Semantics of Programming Languages

Andrzej Murawski and Nikos Tzevelekos

Lecture 1: Introduction, PCF, Turing completeness

Semantics

Giving a (the?) semantics of a programming language $\mathcal{L}$

$= $ Assigning **meanings** to programs of $\mathcal{L}$

What are "meanings"?

- In formal semantics, meanings are mathematical objects.
- These are meant to be an idealisation or abstraction of intuitive meaning.

Put simply, semantics is about formally answering the question:

That's a splendid program – but what does it really do?

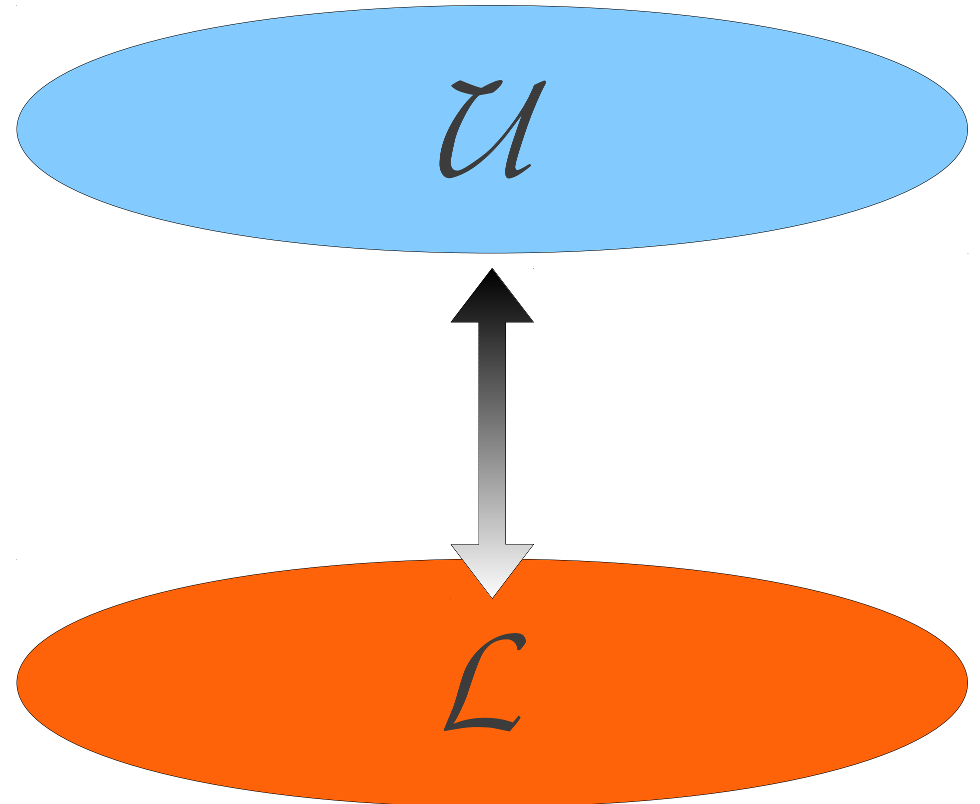
Why semantics?

- Gives insight into the structure and inter-relationship of programming constructs, hence supports rational and sound language design.
- Provides a mathematical foundation for *program analysis*: specifying, synthesising, verifying, transforming, interpreting, compiling, optimising, etc. programs.

Formal semantics provides a basis for viewing programs as mathematical objects. That is, it provides a *calculus of programs*.

Syntax vs Semantics

- Semantics: Mathematical universe of discourse $\mathcal{U}$
- Syntax: Programming language $\mathcal{L}$



From one point of view, we use $\mathcal{L}$ to talk about $\mathcal{U}$.

From another point of view, we use $\mathcal{U}$ to help us understand what we are saying in $\mathcal{L}$.

Example: a Simple Language of Arithmetic (SLA)

Consider the following syntax, defining the set of *expressions* $\mathcal{E}$ and the set of *numbers* $\mathcal{N}$.

$$\begin{array}{lcl} E & ::= & N \mid E_1 + E_2 \mid E_1 - E_2 \\ N & ::= & 0 \mid \mathtt{succ}(N) \end{array}$$

The above notation means that:

- The set of numbers is the smallest set $\mathcal{N}$ such that:
  - $0 \in \mathcal{N}$,
  - if $N \in \mathcal{N}$ then $\mathtt{succ}(N) \in \mathcal{N}$.
- The set of *expressions* is the smallest set $\mathcal{E}$ such that:
  - $N \in \mathcal{E}$, for any $N \in \mathcal{N}$,
  - if $E_1, E_2 \in \mathcal{E}$ then $E_1 + E_2, E_1 - E_2 \in \mathcal{E}$.

Main approaches to semantics: (I) Denotational

- Fix a mathematical structure (a *semantic algebra*) as a universe of meanings, and
- to each syntactic syntactic construct assign an interpretation as an operator in this universe.

Example (SLA)

$$\langle\, \mathbb{Z}\ ,\ 0\ ,\ _ +^{\mathbb{Z}} 1\ ,\ +^{\mathbb{Z}}\ ,\ -^{\mathbb{Z}}\ \rangle$$

We define the semantic function $[\![_]\!] : \mathcal{E} \to \mathbb{Z}$ as:

- $[\![\mathsf{0}]\!] = 0$
- $[\![\mathrm{succ}(N)]\!] = [\![N]\!] +^{\mathbb{Z}} 1$
- $[\![E_1 + E_2]\!] = [\![E_1]\!] +^{\mathbb{Z}} [\![E_2]\!]$
- $[\![E_1 - E_2]\!] = [\![E_1]\!] -^{\mathbb{Z}} [\![E_2]\!]$

$$\begin{aligned} [\![\mathsf{2}]\!] &= [\![\mathrm{succ}(\mathrm{succ}\,\mathsf{0})]\!] \\ &= [\![\mathrm{succ}\,\mathsf{0}]\!] +^{\mathbb{Z}} 1 \\ &= [\![\mathsf{0}]\!] +^{\mathbb{Z}} 1 +^{\mathbb{Z}} 1 = 0 +^{\mathbb{Z}} 1 +^{\mathbb{Z}} 1 = 2 \end{aligned}$$

$$[\![\mathrm{succ}(\mathsf{0}) + \mathrm{succ}(\mathsf{0})]\!] = \cdots = 2$$

Main approaches to semantics: (I) Denotational

Characteristics of denotational semantics

- Meanings are *syntax-independent* objects of a semantic universe
- The semantic function is *global* and *compositional*

Globality:

- Each (whole) syntactic construct mapped to a semantic operator $(+^{\mathbb{Z}}, -^{\mathbb{Z}}, \ldots)$.

Compositionality:

- The meaning of each expression is defined as a function of the meanings of its parts:

$$[\![OP(E_1, \ldots, E_n)]\!] = [\![OP]\!]([\![E_1]\!], \ldots, [\![E_n]\!])$$

Main approaches to semantics: (II) Operational

- Give a method or process for calculating the *value* or *output* or *behaviour* of each program.
- This is generally done by giving rules for manipulating syntax.

Example (SLA): Let *values* be given by: $V \;::=\; N \mid \mathsf{0} - \mathtt{succ}(N)$.
Define the evaluation relation $\Downarrow$ by:

$$\frac{}{N \Downarrow N}\ (R_N) \qquad \frac{}{\mathsf{0} - \mathtt{succ}(N) \Downarrow \mathsf{0} - \mathtt{succ}(N)}\ (R_{N-})$$

$$\frac{E_1 \Downarrow \mathsf{0} \quad E_2 \Downarrow N}{E_1 + E_2 \Downarrow N}\ (R_{0+}) \qquad \frac{E_1 \Downarrow \mathsf{0} \quad E_2 \Downarrow N}{E_1 - E_2 \Downarrow \mathsf{0} - N}\ (R_{0-})$$

$$\frac{E_1 \Downarrow \mathtt{succ}(N) \quad N + E_2 \Downarrow M}{E_1 + E_2 \Downarrow \mathtt{succ}(M)}\ (R_{S+}) \qquad \frac{E_1 \Downarrow \mathtt{succ}(N) \quad N - E_2 \Downarrow M}{E_1 - E_2 \Downarrow \mathtt{succ}(M)}\ (R_{S-})$$

$$\vdots \qquad\qquad \vdots$$

Main approaches to semantics: (II) Operational

Characteristics of operational semantics:

- Meanings are *syntax-dependent* operations of programs
- The semantics is *local* and *non-compositional*

Locality:

- Meaning is built up via "small" computational steps.

Non-compositionality:

- The overall meaning of a program is *not* given directly as a function of the meanings of its parts.

Comparison

Denotational semantics brings out global mathematical structure.

Operational semantics brings out computational microstructure.

So both are good for something.

- We want to use both and be able to move between them
- This motivates the need for **correspondence theorems**, which tell us that they give *equivalent* accounts of meaning.

Main approaches to semantics: (III) Axiomatic

- In contrast to denotational and operational semantics, axiomatic semantics gives *implicit* accounts of meaning,
- we seek to characterise programs in terms of what we can say about them.

Example (SLA)

$$
\begin{aligned}
X + Y &= Y + X \\
X + 0 &= X \\
\mathtt{succ}(X) + Y &= \mathtt{succ}(X + Y) \\
&\vdots
\end{aligned}
$$

Main approaches to semantics: (III) Axiomatic

Relationship with other approaches

- We can use operational and/or denotational semantics to interpret the axioms and proof rules, and give a basis for proving them sound and complete.
- We can ask if the logic determines a specific model.

For example, $X + Y = Y + X$ is valid wrt the denotational semantics $[\![_]\!]$ we defined: for all N_1, N_2,

$$
\begin{aligned}
[\![N_1 + N_2]\!] &= [\![N_1]\!] +^{\mathbb{Z}} [\![N_2]\!] && \text{by definition of } [\![_]\!] \\
&= [\![N_2]\!] +^{\mathbb{Z}} [\![N_1]\!] && \text{by associativity of } +^{\mathbb{Z}} \\
&= [\![N_2 + N_1]\!] && \text{by definition of } [\![_]\!]
\end{aligned}
$$

We will not see more axiomatic semantics in this course.

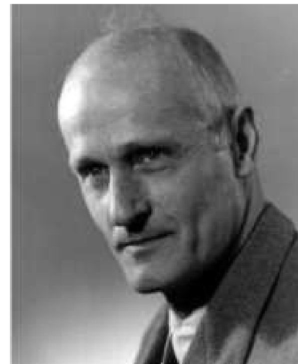
Outline

1. PCF, Turing completeness
2. Equivalence in PCF, complete partial orders
3. Sound models of PCF
4. Definability, logical relations
5. Full abstraction

PCF: Programming with Computable Functionals

Introduced by Gordon Plotkin in the 70's, based on Dana Scott's logic LCF.

It is the simplest **functional language** which is **Turing complete**, that is, it can express *all computable functions*.
As all functional languages, it is based on the *lambda-calculus*
(A. Church; H. Curry, S.C. Kleene, A. Turing, . . .).



PCF: types

Types give discipline to programs:

- by distinguishing between functions and function arguments, and
- making sure that functions are applied to correct kind of arguments.

The types for PCF are given by the grammar:

$$A \quad ::= \quad \texttt{bool} \mid \texttt{nat} \mid A_1 \to A_2$$

Notes:

- `bool` and `nat` are called **base types**; they correspond to boolean truth values and natural numbers respectively.
- $A_1 \to A_2$ are **function types**; they correspond to functions with domain A_1 and codomain A_2.

PCF: variables, contexts, constants

We fix a countably infinite set Var of **variables**. We range over variables by $x, y, z, \ldots$.

A **context** is a finite set of variable-type pairs:

$$\Gamma = \{\, x_1 : A_1, \ldots, x_n : A_n\}$$

If $x : A$ is *compatible* with Γ (i.e. $(x : B) \in \Gamma \implies A = B$), we write $\Gamma, x : A$ for the context $\Gamma \cup \{x : A\}$.

Moreover, we define a set $Cons$ of **constants**:

$$\mathtt{c} \;\; ::= \;\; \mathtt{t} \mid \mathtt{f} \mid n \mid \mathtt{zero?} \mid \mathtt{succ} \mid \mathtt{pred} \mid \mathtt{cond}_{\,\mathtt{bool}} \mid \mathtt{cond}_{\,\mathtt{nat}} \mid \mathbf{Y}_A$$

for each $n \in \mathbb{N}$ and each type A (so $Cons$ is also infinite).

PCF: typing

A **typing judgement** is a triple of the form:

$$\Gamma \vdash M : A$$

where M are going to be the terms of our language, PCF. The meaning of a typing judgement as above is:

> Given that x_1 has type A_1, x_2 has type A_2, ... and x_n has type A_n; we can deduce that M is a valid PCF term of type A.

The terms of PCF are produced by a set of rules, called **typing rules**, deriving such judgements.

A rule tree producing a typing judgement $\Gamma \vdash M : A$ is called a **typing derivation** of M.

PCF: terms

Here are the typing rules for PCF.

- Variables

$$\frac{}{\Gamma \vdash x : A}\ (x : A) \in \Gamma$$

- Constants

$$\frac{}{\Gamma \vdash \mathtt{t} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash \mathtt{f} : \mathtt{bool}} \qquad \frac{}{\Gamma \vdash n : \mathtt{nat}}$$

$$\frac{}{\Gamma \vdash \mathtt{succ}\ : \mathtt{nat} \to \mathtt{nat}} \quad \frac{}{\Gamma \vdash \mathtt{pred}\ : \mathtt{nat} \to \mathtt{nat}} \quad \frac{}{\Gamma \vdash \mathtt{zero?}\ : \mathtt{nat} \to \mathtt{bool}}$$

$$\frac{}{\Gamma \vdash \mathtt{cond}_{A_b} : \mathtt{bool} \to (A_b \to (A_b \to A_b))}\ A_b \in \{\mathtt{bool}, \mathtt{nat}\}$$

$$\frac{}{\Gamma \vdash \mathbf{Y}_A : (A \to A) \to A}$$

PCF: terms (ctd)

Here are the typing rules for PCF (ctd).

- Lambda-abstraction

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A . M : A \to B}$$

- Application

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B}$$

Observe that terms are the **typable** elements of the following grammar.

$$M, N \quad ::= \quad x \mid \mathsf{c} \mid \lambda x^A . M \mid M\, N$$

Example terms

- $\lambda x^A.x$

$$\dfrac{\overline{x : A \vdash x : A}}{\vdash \lambda x^A.x : A \to A}$$

This is the identity function (of type A): it takes an input and returns that as the output

- $\lambda x^A.(\lambda y^B.x)$

$$\dfrac{\dfrac{\overline{x : A, y : B \vdash x : A}}{x : A \vdash \lambda y^B.x : B \to A}}{\vdash \lambda x^A.(\lambda y^B.x) : A \to (B \to A)}$$

This is the first projection function: it takes two consecutive inputs and returns the first one.

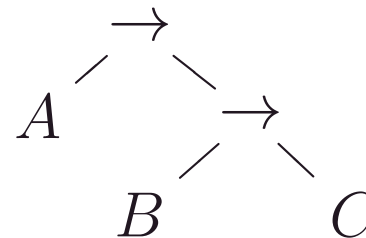
- $\lambda x^{\mathtt{nat}}.\mathtt{succ}\, x$

$$???$$

Save some notation

So far we have been using parentheses informally, in order to express our intended (*abstract*) syntax concretely. That is,

$A \to (B \to C)$ represents



We will continue using them, but be more economic about them:

- $A_1 \to A_2 \to A_3$ stands for $A_1 \to (A_2 \to A_3)$
  (arrows associate to the right)

- $M_1\,M_2\,M_3$ stands for $(M_1\,M_2)\,M_3$
  (applications associate to the left)

- $\lambda x.\,M_1\,M_2$ stands for $\lambda x.(M_1\,M_2)$; $\lambda x.\lambda y.M$ stands for $\lambda x.(\lambda y.M)$
  (scope of λ goes as much to the right as possible)

More on notation

- Thus, for example,

$$\lambda x_1^{A_1}.\lambda x_2^{A_2}\ldots\lambda x_n^{A_n}.\, M_1\, M_2 \cdots M_m$$
$$\equiv \lambda x_1^{A_1}.(\lambda x_2^{A_2}\ldots(\lambda x_n^{A_n}.\,(\cdots(M_1\, M_2)\cdots M_m))\cdots)$$

$$A_1 \to A_2 \to \cdots \to A_n \equiv (\cdots(A_1 \to A_2) \to \cdots) \to A_n$$

- Each type A can be uniquely written in one of the forms:

$$A = A_1 \to \cdots \to A_n \to \texttt{bool}$$
$$A = A_1 \to \cdots \to A_n \to \texttt{nat}$$

for some $n \in \mathbb{N}$ and types $A_1, \ldots, A_n$.

- We shall often omit type subscripts and superscripts and write just $\lambda x.M$, $\texttt{cond}\ M\ N_1\ N_2$, etc.

Variable binding

Variables appearing inside a term *under a* λ are called **bound**:

- x is **bound** in $\lambda x.M$, in the same way that y is bound in $\int f(y)dy$.

On the other hand, each term M has a set of **free variables**, $\mathsf{FV}(M)$, defined by:

$$\begin{aligned}
\mathsf{FV}(\mathsf{c}) &= \emptyset \\
\mathsf{FV}(x) &= \{x\} \\
\mathsf{FV}(M\ N) &= \mathsf{FV}(M) \cup \mathsf{FV}(N) \\
\mathsf{FV}(\lambda x.M) &= \mathsf{FV}(M) \setminus \{x\}
\end{aligned}$$

E.g:

$$\mathsf{FV}(\,x\lambda y.(\lambda z.z)xy\,) = \{x\}$$

A term with no free variables is called **closed**.

Alpha Conversion

Two terms M and N are said to be **α-convertible**, written $M \equiv_\alpha N$ or just $M \equiv N$, if:

- one can derive the same term from both purely by renaming bound variables to *fresh* variables.

Examples: $\lambda x.\,\texttt{succ}\,x \equiv \lambda y.\,\texttt{succ}\,y, \quad x(\lambda x.x) \equiv x(\lambda y.y), \ldots$

NB: We consider terms which are α-convertible to be identical at the syntactic level – to us they are **the same term**!

We adopt the **variable convention** (aka the *Barendregt convention*):

> *In any definition, theorem or proof in which only finitely or countably many terms appear, we silently α-convert them so that the bound variables of each term are not the same as the bound variables of any other term, or the free variables of any term.*

Alpha Conversion formally

Formally, we define *variable swapping* on terms as follows.

$$\begin{aligned} (y\ x) \bullet \mathsf{c} &\equiv \mathsf{c} \\ (y\ x) \bullet st &\equiv ((y\ x) \bullet s)((y\ x) \bullet t) \\ (y\ x) \bullet \lambda z.s &\equiv \lambda((y\ x) \bullet z).((y\ x) \bullet s) \end{aligned}$$

$$(y\ x) \bullet z \equiv \begin{cases} y & \text{if } z \equiv x \\ x & \text{if } z \equiv y \\ z & \text{otherwise} \end{cases}$$

Then, $\equiv_\alpha$ is the relation on terms defined by:

$$\frac{}{\mathsf{c} \equiv_\alpha \mathsf{c}} \qquad \frac{}{x \equiv_\alpha x} \qquad \frac{M \equiv_\alpha M' \quad N \equiv_\alpha N'}{M\,N \equiv_\alpha M'N'}$$

$$\frac{(y\ x) \bullet M \equiv_\alpha (y\ x') \bullet M'}{\lambda x.M \equiv_\alpha \lambda x'.M'}\ y \notin \mathsf{FV}(M, M')$$

Substitution

What is the result of *applying* a function $\lambda x.M$ to an argument N?
– Substitution!

We **substitute** of a term N for a variable x inside a term M as follows.

$$\mathsf{c}[N/x] \equiv \mathsf{c}$$

$$y[N/x] \equiv \begin{cases} y & \text{if } x \not\equiv y \\ N & \text{if } x \equiv y \end{cases}$$

$$(M_1\ M_2)[N/x] \equiv (M_1[N/x])(M_2[N/x])$$

$$(\lambda y.M)[N/x] \equiv \lambda y.(M[N/x]) \ \text{ assuming } y \not\equiv x \text{ and } y \notin \mathsf{FV}(N)$$

Note that we can only assume that $y \notin \mathsf{FV}(x, N)$, in the final clause, because of the variable convention. This kind of substitution is usually termed as *capture-avoiding*.

Fixpoints

The constant $\mathbf{Y}^A$ is a **fixpoint operator**:

$$\frac{\Gamma \vdash \mathbf{Y} : (A \to A) \to A \qquad \Gamma \vdash M : A \to A}{\Gamma \vdash \mathbf{Y} M : A} \quad \rightsquigarrow \quad M(\mathbf{Y} M) = \mathbf{Y} M$$

For example, consider the following recursive definition of addition.

$$n + m = \begin{cases} m & \text{if } n = 0 \\ \sigma(\pi(n) + m) & \text{if } n > 0 \end{cases}$$

where $\sigma, \pi : \mathbb{N} \to \mathbb{N}$ are the successor and predecessor functions respectively. In PCF, we can *define* a term for addition:

$$\mathtt{plus} \equiv \mathbf{Y}^{\mathtt{nat}\to\mathtt{nat}}(\lambda f^{\mathtt{nat}\to\mathtt{nat}}.\lambda x.\lambda y.\, \mathtt{cond}\ (\mathtt{zero?}\ x)\ y\ (\mathtt{succ}\,(f(\mathtt{pred}\,x)y))\,)$$

so $\mathtt{plus} \ = \ \lambda x.\lambda y.\, \mathtt{cond}\ (\mathtt{zero?}\ x)\ y\ (\mathtt{succ}\,(\mathtt{plus}\,(\mathtt{pred}\,x)y)))$

More fixpoints

Now consider the following recursive definition of multiplication.

$$n \times m = \begin{cases} 0 & \text{if } n = 0 \\ (\pi(n) \times m) + m & \text{if } n > 0 \end{cases}$$

where $\pi : \mathbb{N} \to \mathbb{N}$ is the predecessor function. In PCF, we can define:

$$\mathtt{mult} \equiv \mathbf{Y}^{\mathtt{nat}\to\mathtt{nat}}(\lambda f^{\mathtt{nat}\to\mathtt{nat}}.\lambda x.\lambda y.\,\mathtt{cond}\ (\mathtt{zero?}\ x)\ 0\ (\mathtt{plus}\,(f\,(\mathtt{pred}\,x)y)y))$$

$$\text{so} \quad \mathtt{mult} \ = \ \lambda x.\lambda y.\,\mathtt{cond}\ (\mathtt{zero?}\ x)\ 0\ (\mathtt{plus}\,(\mathtt{mult}\,(\mathtt{pred}\,x)y)y)$$

Now, what about the following function?

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (\pi(n))! & \text{if } n > 0 \end{cases}$$

Operational semantics

Define the set of closed **values**:

$$V \quad ::= \quad \mathsf{c} \mid \lambda x.M \qquad (\mathsf{FV}(\lambda x.M) = \emptyset)$$

The operational semantics is given via the **evaluation** relation, which relates closed terms to closed values and is given as follows.

- Values

$$\frac{}{V \Downarrow V}$$

- Application

$$\frac{M \Downarrow V' \quad V'N \Downarrow V}{M\,N \Downarrow V}\ M \text{ not a value}$$

- Substitution

$$\frac{M[N/x] \Downarrow V}{(\lambda x.M)N \Downarrow V}$$

Operational semantics (ctd)

- Arithmetic

$$\frac{M \Downarrow n}{\texttt{succ}\, M \Downarrow n+1} \qquad \frac{M \Downarrow n+1}{\texttt{pred}\, M \Downarrow n} \qquad \frac{M \Downarrow 0}{\texttt{pred}\, M \Downarrow 0}$$

$$\frac{M \Downarrow n+1}{\texttt{zero?}\ M \Downarrow \texttt{f}} \qquad \frac{M \Downarrow 0}{\texttt{zero?}\ M \Downarrow \texttt{t}}$$

- Conditionals

$$\frac{M \Downarrow \texttt{t}}{\texttt{cond}_{A_b}\, M \Downarrow \lambda x^{A_b}.\lambda y^{A_b}.x} \qquad \frac{M \Downarrow \texttt{f}}{\texttt{cond}_{A_b}\, M \Downarrow \lambda x^{A_b}.\lambda y^{A_b}.y}$$

- Fixpoints

$$\frac{M(\mathbf{Y}_A M) \Downarrow V}{\mathbf{Y}_A M \Downarrow V}$$

Derived rules for conditionals

In practice, it will be convenient to use the derived rules:

$$\frac{M \Downarrow \mathtt{t} \quad N_1 \Downarrow V}{\mathtt{cond}\ M\ N_1\ N_2 \Downarrow V} \qquad \frac{M \Downarrow \mathtt{f} \quad N_2 \Downarrow V}{\mathtt{cond}\ M\ N_1\ N_2 \Downarrow V}$$

These are derived e.g. as follows.

$$\dfrac{\dfrac{\dfrac{M \Downarrow \mathtt{t}}{\mathtt{cond}\ M \Downarrow \lambda x.\lambda y.x} \quad \dfrac{\overline{\lambda y.N_1 \Downarrow \lambda y.N_1}}{(\lambda x.\lambda y.x)N_1 \Downarrow \lambda y.N_1}}{(\mathtt{cond}\ M)\ N_1 \Downarrow \lambda y.N_1} \quad \dfrac{N_1 \Downarrow V}{(\lambda y.N_1)N_2 \Downarrow V}}{(\mathtt{cond}\ M\ N_1)\ N_2 \Downarrow V}$$

Derived rules for multiple application

$$\frac{M \Downarrow V' \quad V' N_1 \cdots N_m \Downarrow V}{M\, N_1 \cdots N_m \Downarrow V}$$

Induction on m. For $m = 0$, $M \Downarrow V' \equiv V$. Otherwise, if $m > 0$, $M \Downarrow V'$ and $V' N_1 \cdots N_m \Downarrow V$, then the latter must be due to some

$$\frac{V' N_1 \cdots N_{m-1} \Downarrow V'' \quad V'' N_m \Downarrow V}{V' N_1 \cdots N_m \Downarrow V}$$

By IH, $M\, N_1 \cdots N_{m-1} \Downarrow V''$, so

$$\frac{M\, N_1 \cdots N_{m-1} \Downarrow V'' \quad V'' N_m \Downarrow V}{M\, N_1 \cdots N_{m-1} N_m \Downarrow V}$$

Other derived rules:

$$\frac{M[N_1/x_1, \ldots, N_m/x_m] \Downarrow V}{(\lambda x_1 \ldots \lambda x_n.M)\, N_1 \cdots N_m \Downarrow V} \qquad \frac{M(\mathbf{Y} M) N_1 \cdots N_m \Downarrow V}{(\mathbf{Y} M)\, N_1 \cdots N_m \Downarrow V}$$

A simple example

We show that $(\lambda x^{\texttt{bool}}.\texttt{cond}\ x\ 24\ 42)(\texttt{zero?}\ 42) \Downarrow 42$.

$$\frac{\vdots}{(\lambda x^{\texttt{bool}}.\texttt{cond}\ x\ 24\ 42)(\texttt{zero?}\ 42) \Downarrow ??}$$

$$\frac{\dfrac{\vdots}{(\texttt{cond}\ x\ 24\ 42)[\texttt{zero?}\ 42/x] \Downarrow ??}}{(\lambda x^{\texttt{bool}}.\texttt{cond}\ x\ 24\ 42)(\texttt{zero?}\ 42) \Downarrow ??}$$

$$\frac{\dfrac{\vdots}{\texttt{cond}\ (\texttt{zero?}\ 42)\ 24\ 42 \Downarrow ??}}{(\lambda x^{\texttt{bool}}.\texttt{cond}\ x\ 24\ 42)(\texttt{zero?}\ 42) \Downarrow ??}$$

$$\frac{\dfrac{\vdots}{\texttt{zero?}\ 42 \Downarrow ??} \quad ??}{\texttt{cond}\ (\texttt{zero?}\ 42)\ 24\ 42 \Downarrow ??}$$

A simple example

We show that $(\lambda x^{\texttt{bool}}.\texttt{cond}\ x\ 24\ 42)(\texttt{zero?}\ 42) \Downarrow 42$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{42 \Downarrow 42}}{\texttt{zero?}\ 42 \Downarrow \texttt{f}} \quad \dfrac{}{42 \Downarrow 42}}{\texttt{cond}\ (\texttt{zero?}\ 42)\ 24\ 42 \Downarrow 42}}{(\lambda x^{\texttt{bool}}.\texttt{cond}\ x\ 24\ 42)(\texttt{zero?}\ 42) \Downarrow 42}$$

Evaluation is deterministic

Lemma. *For any term* $\vdash M : A$ *and values* V, V', *if* $M \Downarrow V$ *and* $M \Downarrow V'$ *with derivations* $\mathcal{D}$ *and* $\mathcal{D}'$ *respectively we have* $\mathcal{D} \equiv \mathcal{D}'$.

Proof. By induction on the size of $\mathcal{D}$. The base case is for $\mathcal{D}$ being a value axiom: $M \equiv V$. Clearly, this is the only derivation for M.
For the inductive step, it must be the case that $M \equiv M_1 M_2$.
If M_1 is not a value, then $\mathcal{D}$ and $\mathcal{D}'$ must end respectively in:

$$\frac{M_1 \Downarrow U \quad U M_2 \Downarrow V}{M_1 M_2 \Downarrow V} \qquad \frac{M_1 \Downarrow U' \quad U' M_2 \Downarrow V'}{M_1 M_2 \Downarrow V'}$$

By IH, the derivation of $M_1 M_2 \Downarrow V$ is the same as that of $M_1 M_2 \Downarrow V'$, and the derivation of $U M_2 \Downarrow V$ is the same as that of $U' M_2 \Downarrow V'$.
Thus, $\mathcal{D} = \mathcal{D}'$. Now, if $M_1 \equiv \mathsf{c}$ then $\mathcal{D}, \mathcal{D}'$ must have the form:

$$\frac{M_2 \Downarrow V_2}{\mathsf{c} M_2 \Downarrow V} \qquad \frac{M_2 \Downarrow V_2'}{\mathsf{c} M_2 \Downarrow V'}$$

and we use the IH as before.

Evaluation is deterministic (ctd)

Lemma. *For any term* $\vdash M : A$ *and values* V, V'*, if* $M \Downarrow V$ *and* $M \Downarrow V'$ *with derivations* $\mathcal{D}$ *and* $\mathcal{D}'$ *respectively we have* $\mathcal{D} \equiv \mathcal{D}'$*.*

Proof. For the inductive step, it must be the case that $M \equiv M_1 M_2$. Finally, if $M_1 \equiv \lambda x.M'$ then $\mathcal{D}, \mathcal{D}'$ must have the form:

$$\frac{M'[M_2/x] \Downarrow V}{(\lambda x.M')M_2 \Downarrow V} \qquad \frac{M'[M_2/x] \Downarrow V'}{(\lambda x.M')M_2 \Downarrow V'}$$

and we use the IH as before. $\square$

Observe that in the proof above we used the fact that, according to the form of the term M, there is a *unique* evaluation rule which may apply to it.

Evaluation is a congruence

Lemma. *Suppose FV*$(M) \subseteq \{x\}$ *and* $N \Downarrow V$. *Then,* $M[N/x] \Downarrow \mathsf{c}$ *iff* $M[V/x] \Downarrow \mathsf{c}$.

The lemma is a corollary of a result which uses contexts and will be proved in the next lecture.

Another example

Let $\texttt{fact} \equiv \mathbf{Y}(\lambda f.\lambda x.\, M)$

$$M \equiv \texttt{cond}\ (\texttt{zero?}\ x)\ 1\ (\texttt{mult}\, x(f(\texttt{pred}\, x)))\,.$$

We show that $\texttt{fact}\, n \Downarrow n!$, for every $n \in \mathbb{N}$, by induction on n.

- For the base case:

$$\dfrac{\dfrac{\dfrac{\overline{(\lambda x.M)[\texttt{fact}\,/f] \Downarrow \lambda x.M[\texttt{fact}\,/f]}}{(\lambda f.\lambda x.M)\texttt{fact} \Downarrow \lambda x.M[\texttt{fact}\,/f]}}{\texttt{fact} \Downarrow \lambda x.M[\texttt{fact}\,/f]} \qquad \dfrac{\dfrac{\mathcal{D}}{M[\texttt{fact}\,/f, 0/x] \Downarrow 1}}{(\lambda x.M[\texttt{fact}\,/f])0 \Downarrow 1}}{\texttt{fact}\, 0 \Downarrow 1}$$

$$\mathcal{D}: \quad \dfrac{\dfrac{\overline{0 \Downarrow 0}}{\texttt{zero?}\ 0 \Downarrow \texttt{t}} \quad \overline{1 \Downarrow 1}}{\texttt{cond}\ (\texttt{zero?}\ 0)\ 1\ (\texttt{mult}\ \cdots) \Downarrow 1}$$

Another example (ctd)

Let $\texttt{fact} \equiv \mathbf{Y}(\lambda f.\lambda x.\, M)$

$$M \equiv \texttt{cond}\ (\texttt{zero?}\ x)\ 1\ (\texttt{mult}\, x(f(\texttt{pred}\, x)))\,.$$

We show that $\texttt{fact}\, n \Downarrow n!$, for every $n \in \mathbb{N}$, by induction on n.

- For the inductive step:

$$\dfrac{\dfrac{\vdots}{\texttt{fact} \Downarrow \lambda x.M[\texttt{fact}\,/f]} \qquad \dfrac{\dfrac{\mathcal{D}}{M[\texttt{fact}\,/f,(n+1)/x] \Downarrow (n+1)!}}{(\lambda x.M[\texttt{fact}\,/f])(n+1) \Downarrow (n+1)!}}{\texttt{fact}\,(n+1) \Downarrow (n+1)!}$$

$$\mathcal{D}: \quad \dfrac{\dfrac{\dfrac{}{n+1 \Downarrow n+1}}{\texttt{zero?}\ n{+}1 \Downarrow \texttt{f}} \qquad \dfrac{\dfrac{}{n+1 \Downarrow n+1} \qquad \dfrac{\texttt{fact}\, n \Downarrow n! \qquad \dfrac{\dfrac{}{n+1 \Downarrow n+1}}{\texttt{pred}\, n{+}1 \Downarrow n}}{\texttt{fact}\,(\texttt{pred}\, n{+}1) \Downarrow n!}\,(Cong.)}{\texttt{mult}\,(n+1)\,(\texttt{fact}\,(\texttt{pred}\, n{+}1)) \Downarrow (n+1)!}\,(Ex.)}{\texttt{cond}\ (\texttt{zero?}\ n{+}1)\ 1\ (\texttt{mult}\,(n+1)(\texttt{fact}\,(\texttt{pred}\, n{+}1))) \Downarrow (n+1)!}$$

Divergence

We say that a closed term M **converges**, written $M \Downarrow$, if there is a value V such that $M \Downarrow V$. If no such value exists, M **diverges**, written $M \Uparrow$.

Because of recursion, in PCF there is divergence at every type. The simplest example is the term:

$$\Omega_A \equiv \mathbf{Y}_A(\lambda x^A.x)$$

It is easy to show $\Omega \Uparrow$. For suppose $\Omega \Downarrow V$, some value V. Then, by determinacy, the derivation must be of the form:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Omega \Downarrow V}}{(\lambda x.x)\Omega \Downarrow V}}{\Omega \Downarrow V}$$

which would lead to an infinite derivation.

Expressiveness

How expressive is PCF? What functions can be represented in it?

We say that a function $\phi : \mathbb{N}^m \to \mathbb{N}$ is **definable** by a term $\vdash M : \underbrace{\texttt{nat} \to \cdots \to \texttt{nat}}_{m} \to \texttt{nat}$ if, for every tuple $\langle n_1, \ldots, n_m \rangle \in \mathbb{N}^m$:

$$M\, n_1 \ldots n_m \Downarrow \phi(n_1, \ldots, n_m)$$

Examples:

- The function $\sigma : \mathbb{N} \to \mathbb{N} = n \mapsto n + 1$ is definable by `succ` .
- The function $\phi : \mathbb{N} \to \mathbb{N} = [0 \mapsto 0,\ n + 1 \mapsto n]$ is definable by `pred` .
- The function $\zeta : \mathbb{N} \to \mathbb{N} = n \mapsto 0$ is definable by $\lambda x^{\texttt{nat}}.0$.
- The function $\Pi^i_j : \mathbb{N}^i \to \mathbb{N} = \langle n_1, \ldots, n_i \rangle \mapsto n_j$, for $1 \leq j \leq i$, is definable by $\lambda x_1^{\texttt{nat}} \ldots \lambda x_i^{\texttt{nat}}.\ x_j$.

Partiality

Because of divergence, in PCF we can also represent **partial functions**.

We say that a partial function $\phi : \mathbb{N}^m \rightharpoonup \mathbb{N}$ is **definable** by a term $\vdash M : \underbrace{\mathtt{nat} \to \cdots \to \mathtt{nat}}_{m} \to \mathtt{nat}$ if, for every tuple $\langle n_1, \ldots, n_m \rangle \in \mathbb{N}^m$:

$$M\, n_1 \ldots n_m \begin{cases} \Downarrow \phi(n_1, \ldots, n_m) & \text{if } \phi(n_1, \ldots, n_m) \text{ defined} \\ \Uparrow & \text{if } \phi(n_1, \ldots, n_m) \text{ undefined} \end{cases}$$

We say that ϕ is PCF-definable if it definable by some (closed) PCF term.

Partial recursive functions

We will show that PCF-definable functions are precisely the partial recursive functions (as in Recursion Theory).

The set of **partial recursive functions** is the least set of partial functions $\phi : \mathbb{N}^n \rightharpoonup \mathbb{N}$ which:

- contains the *initial functions*,
- is closed under *function composition*,
- is closed under *primitive recursion*,
- is closed under *minimalisation*.

By the **Church-Turing thesis**, this class of functions represents the set of all **computable** functions. As a result, PCF is **Turing complete**.

Exercises

1. Prove the following properties of typing.
   - If $\Gamma \vdash M : A$ and $x : B$ is compatible with Γ then $\Gamma, x : B \vdash M : A$ (*Weakening*).
   - If a term M has a typing derivation $\Gamma \vdash M : A$ then the latter is unique (*Unique typing*).
2. Find the bug in the following evaluation.

$$\dfrac{\dfrac{\overline{\lambda y.y \Downarrow \lambda y.y}}{(\lambda x.\lambda y.x)y \Downarrow \lambda y.y} \quad \dfrac{\overline{z \Downarrow z}}{(\lambda y.y)z \Downarrow z}}{(\lambda x.\lambda y.x)yz \Downarrow z}$$

3. Show the derived evaluation rules for multiple application for the cases of $\lambda x_1 \ldots \lambda x_m.M$ and $\mathbf{Y}M$.
4. Show that, for any $n, m \in \mathbb{N}$, $\texttt{plus}\, m\, n \Downarrow m + n$ and $\texttt{mult}\, m\, n \Downarrow m \times n$.