COMP4161: Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Christine Rizkallah, Miki Tanaka
S2/2018

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

→ Foundations & Principles
  • Lambda Calculus, natural deduction [1, 2]
  • Higher Order Logic [3a]
  • Term rewriting [4]

→ Proof & Specification Techniques
  • Inductively defined sets, rule induction [5]
  • Datatypes, recursion, induction [6, 7]
  • Hoare logic, proofs about programs, invariants [8b, 9]
  • (mid-semester break)
  • C verification [10]
  • CakeML, Isar [11c]
  • Concurrency [12]

a1 due; a2 due; a3 due
\[ \lambda \text{ calculus is inconsistent} \]

Can find term \( R \) such that \( R \ R \ \Rightarrow_{\beta} \ \text{not}(R \ R) \)

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” $\lambda$ term.

Examples:

1. For term $t$ has type $\alpha$ write $t :: \alpha$
2. If $x$ has type $\alpha$ then $\lambda x. x$ is a function from $\alpha$ to $\alpha$
   Write: $(\lambda x. x) :: \alpha \Rightarrow \alpha$
3. 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 \( \lambda \to \)

**Terms:**
\[
\begin{align*}
t & ::= v \mid c \mid (t \; t) \mid (\lambda x. \; t)
\end{align*}
\]

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

**Types:**
\[
\begin{align*}
\tau & ::= b \mid \nu \mid \tau \Rightarrow \tau
\end{align*}
\]

\( b \in \{\text{bool, int, } \ldots\} \) base types

\( \nu \in \{\alpha, \beta, \ldots\} \) type variables

\[
\alpha \Rightarrow \beta \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma)
\]

**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:
\[ \Gamma \vdash x :: \Gamma(x) \]

Application:
\[ \Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau \quad \Gamma \vdash t_2 :: \tau_2 \]
\[ \Gamma \vdash (t_1 \ t_2) :: \tau \]

Abstraction:
\[ \Gamma \vdash t :: \tau \]
\[ \Gamma \vdash (\lambda x. \ t) :: \tau_x \Rightarrow \tau \]
Example Type Derivation:

\[
\begin{align*}
[x \leftarrow \alpha, y \leftarrow \beta] & \vdash x :: \alpha \\
[x \leftarrow \alpha] & \vdash \lambda y. \ x :: \beta \Rightarrow \alpha \\
[] & \vdash \lambda x \ y. \ x :: \alpha \Rightarrow \beta \Rightarrow \alpha 
\end{align*}
\]
More complex Example

\[
\Gamma \vdash f :: \alpha \Rightarrow (\alpha \Rightarrow \beta) \quad \Gamma \vdash x :: \alpha \\
\Gamma \vdash f \times :: \alpha \Rightarrow \beta \\
\Gamma \vdash x \times :: \beta \\
[f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta] \vdash \lambda x. f \times x :: \alpha \Rightarrow \beta \\
[] \vdash \lambda f \times. f \times x :: (\alpha \Rightarrow \alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta
\]

\[\Gamma = [f \leftarrow \alpha \Rightarrow \alpha \Rightarrow \beta, x \leftarrow \alpha]\]
More general Types

A term can have more than one type.

Example: \[ \frac{}{\lambda x. x :: \text{bool} \Rightarrow \text{bool}} \]
\[ \frac{}{\lambda x. x :: \alpha \Rightarrow \alpha} \]

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:
\[ \Gamma \vdash t :: \tau \implies \exists \sigma. \Gamma \vdash t :: \sigma \land (\forall \sigma'. \Gamma \vdash t :: \sigma' \implies \sigma' \preceq \sigma) \]

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: $\Gamma \vdash s :: \tau \land s \rightarrow^\beta t \implies \Gamma \vdash t :: \tau$

This property is called subject reduction
What about termination?

\[ \beta \text{ reduction in } \lambda \rightarrow \text{ always terminates.} \]

(Alan Turing, 1942)

\[ \Rightarrow \ \alpha \beta \eta \text{ is decidable} \]
To decide if \( s \equiv_{\beta} t \), reduce \( s \) and \( t \) to normal form (always exists, because \( \rightarrow_{\beta} \) terminates), and compare result.

\[ \Rightarrow \ \alpha \beta \eta \text{ is decidable} \]
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 $Y \ t \rightarrow^\beta \ t \ (Y \ t)$ 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)
Types and Terms in Isabelle

Types: \[ \tau ::= b \mid \nu \mid \nu :: C \mid \tau \Rightarrow \tau \mid (\tau, \ldots, \tau) K \]
- \(b \in \{\text{bool, int, \ldots}\}\) base types
- \(\nu \in \{\alpha, \beta, \ldots\}\) type variables
- \(K \in \{\text{set, list, \ldots}\}\) type constructors
- \(C \in \{\text{order, linord, \ldots}\}\) 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: \texttt{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

```haskell
class order =
    assumes order_refl: "x \leq x"
    assumes order_trans: "[x \leq y; y \leq z] \implies x \leq z"
...
```

→ theorems can be proved in the abstract

```haskell
lemma order_less_trans:
" \land x ::\:'a :: order. [[x < y; y < z]] \implies x < z"
```

→ can be used for subtyping

```haskell
class linorder = order +
    assumes linorder_linear: "x \leq y \lor y \leq x"
```

→ can be instantiated

```haskell
instance nat :: "\{order, linorder\}" by ...
```
Schematic Variables

\[
\begin{array}{c}
X \\
\hline \\
X \land Y
\end{array}
\]

→ 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:**
\[
\begin{align*}
?X \land ?Y & =_{\alpha\beta\eta} x \land x & [?X \leftarrow x, ?Y \leftarrow x] \\
?P \ x & =_{\alpha\beta\eta} x \land x & [?P \leftarrow \lambda x. x \land x] \\
P \ (?f \ x) & =_{\alpha\beta\eta} ?Y \ x & [?f \leftarrow \lambda x. x, ?Y \leftarrow P]
\end{align*}
\]

**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 $?f \ t_1 \ldots \ t_n$
- and the $t_1 \ldots \ t_n$ are $\eta$-convertible into $n$ distinct bound variables
We have learned so far...

- Simply typed lambda calculus: $\lambda \rightarrow$
- 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