COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein
Formal Methods

$$\lambda^{\longrightarrow}$$

So, what can you do with λ calculus?

λ calculus is very expressive, you can encode:

- ➜ logic, set theory
- ➜ turing machines, functional programs, etc.

Examples:

$\texttt{true} \equiv \lambda x\ y.\ x$	$\texttt{if}\ \texttt{true}\ x\ y \longrightarrow_\beta^* x$
$\texttt{false} \equiv \lambda x\ y.\ y$	$\texttt{if}\ \texttt{false}\ x\ y \longrightarrow_\beta^* y$
$\texttt{if} \equiv \lambda z\ x\ y.\ z\ x\ y$	

Now, `not`, `and`, `or`, etc is easy:

$\texttt{not} \equiv \lambda x.\ \texttt{if}\ x\ \texttt{false}\ \texttt{true}$
$\texttt{and} \equiv \lambda x\ y.\ \texttt{if}\ x\ y\ \texttt{false}$
$\texttt{or} \equiv \lambda x\ y.\ \texttt{if}\ x\ \texttt{true}\ y$

More Examples

Encoding natural numbers (Church Numerals)

$0 \equiv \lambda f\ x.\ x$
$1 \equiv \lambda f\ x.\ f\ x$
$2 \equiv \lambda f\ x.\ f\ (f\ x)$
$3 \equiv \lambda f\ x.\ f\ (f\ (f\ x))$
$\ldots$

Numeral n takes arguments f and x, applies f n-times to x.

$\texttt{iszero} \equiv \lambda n.\ n\ (\lambda x.\ \texttt{false})\ \texttt{true}$
$\texttt{succ} \equiv \lambda n\ f\ x.\ f\ (n\ f\ x)$
$\texttt{add} \equiv \lambda m\ n.\ \lambda f\ x.\ m\ f\ (n\ f\ x)$

Fix Points

$(\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ \ t \longrightarrow_\beta$
$(\lambda f.\ f\ ((\lambda x\ f.\ f\ (x\ x\ f))\ \ (\lambda x\ f.\ f\ (x\ x\ f))\ f))\ \ t \longrightarrow_\beta$
$t\ \ ((\lambda x\ f.\ f\ (x\ x\ f))\ (\lambda x\ f.\ f\ (x\ x\ f))\ t)$

$\mu = (\lambda xf.\ f\ (x\ x\ f))\ (\lambda xf.\ f\ (x\ x\ f))$
$\mu\ t \longrightarrow_\beta t\ (\mu\ t) \longrightarrow_\beta t\ (t\ (\mu\ t)) \longrightarrow_\beta t\ (t\ (t\ (\mu\ t))) \longrightarrow_\beta \ldots$

$(\lambda xf.\ f\ (x\ x\ f))\ (\lambda xf.\ f\ (x\ x\ f))$ is Turing's fix point operator

Nice, but ...

NICTA

As a mathematical foundation, λ does not work. **It is inconsistent.**

- ➜ **Frege** (Predicate Logic, $\sim$ 1879): allows arbitrary quantification over predicates
- ➜ **Russel** (1901): Paradox $R \equiv \{X | X \notin X\}$
- ➜ **Whitehead & Russel** (Principia Mathematica, 1910-1913): Fix the problem
- ➜ **Church** (1930): λ calculus as logic, `true`, `false`, $\wedge$, ... as λ terms

Problem:

with	$\{x \mid P\ x\} \equiv \lambda x.\ P\ x$	$x \in M \equiv M\ x$
you can write	$R \equiv \lambda x.\ \texttt{not}\ (x\ x)$	
and get	$(R\ R) =_\beta \texttt{not}\ (R\ R)$	

Slide 5

We have learned so far...

NICTA

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

Slide 6

NICTA

Isabelle Demo

Slide 7

Content

NICTA

- ➜ Intro & motivation, getting started with Isabelle
- ➜ **Foundations & Principles**
  - **Lambda Calculus**
  - **Higher Order Logic, natural deduction**
  - Term rewriting
- ➜ Proof & Specification Techniques
  - Datatypes, recursion, induction
  - Inductively defined sets, rule induction
  - Calculational reasoning, mathematics style proofs
  - Hoare logic, proofs about programs

Slide 8

λ calculus is inconsistent

From last lecture:
Can find term R such that $R\ R\ =_\beta\ \texttt{not}(R\ R)$

There are more terms that do not make sense:
$1\ 2$, $\texttt{true false}$, etc.

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

Slide 9

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: $(\lambda x.\ x) :: \alpha \Rightarrow a$
- 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 $(s\ t) :: \beta$

Slide 10

That's about it

Slide 11

Now formally again

Slide 12

Syntax for $\lambda^\to$

NICTA

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

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

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

Context Γ:

Γ: function from variable and constant names to types.

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

Slide 13

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 Γ and τ such that $\Gamma \vdash t :: \tau$

Slide 14

Type Checking Rules

NICTA

Variables: $\quad \dfrac{}{\Gamma \vdash x :: \Gamma(x)}$

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

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

Slide 15

Example Type Derivation:

NICTA

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

Slide 16

More complex Example

NICTA

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

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