NICTA

# COMP 4161

NICTA Advanced Course

## Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray

$$\lambda^{\rightarrow}$$

# Exercises from last time

- Reduce $(\lambda x.\ y\ (\lambda v.\ x\ v))\ (\lambda y.\ v\ y)$ to $\beta\eta$ normal form.
- Find an encoding for function $fs$, $sn$, and $pair$ such that $fs\ (pair\ a\ b)\ =_\beta\ a$ and $sn\ (pair\ a\ b)\ =_\beta\ b$.
- (harder) Find an encoding of list objects, i.e. for the function $cons$ and $nil$. Then find an encoding for $map$ (that is, map $f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$), and for $foldl$ (that is, foldl $f\ i\ [x_1, \ldots, x_n] = f\ x_1\ (f\ x_2\ (f\ x_3\ (\ldots (f\ x_n\ i))) \ldots))$

# Content

Rough timeline

➜ Intro & motivation, getting started [1]

➜ Foundations & Principles

- Lambda Calculus, natural deduction [2,3,4[a]]
- Higher Order Logic [5,6[b],7]
- Term rewriting [8,9,10[c]]

➜ Proof & Specification Techniques

- Isar [11,12[d]]
- Inductively defined sets, rule induction [13[e],15]
- Datatypes, recursion, induction [16,17[f],18,19]
- Calculational reasoning, mathematics style proofs [20]
- Hoare logic, proofs about programs [21[g],22,23]

[a] a1 out; [b] a1 due; [c] a2 out; [d] a2 due; [e] session break; [f] a3 out; [g] a3 due

# $\lambda$ calculus is inconsistent

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

There are more terms that do not make sense:

$1\ 2$, $\mathtt{true}\ \mathtt{false}$, 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: $(\lambda x.\ x) :: \alpha \Rightarrow a$
- ➜ 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 $(s\ t) :: \beta$

# That's about it

# NOW FORMALLY AGAIN

# Syntax for $\lambda^{\rightarrow}$

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

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

**Types:** $\tau \quad ::= \quad \mathsf{b} \mid \nu \mid \tau \Rightarrow \tau$

$\mathsf{b} \in \{\texttt{bool}, \texttt{int}, \ldots\}$ base types

$\nu \in \{\alpha, \beta, \ldots\}$ 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 $\Gamma$:** $\quad \Gamma \vdash t :: \tau$

# Examples

NICTA

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

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

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

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

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

# Type Checking Rules

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

Application: $$\frac{\Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau_1 \quad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1\ t_2) :: \tau_1}$$

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

## Example Type Derivation:

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

# More complex Example

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta)} \quad \dfrac{}{\Gamma \vdash x :: \alpha}}{\Gamma \vdash {\color{blue} f\ x} :: \alpha \Rightarrow \beta} \qquad \dfrac{}{\Gamma \vdash {\color{red} x} :: \alpha}}{\Gamma \vdash {\color{blue} f\ x}\ {\color{red} x} :: \beta}}{\dfrac{[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x.\ f\ x\ x :: \alpha \Rightarrow \beta}{[] \vdash \lambda f\ x.\ f\ x\ x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta}}
$$

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

## More general Types

A term can have more than one type.

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

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

Some types are more general than others:

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

**Examples:**

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

## Most general Types

**Fact:** each type correct term has a most general type

**Formally:**

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

It can be found by executing the typing rules backwards.

- ➜ **type checking:** checking if $\Gamma \vdash t :: \tau$ for given $\Gamma$ 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:** $\Gamma \vdash s :: \tau \ \wedge \ s \longrightarrow_\beta t \Longrightarrow \Gamma \vdash t :: \tau$

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 $\longrightarrow_\beta$ 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?)

# Types and Terms in Isabelle

**Types:** $\tau \quad ::= \quad \mathsf{b} \mid {'\nu} \mid {'\nu :: C} \mid \tau \Rightarrow \tau \mid (\tau, \ldots, \tau)\ K$

$\mathsf{b} \in \{\texttt{bool}, \texttt{int}, \ldots\}$ base types

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

$K \in \{\texttt{set}, \texttt{list}, \ldots\}$ type constructors

$C \in \{\texttt{order}, \texttt{linord}, \ldots\}$ type classes

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

$v, x \in V, \quad c \in C, \quad V, C$ 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: $\alpha :: order$

➜ **schematic variables**: variables that can be instantiated.

# Type Classes

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

  **axclass** order $<$ ord

  order_refl: "$x \leq x$"

  order_trans: "$[\![x \leq y; y \leq z]\!] \Longrightarrow x \leq z$"

  ...

- theorems can be proved in the abstract

  **lemma** order_less_trans: "$\bigwedge x :: 'a :: order.\ [\![x < y; y < z]\!] \Longrightarrow x < z$"

- can be used for subtyping

  **axclass** linorder $<$ order

  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** (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 $\sigma$ 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

NICTA

- 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 from $?f\ t_1\ \ldots\ t_n$
- and the $t_1\ \ldots\ t_n$ are $\eta$-convertible into $n$ distinct bound variables

# We have learned so far...

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

# Exercises

- Construct a type derivation tree for the term $\lambda x\ y\ z.\ z\ x\ (y\ x)$
- Find a unifier (substitution) such that $\lambda x\ y\ z.\ ?F\ y\ z = \lambda x\ y\ z.\ z\ (?G\ x\ y)$