Implementing Recursion via State

CS 350

Dr. Joseph Eremondi

Last updated: August 3, 2024

Recursion via State

Overview

- Goals

Overview

- Goals
    - To see how to implement an interpreter for a language with recursion

Overview

- Goals
  - To see how to implement an interpreter for a language with recursion
  - To see how recursion interacts with environments and stores

Overview

- Goals
    - To see how to implement an interpreter for a language with recursion
    - To see how recursion interacts with environments and stores
- Key Concepts

Overview

- Goals
  - To see how to implement an interpreter for a language with recursion
  - To see how recursion interacts with environments and stores
- Key Concepts
  - Landin's Knot

Recursion in Curly-Rec

- When we are allowed to refer to `x` while defining the value that is assigned to `x`

Recursion in Curly-Rec

- When we are allowed to refer to `x` while defining the value that is assigned to `x`
- We'll do this with a special `let` form

Recursion in Curly-Rec

- When we are allowed to refer to `x` while defining the value that is assigned to `x`
- We'll do this with a special `let` form
    - `{letrec x <expr> <expr>}`

Recursion in Curly-Rec

- When we are allowed to refer to `x` while defining the value that is assigned to `x`
- We'll do this with a special `let` form
    - `{letrec x <expr> <expr>}`
        - Gives `{letrec x e1 e2}` gives `x` the value `e1` then evaluates `e2` with `x` in scope

Recursion in Curly-Rec

- When we are allowed to refer to `x` while defining the value that is assigned to `x`
- We'll do this with a special `let` form
    - `{letrec x <expr> <expr>}`
        - Gives `{letrec x e1 e2}` gives `x` the value `e1` then evaluates `e2` with `x` in scope
        - Exactly like `letvar`, except **x is also in scope in e1**

Bad Recursion

- We can't give `x` a value if we need to evaluate `x` to get that value

Bad Recursion

- We can’t give `x` a value if we need to evaluate `x` to get that value
- E.g.

Bad Recursion

- We can’t give `x` a value if we need to evaluate `x` to get that value
- E.g.
  - `{letrec x {+ x 1} ....}`

Bad Recursion

- We can't give `x` a value if we need to evaluate `x` to get that value
- E.g.
  - `{letrec x {+ x 1} ....}`
  - Should raise an error or loop forever

Bad Recursion

- We can’t give `x` a value if we need to evaluate `x` to get that value
- E.g.
    - `{letrec x {+ x 1} ....}`
    - Should raise an error or loop forever
    - In a later lecture we’ll see a way to give this semantics

Good Recursion

- What can we define with recursion?

Good Recursion

- What can we define with recursion?
    - Definitions that refer to x but don’t try to evaluate it

Good Recursion

- What can we define with recursion?
    - Definitions that refer to x but don’t try to evaluate it
    - **Recursive occurrences of the variable must be in the body of a lambda**

Good Recursion

- What can we define with recursion?
  - Definitions that refer to x but don’t try to evaluate it
  - **Recursive occurrences of the variable must be in the body of a lambda**
    - Lambda bodies aren’t evaluated until call time

Interpreting Recursion: The Problem

- To interpret `{letrec x e1 e2}` recursively

Interpreting Recursion: The Problem

- To interpret `{letrec x e1 e2}` recursively
  - Need `x` in scope when interpreting `e1`

Interpreting Recursion: The Problem

- To interpret `{letrec x e1 e2}` recursively
  - Need `x` in scope when interpreting `e1`
  - Can’t put `e1`’s value because we haven’t computed it yet

Interpreting Recursion: The Problem

- To interpret `{letrec x e1 e2}` recursively
  - Need `x` in scope when interpreting `e1`
  - Can't put `e1`'s value because we haven't computed it yet
  - Can't compute `e1`'s value because we need something for

Interpreting Recursion: Idea

- With Mutable Variables, the environment stores *locations*, not values

Interpreting Recursion: Idea

- With Mutable Variables, the environment stores *locations*, not values
- For `{letrec x e1 e2}`

Interpreting Recursion: Idea

- With Mutable Variables, the environment stores *locations*, not values
- For `{letrec x e1 e2}`
  - Generate a new location for the recursive `x` being defined

Interpreting Recursion: Idea

- With Mutable Variables, the environment stores *locations*, not values
- For `{letrec x e1 e2}`
    - Generate a new location for the recursive `x` being defined
        - Put a dummy value at it

Interpreting Recursion: Idea

- With Mutable Variables, the environment stores *locations*, not values
- For `{letrec x e1 e2}`
    - Generate a new location for the recursive `x` being defined
        - Put a dummy value at it
    - Interpret `e1` in the environment extended with `x`'s location

Interpreting Recursion: Idea

- With Mutable Variables, the environment stores *locations*, not values
- For `{letrec x e1 e2}`
    - Generate a new location for the recursive `x` being defined
        - Put a dummy value at it
    - Interpret `e1` in the environment extended with `x`'s location
        - If ever fetch from that location, get the dummy value and raise an error

Interpreting Recursion: Idea

- With Mutable Variables, the environment stores *locations*, not values
- For `{letrec x e1 e2}`
    - Generate a new location for the recursive `x` being defined
        - Put a dummy value at it
    - Interpret `e1` in the environment extended with `x`'s location
        - If ever fetch from that location, get the dummy value and raise an error
        - Shouldn't ever fetch if self-references are in the bodies of functions

Interpreting Recursion: Idea

- With Mutable Variables, the environment stores *locations*, not values
- For `{letrec x e1 e2}`
  - Generate a new location for the recursive `x` being defined
    - Put a dummy value at it
  - Interpret `e1` in the environment extended with `x`'s location
    - If ever fetch from that location, get the dummy value and raise an error
    - Shouldn't ever fetch if self-references are in the bodies of functions
  - Then, **update the store to contain the value of `e1` at the location of `x`**

Interpreting Recursion: Idea

- With Mutable Variables, the environment stores *locations*, not values
- For `{letrec x e1 e2}`
  - Generate a new location for the recursive `x` being defined
    - Put a dummy value at it
  - Interpret `e1` in the environment extended with `x`'s location
    - If ever fetch from that location, get the dummy value and raise an error
    - Shouldn't ever fetch if self-references are in the bodies of functions
  - Then, **update the store to contain the value of `e1` at the location of `x`**
  - Then, interpret the body `e2` with this updated store

Interpreting Recursion: Code

Interpreting Recursion: Code

```
(define (interp [env : Env]
                [e : Expr]
                [sto : Store]) : Result
  (type-case Expr e
   [(LetRec x xexpr body)
     (let* ([x-loc (new-loc sto)] ;; Location for x
            [dummy-sto ;; Put a dummy value at x's location
               (override-store (cell x-loc
                                     (Var 'recursionError)) sto)])
        (with ([x-val x-sto]
               ;; Interpret xexpr in env with x's location
                 (interp (extendEnv (bind x x-loc) env)
                          xexpr
                          dummy-sto))
         ;; Interpret body in env with x's location
         ;; and store with x's newly computed value
         ;; plus any side-effects from xexpr
         (interp (extendEnv (bind x x-loc) env) body
                 (override-store (cell x-loc x-val) x-sto))))]
```

Example

Example

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}
        {fact 3}}
```

- To evaluate `letrec` we:

Example

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}}
        {fact 3}}
```

- To evaluate `letrec` we:
    - Make a new location `0` for `fact`

Example

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}
        {fact 3}}
```

- To evaluate `letrec` we:
  - Make a new location `0` for `fact`
  - Evaluate the value for `fact`

Example

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}
        {fact 3}}
```

- To evaluate `letrec` we:
  - Make a new location `0` for `fact`
  - Evaluate the value for `fact`
    - Env: `fact := 0`

Example

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}
        {fact 3}}
```

- To evaluate `letrec` we:
  - Make a new location `0` for `fact`
  - Evaluate the value for `fact`
    - Env: `fact := 0`
    - Store: `0 => (Var 'recursionError)` ~~

Example (ctd)

- Evaluating function produces closure:

Example (ctd)

- Evaluating function produces closure:

Example (ctd)

- Evaluating function produces closure:

```
(ClosureV
   (Fun 'x (Ifo (Var 'x)
                (NumLit 1)
                (Times (Var 'x)
                       (Call (Var 'fact)
                             (Plus (Var 'x)
                                   (Times (NumLit 1) (NumLit -1))))))
   (fact := 0))
```

- Never fetch from location 0

Example (ctd)

- Evaluating function produces closure:

```
(ClosureV
   (Fun 'x (If0 (Var 'x)
                (NumLit 1)
                (Times (Var 'x)
                       (Call (Var 'fact)
                             (Plus (Var 'x)
                                   (Times (NumLit 1) (NumLit -1))))))
   (fact := 0))
```

- Never fetch from location 0
  - Interp of function doesn't interp body of function

Example (ctd)

- Evaluating function produces closure:

```
(ClosureV
   (Fun 'x (If0 (Var 'x)
                (NumLit 1)
                (Times (Var 'x)
                       (Call (Var 'fact)
                             (Plus (Var 'x)
                                   (Times (NumLit 1) (NumLit -1)))))))
   (fact := 0))
```

- Never fetch from location 0
  - Interp of function doesn't interp body of function
- Closure captures environment with `fact := 0`

Example (ctd)

- Evaluating function produces closure:

```
(ClosureV
   (Fun 'x (Ifo (Var 'x)
                (NumLit 1)
                (Times (Var 'x)
                       (Call (Var 'fact)
                             (Plus (Var 'x)
                                   (Times (NumLit 1) (NumLit -1)))))))
   (fact := 0))
```

- Never fetch from location 0
    - Interp of function doesn’t interp body of function
- Closure captures environment with `fact := 0`
    - Only captures environment, **not store**

Example (ctd)

- Then tie the knot

Example (ctd)

- Then tie the knot
    - Env: `fact := 0`

Example (ctd)

- Then tie the knot
    - Env: `fact := 0`
    - Store: `0 ==> (ClosureV (Fun 'x ....)  (fact := 0))`

Example (ctd)

- Then tie the knot
  - Env: `fact := 0`
  - Store: `0 ==> (ClosureV (Fun ’x ....) (fact := 0))`
    - Updated with value for `fact`

Example (ctd)

- Then tie the knot
    - Env: `fact := ⊙`
    - Store: `⊙ ==> (ClosureV (Fun 'x ....) (fact := ⊙))`
        - Updated with value for `fact`
- Cyclic data structure:

Example (ctd)

- Then tie the knot
  - Env: `fact := 0`
  - Store: `0 ==> (ClosureV (Fun ’x ....) (fact := 0))`
    - Updated with value for `fact`
- Cyclic data structure:
  - Store contains closure at location `0`

Example (ctd)

- Then tie the knot
    - Env: `fact := 0`
    - Store: `0 ==> (ClosureV (Fun ’x ....) (fact := 0))`
        - Updated with value for `fact`
- Cyclic data structure:
    - Store contains closure at location `0`
    - Closure stores environment `(fact := 0)`

Example (ctd)

- Then tie the knot
    - Env: `fact := 0`
    - Store: `0 ==> (ClosureV (Fun ’x ....)  (fact := 0))`
        - Updated with value for `fact`
- Cyclic data structure:
    - Store contains closure at location `0`
    - Closure stores environment `(fact := 0)`
    - That environment points to `0` in store

Example (ctd)

- Then tie the knot
  - Env: `fact := 0`
  - Store: `0 ==> (ClosureV (Fun ’x ....) (fact := 0))`
    - Updated with value for `fact`
- Cyclic data structure:
  - Store contains closure at location `0`
  - Closure stores environment `(fact := 0)`
  - That environment points to `0` in store
  - Store contains closure at location `0` ....

Example (ct)

Example (ct)

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}}
        {fact 3}}
```

- Finally evaluate body in updated store

Example (ct)

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}
        {fact 3}}
```

- Finally evaluate body in updated store
  - Env: `fact := 0`

Example (ct)

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}
        {fact 3}}
```

- Finally evaluate body in updated store
  - Env: `fact := 0`
- `3` evaluates to `(NumLit 3)`, `fact` evaluates to closure from location 0

Example (ct)

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}
        {fact 3}}
```

- Finally evaluate body in updated store
    - Env: `fact := 0`
- 3 evaluates to `(NumLit 3)`, `fact` evaluates to closure from location 0
- Call evaluates closure body

Example (ct)

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}
        {fact 3}}
```

- Finally evaluate body in updated store
  - Env: `fact := 0`
- `3` evaluates to `(NumLit 3)`, `fact` evaluates to closure from location `0`
- Call evaluates closure body
  - Environment `x := 1, fact := 0`

Example (ct)

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}
        {fact 3}}
```

- Finally evaluate body in updated store
    - Env: `fact := 0`
- `3` evaluates to `(NumLit 3)`, `fact` evaluates to closure from location `0`
- Call evaluates closure body
    - Environment `x := 1, fact := 0`
    - Store `0 ==> (ClosureV ....), 1 ==> (NumV 3)`

Example (ct)

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}
        {fact 3}}
```

- Finally evaluate body in updated store
    - Env: `fact := 0`
- `3` evaluates to `(NumLit 3)`, `fact` evaluates to closure from location 0
- Call evaluates closure body
    - Environment `x := 1, fact := 0`
    - Store `0 ==> (ClosureV ....), 1 ==> (NumV 3)`
- `If0` in closure body goes to branch with call

Example (ct)

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}
        {fact 3}}
```

- Finally evaluate body in updated store
  - Env: `fact := 0`
- 3 evaluates to `(NumLit 3)`, `fact` evaluates to closure from location 0
- Call evaluates closure body
  - Environment `x := 1, fact := 0`
  - Store `0 ==> (ClosureV ....), 1 ==> (NumV 3)`
- `If0` in closure body goes to branch with call
- `fact` in call evaluates to *the same closure* at location 0

Example (ct)

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}
        {fact 3}}
```

- Finally evaluate body in updated store
    - Env: `fact := 0`
- `3` evaluates to `(NumLit 3)`, `fact` evaluates to closure from location 0
- Call evaluates closure body
    - Environment `x := 1, fact := 0`
    - Store `0 ==> (ClosureV ....), 1 ==> (NumV 3)`
- `If0` in closure body goes to branch with call
- `fact` in call evaluates to *the same closure* at location 0
    - Evaluation repeats, but with `x` bound to location with `NumV 2`

Example (ct)

```
{letrec fact {fun {x}
                  {if0 x
                       1
                       {* x {fact {- x 1}}}}}
        {fact 3}}
```

- Finally evaluate body in updated store
  - Env: `fact := 0`
- `3` evaluates to `(NumLit 3)`, `fact` evaluates to closure from location 0
- Call evaluates closure body
  - Environment `x := 1, fact := 0`
  - Store `0 ==> (ClosureV ....), 1 ==> (NumV 3)`
- `If0` in closure body goes to branch with call
- `fact` in call evaluates to *the same closure* at location 0
  - Evaluation repeats, but with `x` bound to location with `NumV 2`
  - Etc. until we reach 0 and don't have a recursive call

Landin’s Knot

- This trick is general

Landin’s Knot

- This trick is general
    - Known as *Landin’s Knot*

Landin’s Knot

- This trick is general
  - Known as *Landin’s Knot*
  - British Computer Scientist Peter Landin

Landin’s Knot

- This trick is general
  - Known as *Landin’s Knot*
  - British Computer Scientist Peter Landin
  - Pioneer of functional programming

Landin's Knot

- This trick is general
    - Known as *Landin's Knot*
    - British Computer Scientist Peter Landin
    - Pioneer of functional programming
        - Inventor of *offside rule* for whitespace-sensitive languages

Landin’s Knot

- This trick is general
    - Known as *Landin’s Knot*
    - British Computer Scientist Peter Landin
    - Pioneer of functional programming
        - Inventor of *offside rule* for whitespace-sensitive languages
        - Invented the term *syntactic sugar*

Landin’s Knot

- This trick is general
  - Known as *Landin’s Knot*
  - British Computer Scientist Peter Landin
  - Pioneer of functional programming
    - Inventor of *offside rule* for whitespace-sensitive languages
    - Invented the term *syntactic sugar*
    - Saw the connection between lambda calculus and programming

Landin’s Knot

- This trick is general
  - Known as *Landin’s Knot*
  - British Computer Scientist Peter Landin
  - Pioneer of functional programming
    - Inventor of *offside rule* for whitespace-sensitive languages
    - Invented the term *syntactic sugar*
    - Saw the connection between lambda calculus and programming
    - Early version of algebraic datatypes

Landin’s Knot

- This trick is general
    - Known as *Landin’s Knot*
    - British Computer Scientist Peter Landin
    - Pioneer of functional programming
        - Inventor of *offside rule* for whitespace-sensitive languages
        - Invented the term *syntactic sugar*
        - Saw the connection between lambda calculus and programming
        - Early version of algebraic datatypes
- You can simulate recursion in any language with

Landin’s Knot

- This trick is general
    - Known as *Landin’s Knot*
    - British Computer Scientist Peter Landin
    - Pioneer of functional programming
        - Inventor of *offside rule* for whitespace-sensitive languages
        - Invented the term *syntactic sugar*
        - Saw the connection between lambda calculus and programming
        - Early version of algebraic datatypes
- You can simulate recursion in any language with
    - First-class functions/closures

Landin’s Knot

- This trick is general
    - Known as *Landin’s Knot*
    - British Computer Scientist Peter Landin
    - Pioneer of functional programming
        - Inventor of *offside rule* for whitespace-sensitive languages
        - Invented the term *syntactic sugar*
        - Saw the connection between lambda calculus and programming
        - Early version of algebraic datatypes
- You can simulate recursion in any language with
    - First-class functions/closures
    - Mutable references/variables

Landin’s Knot

- This trick is general
    - Known as *Landin’s Knot*
    - British Computer Scientist Peter Landin
    - Pioneer of functional programming
        - Inventor of *offside rule* for whitespace-sensitive languages
        - Invented the term *syntactic sugar*
        - Saw the connection between lambda calculus and programming
        - Early version of algebraic datatypes
- You can simulate recursion in any language with
    - First-class functions/closures
    - Mutable references/variables
- Useful in typed languages that can’t give the Y-combinator a type