COMP 4161
NICTA Advanced Course

Advanced Topics in Software Verification

Gerwin Klein, June Andronick, Toby Murray
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

→ 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\text{ 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.

\textbf{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 _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:**

\[
t ::= v \mid c \mid (t \ t) \mid (\lambda x. \ t)
\]

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

**Types:**

\[
\tau ::= b \mid \nu \mid \tau \Rightarrow \tau
\]

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

\( \nu \in \{\alpha, \beta, ...\} \) 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: \[ \Gamma \vdash t_1 :: \tau_2 \Rightarrow \tau_1 \quad \Gamma \vdash t_2 :: \tau_2 \]
\[ \Gamma \vdash (t_1 t_2) :: \tau_1 \]

Abstraction: \[ \Gamma[x \leftarrow \tau_1] \vdash t :: \tau_2 \]
\[ \Gamma \vdash (\lambda x. t) :: \tau_1 \Rightarrow \tau_2 \]
Example Type Derivation:

\[ [x \leftarrow \alpha, y \leftarrow \beta] \vdash x :: \alpha \]
\[ [x \leftarrow \alpha] \vdash \lambda y. x :: \beta \Rightarrow \alpha \]
\[ [] \vdash \lambda x \ y. \ x :: \alpha \Rightarrow \beta \Rightarrow \alpha \]
More complex Example

Γ ⊢ f :: α ⇒ (α ⇒ β)  Γ ⊢ x :: α
  Γ ⊢ f x :: α ⇒ β      Γ ⊢ x :: α
  Γ ⊢ f x x :: β

[f ← α ⇒ α ⇒ β] ⊢ λx. f x x :: α ⇒ β

[] ⊢ λf x. f x x :: (α ⇒ α ⇒ β) ⇒ α ⇒ β

Γ = [f ← α ⇒ α ⇒ β, x ← α]
More general Types

A term can have more than one type.

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

Some types are more general than others:

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

Examples:

\( \text{int} \Rightarrow \text{bool} \preceq \alpha \Rightarrow \beta \preceq \beta \Rightarrow \alpha \not\preceq \alpha \Rightarrow \alpha \)
Most general Types

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

**Formally:**

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

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?

\( \beta \) reduction in \( \lambda \rightarrow \) always terminates.

(Alan Turing, 1942)

\( \rightarrow =_{\beta} \) is decidable

To decide if \( s =_{\beta} t \), reduce \( s \) and \( t \) to normal form (always exists, because \( \lambda \rightarrow_{\beta} \) terminates), and compare result.

\( \rightarrow =_{\alpha \beta \eta} \) 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 $Y \ t \ \rightarrow_{\beta} \ t \ (Y \ t)$ as only constant.

→ $Y$ is called fix point operator
→ used for recursion
→ lose decidability (what does $Y \ (\lambda x.x)$ reduce to?)
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: \( t ::= v \mid c \mid ?v \mid (t t) \mid (\lambda x. t) \)

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

- **type constructors**: construct a new type out of a parameter type.
  Example: \( \text{int list} \)

- **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

  **axclass** order < ord
  - order_refl: ""x ≤ x"
  - order_trans: ""[x ≤ y; y ≤ z] \implies x ≤ z"
  
- theorems can be proved in the abstract

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

- can be used for subtyping

  **axclass** linorder < order
  - linorder_linear: ""x ≤ y \lor y ≤ x"

- can be instantiated

  **instance** nat :: ""{order, linorder}"" by \ldots
Schematic Variables

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

→ \( X \) 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 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: \( \lambda \rightarrow \)
- 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
Exercises

→ Construct a type derivation tree for the term \( \lambda x \ y \ z. \ z \ x \ (y \ x) \)

→ Find a unifier (substitution) such that \( \lambda x \ y \ z. \ ?F \ y \ z = \lambda x \ y \ z. \ z \ (?G \ x \ y) \)