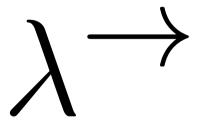


COMP 4161 NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray, Rafal Kolanski



Last time...

NICTA

- → λ calculus syntax
- ➔ free variables, substitution
- $\rightarrow \beta$ reduction
- → α and η conversion
- → β reduction is confluent
- → λ calculus is expressive (turing complete)
- → λ calculus is inconsistent

Content

Content	
Content	NICTA
→ Intro & motivation, getting started	[1]
→ Foundations & Principles	
 Lambda Calculus, natural deduction 	[1,2]
Higher Order Logic	[3 ^{<i>a</i>}]
Term rewriting	[4]
➔ Proof & Specification Techniques	
 Inductively defined sets, rule induction 	[5]
 Datatypes, recursion, induction 	[6 ^b , 7]
 Code generation, type classes 	[7]
 Hoare logic, proofs about programs, refinement 	[8,9 ^c ,10 ^d]
 Isar, locales 	[11,12]

 a a1 due; b a2 due; c session break; d a3 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: $(\lambda x. x) :: \alpha \Rightarrow \alpha$

```
\rightarrow for s t to be sensible:
```

 \boldsymbol{s} must be function

t must be right type for parameter

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



THAT'S ABOUT IT

Copyright NICTA 2012, provided under Creative Commons Attribution License



Now Formally Again

Copyright NICTA 2012, provided under Creative Commons Attribution License



Terms: $t ::= v \mid c \mid (t t) \mid (\lambda x. t)$ $v, x \in V, c \in C, V, C$ sets of names

Types: $\tau ::= b \mid \nu \mid \tau \Rightarrow \tau$ $b \in \{bool, int, ...\}$ base types $\nu \in \{\alpha, \beta, ...\}$ type variables

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

Context Γ :

 Γ : function from variable and constant names to types.

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

Examples



 $\Gamma \vdash (\lambda x. \ x) :: \alpha \Rightarrow \alpha$

 $[y \leftarrow \texttt{int}] \vdash y :: \texttt{int}$

 $[z \leftarrow \texttt{bool}] \vdash (\lambda y. \ y) \ z :: \texttt{bool}$

$$[] \vdash \lambda f \ x. \ f \ x :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$$

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



Variables:
$$\overline{\Gamma \vdash x :: \Gamma(x)}$$

Application:	$\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau_1 \Gamma \vdash t_2 :: \tau_2$
	$\Gamma \vdash (t_1 \ t_2) :: \tau_1$

Abstraction:	$\Gamma[x \leftarrow \tau_1] \vdash t :: \tau_2$
	$\overline{\Gamma \vdash (\lambda x. t) :: \tau_1 \Rightarrow \tau_2}$



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



$$\begin{array}{c} \overline{\Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta)} & \overline{\Gamma \vdash x :: \alpha} \\ \\ \hline \Gamma \vdash f x :: \alpha \Rightarrow \beta & \overline{\Gamma \vdash x :: \alpha} \\ \hline \Gamma \vdash f x x :: \beta \\ \hline \hline [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. \ f \ x \ x :: \alpha \Rightarrow \beta \\ \hline \hline [] \vdash \lambda f \ x. \ f \ x \ x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta \end{array}$$

$$\Gamma = [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta, x \leftarrow \alpha]$$



A term can have more than one type.

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

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

Some types are more general than others:

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

Examples:

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



Fact: each type correct term has a most general type

Formally:

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

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.



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**



 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.

$ightarrow =_{lphaeta\eta}$ is decidable

This is why Isabelle can automatically reduce each term to $\beta\eta$ normal form.



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
- → used for recursion
- → lose decidability (what does $Y (\lambda x. x)$ reduce to?)



Types: $\tau ::= b \mid '\nu \mid '\nu :: C \mid \tau \Rightarrow \tau \mid (\tau, ..., \tau) K$ $b \in \{bool, int, ...\}$ 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: " $\bigwedge x :::'a :: order. [x < y; y < z] \implies x < z$ "

 \rightarrow can be used for subtyping

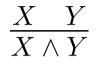
```
class linorder = order +
```

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

 \rightarrow can be instantiated

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





 \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
- \rightarrow 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

In Isabelle:

Unification:

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

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

Examples:

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

Higher Order: schematic variables can be functions.





- → Unification modulo $\alpha\beta$ (Higher Order Unification) is semi-decidable
- → 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:

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



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