Lecture 3: Functional languages

Operational Semantics & Typing systems

Simon Castellan

SOS, Master Recherche Science Informatique, U. Rennes 1

2020-2021

Functional languages

- Functions become first-class objects, eg.

```
let rec map f = function
| [] -> []
| t :: q -> f t :: map f q
in
map (fun x -> x + 1) [1; 2; 3]
```

- Data is usually structured around sum types and product types:

```
type bool = True | False
type coord = int * int
```

- Examples: OCaml, Haskell, Scala, ...
- Most of them come with a **typing system** preventing easy mistakes:

```
map (fun x -> x + 1) ["foo"]
```

Typing systems are basic forms of automatic program analysis:

Well-typed programs do not go wrong.

This lecture

1. Presentation of the λ-calculus, an idealised core functional language

   It's functions all the way down

2. Presentation of the simple types and their properties
3. Discuss some extensions.

References

- Benjamin C. Pierce : Types and Programming Languages. MIT Press, 2002
- Luca Cardelli. Type systems. Handbook of Computer Science and Engineering. CRC Press, 1996.

Outline

1. Lambda calculus
   - Syntax
   - Semantics
2. Simply Typed Lambda Calculus
3. Extensions

Outline

1. Lambda calculus
   - Syntax
   - Semantics

2. Simply Typed Lambda Calculus

3. Extensions

The lambda calculus

- Idealised functional languages focussing on functions:
  - Function application: $f\ x$
  - Function definition $(\lambda\ \mathbf{x}.\ ...)$
  - Rules for evaluating functions called β**-reduction**

$$(\lambda x.\, x+1)\, 2 \to 2+1 \qquad (\to 3)$$

The lambda calculus

- Idealised functional languages focussing on functions:
  - Function application: $f\ x$
  - Function definition $(\lambda\ \mathbf{x}. \dots)$
  - Rules for evaluating functions called β**-reduction**

$$(\lambda x.\, x + 1)\, 2 \to 2 + 1 \qquad (\to 3)$$

- Lambda calculus introduced by Church in 30's concurrently with Turing machine.
- Original motivation: foundations of mathematics.

Syntax of the pure λ-calculus

Expressions (or terms):

$$\begin{array}{rcll} t & ::= & \mathtt{x} & \textit{variable} \\ & | & (\lambda\, \mathtt{x}.\, t) & \textit{lambda abstraction} \\ & | & (t\ t) & \textit{application} \end{array}$$

Some notations:

- $\lambda\ \mathtt{x}\ \mathtt{y}.t$ for $\lambda\ \mathtt{x}.(\lambda\ \mathtt{y}.\ t)$.
- Application has higher precedence than abstraction: $\lambda\ \mathtt{x}.\ \mathtt{x}\ \ \mathtt{y}$ reads $\lambda\ \mathtt{x}.(\ \mathtt{x}\ \ \mathtt{y})$
- Application is left-associative: $t_1\ t_2\ t_3$ reads $((t_1\ t_2)\ t_3)$
- Abstraction right-associative: $\lambda\, \mathtt{x}.\ \lambda\, \mathtt{y}.\ \lambda\, \mathtt{z}.\ \ldots$ reads $(\lambda\, \mathtt{x}.\ (\lambda\, \mathtt{y}.\ (\lambda\, \mathtt{z}.\ \ldots)))$

Syntax of the pure λ-calculus

Expressions (or terms):

$$
\begin{array}{rcll}
t & ::= & \mathtt{x} & \textit{variable} \\
 & | & (\lambda\, \mathtt{x}.\, t) & \textit{lambda abstraction} \\
 & | & (t\ t) & \textit{application}
\end{array}
$$

Some notations:

- $\lambda\ \mathtt{x}\ \mathtt{y}.t$ for $\lambda\ \mathtt{x}.(\lambda\ \mathtt{y}.\ t)$.
- Application has higher precedence than abstraction: $\lambda\ \mathtt{x}.\ \mathtt{x}\ \ \mathtt{y}$ reads $\lambda\ \mathtt{x}.(\ \mathtt{x}\ \ \mathtt{y})$
- Application is left-associative: $t_1\ t_2\ t_3$ reads $((t_1\ t_2)\ t_3)$
- Abstraction right-associative: $\lambda\ \mathtt{x}.\ \lambda\ \mathtt{y}.\ \lambda\ \mathtt{z}.\ \ldots$ reads $(\lambda\ \mathtt{x}.\ (\lambda\ \mathtt{y}.\ (\lambda\ \mathtt{z}.\ \ldots)))$

- Minimal language, but still Turing-complete!
- In particular: integers, and structured types can be encoded in it, eg. integer $n \in \mathbb{N}$ is encoded as $\lambda f.\lambda x.f(f \ldots (fx))$ with n applications of f.
- For now, we consider the **untyped version**.

Examples of terms

- The identity function `id` = λ `x. x`
- Constant functions K = λ `x.` λ `y. x`
- Generalized application S = λ `f.` λ `g.` λ `x. f x ( g x)`
- Double : λ `f.` λ `x. f ( f x)`
- Omega = (λ `x. x x`) (λ `x. x x`)

Variable scope (1/2)

Definition 1 (Variable binding)

An abstraction (λ `x.` `t`) **binds** the variable `x` in its body `t`.
Variable `x` is then said to be **bound** in `t`.

The variables bound by lambdas are “placeholders” and can be renamed without changing the term.

Example: (λ `x. x`) and (λ `y. y`) represent the same function.

Definition 2 (α equivalence, informally)

Lambda terms that are equal up to renaming of bound variables are said to be **alpha-equivalent**.

NB: In the following, we will consider Lambda terms modulo α-equivalence.

Variable scope (2/2)

A variable is free in a term if is not bound by an enclosing abstraction.

Definition 3 (Free variables)

The set $FV(t)$ of **free variables** of a term t is inductively defined as

$$\begin{array}{lcl} FV(\,\mathbf{x}\,) & = & \{\,\mathbf{x}\,\} \\ FV(t_1\ t_2) & = & FV(t_1) \cup FV(t_2) \\ FV(\lambda\,\mathbf{x}.t) & = & FV(t) \setminus \{\,\mathbf{x}\} \end{array}$$

Definition 4 (Closed term)

A lambda term is said to be closed if it has no free variable.

Operational semantics: β-reduction

The (only) computation step is β-reduction: calling a function.

Definition 5 (β-reduction)

$$(\lambda\, \mathtt{x}.t)\ u \to_\beta t[\,\mathtt{x} := u]$$

Definition 6 (Substitution)

$t[\,\mathtt{x} := u]$ denotes the term t in which we substituted u for variable $\mathtt{x}$.

$$\begin{array}{rcll}
\mathtt{x}[\,\mathtt{x} := t] & = & t & \\
\mathtt{y}[\,\mathtt{x} := t] & = & \mathtt{y} & \\
t_1\ t_2[\,\mathtt{x} := t] & = & t_1[\,\mathtt{x} := t]\ t_2[\,\mathtt{x} := t] & \\
(\lambda\, \mathtt{y}.t_1)[\,\mathtt{x} := t] & = & \lambda\, \mathtt{y}.\ t_1[\,\mathtt{x} := t] & \qquad \mathtt{y} \neq \mathtt{x} \text{ and } \mathtt{y} \notin FV(t)
\end{array}$$

and **alpha conversion** of $(\lambda\, \mathtt{y}.t_1)$ to make side condition satisfied.

Side conditions for binders:

- don't break abstractions: $(\lambda\, \mathtt{x}.\lambda\, \mathtt{x}.\ \mathtt{x})\ \mathtt{y} \not\to_\beta \lambda\, \mathtt{x}.(\ \mathtt{x}[\,\mathtt{x} := \mathtt{y}]) = \lambda\, \mathtt{x}.\ \mathtt{y}$
- avoid variable capture: $(\lambda\, \mathtt{x}.\lambda\, \mathtt{z}.\ \mathtt{x})\ \mathtt{z} \not\to_\beta \lambda\, \mathtt{z}.(\ \mathtt{x}[\,\mathtt{x} := \mathtt{z}]) = \lambda\, \mathtt{z}.\ \mathtt{z}$

β-reduction: exercises

Suppose for the moment that β-reduction can apply anywhere in a term.

Exercise 2.1 (In class)

Show that Omega reduces to itself.

Exercise 2.2

Reduce the lambda expression $(S \quad K) \quad K$.
Indication: don't expand the definition of K too early.

β-reduction: Confluence

Definition 7 (Redex)

A sub-term of the form $(\lambda x.t)\ u$ is called a **redex** (reducible expression). This is where β-reduction rule applies.

Definition 8 (β normal form)

A term is in **normal form** if no β-reduction can apply *inside* the term.

A term may have **more than one** redex. Ex: the term $\underline{\mathtt{id}\ (\underline{\mathtt{id}\ (\lambda\,\mathtt{z}.\underline{\mathtt{id}\ \mathtt{z}})})}$ where $\mathtt{id} = \lambda\ \mathtt{x}.\ \mathtt{x}$ contains 3 redexes. There are thus different **strategies** for evaluating a lambda term.

Theorem 9 (Church-Rosser or confluence for $\rightarrow_\beta^*$)

If $t \rightarrow_\beta^* t_1$ and $t \rightarrow_\beta^* t_2$, then there exists t' such that $t_1 \rightarrow_\beta^* t'$ and $t_2 \rightarrow_\beta^* t'$.

This implies the unicity of β-normal forms, modulo α-equivalence.

Operational semantics : evaluation order

There are different **strategies** for evaluating a lambda term.

Full β-reduction.

Non-deterministically reduces any possible redex.

Normal order reduction.

Reduce the left-most outer-most[1] redex first.

Intuition: don't evaluate arguments before the function is actually called.

Example: $\underline{\mathtt{id}\ (\mathtt{id}\ (\lambda\, \mathtt{z}.\mathtt{id}\ \mathtt{z}))} \rightarrow \underline{\mathtt{id}\ (\lambda\, \mathtt{z}.\mathtt{id}\ \mathtt{z})} \rightarrow \lambda\, \mathtt{z}.\underline{(\mathtt{id}\ \mathtt{z})} \rightarrow \lambda\, \mathtt{z}.\ \mathtt{z}$

Call-by-name (and the memoized variant of call-by-need).

Normal order reduction but never under a λ-abstraction.

Example: $\underline{\mathtt{id}\ (\mathtt{id}\ (\lambda\, \mathtt{z}.\mathtt{id}\ \mathtt{z}))} \rightarrow \underline{\mathtt{id}\ (\lambda\, \mathtt{z}.\mathtt{id}\ \mathtt{z})} \rightarrow \lambda\, \mathtt{z}.(\mathtt{id}\ \mathtt{z})$

[1] A outer-most redex is a redex not contained in another redex

Operational semantics : evaluation order

Call-by-value.

Intuitively: evaluate the arguments to functions before applying the function.

Evaluate outermost redex whose argument (right term) is in normal form.
No evaluation under λ-abstractions.

In our example, call-by-value leads to:

$$\mathtt{id}\ (\underline{\mathtt{id}\ (\lambda\, z.\mathtt{id}\ z)}) \to \underline{\mathtt{id}\ (\lambda\, z.\mathtt{id}\ z)} \to \lambda\, z.\mathtt{id}\ z$$

Most programming languages use this strategy (simpler to implement, predictable wrt. side-effects)

Try the lambda calculus reduction workbench!
`http://www.itu.dk/people/sestoft/lamreduce/index.html`

Operational semantics with CBV

Exercise 2.3 (In class)

Define a transition system for the pure lambda calculus such that its transition relation follows the call-by-value strategy.
Is it equivalent to call-by-name?

Running λ-terms: Closures.

Executing λ-terms using β-reduction is not **efficient** due to **substitutions**.

Running λ-terms: Closures.

Executing λ-terms using β-reduction is not **efficient** due to **substitutions**.

- A first idea: add an environment $\rho \in \mathbf{Env} := \mathbf{Var} \to \mathbf{Term}$:

$$\begin{array}{rcl} \langle (\lambda x.\, t)\; u \rangle & \to & \langle t, \rho[x := u] \rangle \\ \langle x, \rho \rangle & \to & \langle \rho(x), \mathbf{??} \rangle \\ & \dots & \end{array}$$

Running λ-terms: Closures.

Executing λ-terms using β-reduction is not **efficient** due to **substitutions**.

- A first idea: add an environment $\rho \in$ **Env** := **Var** → **Term**:

$$\begin{array}{rcl} \langle (\lambda x.\, t)\, u \rangle & \to & \langle t, \rho[x := u] \rangle \\ \langle x, \rho \rangle & \to & \langle \rho(x), \rho \rangle \\ & \ldots & \end{array}$$

  Problem: x evaluates in an invalid environment:

$$\langle (\lambda y.\lambda f.\, f\, 1)\, 2\, (\lambda x.\, x + y), \qquad y \mapsto 0 \rangle$$

Running λ-terms: Closures.

Executing λ-terms using β-reduction is not **efficient** due to **substitutions**.

- A first idea: add an environment $\rho \in \mathbf{Env} := \mathbf{Var} \to \mathbf{Term}$:

$$\begin{array}{rcl} \langle (\lambda x.\, t)\, u \rangle & \to & \langle t, \rho[x := u] \rangle \\ \langle x, \rho \rangle & \to & \langle \rho(x), \rho \rangle \\ & \dots & \end{array}$$

  Problem: x evaluates in an invalid environment:

$$\langle (\lambda y. \lambda f. f\ 1)\ 2\ (\lambda x.\, x + y), \qquad y \mapsto 0 \rangle$$

- We use **closures**:

$$\begin{array}{rcl} \mathbf{Closure} & := & \mathbf{Term} \times \mathbf{Env} \\ \mathbf{Env} & := & \mathbf{Var} \to \mathbf{Closure} \end{array}$$

$$\begin{array}{rcl} \langle (\lambda x.\, t)\, u, \rho \rangle & \to & \langle t, \rho[x := (u, \rho)] \rangle \\ \langle x, \rho \rangle & \to & \rho(x) \end{array}$$

  In the previous example, the environment stores the closure

$$(\lambda x. x + y, y \mapsto 0)$$

Outline

1. Lambda calculus
    - Syntax
    - Semantics
2. Simply Typed Lambda Calculus
3. Extensions

Outline

1. Lambda calculus
   - Syntax
   - Semantics
2. Simply Typed Lambda Calculus
3. Extensions

Problems of the untyped λ-calculus

- **Historical problem**: some terms diverge.
- **Pragmatic problem**: When extending with concrete datatypes, lots of blocking configurations: e.g. $1\,2$

$\leadsto$ Simple-type discipline: typing system guaranteeing:

- All programs terminate (we lose Turing-completeness)
- No unwanted blocking configurations with datatypes.

What is a type system

- Type theory was invented to eliminate certain logical paradoxes by classifying certain logical constructions as non-sense.
- Static semantics (can be computed without running the program)
- Lots of languages have some kind of type systems: C, Ada, Caml, Java.
- Others, like LISP and Prolog, are **un-typed** languages (even though typed versions exist).
- A possible definition of a type system:

  > *A* ***type system*** *is a syntactic and efficient method for proving the absence of certain kinds of program behaviour, by classifying expressions according to the value they compute.*

What are types used for?

1. **Error detection**. Application of a function to wrong number of arguments, application of integer functions to floats, use of undeclared variables in expressions, functions that do not return values, division by zero array indices out of bounds...
2. **Abstraction**. Facilitate the structuring of program into modules.
3. **Documentation**.
4. **Language safety**. Is the level of abstraction promised by a high-level language really ensured (eg. no low-level access to elements of an array)? Caml is safe; C isn't.
5. **Performance**. Information about the type of an expression enables the compiler to generate more efficient code (optimal choice of numerical operators, elimination of certain run-time checks).
6. **Program static analysis** : more generally, types keep track of static information about run-time values. A type checker can prevent insecure information flows, nonterminating recursion, or sorting algorithms that don't sort...

The simply-typed lambda calculus

We consider an extended version with booleans and integers.

Syntax:

Terms :

$$\begin{array}{rcl} t & ::= & \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}\ t\ \mathtt{then}\ t\ \mathtt{else}\ t \\ & \mid & \mathtt{0} \mid \mathtt{succ}\ t \mid \mathtt{pred}\ t \mid \mathtt{iszero}\ t \\ & \mid & \mathtt{x} \\ & \mid & \lambda\, \mathtt{x} : A.\, t \\ & \mid & t\ t \end{array}$$

with types annotating abstraction variables.

(Simple) Types :

$$A \quad ::= \quad \mathtt{Nat} \mid \mathtt{Bool} \mid A \to A.$$

Operational semantics

States : terms

Values (final configurations) :

$$
\begin{array}{lcl}
nv & ::= & 0 \mid \texttt{succ}\ (nv) \\
v & ::= & nv \mid \texttt{true} \mid \texttt{false} \mid \lambda\, \texttt{x} : A.t
\end{array}
$$

Transition relation:

$$\texttt{pred}\ 0 \to 0 \qquad \texttt{pred}\ (\ \texttt{succ}\ nv) \to nv$$

$$\texttt{iszero}\ 0 \to \texttt{true} \qquad \texttt{iszero}\ (\ \texttt{succ}\ nv) \to \texttt{false}$$

$$\frac{t \to t'}{\texttt{succ}\ t \to \texttt{succ}\ t'} \qquad \frac{t \to t'}{\texttt{pred}\ t \to \texttt{pred}\ t'} \qquad \frac{t \to t'}{\texttt{iszero}\ t \to \texttt{iszero}\ t'}$$

$$\texttt{if true then}\ t_1\ \texttt{else}\ t_2 \to t_1 \qquad \texttt{if false then}\ t_1\ \texttt{else}\ t_2 \to t_2$$

$$\frac{t \to t'}{\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \to \texttt{if}\ t'\ \texttt{then}\ t_1\ \texttt{else}\ t_2}$$

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \qquad \frac{t_2 \to t_2'}{v\ t_2 \to v\ t_2'} \qquad (\lambda\, \texttt{x} : A.t)\ v \to t[\ \texttt{x} := v]$$

Typing rules

Judgments of the form

$$\Gamma \vdash t : A \qquad \text{“term } t \text{ has type } A \text{ in the context } \Gamma\text{”}$$

Γ is an *context* $\{\, \mathtt{x} : A_1,\ \mathtt{y} : A_2, \ldots\}$, in which variables appear at most once.

The ternary relation $\Gamma \vdash t : A$ is defined inductively by a rule system.
Justifying the well-typing of an expression : exhibit a finite derivation tree.

$$\textsc{Tt}\,\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{Bool}} \qquad \textsc{Tf}\,\frac{}{\Gamma \vdash \mathtt{false} : \mathtt{Bool}} \qquad \textsc{Tz}\,\frac{}{\Gamma \vdash 0 : \mathtt{Nat}}$$

$$\textsc{Tp}\,\frac{\Gamma \vdash t : \mathtt{Nat}}{\Gamma \vdash \mathtt{pred}\, t : \mathtt{Nat}} \qquad \textsc{Tiz}\,\frac{\Gamma \vdash t : \mathtt{Nat}}{\Gamma \vdash \mathtt{iszero}\, t : \mathtt{Bool}}$$

$$\textsc{Tsu}\,\frac{\Gamma \vdash t : \mathtt{Nat}}{\Gamma \vdash \mathtt{succ}\, t : \mathtt{Nat}} \qquad \textsc{Tif}\,\frac{\Gamma \vdash t : \mathtt{Bool} \qquad \Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash \mathtt{if}\, t\, \mathtt{then}\, t_1\, \mathtt{else}\, t_2 : A}$$

$$\textsc{Tvar}\,\frac{\mathtt{x} : A \in A}{\Gamma \vdash \mathtt{x} : A} \qquad \textsc{Tabs}\,\frac{\Gamma, \mathtt{x} : A \vdash t : B}{\Gamma \vdash \lambda\, \mathtt{x} : A.t : A \to B}$$

$$\textsc{Tapp}\,\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1\ t_2 : B}$$

Exercises

Exercise 3.1 (In class)

Show that $(\lambda\ \texttt{x}\!: \texttt{Bool}\ .\ \texttt{x})\ \texttt{true}$ *is well-typed*

Exercise 3.2 (In class)

Check that

$$\texttt{f}\!: \texttt{Bool} \to \texttt{Bool} \vdash$$
$$\lambda\ \texttt{x}\!: \texttt{Bool}\ .\ \texttt{f}(\,\texttt{if}\ \texttt{x}\ \texttt{then}\ \texttt{false}\ \texttt{else}\ \texttt{x})\,:\ \texttt{Bool} \to \texttt{Bool}$$

Type safety

Theorem 10 (Type safety)

For all well-typed term t, either

1. *t diverges: there is an infinite sequence of reduction starting from t*
2. *t normalises to a value.*

This means that reduction will not get stuck on for instance `succ true`.

Type safety

Theorem 10 (Type safety)

For all well-typed term t, either

1. *t diverges: there is an infinite sequence of reduction starting from t*
2. *t normalises to a value.*

This means that reduction will not get stuck on for instance `succ true`.

Lemma 11 (Progress)

If $\vdash t : A$ then either t is a value, or it can do a reduction $t \to t'$.

Lemma 12 (Subject reduction)

If $\Gamma \vdash t : A$ and $t \to t'$ then $\Gamma \vdash t' : A$.

Some properties of the type system

Lemma 13

1. *If* $\Gamma \vdash \texttt{true} : A$*, then* $A = \texttt{Bool}$.
2. *If* $\Gamma \vdash \texttt{succ}\, t : A$*, then* $A = \texttt{Nat}$ *and* $\Gamma \vdash t : \texttt{Nat}$.
3. $\ldots$

Some properties of the type system

Lemma 13

1. *If* $\Gamma \vdash$ `true` $: A$*, then* $A =$ `Bool` .
2. *If* $\Gamma \vdash$ `succ` $t : A$*, then* $A =$ `Nat` *and* $\Gamma \vdash t :$ `Nat` .
3. ...

Lemma 14 (Unicity of types)

Let Γ *be a context and let* t *be a term with all free variables defined in* Γ*. Then, there exists* ***at most*** *one type* T *such that* $\Gamma \vdash t : T$*.*

Lemma 15 (Canonical forms)

Consider a well-typed closed value $\vdash v : A$*.*

- *If* $A =$ `Nat` *, then* v *is a numerical value (defined by* $nv ::= 0 \mid$ `succ` nv*).*
- *If* $A =$ `Bool` *, then* v *is either* `true` *or* `false`
- *If* $A = A_1 \to A_2$*, then* v *is of the form form* $\lambda\, \mathtt{x} : A_1 . t$ *with* $\mathtt{x} : A_1 \vdash t : A_2$*.*

Progress (proof 1/2)

Theorem (Progress)

*For any closed $\vdash t : A$, **either** t is a value **or** there exists t' such that $t \to t'$.*

Proof: By induction on the derivation of the typing $\vdash t : A$.

Base cases are either values or non-closed terms. Immediate.

Case $\texttt{succ}\ t$. In this case, $T = \texttt{Nat}$ and we have that $t : \texttt{Nat}$. By induction hypothesis, two possibilities. Either t is a value and, by the Canonical Form lemma, an integer, in which case $\texttt{succ}\ t$ is also a value. Or $t \to t'$ and by the semantic rules for $\to$ we have $\texttt{succ}\ t \to \texttt{succ}\ t'$.

Case `if` is an exercise.

Progress (proof 2/2)

Case $t_1\ t_2$. By induction hypothesis (t_1), we get :

1. either $t_1 \rightarrow t_1'$. In this case, $t_1\ t_2 \rightarrow t_1'\ t_2$ by the semantic rule.
2. or t_1 is a value v_1. Now (IH about the type derivation for t_2), there are two cases :
   - If $t_2 \rightarrow t_2'$, then $v_1\ t_2 \rightarrow v_1\ t_2'$ by semantic rule.
   - If t_2 is also a value, say v_2, then by Canonical Form lemma, v_1 is of the form $\lambda\ \mathbf{x} : A_1.u$ and the rule for β-reduction applies.

Other Cases exercises.

Substitution lemma (proof 1/2)

To prove subject reduction, we need:

Lemma (Substitution lemma)

If $\Gamma, \mathtt{x} : A \vdash t : B$ *and* $\Gamma \vdash u : A$*, then* $\Gamma \vdash t[\mathtt{x} := u] : B$*.*

Proof : By induction on the derivation of the judgment $\Gamma, \mathtt{x} : A \vdash t : B$.

Case : $t = \mathtt{y}$ inferred from the judgment $\mathtt{y} : B \in \Gamma, \mathtt{x} : A$. Two sub-cases to consider:

- $\mathtt{x} = \mathtt{y}$. Then $A = B$ and $t[\mathtt{x} := u] = \mathtt{x}[\mathtt{x} := u] = u$, so we need to prove $\Gamma \vdash u : A$. But this typing is one of the hypotheses.
- $\mathtt{x} \neq \mathtt{y}$. Then, $\mathtt{y}[\mathtt{x} := u] = \mathtt{y}$ and since $\mathtt{y} : B \in \Gamma$, we have $\Gamma \vdash \mathtt{y}[\mathtt{x} := u] : B$.

Substitution lemma (proof 2/2)

Case: $t = \lambda\,\mathtt{y} : C.t_1$. We can assume that $\mathtt{y}$ is not bound in Γ, $\mathtt{x} \neq \mathtt{y}$, and that $\mathtt{y} \notin FV(u)$ (check, using alpha-renaming).

In that case, $B = C \to D$ and the premise is $\Gamma,\ \mathtt{x} : A,\ \mathtt{y} : C \vdash t_1 : D$.

But then we can also infer $\Gamma,\ \mathtt{y} : C,\ \mathtt{x} : A \vdash t_1 : D$ (check!) (*)

As $\Gamma \vdash u : A$ is derivable and $\mathtt{y}$ not bound in Γ, then $\Gamma,\ \mathtt{y} : C \vdash u : A$ (check!)

The induction hypothesis can now be applied (*) to give

$$\Gamma,\ \mathtt{y} : C \vdash t_1[\,\mathtt{x} := u] : D$$

and, by the typing rule for abstraction, we infer

$$\Gamma \vdash \lambda\,\mathtt{y} : C.t_1[\,\mathtt{x} := u] : C \to D.$$

Now,

$$\lambda\,\mathtt{y} : C.(t_1[\,\mathtt{x} := u]) = (\lambda\,\mathtt{y} : C.t_1)[\,\mathtt{x} := u] = t[\,\mathtt{x} := u]$$

Case: Application $t = t_1\ t_2$. Exercise.

Subject Reduction (proof 1/2)

Theorem 16 (Invariance)

If $\Gamma \vdash t : A$ *and* $t \to t'$, *then* $\Gamma \vdash t' : A$.

Proof : By induction on the derivation of $t \to t'$. We use the substitution lemma for the case of the β-reduction.

Normalisation theorem

Theorem 17 (Normalisation of the simply-typed λ-calculus)

If $\Gamma \vdash t : A$ then there are no infinite reduction sequences starting from t.

- The simply-typed λ-calculus is not Turing-complete but can be used in logic
- Hence the need for recursion in functional languages.
- Proof is complex for logical reasons (Complex induction)

Proof outline

Induction on terms does not work: termination is not **compositional**:

$$M, N \text{ terminating} \not\Rightarrow M\,N \text{ terminating}$$

$\rightsquigarrow$ We need to find a stronger inductive invariant.

Proof outline

Induction on terms does not work: termination is not **compositional**:

$$M, N \text{ terminating} \not\Rightarrow M\,N \text{ terminating}$$

$\leadsto$ We need to find a stronger inductive invariant.

Idea: define a notion of **good** inhabitants of a type:

$$\begin{array}{rcl} [\![\mathsf{Nat}]\!] & := & \{\vdash M : \mathsf{Nat} \mid M \text{ terminates}\} \\ [\![A \to B]\!] & := & \{\vdash M : A \to B \mid \forall N \in [\![A]\!], M\,N \in [\![B]\!]\} \end{array}$$

This invariant is indeed stronger:

Lemma 18

If $M \in [\![A]\!]$, then M is terminating.

We can then do our induction:

Lemma 19

For all $\Gamma \vdash M : A$, and for all $(v_x \in [\![B]\!])_{x:B\in\Gamma}$, then $[\![M[\vec{x} := v_{\vec{x}}]]\!] \in [\![A]\!]$

Outline

1. Lambda calculus
    - Syntax
    - Semantics
2. Simply Typed Lambda Calculus
3. Extensions

Outline

1. Lambda calculus
   - Syntax
   - Semantics
2. Simply Typed Lambda Calculus
3. Extensions

Extensions : products

Most languages have constructions for building complex data structures.

Pairs (products).

Expressions $t ::= \ldots \mid (t, t) \mid \mathsf{fst}(t) \mid \mathsf{snd}(t)$

Values $v ::= \ldots \mid (v, v)$

Types $A ::= \ldots \mid A \times A.$

Evaluation

$$\mathtt{fst}(v_1, v_2) \to v_1 \qquad \mathsf{snd}(v_1, v_2) \to v_2 \qquad \frac{t_1 \to t_1'}{\mathtt{fst}(t_1) \to \mathtt{fst}(t_1')} \ldots \frac{t_1 \to t_1'}{(t_1, t_2) \to (t_1', t_2)}$$

Typing rules

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash (t, u) : A \times B} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathtt{fst}(t) : A} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathtt{snd}(t) : B}$$

Extensions : sums (1/2)

Sums

Example :

```
type open_file_result =
  Opened of file_handle
| Error of string
```

Expressions $t ::= \dots \mid \texttt{inl}\ t \mid \texttt{inr}\ t \mid (\,\texttt{case}\ t\ \texttt{of}\ \texttt{inl}\ \texttt{x} \rhd t \mid \texttt{inr}\ \texttt{x} \rhd t)$

Values $v ::= \dots \mid \texttt{inl}\ v \mid \texttt{inr}\ v$

Types $T ::= \dots \mid T + T$

Extensions : sums (2/2)

Evaluation

$$\texttt{case inl}\ v_0\ \texttt{of inl}\ \ \texttt{x}_1 \rhd t_1 \mid \texttt{inr}\ \ \texttt{x}_2 \rhd t_2 \rightarrow t_1[\,\texttt{x}_1 := v_0]$$

$$\texttt{case inr}\ v_0\ \texttt{of inl}\ \ \texttt{x}_1 \rhd t_1 \mid \texttt{inr}\ \ \texttt{x}_2 \rhd t_2 \rightarrow t_2[\,\texttt{x}_2 := v_0]$$

$$\frac{t \rightarrow t'}{\texttt{case}\ t\ \texttt{of inl}\ \ \texttt{x} \rhd t_1 \mid \texttt{inr}\ \ \texttt{x} \rhd t_2 \rightarrow\ \texttt{case}\ t'\ \texttt{of inl}\ \ \texttt{x} \rhd t_1 \mid \texttt{inr}\ \ \texttt{x} \rhd t_2}$$

$$\frac{t \rightarrow t'}{\texttt{inr}\ t \rightarrow\ \texttt{inr}\ t'} \qquad \frac{t \rightarrow t'}{\texttt{inl}\ t \rightarrow\ \texttt{inl}\ t'}$$

Typing rules

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash\ \texttt{inl}\ t : A + B} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash\ \texttt{inr}\ t : A + B}$$

$$\frac{\Gamma \vdash t : A + B \quad \Gamma,\ \texttt{x}_1 : A \vdash t_1 : C \quad \Gamma,\ \texttt{x}_2 : B \vdash t_2 : C}{\Gamma \vdash\ \texttt{case}\ t\ \texttt{of inl}\ \ \texttt{x}_1 \rhd t_1 \mid \texttt{inr}\ \ \texttt{x}_2 \rhd t_2 : C}$$

NB : With sum types, we no longer have type unicity

Typing recursive functions

The theoretical formalisation of recursive functions is through **fixpoints**:

```
let rec fac = fun n -> if n < 2 then 1 else n * fac (n-1)
```

can be seen as the fixpoint of the function

```
fun fac -> (fun n -> if n < 2 then 1 else n * fac (n - 1))
```

Definition 20

A **fixpoint combinator** is a term `fix` such that `fix` $M \to M($ `fix` $M)$.

In the untyped call-by-value λ-calculus, there are fixpoint combinators:

$$\texttt{fix} = \lambda f.(\lambda x.f\ (\lambda y.x\ x\ y))(\lambda x.f\ (\lambda y.x\ x\ y))$$

Syntactic fixpoint

Of course, in a typed world, we have to add a primitive for that:

Expressions $t ::= \ldots \mid \texttt{fix}\, t$

Evaluation $\texttt{fix}\,(\lambda\, \texttt{x} : A.t) \to t[\,\texttt{x} := \texttt{fix}\,(\lambda\, \texttt{x} : A.t)]$ $\dfrac{t \to t'}{\texttt{fix}\, t \to \texttt{fix}\, t'}$

Typing rules $\dfrac{\Gamma \vdash t : A \to A}{\Gamma \vdash \texttt{fix}\, t : A}$

`let rec f x = M in N` becomes $N(\texttt{fix}\,(\lambda f x. M))$

Subtyping

Without subtyping, typing rules can be very rigid (types of arguments for functions must match exactly):

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash t\ u : B}$$

Example of limitation: records (and more generally OO-features).

S is a subtype of T (written $S <: T$) means that any term of type S can safely be used in a context where a term of type T is expected (or, every value described by S is also described by T, S is more informative).

Add a new typing rule (subsumption):

$$\frac{A \vdash t : S \quad S <: T}{A \vdash t : T}$$

Subtyping relation: should be reflexive and transitive.

$$\frac{}{S <: S} \qquad \frac{S <: U \quad U <: T}{S <: T}$$

Types for program analysis

Replace types by other program properties, for example, sign, interval, ...

The order $\sqsubseteq$ on the properties gives rise to a subtyping relation (reflexive and transitive).

$$\frac{A \vdash t_1 : P_1 \to P_2 \qquad A \vdash t_2 : P \qquad P \sqsubseteq P_1}{A \vdash t_1\ t_2 : P_2}$$

Other application : types represent the secrecy level of the data manipulated by a program (Cf. lecture on Static Information Flow Control).