Lecture 6: The Untyped Lambda Calculus
Semantics and Implementation

Polyvios Pratikakis

Computer Science Department, University of Crete

Type Systems and Static Analysis
Last class

- **Lambda calculus, cf.1930s**
  - Simple, core language: everything is a function
  - Can express all computation
  - Can encode complex language features as syntactic sugar
  - Simple semantics, one instruction: function application
Defined in one slide

- **Syntax:**

  \[ e ::= x \quad \text{Variables} \]
  \[ \mid \lambda x. e \quad \text{Function definition} \]
  \[ \mid e \ e \quad \text{Function application} \]

- **Nondeterministic small-step semantics:**

  \[
  (\lambda x. e_1) \ e_2 \rightarrow e_1[x \mapsto e_2]
  \]

  \[
  e \rightarrow e'
  \]

  \[
  (\lambda x. e) \rightarrow (\lambda x. e')
  \]

  \[
  e_1 \rightarrow e'_1
  \]

  \[
  e_1 \ e_2 \rightarrow e'_1 \ e_2
  \]

  \[
  e_2 \rightarrow e'_2
  \]

  \[
  e_1 \ e_2 \rightarrow e_1 \ e'_2
  \]
Fun with encodings

- Church integers: \( \lambda s.\lambda z.\langle\text{apply } s \text{ on } z \text{ for } n \text{ times}\rangle \)
- Booleans: true = \( \lambda t.\lambda f.t \) and false = \( \lambda t.\lambda f.f \)
- Pairs: \( (a, b) = \lambda p.p \ a \ b \)
- In general, encode data as a function that takes an action, and applies it on the data
- How about lists?
  - \( [] = \lambda f.\lambda n.n \)
  - \( a :: b = \lambda a.\lambda b.\lambda f.\lambda n.f \ a \ (b \ f \ n) \)
- Examples:
  - Predecessor function
  - Addition and subtraction
  - Check a list for empty
  - Head and tail function for lists
Example: Predecessor function for ints

- We want pred 0 to evaluate to 0, pred 1 to 0, pred 2 to 1, etc.
- Remove one application of s from the chain \( s(s(s\ldots(s\ z)) \)
- Unfortunately not very easy for Church integers
- Solution: rebuild the given number up to the previous number
  - Similar to encoding of integers: base, inductive case
  - Use pairs of predecessor, number: \((\text{pred } n, n)\)
  - Base case, or “zero”—start with pred 0, which is 0:
    - \( \ast \quad \text{zz} = (0, 0) \)
  - Inductive case, or “successor”—construct the next pair \((n, \text{succ } n)\) from the previous \((\text{pred } n, n)\)
    - \( \ast \quad \text{ss} = \lambda p. (\text{snd } p, (\text{succ } (\text{snd } p)) \)
  - pred \( m \) is the first item of the \( m \)-th pair
    - \( \ast \quad \text{pred} = \lambda m. (\text{fst } (m \ \text{ss} \ \text{zz})) \)
Example: plus and minus

- Plus: given two numbers \( m \) and \( n \), construct a number \( m + n \)
  - Replace zero in \( m \) with \( n \): \( \text{plus} = \lambda m.\lambda n.\lambda s.\lambda z. n \ (m \ s \ z) \)

- Minus is a bit more complex
- \( m - n \): apply \( \text{pred} \) on \( m \), \( n \) times
  - But, \( n \) takes a function \( s \) and a \( z \) and applies \( s \) on \( z \) for \( n \) times
  - Just call it with \( s = \text{pred} \), and \( z = m \): \( \text{minus} = \lambda m.\lambda n. n \ \text{pred} \ m \)
  - Will apply \( \text{pred} \) on \( m \) for \( n \) times: \( m - n \)
Terminology reminder

- **Combinator**, or **closed term**: a term with no free variables
- **Normal form**: a term that cannot be reduced further
  - Normal form of a term is unique
  - Does not always exist, a term may run forever
  - Is not always reached, depending on evaluation order
- A **redex** is a subterm that can be reduced: \((\lambda x.e) \ e'\)
- Equivalent terms **up to \(\alpha\)-conversion**: they can be made equal by renaming bound variables
- Substitution \(e[e'/x]\) or \(e[x \mapsto e']\): replace all occurrences of \(x\) in \(e\) by \(e'\).
  - **Capture-avoiding**: \(e'\) does not have free variables that become bound because of substitution
  - Always possible, using \(\alpha\)-conversion to rename variables
Evaluation strategies

- Full $\beta$-reduction: nondeterministic semantics
- Normal order: always reduce leftmost, outermost redex
- Call-by-name (lazy): no reductions under $\lambda$, only at the top-level
  - Call-by-need (used in haskell): remember term substitutions and replace all copies of an evaluated term in the AST with the value
  - Instead of AST: abstract syntax graph
- Call-by-value (eager): reduce only outermost redexes where the argument is a value
Lazy semantics

- Small-step:

\[
\begin{align*}
(\lambda x. e_1) e_2 & \rightarrow e_1[x \mapsto e_2] \\
 e_1 \rightarrow e_1'
\end{align*}
\]

\[
\begin{align*}
 e_1 e_2 & \rightarrow e_1' e_2
\end{align*}
\]

- Big-step:

\[
\begin{align*}
(\lambda x. e) & \downarrow (\lambda x. e) \\
 e_1 \downarrow (\lambda x. e) e[x \mapsto e_2] & \downarrow e'
\end{align*}
\]

\[
\begin{align*}
 e_1 e_2 & \downarrow e'
\end{align*}
\]
Eager semantics

- Define values as:
  \[ v ::= \lambda x.e \]

- Small-step:
  \[
  \frac{e_1 \rightarrow e'_1}{e_1 \ e_2 \rightarrow e'_1 \ e_2}
  \]

- Big-step:
  \[
  \frac{e_1 \downarrow (\lambda x.e) \quad e_2 \downarrow v_2 \quad e[x \mapsto v_2] \downarrow v}{(\lambda x.e) \downarrow (\lambda x.e)}
  \]

  \[
  \frac{e_1 \ e_2 \downarrow v}{\frac{(\lambda x.e) \ v \rightarrow e[x \mapsto v]}{v_2}}
  \]
In code

- All so far is syntax driven: look at the syntax, decide which rule to apply
- The same for all helper function definitions: \( FV(e) \), \( subst(e, x, e') \), etc.
- OCaml datatypes and pattern matching helps with that
- The abstract syntax tree:

```ocaml
type exp =
  | Var of string
  | Fun of string * exp
  | App of exp * exp
```

\[
e ::= \begin{align*}
  x & \text{ Variables} \\
  \lambda x.e & \text{ Function definition} \\
  e e & \text{ Function application}
\end{align*}
\]