Short Introduction to the λ-calculus

Functionnal programming

```
let rec map f l = match l with
    [] -> []
  | h::q -> f h::(map f q);;
- : map : ('a -> 'b) -> 'a list -> 'b list

map (fun x -> x+1) [1;2;3];;
- : int list = [2;3;4]
```

- Functions as arguments
- Lisp, Scheme, OCaml, SML, Haskell, Javascript, Python, Scala, F#, Clojure, C++11, Java 8, ...

Lambda-calculus

- A core language for studying functional programming
- Turing complete
- Termination is undecidable
  - Types
  - Type inference
- Reduction strategies
- Equivalences of terms
- ...

A core language

Syntax

$$t ::= x \mid \lambda x.t \mid t\ t$$

- Variable
- λ-abstraction (function)
- application

A core language

Syntax

$$t ::= x \mid \lambda x.t \mid t\ t$$

- Variable
- λ-abstraction (function)
- application

Example:

$$\lambda f.\lambda g.\lambda x.((f\ x)\ x)\ (g\ g)$$

A core language

Syntax

$$t ::= x \mid \lambda x.t \mid t\ t$$

- Variable
- λ-abstraction (function)
- application

Example:

$$\lambda f.\lambda g.\lambda x.((f\ x)\ x)\ (g\ g)$$

Notation simplifications:

A core language

Syntax

$$t ::= x \mid \lambda x.t \mid t\ t$$

- Variable
- λ-abstraction (function)
- application

Example:

$$\lambda fgx.((f\ x)\ x)\ (g\ g)$$

Notation simplifications:

- $\lambda x.\lambda y.t \rightarrow \lambda xy.t$

A core language

Syntax

$$t ::= x \mid \lambda x.t \mid t\ t$$

- Variable
- λ-abstraction (function)
- application

Example:

$$\lambda fgx.f\ x\ x\ (g\ g)$$

Notation simplifications:

- $\lambda x.\lambda y.t \rightarrow \lambda xy.t$
- Application is right-associative: $t_1\ t_2\ t_3$ is parsed $(t_1\ t_2)\ t_3$

Bound variables

- λ-abstractions bind variables

$$\lambda x.(\ldots x \ldots \lambda x.(\ldots x \ldots))$$

- free variable: not bound

Bound variables

- λ-abstractions bind variables

$$\lambda x.(\ldots x \ldots \lambda x.(\ldots x \ldots))$$

- free variable: not bound
- α-conversion: renaming of *bound* variables

$$\lambda x.(\ldots x \ldots \lambda y.(\ldots y \ldots))$$

- Terms are usually equated up-to α-conversion: $\lambda x.x = \lambda y.y$

Bound variables

- λ-abstractions bind variables

$$\lambda x.(\ldots x \ldots \lambda x.(\ldots x \ldots))$$

- free variable: not bound
- α-conversion: renaming of *bound* variables

$$\lambda x.(\ldots x \ldots \lambda y.(\ldots y \ldots))$$

- Terms are usually equated up-to α-conversion: $\lambda x.x = \lambda y.y$
- Barendregt's convention: bound variables are chosen distinct from free variables

$$(\lambda x.x)\ x \qquad \text{☹}$$
$$(\lambda x.x)\ y \qquad \text{☺}$$

- Sometimes bound variables pairwise distinct also

Bound variables

- What is the binder of x in $\lambda x y.f\ y\ (\lambda x.y)\ x$?

Bound variables

- What is the binder of x in $\lambda xy.f\ y\ (\lambda x.y)\ x$? red
- Is y free in $\lambda xy.f\ y\ (\lambda x.y)\ x$?

Bound variables

- What is the binder of x in $\lambda xy.f\ y\ (\lambda x.y)\ x$? red
- Is y free in $\lambda xy.f\ y\ (\lambda x.y)\ x$? no
- What are the free variables of $\lambda xy.f\ y\ (\lambda x.y)\ x$?

Bound variables

- What is the binder of x in $\lambda xy.f\ y\ (\lambda x.y)\ x$? red
- Is y free in $\lambda xy.f\ y\ (\lambda x.y)\ x$? no
- What are the free variables of $\lambda xy.f\ y\ (\lambda x.y)\ x$? $\{\mathrm{f}\}$
- Is $\lambda xy.f\ y\ (\lambda x.y)\ x$ α-equivalent to $\lambda zy.f\ y\ (\lambda x.y)\ x$?

Bound variables

- What is the binder of x in $\lambda xy.f\ y\ (\lambda x.y)\ x$? red
- Is y free in $\lambda xy.f\ y\ (\lambda x.y)\ x$? no
- What are the free variables of $\lambda xy.f\ y\ (\lambda x.y)\ x$? {f}
- Is $\lambda xy.f\ y\ (\lambda x.y)\ x$ α-equivalent to $\lambda zy.f\ y\ (\lambda x.y)\ x$? no
- Is $\lambda xy.f\ y\ (\lambda x.y)\ x$ α-equivalent to $\lambda xy.f\ y\ (\lambda z.y)\ x$?

Bound variables

- What is the binder of x in $\lambda xy.f\ y\ (\lambda x.y)\ x$? red
- Is y free in $\lambda xy.f\ y\ (\lambda x.y)\ x$? no
- What are the free variables of $\lambda xy.f\ y\ (\lambda x.y)\ x$? $\{\mathrm{f}\}$
- Is $\lambda xy.f\ y\ (\lambda x.y)\ x$ α-equivalent to $\lambda zy.f\ y\ (\lambda x.y)\ x$? no
- Is $\lambda xy.f\ y\ (\lambda x.y)\ x$ α-equivalent to $\lambda xy.f\ y\ (\lambda z.y)\ x$? yes
- Is $\lambda xy.f\ y\ (\lambda x.y)\ x$ α-equivalent to $\lambda yz.f\ z\ (\lambda x.z)\ y$?

Bound variables

- What is the binder of x in $\lambda xy.f\ y\ (\lambda x.y)\ x$? red
- Is y free in $\lambda xy.f\ y\ (\lambda x.y)\ x$? no
- What are the free variables of $\lambda xy.f\ y\ (\lambda x.y)\ x$? $\{\mathrm{f}\}$
- Is $\lambda xy.f\ y\ (\lambda x.y)\ x$ α-equivalent to $\lambda zy.f\ y\ (\lambda x.y)\ x$? no
- Is $\lambda xy.f\ y\ (\lambda x.y)\ x$ α-equivalent to $\lambda xy.f\ y\ (\lambda z.y)\ x$? yes
- Is $\lambda xy.f\ y\ (\lambda x.y)\ x$ α-equivalent to $\lambda yz.f\ z\ (\lambda x.z)\ y$? yes
- Is $\lambda xy.f\ y\ (\lambda x.y)\ x$ α-equivalent to $\lambda xz.g\ z\ (\lambda x.z)\ x$?

Bound variables

- What is the binder of x in $\lambda xy.f\ y\ (\lambda x.y)\ x$? red
- Is y free in $\lambda xy.f\ y\ (\lambda x.y)\ x$? no
- What are the free variables of $\lambda xy.f\ y\ (\lambda x.y)\ x$? $\{\mathrm{f}\}$
- Is $\lambda xy.f\ y\ (\lambda x.y)\ x$ α-equivalent to $\lambda zy.f\ y\ (\lambda x.y)\ x$? no
- Is $\lambda xy.f\ y\ (\lambda x.y)\ x$ α-equivalent to $\lambda xy.f\ y\ (\lambda z.y)\ x$? yes
- Is $\lambda xy.f\ y\ (\lambda x.y)\ x$ α-equivalent to $\lambda yz.f\ z\ (\lambda x.z)\ y$? yes
- Is $\lambda xy.f\ y\ (\lambda x.y)\ x$ α-equivalent to $\lambda xz.g\ z\ (\lambda x.z)\ x$? no

Substitutions

- Result of the substitution of x for y in $\lambda x.y$?

Substitutions

- Result of the substitution of x for y in $\lambda x.y$?
- Capture-avoiding substitution $t\{^u/_x\}$

$$
\begin{aligned}
x\{^u/_x\} &= u \\
y\{^u/_x\} &= y \text{ if } x \neq y \\
(t_1\ t_2)\{^u/_x\} &= t_1\{^u/_x\}\ t_2\{^u/_x\} \\
(\lambda y.t)\{^u/_x\} &= \lambda y.t\{^u/_x\} \text{ if } y \neq x \wedge y \notin \mathsf{fv}(u)
\end{aligned}
$$

- Use α-conversion $(\lambda x.y)\{^x/_y\} = \lambda z.x$

Reduction semantics

Only one rule (β-reduction):

$$(\lambda x.t)\ u \to_\beta t\{u/x\}$$

Reduction semantics

Only one rule (β-reduction):

$$(\lambda x.t)\ u \to_\beta t\{u/x\}$$

- Is λ-calculus a term rewriting system?

Reduction semantics

Only one rule (β-reduction):

$$(\lambda x.t)\ u \to_\beta t\{u/x\}$$

- Is λ-calculus a term rewriting system?
  no (de Bruijn indices + explicit substitutions needed)
- How many redexes in $(\lambda x.(\lambda y.y)\ (x\ x))\ (\lambda z.(\lambda x.x)\ \lambda y.z)$?

Reduction semantics

Only one rule (β-reduction):

$$(\lambda x.t)\ u \to_\beta t\{u/x\}$$

- Is λ-calculus a term rewriting system?
  no (de Bruijn indices + explicit substitutions needed)
- How many redexes in $(\lambda x.(\lambda y.y)\ (x\ x))\ (\lambda z.(\lambda x.x)\ \lambda y.z)$? 3
- Write the derivation tree of
  $(\lambda x.(\lambda y.y)\ (x\ x))\ (\lambda z.(\lambda x.x)\ \lambda y.z)$.

Reduction semantics

Only one rule (β-reduction):

$$(\lambda x.t)\ u \to_\beta t\{u/x\}$$

- Is λ-calculus a term rewriting system?
  no (de Bruijn indices + explicit substitutions needed)
- How many redexes in $(\lambda x.(\lambda y.y)\ (x\ x))\ (\lambda z.(\lambda x.x)\ \lambda y.z)$? 3
- Write the derivation tree of
  $(\lambda x.(\lambda y.y)\ (x\ x))\ (\lambda z.(\lambda x.x)\ \lambda y.z)$.
- Reduce the term $(\lambda x.x\ x)\ (\lambda x.x\ x)$.

Representing data

Everything is a function

Reasoning about data in λ-calculus:

Everything is a function

Reasoning about data in λ-calculus:

- Extend the calculus (meh)

Everything is a function

Reasoning about data in λ-calculus:

- Extend the calculus (meh)
- Encoding in the calculus

Booleans

$$True \equiv \lambda xy.x$$

$$False \equiv \lambda xy.y$$

Booleans

$$True \equiv \lambda xy.x$$

$$False \equiv \lambda xy.y$$

Representing *if*:

$$if \equiv \lambda btu.\ldots.$$

Booleans

$$True \equiv \lambda xy.x$$

$$False \equiv \lambda xy.y$$

Representing *if*:

$$if \equiv \lambda btu.b\ t\ u$$

Booleans

$$True \equiv \lambda xy.x$$

$$False \equiv \lambda xy.y$$

Representing *if*:

$$if \equiv \lambda btu.b\ t\ u$$

Using *if*, encode *and*, *or*, *not*

Booleans

$$True \equiv \lambda xy.x$$
$$False \equiv \lambda xy.y$$

Representing *if*:

$$if \equiv \lambda btu.b\ t\ u$$

Using *if*, encode *and*, *or*, *not*

$$and \equiv \lambda b_1 b_2.if\ b_1\ b_2\ False$$
$$or \equiv \lambda b_1 b_2.if\ b_1\ True\ b_2$$
$$not \equiv \lambda b.if\ b\ False\ True$$

Pairs

$$(t, u) \equiv \lambda f.f\ t\ u$$

f: means to access t or u.

Propose a term which builds a pair from two arguments

Propose an encoding of the projections π_1, π_2

Pairs

$$(t, u) \equiv \lambda f.f\ t\ u$$

f: means to access t or u.

Propose a term which builds a pair from two arguments

$$\lambda tuf.f\ t\ u$$

Propose an encoding of the projections π_1, π_2

$$\pi_1 \equiv \lambda xy.x$$
$$\pi_2 \equiv \lambda xy.y$$

(Yes, it is *True* and *False*)

Easily generalized to tuples

Church numerals

$$
\begin{aligned}
0 &\equiv \lambda fx.x \\
1 &\equiv \lambda fx.f\ x \\
2 &\equiv \lambda fx.f\ (f\ x) \\
3 &\equiv \lambda fx.f\ (f\ (f\ x)) \\
&\vdots \\
n &\equiv \lambda fx.f^n\ x
\end{aligned}
$$

Definition of *succ*

Church numerals

$$
\begin{aligned}
0 &\equiv \lambda fx.x \\
1 &\equiv \lambda fx.f\ x \\
2 &\equiv \lambda fx.f\ (f\ x) \\
3 &\equiv \lambda fx.f\ (f\ (f\ x)) \\
&\vdots \\
n &\equiv \lambda fx.f^n\ x
\end{aligned}
$$

Definition of *succ*

$$succ \equiv \lambda nfx.f(n\ f\ x)$$

Propose an alternate definition of *succ*

Church numerals

$$
\begin{aligned}
0 &\equiv \lambda fx.x \\
1 &\equiv \lambda fx.f\ x \\
2 &\equiv \lambda fx.f\ (f\ x) \\
3 &\equiv \lambda fx.f\ (f\ (f\ x)) \\
&\vdots \\
n &\equiv \lambda fx.f^n\ x
\end{aligned}
$$

Definition of *succ*

$$succ \equiv \lambda nfx.f(n\ f\ x)$$

Propose an alternate definition of *succ*

$$succ \equiv \lambda nfx.n\ f\ (f\ x)$$

Church numerals

$$0 \equiv \lambda fx.x$$

$$n \equiv \lambda fx.f^n\ x$$

Addition

Church numerals

$$0 \equiv \lambda fx.x$$
$$n \equiv \lambda fx.f^n\ x$$

Addition

$$+ \equiv \lambda npfx.n\ f\ (p\ f\ x)$$

Multiplication, exponentiation

$$* \equiv \lambda npf.m\ (n\ f)$$
$$exp \equiv \lambda np.p\ n$$

Propose a term $if0$ that returns $True$ iff it is applied to 0

Church numerals

$$0 \equiv \lambda fx.x$$
$$n \equiv \lambda fx.f^n\ x$$

Addition

$$+ \equiv \lambda npfx.n\ f\ (p\ f\ x)$$

Multiplication, exponentiation

$$* \equiv \lambda npf.m\ (n\ f)$$
$$exp \equiv \lambda np.p\ n$$

Propose a term $if\,0$ that returns $True$ iff it is applied to 0

$$if\,0 \equiv \lambda n.n\ (\lambda x.False)\ True$$

Normal forms, confluence, reduction strategies

Normal forms, confluence

What are the normal forms of the calculus?

Normal forms, confluence

What are the normal forms of the calculus?

$$\lambda x_1 \ldots x_m..y \;\; t_1 \;\; \ldots t_n$$

where $m \geq 0$, $n \geq 0$, t_i normal forms

Unicity of the normal forms?

Normal forms, confluence

What are the normal forms of the calculus?

$$\lambda x_1 \ldots x_m..y \;\; t_1 \;\; \ldots t_n$$

where $m \geq 0$, $n \geq 0$, t_i normal forms

Unicity of the normal forms?

Theorem

$\rightarrow_\beta$ is confluent.

Proof.

Many different proofs, one using a rewriting theorem □

Remember, remember . . .

What if the system is non-teminating and non-orthogonal?

Theorem Consider a reduction relation $\rightarrow_R$ and let $\rightarrow_D$ s.t.

$\rightarrow_R \subseteq \rightarrow_D \subseteq \overset{*}{\rightarrow}_R$

$\rightarrow_D$ has the diamond property

Then, $\rightarrow_R$ is confluent.

Parallel reduction $\overset{\parallel}{\to}$

$$\frac{}{t \overset{\parallel}{\to} t} \qquad \frac{t \overset{\parallel}{\to} t' \qquad u \overset{\parallel}{\to} u'}{t\ u \overset{\parallel}{\to} t'\ u'} \qquad \frac{t \overset{\parallel}{\to} t'}{\lambda x.t \overset{\parallel}{\to} \lambda x.t'}$$

$$\frac{t \overset{\parallel}{\to} t' \qquad u \overset{\parallel}{\to} u'}{\lambda x.t\ u \overset{\parallel}{\to} t'\{u'/x\}}$$

We have

- $\to_\beta \subseteq \overset{\parallel}{\to} \subseteq \overset{*}{\to}_\beta$
- Diamond property?

Stronger than diamond

Lemma (Takahashi)

For all t, there exists t^* such that for all u, $t \xrightarrow{\parallel} u$ implies $u \xrightarrow{\parallel} t^*$.

Intuitively, t^* is t where all the β-redexes are reduced.

Evaluation strategies

- Confluence: the reduction order does not matter
- In practice: evaluation strategy
- Call-by-name (like, well, hem . . .)
  - Reduce the term in function position only
- Call-by-value (all languages except Haskell)
  - Reduce also the argument
- Call-by-need (Haskell)
  - Reduce a term when needed, and at most once

Call-by-name

$$\frac{}{(\lambda x.t)\ u \to_n t\{u/x\}} \qquad \frac{t \to_n t'}{t\ u \to_n t'\ u}$$

Remark: no reduction under λ

Call-by-name

$$\frac{}{(\lambda x.t)\ u \to_n t\{u/x\}} \qquad \frac{t \to_n t'}{t\ u \to_n t'\ u}$$

Remark: no reduction under λ

Reduce $(\lambda xyz.x\ (y\ z))\ (\lambda x.f\ x)\ ((\lambda x.x)\ g)\ ((\lambda y.g)\ c)$ using $\to_n$

Call-by-name

$$\frac{}{(\lambda x.t)\ u \to_n t\{u/x\}} \qquad \frac{t \to_n t'}{t\ u \to_n t'\ u}$$

Remark: no reduction under λ

Reduce $(\lambda xyz.x\ (y\ z))\ (\lambda x.f\ x)\ ((\lambda x.x)\ g)\ ((\lambda y.g)\ c)$ using $\to_n$
Let $Y = \lambda g.(\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))$. Reduce $Y\ f$

Call-by-value

$$\frac{}{(\lambda x.t)\ v \to_v t\{v/x\}} \qquad \frac{t \to_v t'}{t\ u \to_n t'\ u} \qquad \frac{t \to_v t'}{v\ t \to_n v\ t'}$$

Values $v ::= x \mid \lambda x.t$

Call-by-value

$$\frac{}{(\lambda x.t)\ v \to_v t\{v/x\}} \qquad \frac{t \to_v t'}{t\ u \to_n t'\ u} \qquad \frac{t \to_v t'}{v\ t \to_n v\ t'}$$

Values $v ::= x \mid \lambda x.t$

Reduce $(\lambda xyz.x\ (y\ z))\ (\lambda x.f\ x)\ ((\lambda x.x)\ g)\ ((\lambda y.g)\ c)$ using $\to_v$

Call-by-name, call-by-value, call-by-need

Call-by-name vs call-by-value

Call-by-name, call-by-value, call-by-need

Call-by-name vs call-by-value

- CBN $>$ CBV: useless (and potentially non terminating) arguments
- CBV $>$ CBN: duplication

Call-by-name, call-by-value, call-by-need

Call-by-name vs call-by-value

- CBN $>$ CBV: useless (and potentially non terminating) arguments
- CBV $>$ CBN: duplication

Call-by-need (lazy)

- $(\lambda x.t)\ u$: store x, u, evaluate t;
- if the value of x is needed, then evaluate u into v, store x, v
- Interactions with side-effects, space complexity

Termination, types

Termination

- Undecidable
- Sufficient conditions: types

  "A well-typed program cannot go wrong"
- Remark: not the only use for types

Simply typed λ-calculus

Syntax:

$$T ::= \alpha \mid \alpha \to \alpha$$

Judgment:

$$\Gamma \vdash t : T$$

where $\Gamma = (x_1 : T_1, \ldots x_n : T_n)$ typing environment

Simply typed λ-calculus

Syntax:

$$T ::= \alpha \mid \alpha \to \alpha$$

Judgment:

$$\Gamma \vdash t : T$$

where $\Gamma = (x_1 : T_1, \ldots x_n : T_n)$ typing environment

$$\frac{}{\Gamma, x : T \vdash x : T}$$

Simply typed λ-calculus

Syntax:

$$T ::= \alpha \mid \alpha \to \alpha$$

Judgment:

$$\Gamma \vdash t : T$$

where $\Gamma = (x_1 : T_1, \ldots x_n : T_n)$ typing environment

$$\frac{}{\Gamma, x : T \vdash x : T} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2}$$

Simply typed λ-calculus

Syntax:

$$T ::= \alpha \mid \alpha \to \alpha$$

Judgment:

$$\Gamma \vdash t : T$$

where $\Gamma = (x_1 : T_1, \ldots x_n : T_n)$ typing environment

$$\frac{}{\Gamma, x : T \vdash x : T} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2}$$

$$\frac{\Gamma \vdash t : T_1 \to T_2 \qquad \Gamma \vdash u : T_1}{\Gamma \vdash t\ u : T_2}$$

Simply typed λ-calculus

Theorem (Progress)

If $\Gamma \vdash t : T$, *then* t *is a value, or* $t \to_\beta t'$ *for some* t'

Theorem (Preservation)

If $t \to_\beta t'$ *and* $\Gamma \vdash t : T$, *then* $\Gamma \vdash t' : T$.

Theorem

If $\Gamma \vdash t : T$, *then* t *is strongly normalizing.*

Proof.

Tait's reducibility predicates (beyond the scope) $\square$

Simply typed λ-calculus

Write the type derivation tree for (type checking)

$$f : \alpha \to \beta \to \gamma, c : \alpha \vdash (\lambda xy.x\ y)\ f\ c\ :\ \beta \to \gamma$$

Simply typed λ-calculus

Write the type derivation tree for (type checking)

$$f : \alpha \to \beta \to \gamma, c : \alpha \vdash (\lambda xy.x\ y)\ f\ c\ :\ \beta \to \gamma$$

Can we find a simpler type?

Simply typed λ-calculus

Write the type derivation tree for (type checking)

$$f : \alpha \to \beta \to \gamma, c : \alpha \vdash (\lambda xy.x\ y)\ f\ c\ :\ \beta \to \gamma$$

Can we find a simpler type? no, not with this Γ

Simply typed λ-calculus

Write the type derivation tree for (type checking)

$$f : \alpha \to \beta \to \gamma, c : \alpha \vdash (\lambda xy.x\ y)\ f\ c\ :\ \beta \to \gamma$$

Can we find a simpler type? no, not with this Γ

And if we can change Γ?

Simply typed λ-calculus

Write the type derivation tree for (type checking)

$$f : \alpha \to \beta \to \gamma, c : \alpha \vdash (\lambda xy.x\ y)\ f\ c\ :\ \beta \to \gamma$$

Can we find a simpler type? no, not with this Γ
And if we can change Γ? yes

$$f : \alpha \to \beta, c : \alpha \vdash (\lambda xy.x\ y)\ f\ c\ :\ \beta$$

Principal type: most general type T (given t, Γ)
Principal typing: most general typing Γ, t (given t)

Theorem
STLC has the principal typing property

Simply typed λ-calculus

Find a type for $(\lambda xy.x\ y)\ (\lambda x.x)$ (type inference)

Simply typed λ-calculus

Find a type for $(\lambda xy.x\ y)\ (\lambda x.x)$ (type inference)

$$\vdash\ (\lambda xy.x\ y)\ (\lambda x.x)\ :\ \alpha \to \alpha$$

Simply typed λ-calculus

Find a type for $(\lambda xy.x\ y)\ (\lambda x.x)$ (type inference)

$$\vdash\ (\lambda xy.x\ y)\ (\lambda x.x)\ :\ \alpha \to \alpha$$

Two different problems (not a universal terminology):

- Type inference: $\Gamma\ \vdash\ t\ :\ ?$
- Typing inference: $?\ \vdash\ t\ :\ ?$

Theorem

Type/typing inference is decidable in STLC, and the resulting type/typing is the principal one

Type systems

- Simple types are not expressive enough: no polymorphism
- ML: limited (prenex) polymorphism

$$\vdash \lambda x.x \ : \ \forall \alpha.(\alpha \to \alpha)$$

```
- :  map :  ('a -> 'b) -> 'a list -> 'b list
```

  - Decidable type inference
  - Principal type (but not principal typing)
  - still limited: $\lambda x..x\ x$ not typable

Type systems

- System F: unrestricted polymorphism (under arrows)

$$\vdash \lambda f.(f\ 1, f\ \mathit{True}) \ :\ (\forall\alpha.(\alpha \to \beta)) \to (\beta, \beta)$$
$$\vdash \lambda x.x\ x \ :\ (\forall\alpha.\alpha) \to (\forall\beta.\beta)$$

  - Type checking and type inference are undecidable
  - No principal type/typing
  - Some SN terms not typable

Type systems

- System F: unrestricted polymorphism (under arrows)

$$\vdash \lambda f.(f\ 1, f\ \mathit{True}) \ :\ (\forall\alpha.(\alpha \to \beta)) \to (\beta, \beta)$$
$$\vdash \lambda x.x\ x \ :\ (\forall\alpha.\alpha) \to (\forall\beta.\beta)$$

  - Type checking and type inference are undecidable
  - No principal type/typing
  - Some SN terms not typable

- Intersection types

$$\vdash \lambda f.(f\ 1, f\ \mathit{True}) \ :\ ((\mathit{Int} \to \beta) \cap (\mathit{Bool} \to \gamma)) \to (\beta, \gamma)$$
$$\vdash \lambda x.x\ x \ :\ (\alpha \cap (\alpha \to \beta)) \to \beta$$

  - Characterize all SN λ-terms
  - Type inference undecidable
  - Principal typing

Back to rewriting: ρ-calculus

- No pattern-matching in λ
- No way to control reduction strategies in TRS
- Combination of the two: ρ-calculus (Cirstea, Kirchner)
- Patterns as first-class items
- Formal base of rule-based programming languages (Tom, ELAN, Maude, ...)
- More details at

`rho.loria.fr`