frac{}{\text{zero } \mathbf{term}}$$

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

$$\frac{M \mathbf{term}}{\text{suc } M \mathbf{term}}$$

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

The reduction of (suc M) is given by its subterm M:

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

## Values: canonical elements

Values are basic forms of terms of each kind of types and they are defined independent of their types in the approach *à la* Curry.

### Definition 9

A **value** is a term of the following form:

$$\frac{}{\text{zero } \mathbf{val}} \quad \frac{M \mathbf{val}}{\text{suc } M \mathbf{val}} \quad \frac{}{\lambda x. M \mathbf{val}}$$

Define **numerals**  $\underline{0}$  for zero and  $\underline{n+1}$  for  $\text{suc } \underline{n}$  inductively.

### Example 10

By this formation, we have well-typed values  $\text{suc } (\text{suc } \text{zero})$ ,  $\lambda x. \text{suc } x$ , and  $\lambda x. x$ , and also ill-typed values  $\text{suc } \lambda x. x$ ,  $\lambda y. y \ y$ .

Moreover, we can do branching according to the argument is zero or not.

$$\frac{\text{M term} \quad \text{M}_0 \text{ term} \quad x \text{ var} \quad \text{M}_1 \text{ term}}{\text{ifz}(\text{M}; \text{M}_0; x. \text{M}_1) \text{ term}}$$

$$\frac{\Gamma \vdash \text{M} : \text{nat} \quad \Gamma \vdash \text{M}_0 : \tau \quad \Gamma, x : \text{nat} \vdash \text{M}_1 : \tau}{\Gamma \vdash \text{ifz}(\text{M}; \text{M}_0; x. \text{M}_1) : \tau} \text{ (ifz)}$$

accompanying with three reductions rules

$$\frac{\text{M} \rightsquigarrow \text{M}'}{\text{ifz}(\text{M}; \text{M}_0; x. \text{M}_1) \rightsquigarrow \text{ifz}(\text{M}'; \text{M}_0; x. \text{M}_1)} (\rightsquigarrow\text{-ifz})$$

$$\frac{}{\text{ifz}(\text{zero}; \text{M}_0; x. \text{M}_1) \rightsquigarrow \text{M}_0} (\rightsquigarrow\text{-ifz}_0)$$

$$\frac{\text{suc M val}}{\text{ifz}(\text{suc M}; \text{M}_0; x. \text{M}_1) \rightsquigarrow \text{M}_1[\text{M}/x]} (\rightsquigarrow\text{-ifz}_1)$$

## Example: predecessor

The predecessor of natural numbers can be defined as

$$\text{pred} := \lambda x. \text{ifz}(x; \underline{0}; y. y) : \text{nat} \rightarrow \text{nat}$$

with the following typing derivation:

$$\frac{\frac{\frac{\Gamma \vdash x : \text{nat} \quad \Gamma \vdash \underline{0} : \text{nat} \quad \Gamma, y : \text{nat} \vdash y : \text{nat}}{\Gamma \vdash \text{ifz}(x; \underline{0}; y. y) : \text{nat}}}{\vdash \lambda x. \text{ifz}(x; \underline{0}; y. y) : \text{nat} \rightarrow \text{nat}}}$$

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

### Exercise.

- 1 Show that  $\text{pred } \underline{0} \rightsquigarrow^* \underline{0}$  and  $\text{pred } \underline{n+1} \rightsquigarrow^* \underline{n}$ .
- 2 Define  $\text{flip} : \text{nat} \rightarrow \text{nat}$  such that  $\text{flip } \underline{0} \rightsquigarrow^* \underline{1}$  and  $\text{flip } \underline{n+1} \rightsquigarrow^* \underline{0}$ .

# Term formation, typing rule, and reduction for general recursion

The Y operator, used to do general recursion, has the same term formation as  $\lambda$ -abstraction and a similar typing rules.

$$\frac{x \text{ var} \quad M \text{ term}}{Yx. M \text{ term}}$$

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

Each occurrence of  $Yx. M$  reduces to an substitution of  $x$  in  $M$  by itself:

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

## Example 11 (Divergent term)

Consider the term  $Yx. x$  which never reduces to any value

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



## Example: calculating the factorials

The factorial of  $n$  is usually defined recursively

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

This is a *fixpoint* of the higher-order function

$F: (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$  defined by

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

for any  $f: \mathbb{N} \rightarrow \mathbb{N}$ , satisfying  $F(\text{fact}) = \text{fact}$ .

The higher-order function  $F: (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$  can be presented in **PCF** as

$$F := \lambda f. F'$$

with

$$F' := \lambda n. \text{ifz}(n; \underline{1}; m. n \times (f\ m)).$$

$\Upsilon f. F''$  is a a fixpoint of  $F$

$$\begin{aligned} (\lambda f. F') (\Upsilon f. F') &\rightsquigarrow F'[(\Upsilon f. F')/f] \\ &= \lambda n. \text{ifz}(n; \underline{1}; m. n \times (\Upsilon f. F')\ m) \end{aligned}$$

**Exercise.** Show that fact  $\underline{n} \rightsquigarrow^* \underline{n!}$  by induction on  $\underline{n}$ .

# Type safety for **PCF**

## Theorem 12 (Progress Theorem)

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

## Theorem 13 (Preservation Theorem)

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

All follow the same pattern in the situation for simply typed lambda calculus.<sup>1</sup>

---

<sup>1</sup>To be proved in **Agda** formally.

## Another reduction relation

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

- simply typed lambda calculus

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

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

- natural numbers

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

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

■ if-zero test

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

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

■ general recursion

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

$$\frac{\frac{\lambda x. \text{ifz}(x; \underline{0}; y. y) \Downarrow \lambda x. \text{ifz}(x; \underline{0}; y. y)}{\lambda x. \text{ifz}(x; \underline{0}; y. y) \underline{3} \Downarrow \underline{2}} \quad \frac{\frac{\frac{\vdots}{\underline{3} \Downarrow \text{suc } \underline{2}} \quad \frac{\frac{\vdots}{y[\underline{2}/y] \Downarrow \underline{2}}}{\text{ifz}(x; \underline{0}; y. y)[\underline{3}/x] \Downarrow \underline{2}}}{\lambda x. \text{ifz}(x; \underline{0}; y. y) \underline{3} \Downarrow \underline{2}}$$

Figure: Derivation of  $\text{pred } \underline{3} \Downarrow \underline{2}$

### Exercise.

- 1 Show that  $\text{fact } \underline{0} \Downarrow \underline{1}$ .
- 2 Show that  $\text{flip } \underline{0} \Downarrow \underline{1}$  and  $\text{flip } \underline{n+1} \Downarrow \underline{0}$ .

## Reduction on values

We shall justify the intended meaning. Whenever  $M \Downarrow V$ , the term  $V$  is always a value; every value is in its simplest form.

### Lemma 14

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

### Proof.

By induction on the derivation of  $M \Downarrow V$ . □

### Lemma 15

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

### Proof.

By induction on the derivation of  $V$  **val**. □

# Agreement of big-step and one-step semantics

## Theorem 16

*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 Show that if  $M \rightsquigarrow N$  and  $N \Downarrow V$  then  $M \Downarrow V$ .
- 3 Show that if  $M \rightsquigarrow^* N$  and  $N \Downarrow V$  then  $M \Downarrow V$ .

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

## Corollary 17 (Preservation Theorem for $\Downarrow$ )

*If  $\vdash M : \tau$  and  $M \Downarrow V$  then  $\vdash V : \tau$ .*