CIS 500:
An ML Implementation of the λ-Calculus

Chapter 7 of TAPL

5 October 2005

Today

- Finished up ideas behind de Bruijn indices
- Cover de Bruijn-based implementation of the λ-calculus

- Question: Why use de Bruijn indices in an implementation?

Today

- Finished up ideas behind de Bruijn indices
- Cover de Bruijn-based implementation of the λ-calculus

- Question: Why use de Bruijn indices in an implementation?
- Answer: Can be easier to make your implementation correct (no need to fiddle with names).

The datatype for λ-terms

Recall the grammar of the λ-calculus:

$\mathtt{t}$	$::=$	$\mathtt{x}$	variables
		$\mathtt{t}_1\ \mathtt{t}_2$	application
		$\lambda\mathtt{x}.\mathtt{t}$	abstraction

The corresponding OCaml datatype:

```
type term =
    TmVar of          int
  | TmApp of          term * term
  | TmAbs of                    term
```

The datatype for λ-terms

Recall the grammar of the λ-calculus:

$$
\begin{array}{lcll}
\mathtt{t} & ::= & \mathtt{x} & \text{variables} \\
 & & \mathtt{t}_1\ \mathtt{t}_2 & \text{application} \\
 & & \lambda\mathtt{x}.\mathtt{t} & \text{abstraction}
\end{array}
$$

The corresponding OCaml datatype:

```
type term =
    TmVar of info * int
  | TmApp of info * term * term
  | TmAbs of info *          term
```

Take 2: Include information for error messages.

The datatype for λ-terms

Recall the grammar of the λ-calculus:

$$\begin{array}{rcll} \mathtt{t} & ::= & \mathtt{x} & \text{variables} \\ & & \mathtt{t}_1\ \mathtt{t}_2 & \text{application} \\ & & \lambda\mathtt{x}.\mathtt{t} & \text{abstraction} \end{array}$$

The corresponding OCaml datatype:

```
type term =
    TmVar of info * int * int
  | TmApp of info * term * term
  | TmAbs of info *           term
```

Take 3: Keep track of context size as sanity check.

The datatype for λ-terms

Recall the grammar of the λ-calculus:

$$
\begin{array}{lll}
\mathtt{t} ::= & \mathtt{x} & \text{variables} \\
& \mathtt{t}_1\ \mathtt{t}_2 & \text{application} \\
& \lambda\mathtt{x}.\mathtt{t} & \text{abstraction}
\end{array}
$$

The corresponding OCaml datatype:

```
type term =
    TmVar of info * int * int
  | TmApp of info * term * term
  | TmAbs of info * string * term
```

Final version: Add in information for printing.

Other pieces of code we need

We're aiming to build an interpreter that evaluates terms.

We still need to handle:

- small-step evaluation
- substitution
- shifting indices
- lexing, parsing, printing

We will ignore lexing, parsing, and printing.

Shifting indices

What's being computed: termShift d t $= \uparrow_0^d (t)$

```
let termShift d t =
   let rec walk c t = match t with
         | TmVar(fi,x,n) →
               if x >= c then TmVar(fi,x+d,n+d)
               else TmVar(fi,x,n+d)
         | TmAbs(fi,x,t1) →
               TmAbs(fi, x, walk (c+1) t1)
         | TmApp(fi,t1,t2) →
               TmApp(fi, walk c t1, walk c t2)
   in
      walk 0 t
```

Shifting indices

A closer look: walk c t $= \uparrow^{d}_{c}(t)$

```
let termShift d t =
   let rec walk c t = match t with
         | TmVar(fi,x,n) →
              if x >= c then TmVar(fi,x+d,n+d)
              else TmVar(fi,x,n+d)
         | TmAbs(fi,x,t1) →
              TmAbs(fi, x, walk (c+1) t1)
         | TmApp(fi,t1,t2) →
              TmApp(fi, walk c t1, walk c t2)
   in
      walk 0 t
```

Shifting indices

Note: For variables, take into account the context.

```
let termShift d t =
   let rec walk c t = match t with
         | TmVar(fi,x,n) →
              if x >= c then TmVar(fi,x+d,n+d)
              else TmVar(fi,x,n+d)
         | TmAbs(fi,x,t1) →
              TmAbs(fi, x, walk (c+1) t1)
         | TmApp(fi,t1,t2) →
              TmApp(fi, walk c t1, walk c t2)
   in
      walk 0 t
```

Defining substitution

What's being computed: `termSubst j s t` $= [j \mapsto s]t$.

```
let termSubst j s t =
  let rec walk c t = match t with
    TmVar(fi,x,n) →
      if x=j+c then termShift c s
      else TmVar(fi,x,n)
   | TmAbs(fi,x,t1) →
      TmAbs(fi, x, walk (c+1) t1)
   | TmApp(fi,t1,t2) →
      TmApp(fi, walk c t1, walk c t2)
  in
    walk 0 t
```

Defining substitution

Note: All the shifting is done in the `TmVar` case.

```
let termSubst j s t =
   let rec walk c t = match t with
      TmVar(fi,x,n) →
         if x=j+c then termShift c s
         else TmVar(fi,x,n)
    | TmAbs(fi,x,t1) →
         TmAbs(fi, x, walk (c+1) t1)
    | TmApp(fi,t1,t2) →
         TmApp(fi, walk c t1, walk c t2)
   in
      walk 0 t
```

Wrapping up substitution

Recall that for evaluation, we only need substitution in the rule

$$(\lambda.\mathtt{t})\,\mathtt{v} \quad \longrightarrow \quad \uparrow^{-1}\left([0 \mapsto \uparrow^{1}(\mathtt{v})]\mathtt{t}\right) \qquad \text{(E-AppAbs)}$$

We can provide a simple wrapper for this special case:

```
(* Substitute v for 0 in t. *)
let termSubstTop v t =
   termShift (-1) (termSubst 0 (termShift 1 v) t)
```

Values

Testing for a value is straightforward.

```
let rec isval ctx t = match t with
    TmAbs(_,_,_) → true
  | _ → false
```

A few observations:

- Could use just `let` instead of `let rec`.
- `ctx` argument is unused. It's included for comparison against interpreters for larger languages.

Defining one-step evaluation

Try the rules in order: E-AppAbs, E-App2, E-App1.

```
let rec eval1 ctx t = match t with
    TmApp(fi,TmAbs(_,x,t12),v2) when isval ctx v2 →
      termSubstTop v2 t12
  | TmApp(fi,v1,t2) when isval ctx v1 →
      let t2' = eval1 ctx t2 in
      TmApp(fi,v1,t2')
  | TmApp(fi,t1,t2) →
      let t1' = eval1 ctx t1 in
      TmApp(fi,t1',t2)
  | _ →
      raise NoRuleApplies
```

The end

- First midterm is one week from today (October 12).
  - Everything up through this lecture may be on the exam.
  - For Monday's lecture: Please bring questions!
- Look out for annoucements concerning new office hours.

- Any questions?