

System F

Alexandre Miquel — PPS & U. Paris 7

`Alexandre.Miquel@pps.jussieu.fr`

Types Summer School 2005

August 15–26 — Göteborg

Introduction

- **System F :** independently discovered by
 - Girard:** System F (1970)
 - Reynolds:** The polymorphic λ -calculus (1974)
- Quite different motivations...
 - Girard:** Interpretation of second-order logic
 - Reynolds:** Functional programming

... connected by the **Curry-Howard isomorphism**
- Significant influence on the development of Type Theory
 - Interpretation of higher-order logic [Girard, Martin-Löf]
 - Type:Type [Martin-Löf 1971]
 - Martin-Löf Type Theory [1972, 1984, 1990, ...]
 - The Calculus of Constructions [Coquand 1984]

Part I

System F: Church-style presentation

System F syntax

Definition

Types $A, B ::= \alpha \mid A \rightarrow B \mid \forall \alpha B$

Terms $t, u ::= x$
 | $\lambda x : A . t$ | tu (term abstr./app.)
 | $\Lambda \alpha . t$ | tA (type abstr./app.)

Notations

- Set of free (term) variables: $FV(t)$
- Set of free type variables: $TV(t), \quad TV(A)$
- Term substitution: $u\{x := t\}$
- Type substitution: $u\{\alpha := A\}, \quad B\{\alpha := A\}$

Perform α -conversion to prevent captures of free (term/type) variables!

System F typing rules

Contexts

$$\Gamma ::= x_1 : A_1, \dots, x_n : A_n$$

Typing judgments

$$\Gamma \vdash t : A$$
$$\frac{}{\Gamma \vdash x : A} \quad (x:A) \in \Gamma$$
$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A. t : A \rightarrow B}$$
$$\frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$
$$\frac{\Gamma \vdash t : B}{\Gamma \vdash \Lambda \alpha. t : \forall \alpha B} \quad \alpha \notin TV(\Gamma)$$
$$\frac{\Gamma \vdash t : \forall \alpha B}{\Gamma \vdash tA : B\{\alpha := A\}}$$

- Declaration of type variables is **implicit** (for each $\alpha \in TV(\Gamma)$)
- Type variables could be declared explicitly: $\alpha : *$ (cf PTS)
- One rule for each syntactic construct \Rightarrow System is **syntax-directed**

Example: the polymorphic identity

- Set: $\text{id} \equiv \Lambda\alpha. \lambda x:\alpha. x$

- One has:

$$\text{id} \quad : \quad \forall\alpha (\alpha \rightarrow \alpha)$$
$$\text{id } B \quad : \quad B \rightarrow B \quad \text{for any type } B$$
$$\text{id } B u \quad : \quad B \quad \text{for any term } u : B$$

- In particular, if we take $B \equiv \forall\alpha (\alpha \rightarrow \alpha)$ and $u \equiv \text{id}$

$$\text{id } (\forall\alpha (\alpha \rightarrow \alpha)) \quad : \quad \forall\alpha (\alpha \rightarrow \alpha) \rightarrow \forall\alpha (\alpha \rightarrow \alpha)$$
$$\text{id } (\forall\alpha (\alpha \rightarrow \alpha)) \text{id} \quad : \quad \forall\alpha (\alpha \rightarrow \alpha)$$

\Rightarrow Type system is **impredicative** (or **cyclic**)

Properties

Substitutivity (for types/terms):

- $\Gamma \vdash u : B \quad \Rightarrow \quad \Gamma\{\alpha := A\} \vdash u\{\alpha := A\} : B\{\alpha := A\}$
- $\Gamma, x : A \vdash u : B, \quad \Gamma \vdash t : A \quad \Rightarrow \quad \Gamma \vdash u\{x := t\} : B$

Uniqueness of type

$$\Gamma \vdash t : A, \quad \Gamma \vdash t : A' \quad \Rightarrow \quad A = A' \quad (\alpha\text{-conv.})$$

Decidability of type checking / type inference

- 1 Given Γ , t and A , decide whether $\Gamma \vdash t : A$ is derivable
- 2 Given Γ and t , compute a type A such that $\Gamma \vdash t : A$ if such a type exists, or fail otherwise.

Both problems are **decidable**

Reduction rules

Two kinds of **redexes**:

$$\begin{array}{ll} (\lambda x : A. t)u \succ t\{x := u\} & \text{1st kind redex} \\ (\Lambda \alpha. t)A \succ t\{\alpha := A\} & \text{2nd kind redex} \end{array}$$

Other combinations of abstraction and application are meaningless (and rejected by typing)

Definitions

- One step β -reduction $t \succ t' \equiv$
contextual closure of both rules above
- β -reduction $t \succ^* t' \equiv$
reflexive-transitive closure of \succ
- β -convertibility $t \simeq t' \equiv$
reflexive-symmetric-transitive closure of \succ

Examples

- The polymorphic identity, again

$$\text{id } B \ u \equiv (\Lambda \alpha . \lambda x : \alpha . x) B \ u \ \succ (\lambda x : B . x) u \ \succ \ u$$

$$\text{id } (\forall \alpha (\alpha \rightarrow \alpha)) \ \text{id } (\forall \alpha (\alpha \rightarrow \alpha)) \ \cdots \ \text{id } (\forall \alpha (\alpha \rightarrow \alpha)) \ \text{id } B \ u \ \succ^* \ u$$

- A little bit more complex example...

$$\begin{aligned} & (\Lambda \alpha . \lambda x : \alpha . \lambda f : \alpha \rightarrow \alpha . \overbrace{f (\cdots (f x) \cdots)}^{32 \text{ times}}) \\ & (\forall \alpha (\alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha)) (\Lambda \alpha . \lambda x : \alpha . \lambda f : \alpha \rightarrow \alpha . f x) \\ & (\lambda n : \forall \alpha (\alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha) . \Lambda \alpha . \lambda x : \alpha . \lambda f : \alpha \rightarrow \alpha . n \alpha (n \alpha x f) f) \end{aligned}$$

$$\succ^* \quad \Lambda \alpha . \lambda x : \alpha . \lambda f : \alpha \rightarrow \alpha . \underbrace{(f \cdots (f x) \cdots)}_{4\,294\,967\,296 \text{ times}}$$

Properties

Confluence

$$t \gamma^* t_1 \wedge t \gamma^* t_2 \quad \Rightarrow \quad \exists t' (t_1 \gamma^* t' \wedge t_2 \gamma^* t')$$

Proof. Roughly the same as for the untyped λ -calculus (adaptation is easy)

Church-Rosser

$$t_1 \simeq t_2 \quad \Leftrightarrow \quad \exists t' (t_1 \gamma^* t' \wedge t_2 \gamma^* t')$$

Subject-reduction

If $\Gamma \vdash t : A$ and $t \gamma^* t'$ then $\Gamma \vdash t' : A$

Proof. By induction on the derivation of $\Gamma \vdash t : A$, with $t \succ t'$ (one step reduction)

Strong normalisation

All well-typed terms of system F are **strongly normalisable**

Proof. Girard and Tait's method of reducibility candidates (postponed)

Part II

Encoding data types

Booleans (1/3)

Encoding of booleans

$$\text{Bool} \equiv \forall \gamma (\gamma \rightarrow \gamma \rightarrow \gamma)$$
$$\text{true} \equiv \Lambda \gamma. \lambda x, y: \gamma. x \quad : \quad \text{Bool}$$
$$\text{false} \equiv \Lambda \gamma. \lambda x, y: \gamma. y \quad : \quad \text{Bool}$$
$$\text{if}_A u \text{ then } t_1 \text{ else } t_2 \equiv u A t_1 t_2$$

Correctness w.r.t. typing

$$\frac{\Gamma \vdash u : \text{Bool} \quad \Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : A}{\Gamma \vdash \text{if}_A u \text{ then } t_1 \text{ else } t_2 : A}$$

Correctness w.r.t. reduction

$$\text{if}_A \text{ true then } t_1 \text{ else } t_2 \xrightarrow{*} t_1$$
$$\text{if}_A \text{ false then } t_1 \text{ else } t_2 \xrightarrow{*} t_2$$

Booleans (2/3)

Objection: We can do the same in the untyped λ -calculus!

$\text{true} \equiv \lambda x, y. x$	}	Same reduction rules as before
$\text{false} \equiv \lambda x, y. y$		
$\text{if } u \text{ then } t_1 \text{ else } t_2 \equiv u t_1 t_2$		

But nothing prevents the following computation:

$$\text{if } \underbrace{\lambda x. x}_{\text{bad bool}} \text{ then } t_1 \text{ else } t_2 \equiv (\lambda x. x) t_1 t_2 \succ \underbrace{t_1 t_2}_{\text{meaningless result}}$$

Question: Does the type discipline of system F avoid this?

Booleans (3/3)

Principle (that should be satisfied by any functional programming language)

When a program P of type A evaluates to a value v , then v has one of the **canonical forms** expected by the type A .

In ML/Haskell, a value produced by a program of type `Bool` will always be `true` or `false` (i.e. the canonical forms of type `bool`).

In system F : Subject-reduction ensures that the normal form of a term of type `Bool` is a term of type `Bool`.

To conclude, it suffices to check that in system F :

Lemma (Canonical forms of type `bool`)

The terms $\text{true} \equiv \Lambda\gamma. \lambda x, y : \gamma. x$ and $\text{false} \equiv \Lambda\gamma. \lambda x, y : \gamma. y$ are the only closed normal terms of type $\text{Bool} \equiv \forall\gamma (\gamma \rightarrow \gamma \rightarrow \gamma)$

Proof. Case analysis on the derivation.

Cartesian product

Encoding of the cartesian product $A \times B$

$$A \times B \equiv \forall \gamma ((A \rightarrow B \rightarrow \gamma) \rightarrow \gamma)$$

$$\langle t_1, t_2 \rangle \equiv \Lambda \gamma . \lambda f : A \rightarrow B \rightarrow \gamma . f \ t_1 \ t_2$$

$$\text{fst} \equiv \lambda p : A \times B . p \ A \ (\lambda x : A . \lambda y : B . x) : A \times B \rightarrow A$$

$$\text{snd} \equiv \lambda p : A \times B . p \ B \ (\lambda x : A . \lambda y : B . y) : A \times B \rightarrow B$$

Correctness w.r.t. typing and reduction

$$\frac{\Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : B}{\Gamma \vdash \langle t_1, t_2 \rangle : A \times B}$$

$$\text{fst} \langle t_1, t_2 \rangle \ \gamma^* \ t_1$$

$$\text{snd} \langle t_1, t_2 \rangle \ \gamma^* \ t_2$$

Lemma (Canonical forms of type $A \times B$)

The closed normal terms of type $A \times B$ are of the form $\langle t_1, t_2 \rangle$, where t_1 and t_2 are closed normal terms of type A and B , respectively.

Disjoint union

Encoding of the disjoint union $A + B$

$$A + B \equiv \forall \gamma ((A \rightarrow \gamma) \rightarrow (B \rightarrow \gamma) \rightarrow \gamma)$$

$$\text{inl}(v) \equiv \Lambda \gamma . \lambda f : A \rightarrow \gamma . \lambda g : B \rightarrow \gamma . f \ v \quad : \quad A + B \quad (\text{with } v : A)$$

$$\text{inr}(v) \equiv \Lambda \gamma . \lambda f : A \rightarrow \gamma . \lambda g : B \rightarrow \gamma . g \ v \quad : \quad A + B \quad (\text{with } v : B)$$

$$\text{case}_C \ u \ \text{of} \ \text{inl}(x) \mapsto t_1 \mid \text{inr}(y) \mapsto t_2 \equiv u \ C \ (\lambda x : A . t_1) \ (\lambda y : B . t_2)$$

Correctness w.r.t. typing and reduction

$$\frac{\Gamma \vdash u : A + B \quad \Gamma, x : A \vdash t_1 : C \quad \Gamma, y : B \vdash t_2 : C}{\Gamma \vdash \text{case}_C \ u \ \text{of} \ \text{inl}(x) \mapsto t_1 \mid \text{inr}(y) \mapsto t_2 : C}$$

$$\begin{array}{l} \text{case}_C \ \text{inl}(v) \ \text{of} \ \text{inl}(x) \mapsto t_1 \mid \text{inr}(y) \mapsto t_2 \quad \gamma^* \quad t_1\{x := v\} \\ \text{case}_C \ \text{inr}(v) \ \text{of} \ \text{inl}(x) \mapsto t_1 \mid \text{inr}(y) \mapsto t_2 \quad \gamma^* \quad t_2\{y := v\} \end{array}$$

Note: Expected canonical forms **only for basic datatypes** A, B

Finite types

Encoding of Fin_n ($n \geq 0$)

$$\text{Fin}_n \equiv \forall \gamma \underbrace{(\gamma \rightarrow \dots \rightarrow \gamma \rightarrow \gamma)}_{n \text{ times}}$$

$$\mathbf{e}_i \equiv \Lambda \gamma . \lambda x_1 : \gamma \dots \lambda x_n : \gamma . x_i \quad : \quad \text{Fin}_n \quad (1 \leq i \leq n)$$

Again, $\mathbf{e}_1, \dots, \mathbf{e}_n$ are the only closed normal terms of type Fin_n .

In particular:

$$\text{Fin}_2 \equiv \forall \gamma (\gamma \rightarrow \gamma \rightarrow \gamma) \equiv \text{Bool} \quad (\text{type of } \mathbf{booleans})$$

$$\text{Fin}_1 \equiv \forall \gamma (\gamma \rightarrow \gamma) \equiv \text{Unit} \quad (\mathbf{unit} \text{ data-type})$$

$$\text{Fin}_0 \equiv \forall \gamma \gamma \equiv \perp \quad (\mathbf{empty} \text{ data-type})$$

(Notice that there is no closed normal term of type \perp .)

Natural numbers

Encoding of the type of Church numerals

$$\text{Nat} \equiv \forall \gamma (\gamma \rightarrow (\gamma \rightarrow \gamma) \rightarrow \gamma)$$

$$\bar{0} \equiv \Lambda \gamma . \lambda x : \gamma . \lambda f : \gamma \rightarrow \gamma . x$$

$$\bar{1} \equiv \Lambda \gamma . \lambda x : \gamma . \lambda f : \gamma \rightarrow \gamma . f \ x$$

$$\bar{2} \equiv \Lambda \gamma . \lambda x : \gamma . \lambda f : \gamma \rightarrow \gamma . f \ (f \ x)$$

⋮

$$\bar{n} \equiv \Lambda \gamma . \lambda x : \gamma . \lambda f : \gamma \rightarrow \gamma . \underbrace{f(\dots(f \ x)\dots)}_{n \text{ times}} \quad : \quad \text{Nat}$$

⋮

Lemma (Canonical forms of type Nat)

The terms $\bar{0}, \bar{1}, \bar{2}, \dots$ are the only closed normal terms of type Nat.

Computing with natural numbers (1/2)

Intuition: Church numeral \bar{n} acts as an iterator:

$$\bar{n} A f x \quad \lambda^* \quad \underbrace{f (\dots (f x) \dots)}_n \quad (f : A \rightarrow A, \quad x : A)$$

- Successor

$$\text{succ} \equiv \lambda n : \text{Nat} . \Lambda \gamma . \lambda x : \gamma . \lambda f : \gamma \rightarrow \gamma . f (n \gamma x f)$$

- Addition

$$\begin{aligned} \text{plus} &\equiv \lambda n, m : \text{Nat} . \Lambda \gamma . \lambda x : \gamma . \lambda f : \gamma \rightarrow \gamma . m \gamma (n \gamma x f) f \\ \text{plus}' &\equiv \lambda n, m : \text{Nat} . m \text{ Nat } n \text{ succ} \end{aligned}$$

- Multiplication

$$\begin{aligned} \text{mult} &\equiv \lambda n, m : \text{Nat} . \Lambda \gamma . \lambda x : \gamma . \lambda f : \gamma \rightarrow \gamma . n \gamma x (\lambda y : \gamma . m \gamma y f) \\ \text{mult}' &\equiv \lambda n, m : \text{Nat} . n \text{ Nat } \bar{0} (\text{plus } m) \end{aligned}$$

Computing with natural numbers (2/2)

- Predecessor function $\text{pred} : \text{Nat} \rightarrow \text{Nat}$

$$\begin{aligned}\text{pred } \bar{0} &\simeq \bar{0} \\ \text{pred } (\overline{n+1}) &\simeq \bar{n}\end{aligned}$$

$$\begin{aligned}\text{fst} &\equiv \lambda p : \text{Nat} \times \text{Nat} . p \text{ Nat } (\lambda x, y : \text{Nat} . x) &: \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \\ \text{snd} &\equiv \lambda p : \text{Nat} \times \text{Nat} . p \text{ Nat } (\lambda x, y : \text{Nat} . y) &: \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \\ \text{step} &\equiv \lambda p : \text{Nat} \times \text{Nat} . \langle \text{snd } p, \text{succ } (\text{snd } p) \rangle &: \text{Nat} \times \text{Nat} \rightarrow \text{Nat} \times \text{Nat} \\ \text{pred} &\equiv \lambda n : \text{Nat} . \text{fst } (n (\text{Nat} \times \text{Nat}) \langle \bar{0}, \bar{0} \rangle \text{step}) &: \text{Nat} \rightarrow \text{Nat}\end{aligned}$$

- Ackerman function $\text{ack} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$

$$\begin{aligned}\text{ack } \bar{0} \quad \bar{m} &\simeq \overline{m+1} \\ \text{ack } (\overline{n+1}) \quad \bar{0} &\simeq \text{ack } \bar{n} \quad \bar{1} \\ \text{ack } (\overline{n+1}) \quad (\overline{m+1}) &\simeq \text{ack } \bar{n} \quad (\text{ack } (\overline{n+1}) \quad \bar{m})\end{aligned}$$

$$\begin{aligned}\text{down} &\equiv \lambda f : (\text{Nat} \rightarrow \text{Nat}) . \lambda p : \text{Nat} . p \text{ Nat } (f \bar{1}) f &: (\text{Nat} \rightarrow \text{Nat}) \rightarrow (\text{Nat} \rightarrow \text{Nat}) \\ \text{ack} &\equiv \lambda n, m : \text{Nat} . n (\text{Nat} \rightarrow \text{Nat}) \text{succ down } m &: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}\end{aligned}$$

- ▷ **SN** theorem guarantees that all well-typed computations terminate

Part III

System F: Curry-style presentation

System F polymorphism

ML/Haskell polymorphism

Types $A, B ::= \alpha \mid A \rightarrow B \mid \dots$ (user datatypes)

Schemes $S ::= \forall \vec{\alpha} B$

The type scheme $\forall \alpha B$ is defined **after** its particular instances $B\{\alpha := A\}$
 \Rightarrow Type system is **predicative**

System F polymorphism

Types $A, B ::= \alpha \mid A \rightarrow B \mid \forall \alpha B$

The type $\forall \alpha B$ and its instances $B\{\alpha := A\}$ are defined **simultaneously**

$\forall \alpha (\alpha \rightarrow \alpha)$ and $\forall \alpha (\alpha \rightarrow \alpha) \rightarrow \forall \alpha (\alpha \rightarrow \alpha)$

\Rightarrow Type system is **impredicative**, or **cyclic**

Extracting pure λ -terms

In Church-style system F , polymorphism is **explicit**:

$$\text{id} \equiv \Lambda\alpha. \lambda x : \alpha. x \quad \text{and} \quad \text{id Nat 2}$$

- Two kind of redexes $(\lambda x : A. t)u$ and $(\Lambda\alpha. t)A$

Idea: Remove type abstractions/applications/annotations

Erasing function $t \mapsto |t|$

$$\begin{array}{ll} |x| = x & \\ |\lambda x : A. t| = \lambda x. |t| & |\Lambda\alpha. t| = |t| \\ |tu| = |t||u| & |tA| = |t| \end{array}$$

- Target language is **pure λ -calculus**
- Second kind redexes are erased, first kind redexes are preserved

Extending the erasing function

Erased terms have a **nice computational behaviour**...

- Only one kind of redex, easy to execute (Krivine's machine)
- Irrelevant part of computation has been removed
- The essence of computation has been preserved (to be justified later)

... but **what is their status** w.r.t. typing?

The erasing function, defined on terms, can be extended to:

- The whole syntax
- The judgements
- The typing rules
- The derivations

⇒ Induces a new formalism: **Curry-style system F**

Church-style system F [Leivant 83]

Types	$A, B ::= \alpha \mid A \rightarrow B \mid \forall \alpha B$
Terms	$t, u ::= x \mid \lambda x:A. t \mid \mu tu \mid \Lambda \alpha. t \mid tA$
Judgments	$\Gamma ::= [] \mid \Gamma, x:A$
Reduction	$(\lambda x:A. t)u \rightarrow t\{x := u\}$ $(\Lambda \alpha. t)A \rightarrow t\{\alpha := A\}$

Remarks:

- Types (and contexts) are unchanged
- Terms are now **pure λ -terms**
- Only one kind of redex

Curry-style system of F-typing rules

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

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x. A t : A \rightarrow B}$$

$$\frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash t u : B}$$

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash \lambda \alpha. t : \forall \alpha. B} \quad \alpha \notin \text{FV}(\Gamma)$$

$$\frac{\Gamma \vdash t : \forall \alpha. B}{\Gamma \vdash t A : B \{ \alpha : A \}}$$

⇒ Rules are no more syntax directed

Curry-style system F: properties

Things that do not change

- Substitutivity + β -subject reduction
- Strong normalisation (postponed)

Things that change

- A term may have several types

$$\begin{aligned}\Delta \equiv \lambda x. x x & : \forall \alpha (\alpha \rightarrow \alpha) \rightarrow \forall \alpha (\alpha \rightarrow \alpha) \\ & : \forall \alpha \alpha \rightarrow \forall \alpha \alpha \\ & : \forall \alpha \alpha \rightarrow \forall \alpha (\alpha \rightarrow \alpha) \\ & : \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \quad (\text{'or' function!})\end{aligned}$$

- No principal type (cf later)
- Type checking/inference becomes **undecidable** [Wells 94]

Erasing and typing

Equivalence between Church and Curry's presentations

- 1 If $\Gamma \vdash t_0 : A$ (Church), then $\Gamma \vdash |t_0| : A$ (Curry)
- 2 If $\Gamma \vdash t : A$ (Curry), then $\Gamma \vdash t_0 : A$ (Church)
for some t_0 s.t. $|t_0| = t$

The erasing function maps:

- | <u>Church's world</u> | | <u>Curry's world</u> | |
|-----------------------|----|----------------------|---------------------------|
| 1. derivations | to | derivations | (isomorphism) |
| 2. valid judgements | to | valid judgements | (surjective only) |



On valid judgements, erasing is **not injective**:

$$\begin{aligned} \lambda f : (\forall \alpha (\alpha \rightarrow \alpha)) . f(\forall \alpha (\alpha \rightarrow \alpha))f & : \forall \alpha (\alpha \rightarrow \alpha) \rightarrow \forall \alpha (\alpha \rightarrow \alpha) \\ \lambda f : (\forall \alpha (\alpha \rightarrow \alpha)) . \lambda \alpha . f(\alpha \rightarrow \alpha)(f\alpha) & : \forall \alpha (\alpha \rightarrow \alpha) \rightarrow \forall \alpha (\alpha \rightarrow \alpha) \\ \rightsquigarrow \lambda f . ff & : \forall \alpha (\alpha \rightarrow \alpha) \rightarrow \forall \alpha (\alpha \rightarrow \alpha) \end{aligned}$$

Erasing and reduction

Second-kind redexes are **erased**, first-kind redexes are **preserved**

$$\begin{array}{l} \text{(Church)} \quad (\Lambda\alpha. \lambda x : \alpha. x) B y \succ (\lambda x : B. x) y \succ y \\ \quad \downarrow \text{Erasing} \\ \text{(Curry)} \quad (\lambda x. x) y \equiv (\lambda x. x) y \succ y \end{array}$$

Fact 1 (Church to Curry):

If $t_0, t'_0 \in \text{Church}$, then

$$t \succ^n t' \Rightarrow |t_0| \succ^p |t'_0| \quad (\text{with } p \leq n)$$

Fact 2 (Curry to Church):

If $t_0 \in \text{Church}$, $t' \in \text{Curry}$ and t_0 **well-typed**, then

$$|t_0| \succ^p t' \Rightarrow \exists t'_0 (|t'_0| = t' \wedge t_0 \succ^n t'_0) \quad (\text{with } n \geq p)$$

Normalisation equivalence

Fact 3 (Combinatorial argument):

- 1 During the contraction of a 1st-kind redex, the number of redexes of both kinds may increase
- 2 During the contraction of a 2nd-kind redex
 - the number of 1st-kind redexes may increase
 - the number of 2nd-kind redexes does not increase
 - the number of **type abstractions** ($\lambda\alpha . t$) **decreases**

Combining facts 1, 2 and 3, we easily prove:

Theorem (Normalisation equivalence):

The following statements are **combinatorially** equivalent:

- 1 All typable terms of syst. F -Church are strongly normalisable
- 2 All typable terms of syst. F -Curry are strongly normalisable

Subtyping

In Curry-style system F , **subtyping** is introduced as a **macro**:

$$A \leq B \equiv x : A \vdash x : B$$

Admissible rules

(Reflexivity, transitivity) $\frac{}{A \leq A} \quad \frac{A \leq B \quad B \leq C}{A \leq C}$

(Polymorphism) $\frac{}{\forall \alpha B \leq B\{\alpha := A\}} \quad \frac{A \leq B}{A \leq \forall \alpha B} \quad \alpha \notin TV(A)$

(Subsumption) $\frac{\Gamma \vdash t : A \quad A \leq B}{\Gamma \vdash t : B}$

Problem with η -redexes in Curry-style system F

- The (desired) subtyping rule for arrow-types

$$\frac{A \leq A' \quad B \leq B'}{A' \rightarrow B \leq A \rightarrow B'}$$

is **not admissible**

- In particular, we have: $f : \text{Nat} \rightarrow \forall \beta \beta \not\vdash f : \forall \alpha \alpha \rightarrow \text{Bool}$
but if we η -expand: $f : \text{Nat} \rightarrow \forall \beta \beta \vdash \lambda x. fx : \forall \alpha \alpha \rightarrow \text{Bool}$
- This shows that:
 - Curry-style system F does not enjoy η -subject reduction
 - This problem is connected with subtyping in arrow-types

The well-typed term: $\lambda x. fx : (\forall \alpha \alpha) \rightarrow \text{Bool}$ (Curry-style)
comes from the term $\lambda x : (\forall \alpha \alpha). f (x \text{ Nat}) \text{ Bool}$ (Church-style)

$\underbrace{\hspace{15em}}_{\text{not an } \eta\text{-redex}}$

System F_η [Mitchell 88]

Extend Curry-style system F with a new rule

$$\frac{\Gamma \vdash \lambda x. tx : A}{\Gamma \vdash t : A} \quad x \notin FV(t)$$

to enforce η -subject reduction

Properties:

- Substitutivity, $\beta\eta$ -subject-reduction, strong normalisation

- Subtyping rule $\frac{A \leq A' \quad B \leq B'}{A' \rightarrow B \leq A \rightarrow B'}$ is now **admissible**

Expansion lemma

If $\Gamma \vdash t : A$ is derivable in F_η , then $\Gamma \vdash t' : A$ is derivable in system F for some η -expansion t' of the term t .

More subtyping

If we set

$$\begin{aligned}\perp &:= \forall \gamma \gamma \\ A \times B &:= \forall \gamma ((A \rightarrow B \rightarrow \gamma) \rightarrow \gamma) \\ A + B &:= \forall \gamma ((A \rightarrow \gamma) \rightarrow (B \rightarrow \gamma) \rightarrow \gamma) \\ \text{List}(A) &:= \forall \gamma (\gamma \rightarrow (A \rightarrow \gamma \rightarrow \gamma) \rightarrow \gamma)\end{aligned}$$

then, in F_η , the following subtyping rules are admissible:

$$\begin{array}{c} \frac{}{\perp \leq A} \qquad \frac{A \leq A'}{\text{List}(A) \leq \text{List}(A')} \\ \\ \frac{A \leq A' \quad B \leq B'}{A \times B \leq A' \times B'} \qquad \frac{A \leq A' \quad B \leq B'}{A + B \leq A' + B'}\end{array}$$



But most typable terms have no **principal type**

Adding intersection types

Extend system F_η with **binary intersections**

Types $A, B ::= \alpha \mid A \rightarrow B \mid \forall \alpha B \mid A \cap B$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash t : B}{\Gamma \vdash t : A \cap B}$$

$$\frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : A}$$

$$\frac{\Gamma \vdash t : A \cap B}{\Gamma \vdash t : B}$$

- $\beta\eta$ -subject reduction, strong normalisation, etc.
- Subtyping rules

$$\frac{}{A \cap B \leq A} \quad \frac{}{A \cap B \leq B} \quad \frac{C \leq A \quad C \leq B}{C \leq A \cap B}$$

- All the **strongly normalising terms** are **typable**...
... but nothing to do with \forall : already true in $\lambda \rightarrow \cap$
- All typable terms have a **principal type**
 $\lambda x : xx. : \forall \alpha \forall \beta ((\alpha \rightarrow \beta) \cap \alpha \rightarrow \beta)$

Part IV

The Strong Normalisation Theorem

The meaning of second-order quantification (1/2)

Question: What is the meaning of $\forall\alpha (\alpha \rightarrow \alpha)$?

First scenario: an **infinite Cartesian product** (à la Martin-Löf)

$$\begin{aligned}\forall\alpha (\alpha \rightarrow \alpha) &\approx \prod_{\alpha \text{ type}} (\alpha \rightarrow \alpha) \\ &\approx (\perp \rightarrow \perp) \times (\text{Bool} \rightarrow \text{Bool}) \times (\text{Nat} \rightarrow \text{Nat}) \times \dots\end{aligned}$$

Since all the types $A \rightarrow A$ are inhabited:

- 1 The cartesian product $\forall\alpha (\alpha \rightarrow \alpha)$ should be **larger** than all the types of the form $A \rightarrow A$
- 2 In particular, $\forall\alpha (\alpha \rightarrow \alpha)$ should be larger than its own function space $\forall\alpha (\alpha \rightarrow \alpha) \rightarrow \forall\alpha (\alpha \rightarrow \alpha) \dots$

... seems to be very confusing!

The meaning of second-order quantification (2/2)

Second scenario: In F -Curry, both rules \forall -intro and \forall -elim

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall \alpha B} \quad \alpha \notin TV(\Gamma) \qquad \frac{\Gamma \vdash t : \forall \alpha B}{\Gamma \vdash t : B\{\alpha := A\}}$$

suggest that \forall is not a cartesian product, but an **intersection**

Taking back our example:

- 1 The intersection $\forall \alpha (\alpha \rightarrow \alpha)$ is **smaller** than all $A \rightarrow A$
- 2 In particular, $\forall \alpha (\alpha \rightarrow \alpha)$ is smaller than its own function space $\forall \alpha (\alpha \rightarrow \alpha) \rightarrow \forall \alpha (\alpha \rightarrow \alpha) \dots$

... our intuition feels much better!

\Rightarrow We will prove **strong normalisation** for Curry-style system F

Remember that $SN(F\text{-Church}) \Leftrightarrow SN(F\text{-Curry})$ (combinatorial equivalence)

Strong normalisation: the difficulty

Try to prove that

$$\Gamma \vdash t : A \Rightarrow t \text{ is SN}$$

by **induction on the derivation** of $\Gamma \vdash t : A$

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

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x. t : A \rightarrow B}$$

$$\frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B}$$

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash t : \forall \alpha B} \quad \alpha \notin TV(\Gamma)$$

$$\frac{\Gamma \vdash t : \forall \alpha B}{\Gamma \vdash t : B\{\alpha := A\}}$$

All the cases successfully pass the test except **application**

Two terms t and u may be SN, whereas tu is not [Take $t \equiv u \equiv \lambda x. xx$]

\Rightarrow The induction hypothesis “ t is SN” is too weak (in general)

Reducibility candidates [Girard 1971]

To prove that

$$\Gamma \vdash t : A \Rightarrow t \text{ is SN},$$

the induction hypothesis “ t is SN” is too weak.

\Rightarrow Should replace it by an invariant that **depends on the type A**

Intuition:

*The more complex the type, the stronger its invariant,
the smaller the set of terms that fulfill this invariant*

Invariants are represented by suitable sets of terms:

- **Reducibility candidates** [Girard], or
- **Saturated sets** [Tait]

Outline of the proof

- 1 Define a suitable notion of **reducibility candidate**
= the sets of λ -terms that will interpret/represent **types**
(Here, we use Tait's **saturated sets**)
- 2 Ensure that the notion of candidate captures the property of **strong normalisation** (which we want to prove)
Each candidate should only contain strongly normalisable λ -terms as elements
- 3 Associate to each type A a reducibility candidate $\llbracket A \rrbracket$
Type constructors ' \rightarrow ' and ' \forall ' have to be reflected at the level of candidates
- 4 Check (by induction) that $\Gamma \vdash t : A$ implies $t \in \llbracket A \rrbracket$
This is actually a little bit more complex, since we must take care of the typing context
- 5 Conclude that any well-typed term t is SN by step 2.

Preliminaries (1/2)

- **Notations:**

Λ \equiv set of all untyped λ -terms (open & closed)

SN \equiv set of all strongly normalisable untyped λ -terms

Var \equiv set of all (term) variables

$TVar$ \equiv set of all type variables

- A **reduct** of a term t is a term t' such that $t \succ t'$ (**one step**)

The number of reducts of a given term is finite and bounded by the number of redexes

- A finite **reduction sequence** of a term t is a finite sequence

$(t_i)_{i \in [0..n]}$ such that $t = t_0 \succ t_1 \succ \dots \succ t_{n-1} \succ t_n$

Infinite reduction sequences are defined similarly, by replacing $[0..n]$ by \mathbb{N}

- Finite reduction sequences of a term t form a tree, called the **reduction tree** of t

Preliminaries (2/2)

Definition (Strongly normalisable terms)

A term t is **strongly normalisable** if all the reduction sequences starting from t are finite

Proposition

The following assertions are equivalent:

- 1 t is strongly normalisable
- 2 All the reducts of t are strongly normalisable
- 3 The reduction tree of t is finite

Saturated sets [Tait]

Definition (Saturated set)

A set $S \subset \Lambda$ is **saturated** if:

$$\text{(SAT1)} \quad S \subset \text{SN}$$

$$\text{(SAT2)} \quad x \in \text{Var}, \quad \vec{v} \in \text{list}(\text{SN}) \Rightarrow x\vec{v} \in S$$

$$\text{(SAT3)} \quad t\{x := u\}\vec{v} \in S, \quad u \in \text{SN} \Rightarrow (\lambda x. t)u\vec{v} \in S$$

- (SAT1) expresses the property we want to prove
- Saturated sets contain **all the variables** (SAT2)
Extra-arguments $\vec{v} \in \text{list}(\text{SN})$ are here for technical reasons
- Saturated sets are closed under **head β -expansion** (SAT3)
Notice the condition $u \in \text{SN}$ to avoid a clash with (SAT1) for K-redexes
- The set of all saturated sets is written **SAT** $[\subset \wp(\text{SN}) \subset \wp(\Lambda)]$

Properties of saturated sets

Proposition (Lattice structure)

- 1 SN is a saturated set
- 2 **SAT** is closed under arbitrary non-empty intersections/unions:

$$I \neq \emptyset, (S_i)_{i \in I} \in \mathbf{SAT}' \Rightarrow \left(\bigcap_{i \in I} S_i \right), \left(\bigcup_{i \in I} S_i \right) \in \mathbf{SAT}$$

(**SAT**, \subset) is a **complete distributive lattice**, with

$\top = \text{SN}$ and $\perp = \{t \in \text{SN} \mid t \succ^* x u_1 \cdots u_n\}$ (Neutral terms)

Realisability arrow: For all $S, T \subset \Lambda$ we set

$$S \rightarrow T := \{t \in \Lambda \mid \forall u \in S \ tu \in T\}$$

Proposition (Closure under realisability arrow)

If $S, T \in \mathbf{SAT}$, then $(S \rightarrow T) \in \mathbf{SAT}$

Interpreting types (1/2)

Principle: Interpret **syntactic types** by **saturated sets**

- Type arrow $A \rightarrow B$ is interpreted by $S \rightarrow T$ (realisability arrow)
- Type quantification $\forall \alpha \dots$ is interpreted by the intersection $\bigcap_{S \in \text{SAT}} \dots$

Remark: this intersection is **impredicative** since S ranges over all saturated sets

Example: $\forall \alpha (\alpha \rightarrow \alpha)$ should be interpreted by $\bigcap_{S \in \text{SAT}} (S \rightarrow S)$

To interpret type variables, use type valuations:

Definition (Type valuations)

A **type valuation** is a function $\rho : \text{TVar} \rightarrow \text{SAT}$

The set of type valuations is written TVAl ($= \text{TVar} \rightarrow \text{SAT}$)

Interpreting types (2/2)

By induction on A , we define a function $\llbracket A \rrbracket : \text{TVal} \rightarrow \mathbf{SAT}$

$$\llbracket A \rightarrow B \rrbracket_\rho = \llbracket A \rrbracket_\rho \rightarrow \llbracket B \rrbracket_\rho \qquad \llbracket \alpha \rrbracket_\rho = \rho(\alpha)$$

$$\llbracket \forall \alpha B \rrbracket_\rho = \bigcap_{S \in \mathbf{SAT}} \llbracket B \rrbracket_{\rho; \alpha \leftarrow S}$$

Note: $(\rho; \alpha \leftarrow S)$ is defined by $\begin{cases} (\rho; \alpha \leftarrow S)(\alpha) = S \\ (\rho; \alpha \leftarrow S)(\beta) = \rho(\beta) \text{ for all } \beta \neq \alpha \end{cases}$

Problem: The implication

$$\Gamma \vdash t : A \quad \Rightarrow \quad t \in \llbracket A \rrbracket_\rho$$

cannot be proved directly. (One has to take care of the context)

\Rightarrow Strengthen induction hypothesis using **substitutions**

Substitutions

Definition (Substitutions)

A **substitution** is a finite list $\sigma = [x_1 := u_1; \dots; x_n := u_n]$ where $x_i \neq x_j$ (for $i \neq j$) and $u_i \in \Lambda$

Application of a substitution σ to a term t is written $t[\sigma]$

Exercise: Define it formally

Definition (Interpretation of contexts)

For all $\Gamma = x_1 : A_1; \dots; x_n : A_n$ and $\rho \in \text{TVal}$ set:

$$\llbracket \Gamma \rrbracket_\rho = \{ \sigma = [x_1 := u_1; \dots; x_n := u_n]; u_i \in \llbracket A_i \rrbracket_\rho \ (i = 1..n) \}$$

Substitutions $\sigma \in \llbracket \Gamma \rrbracket_\rho$ are said to be **adapted** to the context Γ (in the type valuation ρ)

The strong normalisation invariant

Lemma (Strong normalisation invariant)

If $\Gamma \vdash t : A$ in Curry-style system F , then

$$\forall \rho \in \text{TVal} \quad \forall \sigma \in \llbracket \Gamma \rrbracket_\rho \quad t[\sigma] \in \llbracket A \rrbracket_\rho$$

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

Exercise: Write down the 5 cases completely

Theorem (Strong normalisation)

The typable terms of F -Curry are strongly normalisable

Proof. Assume $x_1 : A_1; \dots; x_n : A_n \vdash t : B$

Corollary (Church-style SN)

The typable terms of F -Church are strongly normalisable

$$\sigma = [x_1 := x_1; \dots; x_n := x_n] \in \llbracket x_1 : A_1; \dots; x_n : A_n \rrbracket_\rho$$

From the lemma we get $t = t[\sigma] \in \llbracket B \rrbracket_\rho$, hence $t \in \text{SN}$ (SAT1)

A remark on impredicativity

In the SN proof, interpretation of \forall relies on the property:

*If $(S_i)_{i \in I}$ ($I \neq \emptyset$) is a family of saturated sets,
then $\bigcap_{i \in I} S_i$ is a saturated set*

in the special case where $I = \mathbf{SAT}$ (**impredicative intersection**)

- In 'classical' mathematics, this construction is legal
 \Rightarrow Standard set theories (Z, ZF, ZFC) are impredicative
- In (Bishop, Martin-Löf's style) constructive mathematics, this principle is rejected, mainly for philosophical reasons:
 - No convincing 'constructive' explanation
 - Suspicion about (this kind of) cyclicity

Impredicativity: An example (1/2)

Assume E is a vector space, S a set of vectors.

How to define the sub-vector space $\bar{S} \subset E$ **generated** by S in E ?

Standard 'abstract' method:

- 1 Consider the set: $\mathfrak{G} = \{F; F \text{ is a sub-vector space of } E \text{ and } F \supset S\}$
- 2 Fact: \mathfrak{G} is non empty, since $E \in \mathfrak{G}$
- 3 Take: $\bar{S} = \bigcap_{F \in \mathfrak{G}} F$
- 4 By definition, S is included in all the sub-spaces of E containing S
- 5 But \bar{S} is itself a sub-vector space of E containing S (so that $\bar{S} \in \mathfrak{G}$)
- 6 So that \bar{S} is actually the smallest of all such spaces

This definition is **impredicative** (step 3) (but legal in 'classical' mathematics)

The set \bar{S} is defined *from* \mathfrak{G} , that already contains \bar{S} as an element
discovered **a fortiori**

Impredicativity: An example (2/2)

But there are other ways of defining \bar{S} ...

- **Standard 'concrete' definition, by linear combinations:**

Let \bar{S} be the set of all vectors of the form $v = \alpha_1 \cdot v_1 + \dots + \alpha_n \cdot v_n$

where (v_i) ranges over all the finite families of elements of S ,

and (α_i) ranges over all the finite families of scalars

- **Inductive definition:**

Let \bar{S} be the set inductively defined by:

- 1 $\vec{0} \in \bar{S}$,
- 2 If $v \in S$, then $v \in \bar{S}$,
- 3 If $v \in \bar{S}$ and α is a scalar, then $\alpha \cdot v \in \bar{S}$
- 4 If $v_1 \in \bar{S}$ and $v_2 \in \bar{S}$, then $v_1 + v_2 \in \bar{S}$.

\Rightarrow Both definitions are **predicative** (and give the same object)