



COMP4161: Advanced Topics in Software Verification



June Andronick, Christine Rizkallah, Miki Tanaka, Johannes Åman Pohjola  
T3/2019

[data61.csiro.au](http://data61.csiro.au)



# Last time...



- $\lambda$  calculus syntax
- free variables, substitution
- $\beta$  reduction
- $\alpha$  and  $\eta$  conversion
- $\beta$  reduction is confluent
- $\lambda$  calculus is expressive (Turing complete)
- $\lambda$  calculus is inconsistent (as a logic)

# Content



- Intro & motivation, getting started [1]
- Foundations & Principles
  - Lambda Calculus, natural deduction [1,2]
  - Higher Order Logic, Isar (part 1) [3<sup>a</sup>]
  - Term rewriting [4]
- Proof & Specification Techniques
  - Inductively defined sets, rule induction [5]
  - Datatypes, recursion, induction, Isar (part 2) [6, 7<sup>b</sup>]
  - Hoare logic, proofs about programs, invariants [8]
  - C verification [9]
  - Practice, questions, exam prep [10<sup>c</sup>]

---

<sup>a</sup>a1 due; <sup>b</sup>a2 due; <sup>c</sup>a3 due

# $\lambda$ 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:

1 2, true false, etc.

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

# Introducing types



**Idea:** assign a type to each “sensible”  $\lambda$  term.

## Examples:

- for *term*  $t$  has type  $\alpha$  write  $t :: \alpha$
- if  $x$  has type  $\alpha$  then  $\lambda x. x$  is a function from  $\alpha$  to  $\alpha$   
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 $\lambda^{\rightarrow}$



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

**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

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

**Context  $\Gamma$ :**

$\Gamma$ : function from variable and constant names to types.

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



# Examples


$$\Gamma \vdash (\lambda x. x) :: \alpha \Rightarrow \alpha$$
$$[y \leftarrow \text{int}] \vdash y :: \text{int}$$
$$[z \leftarrow \text{bool}] \vdash (\lambda y. y) z :: \text{bool}$$
$$[] \vdash \lambda f x. f x :: (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta$$

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

# Type Checking Rules



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

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

# Example Type Derivation:



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

# More complex Example

$$\begin{array}{c}
 \frac{\overline{\Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta)} \quad \overline{\Gamma \vdash x :: \alpha}}{\Gamma \vdash f \text{ } x :: \alpha \Rightarrow \beta} \quad \overline{\Gamma \vdash x :: \alpha} \\
 \frac{\Gamma \vdash f \text{ } x :: \alpha \Rightarrow \beta}{\Gamma \vdash f \text{ } x \text{ } x :: \beta} \\
 \frac{\overline{[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. f \text{ } x \text{ } x :: \alpha \Rightarrow \beta}}{\boxed{\phantom{}} \vdash \lambda f \text{ } x. f \text{ } x \text{ } x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}
 \end{array}$$

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

# More general Types



A term can have more than one type.

**Example:**  $\Box \vdash \lambda x. x :: \text{bool} \Rightarrow \text{bool}$   
 $\Box \vdash \lambda x. x :: \alpha \Rightarrow \alpha$

Some types are more general than others:

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

**Examples:**

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

# 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  $\Gamma$  and  $\tau$
- **type inference:** computing  $\Gamma$  and  $\tau$  such that  $\Gamma \vdash t :: \tau$

**Type checking and type inference on  $\lambda^{\rightarrow}$  are decidable.**

# What about $\beta$ reduction?



**Definition of  $\beta$  reduction stays the same.**

**Fact:** Well typed terms stay well typed during  $\beta$  reduction

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

This property is called **subject reduction**

# What about termination?

$\beta$  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  $\rightarrow_{\beta}$  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  $\lambda^{\rightarrow}$ !**

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  $Y\ t \longrightarrow_{\beta} t\ (Y\ t)$  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)

# Types and Terms in Isabelle



**Types:**  $\tau ::= b \mid '\nu \mid '\nu :: C \mid \tau \Rightarrow \tau \mid (\tau, \dots, \tau) K$

$b \in \{\text{bool}, \text{int}, \dots\}$  base types

$\nu \in \{\alpha, \beta, \dots\}$  type variables

$K \in \{\text{set}, \text{list}, \dots\}$  type constructors

$C \in \{\text{order}, \text{linord}, \dots\}$  type classes

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

$v, x \in V, \quad c \in C, \quad V, C$  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.

# Type Classes



- similar to Haskell's type classes, but with semantic properties

**class** order =

**assumes** order\_refl: " $x \leq x$ "

**assumes** order\_trans: " $\llbracket x \leq y; y \leq z \rrbracket \implies x \leq z$ "

    ...

- theorems can be proved in the abstract

**lemma** order\_less\_trans:

" $\bigwedge x :: 'a :: \text{order}. \llbracket x < y; y < z \rrbracket \implies x < z$ "

- can be used for subtyping

**class** linorder = order +

**assumes** linorder\_linear: " $x \leq y \vee y \leq x$ "

- can be instantiated

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

# 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** ( $\lambda x$ ), and **schematic** ( $\lambda x$ ) variables.

**Only schematic variables can be instantiated.**

Free converted into schematic after proof is finished.

# Higher Order Unification



## Unification:

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

## In Isabelle:

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

## Examples:

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

**Higher Order:** schematic variables can be functions.

# 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  $\beta$  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  $\eta$ -convertible into  $n$  distinct bound variables

# We have learned so far...



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