



COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Toby Murray, June Andronick, Gerwin Klein



Slide 1



Last time...

- λ calculus syntax
- free variables, substitution
- β reduction
- α and η conversion
- β reduction is confluent
- λ calculus is expressive (turing complete)
- λ calculus is inconsistent

Slide 2



Content

- Intro & motivation, getting started [1]
- Foundations & Principles
 - Lambda Calculus, natural deduction [1,2]
 - Higher Order Logic [3^a]
 - Term rewriting [4]
- Proof & Specification Techniques
 - Inductively defined sets, rule induction [5]
 - Datatypes, recursion, induction [6, 7]
 - Hoare logic, proofs about programs, C verification [8^b,9]
 - (mid-semester break)
 - Writing Automated Proof Methods [10]
 - `lsar`, `codegen`, `typeclasses`, `locales` [11^c,12]

^aa1 due; ^ba2 due; ^ca3 due

Slide 3



λ calculus is inconsistent

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

There are more terms that do not make sense:

`! 2`, `true false`, etc.

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

Slide 4

Introducing types

Idea: assign a type to each “sensible” λ term.

Examples:

- 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 function
 - t must be right type for parameter
- If $s :: \alpha \Rightarrow \beta$ and $t :: \alpha$ then $(s t) :: \beta$

Slide 5



NOW FORMALLY AGAIN

Slide 7

THAT'S ABOUT IT

Slide 6



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

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

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

Context Γ :

Γ : function from variable and constant names to types.

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

Slide 8

Examples

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

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

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

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

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

Slide 9



Type Checking Rules

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

Application: $\frac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1 t_2) :: \tau}$

Abstraction: $\frac{\Gamma[x \leftarrow \tau_x] \vdash t :: \tau}{\Gamma \vdash (\lambda x. t) :: \tau_x \Rightarrow \tau}$

Slide 10



Example Type Derivation:

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

Slide 11



More complex Example

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

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

Slide 12



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:

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

Examples:

$\text{int} \Rightarrow \text{bool} \lesssim \alpha \Rightarrow \beta \lesssim \beta \Rightarrow \alpha \not\lesssim \alpha \Rightarrow \alpha$

Slide 13

Most general Types

Fact: each type correct term has a most general type

Formally:

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

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 $\lambda \rightarrow$ are decidable.

Slide 14



What about β reduction?

Definition of β reduction stays the same.

Fact: Well typed terms stay well typed during β reduction

Formally: $\Gamma \vdash s :: \tau \wedge s \rightarrow_{\beta} t \implies \Gamma \vdash t :: \tau$

This property is called **subject reduction**

Slide 15

What about termination?

β reduction in $\lambda \rightarrow$ always terminates.



(Alan Turing, 1942)

- **$=_{\beta}$ is decidable**
To decide if $s =_{\beta} t$, reduce s and t to normal form (always exists, because \rightarrow_{β} terminates), and compare result.
- **$=_{\alpha\beta\eta}$ is decidable**
This is why Isabelle can automatically reduce each term to $\beta\eta$ normal form.

Slide 16



What does this mean for Expressiveness?



Not all computable functions can be expressed in $\lambda \rightarrow$!

How can typed functional languages then be Turing complete?

Fact:

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

- Y is called fix point operator
- 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)

Slide 17

Types and Terms in Isabelle



Types: $\tau ::= b \mid 'v \mid 'v :: C \mid \tau \Rightarrow \tau \mid (\tau, \dots, \tau) K$
 $b \in \{\text{bool}, \text{int}, \dots\}$ base types
 $v \in \{\alpha, \beta, \dots\}$ type variables
 $K \in \{\text{set}, \text{list}, \dots\}$ type constructors
 $C \in \{\text{order}, \text{linord}, \dots\}$ 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: $\alpha :: \text{order}$
- **schematic variables:** variables that can be instantiated.

Slide 18

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"  
  ...
```

- theorems can be proved in the abstract

```
lemma order_less_trans: "∧ x :: 'a :: order. [x < y; y < z] ⇒ x < z"
```

- can be used for subtyping

```
class linorder = order +  
  assumes linorder_linear: "x ≤ y ∨ y ≤ x"
```

- can be instantiated

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

Slide 19

Schematic Variables



$$\frac{X \quad Y}{X \wedge Y}$$

- X and Y must be **instantiated** to apply the rule

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

- x is free
- 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.

Slide 20

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:

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

Higher Order: schematic variables can be functions.

Slide 21

Higher Order Unification



- 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:

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

Slide 22

We have learned so far...



- 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

Slide 23