Lecture 27
Fixpoints and Recursion

3 May 2012
Recursion in $\lambda$-calculus

Last time: encoded booleans and numbers in $\lambda$-calculus. Can we use these to express the factorial function?

```ocaml
let rec fact n =
  if n=0 then 1 else n * fact (n-1)
```

Yes... but we need a way to define recursive functions! What about "Landin's knot"?

```ocaml
let fact =
  let g : (int -> int) = ref (fun n -> 42) in
  let f n =
    if n=0 then 1 else n * !g (n-1) in
  g := f; fun n -> !g n
```

Won't work—$\lambda$-calculus doesn't have references!
Recursion in $\lambda$-calculus

Last time: encoded booleans and numbers in $\lambda$-calculus.
Can we use these to express the factorial function?

```ml
let rec fact n =
  if n=0 then 1 else n * fact (n-1)
```

Yes... but we need a way to define recursive functions!
Recursion in $\lambda$-calculus

Last time: encoded booleans and numbers in $\lambda$-calculus. Can we use these to express the factorial function?

\[
\text{let rec fact n = if n=0 then 1 else n * fact (n-1)}\]

Yes... but we need a way to define recursive functions!

What about “Landin’s knot”?

\[
\text{let fact = let g : (int -> int) = ref (fun n -> 42) in let f n = if n=0 then 1 else n * !g (n-1) in g := f; fun n -> !g n}\]

Won’t work—$\lambda$-calculus doesn’t have references!
let t_fact g n = if n=0 then 1 else n * g (n-1)

let fact0 = (fun n -> 42) (* {} ok *)
let fact1 = t_fact fact0 (* {0} ok *)
let fact2 = t_fact fact1 (* {0,1} ok *)
let fact3 = t_fact fact2 (* {0,1,2} ok *)
. . .
let fact = t_fact fact (* {0,1,2,...} ok *)
Fixpoints

let t_fact g n = if n=0 then 1 else n * g (n-1)

let fact0 = (fun n -> 42) (* {} ok *)
let fact1 = t_fact fact0 (* {0} ok *)
let fact2 = t_fact fact1 (* {0,1} ok *)
let fact3 = t_fact fact2 (* {0,1,2} ok *)

let fact = t_fact fact (* {0,1,2,...} ok *)

Definition (Fixpoint)

A fixpoint $x$ of a function $f$ satisfies $f(x) = x$.

So we want to find a fixpoint of $t\_fact$. 
Recall the \( \lambda \)-calculus term

\[
\textit{omega} \overset{\Delta}{=} (\lambda x. x x)(\lambda x. x x)
\]

which \( \beta \)-converts to itself in one step.
Fixpoints in $\lambda$-calculus

Recall the $\lambda$-calculus term

$$\omega \triangleq (\lambda x. x x)(\lambda x. x x)$$

which $\beta$-converts to itself in one step.

If we interpose a $\lambda$-calculus term $F$ we get

$$(\lambda x. F (x x))(\lambda x. F (x x))$$

$$\Rightarrow F ((\lambda x. F (x x))(\lambda x. F (x x)))$$

That is, a fixed point of $W$!
Fixpoints in $\lambda$-calculus

Recall the $\lambda$-calculus term

$$omega \triangleq (\lambda x. x x)(\lambda x. x x)$$

which $\beta$-converts to itself in one step.

If we interpose a $\lambda$-calculus term $F$ we get

$$(\lambda x. F (x x))(\lambda x. F (x x))$$

$$\Rightarrow F ((\lambda x. F (x x))(\lambda x. F (x x)))$$

That is, a fixed point of $W$!

The famous $Y$ combinator is just

$$Y \triangleq \lambda f. (\lambda x. f (x x)) \ (\lambda x. f (x x))$$
Factorial in $\lambda$-calculus

$$Y \triangleq \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

$$t_{\text{fact}} \triangleq \lambda g. \lambda n. \text{cond} (\text{iszero } n) \bar{1} (\text{mul } n (g (\text{predn})))$$

$$\text{fact} \triangleq Y \ t_{\text{fact}}$$
Factorial in $\lambda$-calculus

\[ Y \triangleq \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \]

\[ t_{\text{fact}} \triangleq \lambda g. \lambda n. \text{cond} (\text{iszero } n) \top (\text{mul } n (g (\text{predn}))) \]

\[ \text{fact} \triangleq Y t_{\text{fact}} \]

Theorem (Correctness of fact)

\[ \forall n. \text{fact } n = n! \]
Factorial in $\lambda$-calculus

$$Y \triangleq \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

$$t_{fact} \triangleq \lambda g. \lambda n. \text{cond} (\text{iszero} n) \overline{1} (\text{mul} n (g \ (\text{predn})))$$

$$\text{fact} \triangleq Y \ t_{fact}$$

Theorem (Correctness of $\text{fact}$)

$$\forall n. \text{fact } \overline{n} = \overline{n}!$$

Proof.

By induction on $n$...
Review
Overview

- Functional Programming
- Data Structures
- Verification and Testing
- Concurrency
- Analysis of Algorithms
- Advanced Topics
Functional Programming

- OCaml Basics (syntax, evaluation)
- Types (tuples, records, variants, polymorphism)
- Higher-order functions (currying)
- Side-effects (printing, exceptions)
- Maps and folds (tail recursion)
- The Substitution Model
Functional Data Structures

- Basic Modules (signatures, structures)
- Basic data structures (stacks, queues, dictionaries)
- Advanced Modules (abstraction functions, representation invariants)
- Trees (red-black)
- Mutability (arrays, union-find, functional arrays)
- The Environment Model
Verification and Testing

- Logic (propositional, predicate)
- Induction
- Verification (total, partial correctness)
Concurrency

- Threads
- Locks and condition variables
Analysis of Algorithms

- Asymptotic complexity
- Recurrences and recursion trees
- Master method
- Substitution method
- Amortized analysis
Advanced Topics

- Memoization
- Locality and Memory Management
- Graph Algorithms
- Type Inference and Unification
- Laziness and Streams
- $\lambda$-calculus
- Fixpoints and Recursion
Thank you!