CS653: Functional Programming
2017-18 II^{nd} Semester

Graph Reduction

Amey Karkare

karkare@cse.iitk.ac.in

http://www.cse.iitk.ac.in/~karkare/cs653

Department of CSE, IIT Kanpur

Agenda

- Representing Programs
- Normal Form
- Towards Compilation

Reducing the number of “built-in” functions

- `s` (a sum constructor) is replaced by PACK_SUM_d_r_s, where
  - `d` is the structure tag of `s`: a unique index assigned to each data constructor corresponding to the type of `s`
  - r_S is the arity of `s`
- UNPACK_SUM_s is replaced by UNPACK_SUM_d_r_s
- `t` (a product constructor) is replaced by PACK_PROD_r_t, where
  - r_t is the arity of `t`
- UNPACK_PROD_t is replaced by UNPACK_PROD_r_t
- SEL_t^i is replaced by $\mathrm{SEL}_{r_t}^i$

Reducing the number of “built-in” functions

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

  `Nil` is replaced by PACK_SUM_1_0
  UNPACK_SUM_`Nil` is replaced by UNPACK_SUM_1_0
  `Cons` is replaced by PACK_SUM_2_2
  UNPACK_SUM_`Cons` is replaced by UNPACK_SUM_2_2

- ```
  data Tree a = Leaf a
              | Branch(Tree a)(Tree a)
  ```

  `Leaf` is replaced by PACK_SUM_1_1
  UNPACK_SUM_`Leaf` is replaced by UNPACK_SUM_1_1
  ```

Reducing the number of “built-in” functions

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

  `Nil` is replaced by PACK_SUM_1_0
  UNPACK_SUM_`Nil` is replaced by UNPACK_SUM_1_0
  `Cons` is replaced by PACK_SUM_2_2
  UNPACK_SUM_`Cons` is replaced by UNPACK_SUM_2_2

- ```
  data Tree a = Leaf a
              | Branch(Tree a)(Tree a)
  ```

  `Leaf` is replaced by PACK_SUM_1_1
  UNPACK_SUM_`Leaf` is replaced by UNPACK_SUM_1_1
  `Branch` is replaced by PACK_SUM_2_2
  ```

Reducing the number of "built-in" functions

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

  `Nil` is replaced by PACK_SUM_1_0
  UNPACK_SUM_`Nil` is replaced by UNPACK_SUM_1_0
  `Cons` is replaced by PACK_SUM_2_2
  UNPACK_SUM_`Cons` is replaced by UNPACK_SUM_2_2

- ```
  data Tree a = Leaf a
              | Branch(Tree a)(Tree a)
  ```

  `Leaf` is replaced by PACK_SUM_1_1
  UNPACK_SUM_`Leaf` is replaced by UNPACK_SUM_1_1
  `Branch` is replaced by PACK_SUM_2_2
  UNPACK_SUM_`Branch` is replaced by UNPACK_SUM_2_2
  ```

Reducing the number of “built-in” functions

- ```
  data Complex = Polar Float Float
               | Rect  Float Float
  ```

`Polar` is replaced by PACK_SUM_1_2
UNPACK_SUM_`Polar` is replaced by UNPACK_SUM_1_2
  ```

Reducing the number of “built-in” functions

- ```
  data Complex = Polar Float Float
               | Rect  Float Float
  ```

`Polar` is replaced by PACK_SUM_1_2
UNPACK_SUM_`Polar` is replaced by UNPACK_SUM_1_2
`Rect` is replaced by PACK_SUM_2_2
  ```

Reducing the number of “built-in” functions

- ```
  data Complex = Polar Float Float
               | Rect  Float Float
  ```

`Polar` is replaced by PACK_SUM_1_2
UNPACK_SUM_`Polar` is replaced by UNPACK_SUM_1_2
`Rect` is replaced by PACK_SUM_2_2
UNPACK_SUM_`Rect` is replaced by UNPACK_SUM_2_2
  ```

Reducing the number of “built-in” functions

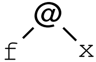
- `data Pair a b = Pair a b`

$$
\begin{aligned}
\texttt{Pair} &\text{ is replaced by PACK\_PROD\_2} \\
\text{UNPACK\_PROD\_}\texttt{Pair} &\text{ is replaced by UNPACK\_PROD\_2} \\
\text{SEL}^1_{Pair} &\text{ is replaced by } \text{SEL}^1_2 \\
\text{SEL}^2_{Pair} &\text{ is replaced by } \text{SEL}^2_2
\end{aligned}
$$

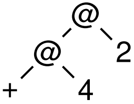
Graph Reduction

Program Representation

- Abstract Syntax Trees
- Application: `(f x)`



- Multiple arguments handled by currying, e.g. (+ 4 2)



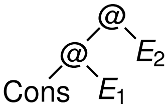
Program Representation

- Abstract Syntax Trees
- λ abstraction: λx.body

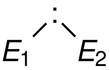
$$
\begin{array}{c}
\lambda x \\
| \\
\text{body}
\end{array}
$$

Cons: Expression and Result

- The graph for the expression (Cons E_1 E_2)



- And its result, the CONS cell



Evaluation Strategy

- Program converted to graph.
- Goal is to reduce the graph to normal form.
- Evaluation strategy is simple:
  - Select the *next* redex to be reduced.
  - Reduce it.

Lazy evaluation

- Two ingredients.
- Argument to functions should be evaluated only when their value is needed
  - not when the function is applied
- Arguments should be evaluated only once.
  - Further uses of the argument within the function should use the value computed the first time.
  - Referential transparency: the result is same as re-evaluating the argument.

Applicative Order vs. Normal Order Reduction

- Applicative Order: Strict semantics $\Rightarrow$ reduce argument before application
- Normal Order: Lazy Semantics

Normal Form: Do we need it?

- Consider an expression E whose result is a CONS cell.
- Evaluating E should not entail evaluating CONS's head and tail.
  - unless required by surrounding expressions/environment
- Thus, we could stop reduction even though some redexes are left in the graph
- A "special" kind of normal form

Weak Head Normal Form (WHNF)

- A λ expression is in *weak head normal form (WHNF)* if and only if it is of the form

$$F\ E_1\ E_2\ \ldots\ E_n$$

  where, $n \geq 0$, and
  - Either F is a variable or data object
  - Or F is a λ abstraction or built-in function and $(F\ E_1\ E_2\ \ldots\ E_m)$ is **not** a redex for any $m \leq n$.
- An expression has no *top-level redex* if and only if it is in WHNF.

Examples

λ **Expr**	**NF?**	**WHNF?**
3	$\checkmark$	$\checkmark$
A CONS cell	$\checkmark$ / $\times$	$\checkmark$
$+$ ($-$ 4 3)	$\times$	$\checkmark$
(λx.+ 5 1)	$\times$	$\checkmark$
$+$ 5 ($-$ 4 3)	$\times$	$\times$

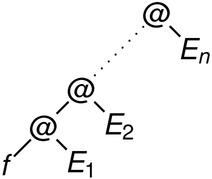
Evaluating Arguments of built-in functions

- Some builtin functions are strict in (one or more of) their arguments
- Need to evaluate their arguments before they can execute
- Examples:
  - $+ (- 4\ 3)\ 5$
  - IF (NOT TRUE) f g h
  - HEAD (CONS 2 NIL)
- The evaluator has to invoke itself recursively to evaluate the arguments of strict built-in functions (to WHNF form)

The Next Top-level Redex

- Our expression can only be of the form

  $$f\ E_1\ E_2\ \ldots\ E_n$$



- f can be a data object, a built-in function, or a λ-abstraction
- Zero or more arguments (E_i)

The Next Top-level Redex

- f is a data object.
  - For example, a number or a CONS cell.
  - Expression is already in WHNF.
  - n should be zero.
  - Otherwise a type error (should not have reached here!)

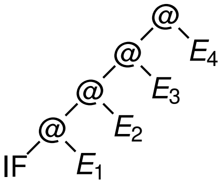
The Next Top-level Redex

- f is a built-in function taking k arguments.
- If $n \geq k$,
  - Choose $(f\ E_1 \ldots E_k)$ for reduction.
- If $n < k$,
  - Expression is already in WHNF.

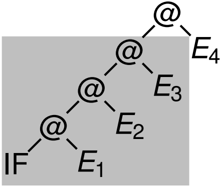
The Next Top-level Redex

- f is a λ abstraction.
- If an argument is available, i.e., $n \geq 1$,
  - Choose $(f\ E_1)$ for reduction.
- If $n = 0$,
  - Expression is already in WHNF.

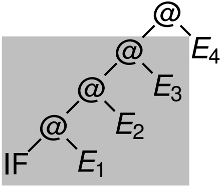
Examples of Next Redex

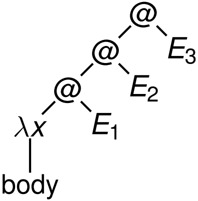


Examples of Next Redex

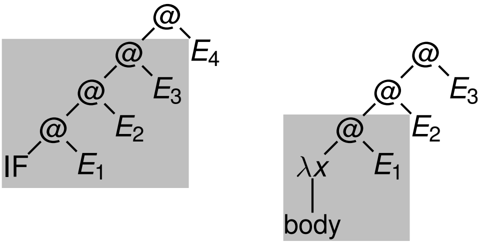


Examples of Next Redex



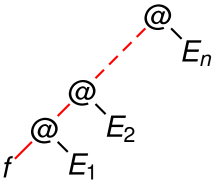


Examples of Next Redex

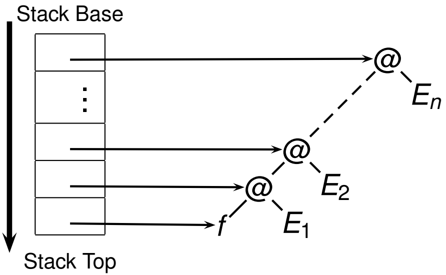


Terminology

- **Spine:** left-branching chain of application nodes (along red lines in the figure)
- **Unwinding a spine:** Act of 'going down' the spine
- **Vertebrae:** Application nodes (@ nodes in the figure) encountered during unwinding.
  - Any @ nodes inside E_is are not vertebrae for *this* expression.
- **Ribs:** the arguments to the vertebrae (E_i)
- **Tip of the Spine:** the extreme bottom of the spine (f)



The Spine Stack



- Stack of pointers to vertebrae
- Stack depth = Number of arguments
- Possible to access inner elements (vertebrae)
- Allows us to overwrite the root of the redex with the result
  - Required for reduction

The Spine Stack

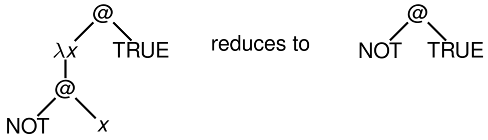
- To recursively evaluate the arguments, we need a new stack
  - the existing stack does not change until the argument evaluation is complete.
  - This new stack can be discarded when the argument evaluation is complete.
- New stack can be built on top of the old one
  - similar to Stack Frames in imperative languages
- Or separate space can be used (called *dump*)

Graph Reduction of λ Expressions

- After having identified the redex, we must do the reduction.
- Reduction $\Rightarrow$ a *local transformation* of the graph representing the expression.
- Successive reductions reduce the graph to the result of the computation.

Reducing a λ Application

- Redex consists of λ abstraction applied to an argument.
- Example:



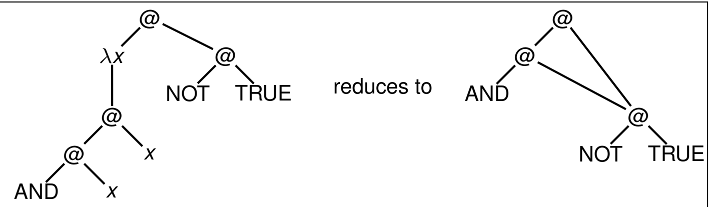
$(\lambda x.\text{NOT } x)$ TRUE $\rightarrow$ NOT TRUE

Reducing a λ Application: Implementation Issues

- The argument may be bulky and/or contain redexes, so do *pointer manipulations*
- The redex may be shared, so *overwrite* the root of the redex with the result
- *Shared* λ abstractions should not be destroyed, so create a copy of the λ body.

Substituting Pointers to the Arguments

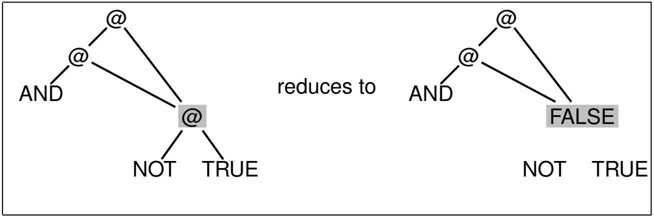
- Example:



$(\lambda x.\mathrm{AND}\ x\ x)\ (\mathrm{NOT\ TRUE}) \to \mathrm{AND}\ (\mathrm{NOT\ TRUE})\ (\mathrm{NOT\ TRUE})$

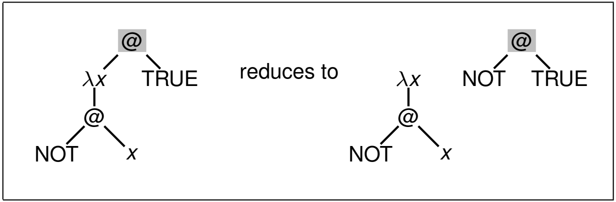
Overwriting the Root of the Redex

- Example:



Constructing a New Instance of the λ Body

- Example:



Instantiation Algorithm

```
instantiate(body, var, value)
  if body is a variable x and var=x then
      return value
  if body is a variable x and var≠x then
      return body
  if body is a constant or built in function then
      return body
  if body is (e1 e2), then
      return (instantiate(e1, var, value)
                    instantiate(e2, var, value))
  if body is λx.e and var=x then
      return body
  if body is λx.e and var≠x then
      return λx.instantiate(e, var, value)
```

The root of the redex is updated with whatever is returned by instantiate(body, var, value)

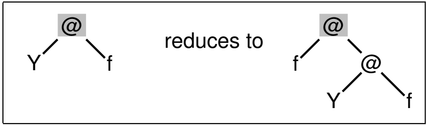
Exercise for Instantiation

For each case, draw the expression tree and show the steps of the instantiate algorithm. Clearly mark/number the nodes to help understand which nodes are modified or overwritten at each stage. Draw all garbage nodes at each stage.

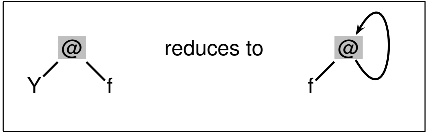
- `(λx.NOT x) t      -- instantiate(NOT x, x, t)`
- `(λxλx.2+x) t    -- instantiate(λx.2+x, x, t)`
- `(λxλy.2+x) t    -- instantiate(λy.2+x, x, t)`
- `(λx.x) t              -- instantiate(x, x, t)`
- `(λx.+ (+ 3 5) x) t`
  `-- instantiate(+ (+ 3 5) x, x, t)`

Implementing Y

- Two ways to implement Y directly
- Recall that Y f = f (Y f)
- First way



- Second way



- This is where *cycle* gets introduced in the graph.

Exercise

- Show the steps in the graph reduction of
  (Y λfλn . if n == 0 then 1 else n * f (n - 1)) 2