

data61.csiro.au

Last time...



- → λ calculus syntax
- \rightarrow free variables, substitution
- $\rightarrow \beta$ reduction
- $\clubsuit~\alpha$ and η conversion
- \clubsuit β reduction is confluent
- → λ calculus is expressive (Turing complete)
- → λ calculus is inconsistent (as a logic)

Content

intent	DATA DATA
→ Intro & motivation, getting started	
 → Foundations & Principles Lambda Calculus, natural deduction Higher Order Logic Term rewriting 	[1,2] [3ª] [4]
 → Proof & Specification Techniques Inductively defined sets, rule induction Datatypes, recursion, induction Hoare logic, proofs about programs, invariants (mid-semester break) C verification CakeML, Isar Concurrency 	[5] [6, 7] $[8^b,9]$ [10] $[11^c]$ [12]

^aa1 due; ^ba2 due; ^ca3 due

λ calculus is inconsistent



Can find term R such that $R R =_{\beta} \operatorname{not}(R R)$

There are more terms that do not make sense:

12, true false, etc.

Solution: rule out ill-formed terms by using types. (Church 1940)

Introducing types

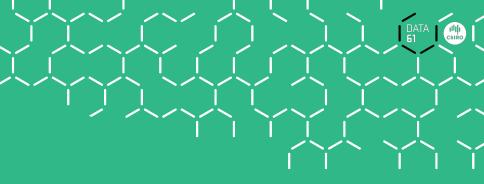


Idea: assign a type to each "sensible" λ term.

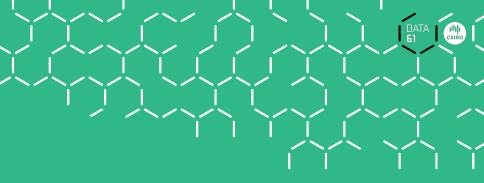
Examples:

- \rightarrow for term t has type α write $t :: \alpha$
- → if x has type α then $\lambda x. x$ is a function from α to α Write: $(\lambda x. x) :: \alpha \Rightarrow \alpha$
- → for s t to be sensible: s must be a function t must be right type for parameter

If
$$s :: \alpha \Rightarrow \beta$$
 and $t :: \alpha$ then $(s \ t) :: \beta$



That's about it



Now formally again

Syntax for λ^{\rightarrow}



Terms:
$$t ::= v \mid c \mid (t \ t) \mid (\lambda x. \ t)$$

 $v, x \in V, c \in C, V, C$ sets of names

$$\alpha \Rightarrow \beta \Rightarrow \gamma \quad = \quad \alpha \Rightarrow (\beta \Rightarrow \gamma)$$

Context Γ:

 Γ : function from variable and constant names to types.

Term *t* has type τ in context Γ : $\Gamma \vdash t :: \tau$

Examples

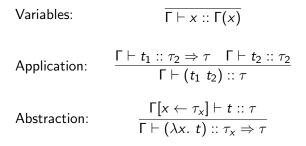


$$\begin{array}{l} \Gamma \vdash (\lambda x. \ x) ::: \alpha \Rightarrow \alpha \\ [y \leftarrow \operatorname{int}] \vdash y :: \operatorname{int} \\ [z \leftarrow \operatorname{bool}] \vdash (\lambda y. \ y) \ z :: \operatorname{bool} \\ [] \vdash \lambda f \ x. \ f \ x :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta \end{array}$$

A term t is **well typed** or **type correct** if there are Γ and τ such that $\Gamma \vdash t :: \tau$

Type Checking Rules





10 | COMP4161 | © Data61, CSIRO: provided under Creative Commons Attribution License

Example Type Derivation:



$$\frac{\overline{[x \leftarrow \alpha, y \leftarrow \beta] \vdash x :: \alpha}}{[x \leftarrow \alpha] \vdash \lambda y. \ x :: \beta \Rightarrow \alpha}$$
$$\overline{[] \vdash \lambda x \ y. \ x :: \alpha \Rightarrow \beta \Rightarrow \alpha}$$

11 | COMP4161 | © Data61, CSIRO: provided under Creative Commons Attribution License

More complex Example



$$\frac{\Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta) \quad \overline{\Gamma \vdash x :: \alpha}}{\Gamma \vdash f :: \alpha \Rightarrow \beta} \quad \overline{\Gamma \vdash x :: \alpha} \\
\frac{\Gamma \vdash f :: \alpha \Rightarrow \beta}{[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. \ f : x :: \alpha \Rightarrow \beta} \\
\frac{\overline{[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. \ f : x :: \alpha \Rightarrow \beta}}{[] \vdash \lambda f : x. \ f : x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}$$

$$\mathsf{F} = [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta, x \leftarrow \alpha]$$

12 | COMP4161 | © Data61, CSIRO: provided under Creative Commons Attribution License

More general Types



A term can have more than one type.

Example: []
$$\vdash \lambda x. x :: \text{bool} \Rightarrow \text{bool}$$

[] $\vdash \lambda x. x :: \alpha \Rightarrow \alpha$

Some types are more general than others:

 $au \lesssim \sigma$ if there is a substitution S such that $au = S(\sigma)$

Examples:

$$\texttt{int} \Rightarrow \texttt{bool} \quad \lesssim \quad \alpha \Rightarrow \beta \quad \lesssim \quad \beta \Rightarrow \alpha \quad \not\lesssim \quad \alpha \Rightarrow \alpha$$

Most general Types



Fact: each type correct term has a most general type

Formally:

 $\Gamma \vdash t :: \tau \implies \exists \sigma. \ \Gamma \vdash t :: \sigma \land (\forall \sigma'. \ \Gamma \vdash t :: \sigma' \Longrightarrow \sigma' \lesssim \sigma)$

It can be found by executing the typing rules backwards.

- → type checking: checking if $\Gamma \vdash t :: \tau$ for given Γ and τ
- → type inference: computing Γ and τ such that $\Gamma \vdash t :: \tau$

Type checking and type inference on λ^{\rightarrow} are decidable.

What about β reduction?



Definition of β reduction stays the same.

Fact: Well typed terms stay well typed during β reduction

Formally: $\Gamma \vdash s :: \tau \land s \longrightarrow_{\beta} t \Longrightarrow \Gamma \vdash t :: \tau$

This property is called **subject reduction**

What about termination?



β reduction in λ^{\rightarrow} always terminates.



(Alan Turing, 1942)

 $\rightarrow =_{\beta}$ is decidable

To decide if $s =_{\beta} t$, reduce *s* and *t* to normal form (always exists, because \longrightarrow_{β} terminates), and compare result.

→ $=_{\alpha\beta\eta}$ is decidable This is why Isabelle can automatically reduce each term to $\beta\eta$ normal form.

What does this mean for Expressiveness?



Not all computable functions can be expressed in λ^{\rightarrow} !

How can typed functional languages then be turing complete?

Fact:

Each computable function can be encoded as closed, type correct λ^{\rightarrow} term using $Y :: (\tau \Rightarrow \tau) \Rightarrow \tau$ with $Y \ t \longrightarrow_{\beta} t \ (Y \ t)$ as only constant.

- \rightarrow Y is called fix point operator
- \rightarrow used for recursion
- → lose decidability (what does $Y(\lambda x. x)$ reduce to?)
- → (Isabelle/HOL doesn't have Y; it supports more restricted forms of recursion)

Types and Terms in Isabelle

$$\begin{bmatrix} DATA \\ \mathbf{51} \end{bmatrix} \bigoplus_{K}$$

Types:
$$\tau ::= b \mid \nu \mid \nu :: C \mid \tau \Rightarrow \tau \mid (\tau, \tau)$$

$$b \in \{bool, int, ...\} base types$$
$$\nu \in \{\alpha, \beta, ...\} type variables$$
$$K \in \{set, list, ...\} type constructors$$
$$C \in \{order, linord, ...\} type classes$$

Terms:
$$t ::= v \mid c \mid ?v \mid (t \ t) \mid (\lambda x. \ t)$$

 $v, x \in V, c \in C, V, C$ sets of names

- → type constructors: construct a new type out of a parameter type. Example: int list
- → type classes: restrict type variables to a class defined by axioms. Example: α :: order
- → schematic variables: variables that can be instantiated.

Type Classes



- → similar to Haskell's type classes, but with semantic properties
 class order =
 assumes order_refl: "x ≤ x"
 assumes order_trans: "[[x ≤ y; y ≤ z]] ⇒ x ≤ z"
- \rightarrow theorems can be proved in the abstract

lemma order_less_trans:

 $" \bigwedge x :::'a :: order. \llbracket x < y; y < z \rrbracket \Longrightarrow x < z"$

→ can be used for subtyping

class linorder = order +

assumes linorder_linear: " $x \le y \lor y \le x$ "

→ can be instantiated

instance nat :: "{order, linorder}" by ...

Schematic Variables





 \rightarrow X and Y must be **instantiated** to apply the rule

But: lemma "x + 0 = 0 + x"

- $\rightarrow x$ is free
- \rightarrow convention: lemma must be true for all x
- → during the proof, x must not be instantiated

Solution:

Isabelle has free (x), bound (x), and schematic (?X) variables.

Only schematic variables can be instantiated.

Free converted into schematic after proof is finished.

Higher Order Unification



Unification:

Find substitution σ on variables for terms s,t such that $\sigma(s) = \sigma(t)$

In Isabelle:

Find substitution σ on schematic variables such that $\sigma(s) =_{\alpha\beta\eta} \sigma(t)$

Examples:

$?X \land ?Y$	$=_{\alpha\beta\eta}$	$x \wedge x$	$[?X \leftarrow x, ?Y \leftarrow x]$
?P x	$=_{\alpha\beta\eta}$	$x \wedge x$	$[?P \leftarrow \lambda x. \ x \land x]$
P (?f x)	$=_{\alpha\beta\eta}$?Y x	$[?f \leftarrow \lambda x. \ x, ?Y \leftarrow P]$

Higher Order: schematic variables can be functions.

Higher Order Unification



- → Unification modulo $\alpha\beta$ (Higher Order Unification) is semi-decidable
- \clubsuit Unification modulo $\alpha\beta\eta$ is undecidable
- → Higher Order Unification has possibly infinitely many solutions

But:

- → Most cases are well-behaved
- → Important fragments (like Higher Order Patterns) are decidable

Higher Order Pattern:

- \clubsuit is a term in β normal form where
- \rightarrow each occurrence of a schematic variable is of the form ? $f t_1 \ldots t_n$
- \rightarrow and the $t_1 \ldots t_n$ are η -convertible into n distinct bound variables

We have learned so far...



- → Simply typed lambda calculus: λ^{\rightarrow}
- → Typing rules for λ^{\rightarrow} , type variables, type contexts
- \clubsuit $\beta\text{-reduction}$ in λ^{\rightarrow} satisfies subject reduction
- \clubsuit $\beta\text{-reduction}$ in λ^{\rightarrow} always terminates
- \rightarrow Types and terms in Isabelle