Public service announcement:

The ACM SIGPLAN
International Conference on
Functional Programming
(ICFP)
is in Boston September 22 to 28.

There are a variety of collocated events, aimed at researchers, practitioners, and/or both.

http://icfpconference.org/icfp2013/
Algebraic types

• Algebraic types are *tagged unions of products*
• Example

```
data Shape = Line      Pnt  Pnt      
  | Triangle  Pnt  Pnt  Pnt
  | Quad      Pnt  Pnt  Pnt  Pnt
```

- new "constructors" (a.k.a. "tags", "disjuncts", "summands")
- a k-ary constructor is applied to k type expressions
Examples of Algebraic types

In Haskell:

```
data Bool = False | True

data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

data Maybe a = Nothing | Just a

data List a = Nil | Cons a (List a)

data Tree a = Leaf a | Node (Tree a) (Tree a)

data Tree’ a b = Leaf’ a
    | Nonleaf’ b (Tree’ a b) (Tree’ a b)

data Course = Course String Int String (List Course)
```

In Haskell: "[]"

name   number  description   pre-reqs
Constructors are functions

• Constructors can be used as functions to create values of the type.

```
let
  l1 :: Shape
  l1 = Line e1 e2

  t1 :: Shape = Triangle e3 e4 e5
  q1 :: Shape = Quad e6 e7 e8 e9

in
  ...
```

where each "eJ" is an expression of type "Pnt"
Pattern-matching on algebraic types

- **Pattern-matching** is used to examine values of an algebraic type.

```haskell
anchorPnt :: Shape -> Pnt
anchorPnt s = case s of
  Line     p1 p2       -> p1
  Triangle p3 p4 p5    -> p3
  Quad     p6 p7 p8 p9 -> p6
```

- A pattern-match has two roles:
  - A test: "does the given value match this pattern?"
  - Binding ("if the given value matches the pattern, bind the variables in the pattern to the corresponding parts of the value")

- Clauses are examined top-to-bottom and left-to-right for pattern matching.
A $\lambda$-calculus with Constants and Let-blocks
(plus introduction to operational semantics)
[Redacted Version]

Adam Chlipala
MIT

6.820
Based on Arvind's 2010 slides

September 11, 2013
Outline

• Big-step versus small-step semantics

• Recursion and the Y combinator

• The $\lambda_{let}$ Calculus
Big-Step Operational Semantics

• Model the execution in an abstract machine
• Basic Notation: Judgments
  \[ \langle \text{configuration} \rangle \rightarrow \text{result} \]
  - describe how a program configuration is evaluated into a result.
  - the configuration is usually a program fragment together with any state.

• Basic Notation: Inference rules
  - define how to derive judgments for an arbitrary program.
  - also called derivation rules or evaluation rules
  - usually defined recursively

Some alternative notations: ↓, →, ⇔, ⇒, etc...
What evaluation order is this?

\[
x \to x
\]

\[
\lambda x. e \to \lambda x. e
\]

\[
e_1 \to \lambda x. e_1' \quad e_1'[\alpha(e_2)/x] \to e_3
\]

\[
e_1 e_2 \to e_3
\]
What evaluation order is this?

\[
x \rightarrow x
\]

\[
\lambda x. e \rightarrow \lambda x. e
\]

\[
e_1 \rightarrow \lambda x. e'_1 \quad e_2 \rightarrow e'_2 \quad e'_1[\alpha(e'_2)/x] \rightarrow e_3
\]

\[
e_1e_2 \rightarrow e_3
\]
Outline

• Big-step versus small-step semantics

• Recursion and the Y combinator

• The $\lambda_{\text{let}}$ Calculus
Recursive functions can be thought of as solutions of fixed point equations:

\[ \text{fact} = \lambda n. \text{Cond} \ (\text{Zero?} \ n) \ 1 \ (\text{Mul} \ n \ (\text{fact} \ (\text{Sub} \ n \ 1))) \]

Suppose

\[ H = \lambda f. \lambda n. \text{Cond} \ (\text{Zero?} \ n) \ 1 \ (\text{Mul} \ n \ (f \ (\text{Sub} \ n \ 1))) \]

then

\[ \text{fact} = H \ \text{fact} \]

fact is a fixed point of function $H$!
Fixed Point Equations

\[ f : D \to D \]
A fixed point equation has the form
\[ f(x) = x \]
Its solutions are called the \textit{fixed points} of \( f \) because if \( x_p \) is a solution then
\[ x_p = f(x_p) = f(f(x_p)) = f(f(f(x_p))) = ... \]

We want to consider fixed-point equations whose solutions are functions, i.e., sets that contain their function spaces.  
\textit{domain theory}  
(which we look at only very briefly here)
An example

Consider
\[
    f \ n = \begin{cases} 
        1 & \text{if } n=0 \\
        \text{else if } n=1 & \text{then } f \ 3 \\
        \text{else } f \ (n-2) 
    \end{cases}
\]

\[
    H = \lambda f. \lambda n. \text{Cond}(n=0, 1, \text{Cond}(n=1, f \ 3, f \ (n-2))
\]

Is there an \( f_p \) such that \( f_p = H \ f_p \)?

Under the assumption of \textit{monotonicity} and \textit{continuity} least fixed points are unique and computable.
Computing a Fixed Point

• Recursion requires repeated application of a function.

• Self application allows us to recreate the original term.

  • Consider: \( \Omega = (\lambda x. x x) (\lambda x. x x) \)

  • Notice \( \beta \)-reduction of \( \Omega \) leaves \( \Omega : \Omega \rightarrow \Omega \)

• Now to get \( F (F (F (F ...))) \) we insert \( F \) in \( \Omega \):
  \[
  \Omega_F = (\lambda x. F(x x)) (\lambda x. F(x x))
  \]
  which \( \beta \)-reduces to:
  \[
  \Omega_F \rightarrow F((\lambda x. F(x x))(\lambda x. F(x x)))
  \rightarrow F \Omega_F \rightarrow F(F \Omega_F) \rightarrow F(F(F \Omega_F)) \rightarrow ...
  \]

• Now \( \lambda \)-abstract \( F \) to get a Fix-Point Combinator:
  \[
  Y \equiv \lambda f.(\lambda x. (f (x x))) (\lambda x.(f (x x)))
  \]
Y: A Fixed Point Operator

\[ Y \equiv \lambda f. (\lambda x. (f \, (x \, x))) \, (\lambda x. (f \, (x \, x))) \]

Notice

\[
\begin{align*}
Y \, F & \quad \rightarrow \quad (\lambda x. F \, (x \, x)) \, (\lambda x. F \, (x \, x)) \\
& \quad \rightarrow \quad F \, ((\lambda x. F \, (x \, x)) \, (\lambda x. F \, (x \, x))) \\
& \quad \rightarrow \quad F \, (Y \, F)
\end{align*}
\]

\[ F \, (Y \, F) = Y \, F \quad \text{(Y F) is a fixed point of F.} \]

Y computes the least fixed point of any function!

There are many different fixed point operators.
Mutual Recursion

odd \ n = \text{if } n==0 \ \text{then} \ False \ \text{else} \ even \ (n-1)
even \ n = \text{if } n==0 \ \text{then} \ True \ \text{else} \ odd \ (n-1)
Self-Application and Paradoxes

Self application, i.e., \((x \ x)\) is dangerous.

Suppose:

\[
u \equiv \lambda y. \ if \ (y \ y) = a \ then \ b \ else \ a
\]

What is \((u \ u)\)?
Recursive programs can be translated into the \( \lambda \)-calculus with constants and combinator Y. However,

- Y is hard to assign a static type.
- Translation is messy in case of mutually recursive functions.

\( \Rightarrow \)

Extend the \( \lambda \)-calculus with recursive let blocks.

The \( \lambda_{\text{let}} \) Calculus
Outline

- Big-step versus small-step semantics
- Recursion and the Y combinator
- The $\lambda_{let}$ Calculus
\( \lambda \)-calculus with Constants & Letrec

\[
E ::= \ x \mid \lambda x. E \mid E \ E \\
    \mid \text{Cond} \ (E, E, E) \\
    \mid \text{OP}_k(E_1,\ldots,E_k) \\
    \mid \text{CN}_0 \\
    \mid \text{CN}_k(E_1,\ldots,E_k) \\
    \mid \text{let } S \text{ in } E
\]

\( \text{OP}_1 ::= \text{negate} \mid \text{not} \mid \ldots \)
\( \text{OP}_2 ::= + \mid \ldots \)
\( \text{CN}_0 ::= \text{Number} \mid \text{Boolean} \)
\( \text{CN}_2 ::= \text{pair} \mid \ldots \)

**Statements**

\[
S ::= \varepsilon \mid x = E \mid S; S
\]

Variables on the LHS in a let expression must be pairwise distinct.
Let-block Statements

“;” is associative and commutative

\[ S_1 ; S_2 \equiv S_2 ; S_1 \]
\[ S_1 ; (S_2 ; S_3) \equiv (S_1 ; S_2 ) ; S_3 \]

\[ \varepsilon ; S \equiv S \]
\[ let \ \varepsilon \ in \ E \equiv E \]
Values and Simple Expressions

Values
\[ V ::= \lambda x. E \mid CN_0 \mid CN_k(SE_1,\ldots,SE_k) \]

Simple expressions
\[ SE ::= x \mid V \]
How to define the operational semantics of $\lambda_{let}$: Environments

An environment-based interpreter.

- An environment where all the (variable name, value) bindings are kept and passed around for expression evaluation.
- When a let expression is encountered, the environment is extended with all the let-bindings. Very complicated if the environment contains unevaluated expressions.
- Not abstract enough – too many concrete data structures and associated functions for proper execution.
How to define the operational semantics of \( \lambda_{let} \): graphs

A let simply represents a wiring diagram, i.e., a graph.

\[
\text{let} \\
\quad f = \lambda x.e_1 \\
\quad y = e_2 e_3 \\
\text{in} \\
\quad (f y) + y
\]

- Quite complicated to explain \( \beta \)-substitution in a graph-based interpreter but good for showing sharing of terms.
How to define the operational semantics of $\lambda_{let}$: via a calculus

- Rewrite rules
  - slightly more complicated than the $\lambda$-calculus
- Reduction Strategy
- Normal forms? Equivalences?
  - do the following terms have the same meaning?

\[
\begin{array}{ccc}
\text{let} & \ x = 5 & \text{in} \\
\text{in} & \ x \\
\text{let} & \ x = 5 & \text{in} \\
\text{in} & \ 5 \\
\text{let} & \ x = 5 & \text{in} \\
\text{in} & \ y = 6 \\
\end{array}
\]
Operational semantics of $\lambda_{let}$: Issue #1

Creating redexes

$$((\text{let } S \text{ in } \lambda x.e_1) \ e_2)$$

How do we juxtapose

$$\ (\lambda x.e_1) \ e_2 \ ?$$

Solution: Lifting rules
Lifting Rules

(let S’ in e’) is the $\alpha$-renamed (let S in e) to avoid name conflicts in the following rules:

\[
\begin{align*}
x &= \text{let } S \text{ in } e & \rightarrow & & x = e’; S’ \\
\text{let } S_1 \text{ in } (\text{let } S \text{ in } e) & \rightarrow & & \text{let } S_1; S’ \text{ in } e’ \\
(\text{let } S \text{ in } e) \ e_1 & \rightarrow & & \text{let } S’ \text{ in } e’ \ e_1 \\
\text{Cond}(\text{(let } S \text{ in } e), e_1, e_2) & \rightarrow & & \text{let } S’ \text{ in } \text{Cond}(e’, e_1, e_2) \\
\text{OP}_k(e_1,\ldots(\text{let } S \text{ in } e),\ldots e_k) & \rightarrow & & \text{let } S’ \text{ in } \text{OP}_k(e_1,\ldots e’,\ldots e_k)
\end{align*}
\]
Operational semantics of $\lambda_{let}$: Issue #2

How to refer to a variable binding

\[
\text{let} \quad f = \lambda x. e_1 \\
y = e_2 e_3 \\
\text{in} \quad (f \, y) + y
\]

**Solution:** Instantiation rules

*How and when f and y refer to their definitions*

\[
((\lambda x. e_1) \, y) + y
\]

*but first need to understand something about “contexts”*
Contexts for Expressions

A context is an expression (or statement) with a “hole” such that if an expression is plugged in the hole the context becomes a legitimate expression:

\[
C[] ::= [] \\
| \lambda x. C[] \\
| C[] E | E C[] \\
| let S in C[] \\
| let SC[] in E
\]

Statement Context for an expression

\[
SC[] ::= x = C[] \\
| SC[] ; S | S; SC[]
\]
Interlude on Evaluation Contexts

Contexts for basic lambda calculus:

\[ C[] ::= [] \]
\[ \mid \lambda x. C[] \]
\[ \bullet \mid C[] E \mid E C[] \]

Beta rule: \((\lambda x. E) \ E' \rightarrow E [E'/x]\)
A free variable in an expression can be instantiated by a *simple expression*.

**Instantiation rule 1**

\[(let \ x = a \ ; \ S \ in \ C[\lambda x]) \rightarrow (let \ x = a \ ; \ S \ in \ C'[\lambda a])\]

- simple expression
- free occurrence of \(x\) in some context \(C\)
- renamed \(C[\lambda ]\) to avoid free-variable capture

**Instantiation rule 2**

\[(x = a \ ; \ SC[\lambda x]) \rightarrow (x = a \ ; \ SC'[\lambda a])\]

**Instantiation rule 3**

\[x = a \quad \text{where} \quad a = C[\lambda x] \rightarrow x = C'[\lambda a]\]
The $\beta$-rule

The normal $\beta$-rule

$$(\lambda x. e) e_a \rightarrow e[e_a/x]$$

is replaced by the following $\beta$-rule

$$(\lambda x. e) e_a \rightarrow let\ t = e_a\ in\ e[t/x]$$

where $t$ is a new variable

*Instantiation rules* are used to refer to the value of variable $t$. 
Primitive Functions and Datastructures

$\delta$-rules

$+ ( \, n, \, m ) \quad \rightarrow \quad n + m$

...

Cond-rules

Cond(True, $e_1$, $e_2$) \quad \rightarrow \quad e_1

Cond(False, $e_1$, $e_2$) \quad \rightarrow \quad e_2

Data-structures

CN_k(e_1, ..., e_k) \quad \rightarrow \quad
Strategy for computing WHNFs

• Conceptually just like normal-order reduction
• When we encounter a let-expression, we evaluate the term to be returned and instantiate variables in the term as necessary.
  – Exact specification can be given using a environment-based interpreter, graphs or big-step semantics (each has its advantages and disadvantages).
That's it for now!

Next time: Some machine-checked theorem-proving about operational semantics, using Coq

Recitation/office hours on Friday
A good chance to get help on PS1 (due on Tuesday)
Free Variables of an Expression

\[ FV(x) = \{x\} \]
\[ FV(E_1 E_2) = FV(E_1) \cup FV(E_2) \]
\[ FV(\lambda x. E) = FV(E) - \{x\} \]
\[ FV(let\ S\ in\ E) = FVS(S) \cup FV(E) - BVS(S) \]

\[ FVS(\varepsilon) = \{\} \]
\[ FVS(x = E; S) = FV(E) \cup FVS(S) \]

\[ BVS(\varepsilon) = \{\} \]
\[ BVS(x = E; S) = \{x\} \cup BVS(S) \]
\(\alpha\) -Renaming \((to\ avoid\ free\ variable\ capture)\)

Assuming \(t\) is a new variable, rename \(x\) to \(t\):

\[
\lambda x. e \equiv \lambda t. (e[t/x]) \\
\text{let } x = e \text{ ; } S \text{ in } e_0 \\
\equiv \text{ let } t = e[t/x] \text{ ; } S[t/x] \text{ in } e_0[t/x]
\]

where \([t/x]\) is defined as follows:

\[
\begin{align*}
x[t/x] & = t \\
y[t/x] & = y \quad \text{if } x \neq y \\
(E_1 \ E_2)[t/x] & = (E_1[t/x] \ E_2[t/x]) \\
(\lambda x. E)[t/x] & = \lambda x. E \\
(\lambda y. E)[t/x] & = \lambda y. E[t/x] \quad \text{if } x \neq y \\
(let \ S \ in \ E)[t/x] & \text{ is defined as follows:} \\
& = \begin{cases} 
(let \ S \ in \ E) & \text{if } x \notin \text{FV}(let \ S \ in \ E) \\
(let \ S[t/x] \ in \ E[t/x]) & \text{if } x \in \text{FV}(let \ S \ in \ E)
\end{cases} \\
(S_1; S_2)[t/x] & = (S_1[t/x]; S_2[t/x]) \\
(y = E)[t/x] & = (y = E[t/x]) \\
\varepsilon[t/x] & = \varepsilon
\end{align*}
\]