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 \(f\), \(s\), and \(\text{pair} \) such that \(f \text{ pair } a b = a\) and \(s \text{ pair } a b = b\).
- (harder) Find an encoding of list objects, i.e. for the functions \(\text{cons}\) and \(\text{nil}\). Then find an encoding for \(\text{map}\) (that is, \(\text{map } f [x_1, \ldots, x_n] = [f x_1, \ldots, f x_n]\)), and for \(\text{foldl}\) (that is, \(\text{foldl } f [x_1, \ldots, x_n] = f x_1 (f x_2 (f x_3 (\ldots (f x_n)) \ldots))\)).

\[\lambda\]

\[\rightarrow\]

\[\lambda\text{ calculus is inconsistent}\]

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.

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: \( (\lambda x. x) :: \alpha \to \alpha \)
- 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 \( (s t) :: \beta \)

**NOW FORMALLY AGAIN**

Syntax for \( \lambda \rightarrow \)

\[
\text{Terms: } t ::= v \mid c \mid (t t) \mid (\lambda x. t) \\
v, x \in V, \ c \in C, \ V, C \text{ sets of names}
\]

\[
\text{Types: } \tau ::= b \mid \nu \mid \tau \Rightarrow \tau \\
b \in \{\text{bool, int,...}\} \text{ base types} \\
\nu \in \{\alpha, \beta, \ldots\} \text{ type variables} \\
\alpha \Rightarrow \beta \Rightarrow \gamma = \alpha \Rightarrow (\beta \Rightarrow \gamma)
\]

Context \( \Gamma \):

\( \Gamma \): function from variable and constant names to types.

**Term \( t \) has type \( \tau \) in context \( \Gamma \): \( \Gamma \vdash t :: \tau \)**
Examples

Γ ⊢ (λx. x) :: α ⇒ α
[y ← int] ⊢ y :: int
[z ← bool] ⊢ (λy. z) :: bool
[] ⊢ λf x. f x :: (α ⇒ β) ⇒ α ⇒ β

A term t is well typed or type correct
if there are Γ and τ such that Γ ⊢ t :: τ

Type Checking Rules

Variables:

Γ ⊢ x :: Γ(x)

Application:

Γ ⊢ t₁ :: t₁, Γ ⊢ t₂ :: t₂
Γ ⊢ (t₁ t₂) :: t₁

Abstraction:

Γ[x ← t₁] ⊢ t :: t₂
Γ ⊢ (λx. t) :: t₁ ⇒ t₂

Example Type Derivation:

Example Type Derivation:

More complex Example

Γ = [f ← α ⇒ β, x ← α]

Γ ⊢ f :: α ⇒ (α ⇒ β) ⊢ x :: α
Γ ⊢ f x :: α ⇒ β ⊢ λx. f x x :: (α ⇒ α ⇒ β) ⇒ α ⇒ β
[Γ ⊢ λf x. f x x :: α ⇒ β]
More general Types

A term can have more than one type.

Example: 

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

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

Some types are more general than others:

\[ \tau \leq \sigma \text{ if there is a substitution } S \text{ such that } \tau = S(\sigma) \]

Examples:

\[ \text{int} \Rightarrow \text{bool} \leq \alpha \Rightarrow \beta \leq \beta \Rightarrow \alpha \leq \alpha \]

Most general Types

Fact: each type correct term has a most general type

Formally:

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

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 \( \lambda \) are decidable.

What about \( \beta \) reduction?

Definition of \( \beta \) reduction stays the same.

Fact: Well typed terms stay well typed during \( \beta \) reduction

Formally:

\[ \Gamma \vdash s :: \tau \land s \rightarrow_\beta t \Rightarrow \Gamma \vdash t :: \tau \]

This property is called subject reduction

What about termination?

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

(Alan Turing, 1942)

⇒ \( \rightarrow_\beta \) is decidable
To decide if \( s \rightarrow_\beta t \), reduce \( s \) and \( t \) to normal form (always exists, because \( \rightarrow_\beta \) terminates), and compare result.
⇒ \( \rightarrow_\beta \alpha \beta \eta \) 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 $Y : (\tau \to \tau) \to \tau$ with $Y \to \gamma \to \gamma (Y \gamma)$ as only constant.

$\to$ $Y$ is called fix point operator
$\to$ used for recursion
$\to$ lose decidability (what does $Y (\lambda x. x)$ reduce to?)

Types and Terms in Isabelle

Types: $\tau ::= b | \nu | \nu : C | \tau \to \tau | (\tau _1, \ldots, \tau _n) K$

$K \in \{\text{set, list, ...}\}$ type constructors

$C \in \{\text{order, linord, ...}\}$ type classes

Terms: $t ::= v | e | ?\nu \in \{t\} \mid (\lambda x. t) \mid (\nu t)$

$e, x \in \nu, \quad e \in C, \quad \nu, C \text{ sets of names}$

$\rightarrow$ type constructors: construct a new type out of a parameter type.
Example: $\text{list}$, $\text{set}$

$\rightarrow$ type classes: restrict type variables to a class defined by axioms.
Example: $\alpha : \text{order}$

$\rightarrow$ schematic variables: variables that can be instantiated.

Type Classes

$\to$ similar to Haskell’s type classes, but with semantic properties

axclass order $<\text{ord}$

order_refl: “$x \leq x$”

order_trans: “[x \leq y; y \leq z] \Rightarrow x \leq z”

…

$\to$ theorems can be proved in the abstract

lemma order_less_trans: “$\lambda x. \nu \text{ ord}: [x < y; y < z] \Rightarrow x < z$”

$\rightarrow$ can be used for subtyping

axclass linorder $<\text{ord}$

linorder_linear: “$x \leq y \lor y \leq x”$ can be instantiated

instance nat $\in \{\text{order, linorder}\}$ by …

Schematic Variables

$\frac{X}{Y}$

$\rightarrow$ $X$ and $Y$ must be instantiated to apply the rule

But: lemma “$x + 0 = 0 + x$”

$\rightarrow x$ is free

$\rightarrow$ convention: lemma must be true for all $x$

$\rightarrow$ 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:
\[
\begin{align*}
?X \land ?Y &= \alpha\beta\eta \ x \land x \\
?P \ x &= \alpha\beta\eta \ x \land x \\
P \ (?f \ x) &= \alpha\beta\eta \ ?Y \ x \\
\end{align*}
\]

Higher Order: schematic variables can be functions.

Slide 21

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

Slide 23

Exercises

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

Slide 24