COMP4161: Advanced Topics in Software Verification

Gerwin Klein, Johannes Åman Pohjola, Christine Rizkallah, Miki Tanaka
T3/2020

data61.csiro.au
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 (as a logic)
Content

→ Foundations & Principles
  • Intro, Lambda calculus, natural deduction [1,2]
  • Higher Order Logic, Isar (part 1) [2,3]
  • Term rewriting [3,4]

→ Proof & Specification Techniques
  • Inductively defined sets, rule induction, datatype induction, primitive recursion [4,5]
  • General recursive functions, termination proofs [7b]
  • Proof automation, Hoare logic, proofs about programs, invariants [8]
  • C verification [9,10]
  • Practice, questions, examp prep [10c]

\(^{a}a1\) due; \(^{b}a2\) due; \(^{c}a3\) due
\( \text{\LaTeX} \) calculus is inconsistent

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

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

Types: $\tau ::= b \mid \nu \mid \tau \Rightarrow \tau$
$b \in \{\text{bool, int, ...}\}$ 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: $\Gamma[x \leftarrow \tau_x] \vdash t :: \tau \\
\Gamma \vdash (\lambda x. \ t) :: \tau_x \Rightarrow \tau$
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

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

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

A term can have more than one type.

Example: \[
\begin{align*}
&\vdash \lambda x. x :: \text{bool} \Rightarrow \text{bool} \\
&\vdash \lambda x. x :: \alpha \Rightarrow \alpha
\end{align*}
\]

Some types are more general than others:

\[\tau \lessapprox \sigma\] if there is a substitution \(S\) such that \(\tau = S(\sigma)\)

Examples:

\[\text{int} \Rightarrow \text{bool} \lessapprox \alpha \Rightarrow \beta \lessapprox \beta \Rightarrow \alpha \nless \alpha \Rightarrow \alpha\]
Most general Types

Fact: each type correct term has a most general type

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

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:** $\Gamma \vdash s :: \tau \land s \xrightarrow{\beta} t \implies \Gamma \vdash t :: \tau$

This property is called **subject reduction**.
What about termination?

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

(Alan Turing, 1942)

$\Rightarrow \quad =_\beta$ is decidable
To decide if $s =_\beta t$, reduce $s$ and $t$ to normal form (always exists, because $\rightarrow_\beta$ terminates), and compare result.

$\Rightarrow \quad =_{\alpha\beta\eta}$ 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 $\lambda \to !$!

How can typed functional languages then be turing complete?

Fact:
Each computable function can be encoded as closed, type correct $\lambda \to$ term using $Y :: (\tau \Rightarrow \tau) \Rightarrow \tau$ with $Y \ t \ \beta t \ (Y \ t)$ 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: \[ \tau ::= \text{b} \mid '\nu \mid '\nu :: \text{C} \mid \tau \Rightarrow \tau \mid (\tau, \ldots, \tau) \text{K} \]
- \text{b} \in \{\text{bool, int,} \ldots\} \text{ base types}\n- \nu \in \{\alpha, \beta, \ldots\} \text{ type variables}\n- \text{K} \in \{\text{set, list,} \ldots\} \text{ type constructors}\n- \text{C} \in \{\text{order, linord,} \ldots\} \text{ type classes}\n
Terms: \[ t ::= \nu \mid c \mid ?\nu \mid (t t) \mid (\lambda x. \ t) \]
- \nu, x \in V, \quad c \in C, \quad V, C \text{ sets of names}\n
→ **type constructors**: construct a new type out of a parameter type.
   Example: \text{int list}\n
→ **type classes**: restrict type variables to a class defined by axioms.
   Example: \text{\alpha :: order}\n
→ **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 ≤ x"
    assumes order_trans: "[x ≤ y; y ≤ z] ⇒ x ≤ z"
    ...
```

→ theorems can be proved in the abstract

```haskell
lemma order_less_trans:
    "∀ x ::′a :: order. [x < y; y < z] ⇒ x < z"
```

→ can be used for subtyping

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

→ can be instantiated

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

\[
\frac{X \quad Y}{X \land Y}
\]

→ \(X\) and \(Y\) must be \textbf{instantiated} to apply the rule

\textbf{But:} \quad \textbf{lemma} \quad “x + 0 = 0 + x”

→ \(x\) is free

→ convention: lemma must be true for all \(x\)

→ \textbf{during the proof}, \(x\) must \textbf{not} be instantiated

\textbf{Solution:}

Isabelle has \textbf{free} (\(x\)), \textbf{bound} (\(x\)), and \textbf{schematic} (?\(X\)) variables.

\textbf{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