# Lecture 7: The Untyped Lambda Calculus

## Writing the interpreter


Polyvios Pratikakis

Computer Science Department, University of Crete


Type Systems and Static Analysis

# Last class

- Semantics:
  - Small-step: $e \to e'$, return the AST rewritten by interpreting one instruction
  - Big-step: $e \downarrow v$, go all the way to a value, *evaluate*
  - Lazy, call-by-name: defer $\beta$-reduction until latest possible point
  - Eager, call-by-value: perform $\beta$-reduction inside sub-terms (but not under $\lambda$) compute argument before applying the function
- Implementation:
  - Free variables
  - $\alpha$-renaming
  - Substitution (capture-avoiding)

# Definitions

- Syntax:

```
type var = string
type value =
  | VFun of var * exp
and exp =
  | EVar of var
  | EVal of value
  | EApp of exp * exp
```

# Free variables (alternative)

$$
\begin{aligned}
FV(x) &= x \\
FV(\lambda x.e) &= FV(e) \setminus \{x\} \\
FV(e_1\ e_2) &= FV(e_1) \cup FV(e_2)
\end{aligned}
$$

```
module VarSet = Set.Make(String)
let rec freevar (ast: exp) : VarSet.t =
  match ast with
    EVar(x) -> VarSet.singleton x
  | EVal(VFun(x, e)) -> VarSet.remove x (freevar e)
  | EApp(e1, e2) -> VarSet.union (freevar e1) (freevar e2)
```

# A little more about modules

- More than namespace
- Abstraction for values (including functions) and types
- Modules have types

```
module type OrderedType =
  sig
    type t
    val compare : t -> t -> int
  end
```

# Module types

- Example

```
module Str : OrderedType =
  struct
    type t = string
    let compare = Pervasives.compare
  end
```

- Actually, built-in:

```
module type Set.OrderedType = sig ... end
module String = struct ... end
```

- `String` defines *at least* the names and types from `Set.OrderedType`

# Functors: functions on modules

```
module Set : sig
  module type OrderedType = sig ... end
  module Make : functor (O: OrderedType) ->
    sig
      type elt = O.t
      type t
      val singleton : elt -> t
      val union : t -> t -> t
      ...
    end
end
```

# Back to implementation: $\alpha$-renaming

- One interesting case: $(\lambda y.e_1)[e/x]$
- Don't recompute $FV(e)$ each time

```
let subst ast e x =
  let fv = freevars e in
  let rec helper ast = ...
  in helper ast
```

- Might need to replace $y$ in $e_1$ with fresh variable
  - Create fresh name $y'$ different from free vars and $x$
  - Keep alpha-renamings in a map: `Map.Make(OrderedType)`, `List.assoc`, etc
  - Renaming map is argument to substitution function

# Creating fresh names

- Use a reference to create unique numbers
- But don't use references if you can avoid it!

```
let next =
  let counter = ref 0 in
  fun str ->
    let n = !counter in
    incr counter;
    str^(string_of_int n)
```

- A function that takes a string and appends a unique number

# A more efficient substitution

```
module StrMap = Map.Make(String)
let subst ast e x =
  let fv = freevars e in
  let rec helper map ast =
    match ast with
      EVar(v) ->
        let v' = StrMap.find v map in
        ...
    | ...
  in helper ast StrMap.empty
```

# Other utility functions

- Test integers

```
let church_from_int n : exp = ...
let scott_from_int n : exp = ...
let church_succ : exp =
  ast_of_string "fun n. fun s. fun z. s (n s z)"
let church_plus : exp =
  ast_of_string "fun m. fun n. fun s. fun z. m s (n s z)"
...
```

- Also try Scott encoding

# Other utility functions, cont'd

- Keep repeating small-step until done:

```
let rec stepper (f: exp -> exp) (e: exp) : exp =
  try (stepper f (f e))
  with Cannot_step _ -> e
```

- Throw away the exception argument: We don't want the subexpression that cannot step (_), we want the whole stopped program e
- Might not terminate

# Next time

- Pure calculus is like assembly
  - Many terms have the same representation
  - Ad-hoc, depending on usage
  - Easy to mix representations, use bool instead of int
- Introduce types
  - A way to separate values into sets, ensure isolation
  - E.g. never mix an integer and a boolean
  - Even when the underlying representation is the same
- Types as relations
  - Prove type-safety: program will not go wrong
  - Property over all executions