CS653: Functional Programming 2017-18 *II*nd Semester

Haskell to Enriched λ Calculus

Amey Karkare

`karkare@cse.iitk.ac.in`

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

Department of CSE, IIT Kanpur

Agenda

- Enriched λ calculus
- Translating Haskell to enriched λ calculus
  - Translating Expressions ($\mathbb{TE}[\![\ ]\!]$)
  - Translating Definitions ($\mathbb{TD}[\![\ ]\!]$)

Acknowledgements

- The slides are based on the book **The Implementation of Functional Programming Languages** by *Simon L. Peyton Jones*
  E-copy: `https://www.microsoft.com/en-us/research/wp-content/uploads/1987/01/slpj-book-1987-small.pdf`

Enriched λ calculus

- Ordinary λ calculus with few extra constructs
- Extra constructs can be expressed in terms of ordinary λ calculus
- Extra layer introduced for ease of translation
- Works as a *correct* reference implementation
  - Far greater efficiency can be achieved by using complicated translation

Enriched λ calculus

- λ calculus
- Constants, built-in operators and values
- let-expressions and letrec-expressions
- built-in value FAIL
- built-in infix operator [] (called FATBAR)
- pattern matching λ abstractions
- case-expressions

Simple let expressions

- Syntax: `let` $v = B$ `in` E
  - v is a variable,
  - B and E are expressions in enriched λ calculus
- Equivalent to: `((`$\lambda v.E$`)` B`)`

Simple letrec expressions

- Syntax:

$$
\begin{array}{ll}
\texttt{letrec} & v_1 = E_1 \\
 & v_2 = E_2 \\
 & \ldots \\
 & v_n = E_n \\
\texttt{in} & \\
 & E
\end{array}
$$

Simple letrec

- Semantics for a *single* definition letrec
  $(\texttt{letrec}\ v = B\ \texttt{in}\ E) \equiv (\texttt{let}\ v = Y(\lambda v.B)\ \texttt{in}\ E)$
- How to extend it to multiple definition letrec?

FAIL and [] (FATBAR)

- Consider Haskell definition

  $f \;\; p_1 = E_1$
  $f \;\; p_2 = E_2$
  $\ldots$
  $f \;\; p_n = E_n$

- Informally, the execution at a call $f\ x$ happens as follows:
  - The first equation's pattern p_1 is matched against x. If it matches, body E_1 is executed.
  - If it fails, second pattern p_2 is tried.
  - and so on
  - If all patterns fail to match, the pattern-match (and hence the execution) fails.
- Note that failure to match a single pattern does not constitute a failure of execution.

FAIL and [] (FATBAR)

- FAIL and [] capture the previous idea.
- The definition of f can be translated to enriched λ calculus as:

$$
\begin{array}{rl}
f = \lambda x.(& ((\lambda p'_1 = E'_1)\ \ x) \\
[] & ((\lambda p'_2 = E'_2)\ \ x) \\
 & \dots \\
[] & ((\lambda p'_n = E'_n)\ \ x) \\
[] & \mathtt{ERROR})
\end{array}
$$

where

- x is a fresh variable ($\notin FV(E_i)$ for any E_i),
- p'_i and E'_i are translations of respective p_i and E_i,
- and

$$
\begin{array}{lllll}
\mathtt{a} & [] & \mathtt{b} = \mathtt{a} & & \mathtt{if}\ \mathtt{a} \neq \perp\ \mathtt{and}\ \mathtt{a} \neq \text{FAIL} \\
\text{FAIL} & [] & \mathtt{b} = \mathtt{b} & & \\
\perp & [] & \mathtt{b} = \perp & &
\end{array}
$$

Pattern Matching λ Abstractions

- $\lambda p.E$ where p is a pattern
- Types of patterns
  - Variable (e.g., x)
  - Constant (e.g., 4)
  - Sum-constructor (e.g., Nil | Cons x xs)
  - Product-constructor (e.g., Pair x y)

Variable Patterns

- If a pattern p is a variable v, then $\lambda p.E$ is the λ abstraction $\lambda v.E$.
- The semantics are those of β-reduction.

Constant Patterns

- If a pattern p is a constant k, then $\lambda p.E$ is $\lambda k.E$.

$$\begin{aligned}
\text{Eval}[\![\lambda k.E]\!]\ a &= \text{Eval}[\![E]\!] && \text{if } a = \text{Eval}[\![k]\!] \\
\text{Eval}[\![\lambda k.E]\!]\ a &= \text{FAIL} && \text{if } a \neq \text{Eval}[\![k]\!] \\
\text{Eval}[\![\lambda k.E]\!]\ \perp &= \perp
\end{aligned}$$

- Examples:

$$\begin{aligned}
\text{Eval}[\![\lambda 1.(+\ 2\ 4)]\!]\ 1 &= 6 \\
\text{Eval}[\![\lambda 1.(+\ 2\ 4)]\!]\ 2 &= \text{FAIL}
\end{aligned}$$

Sum-constructor Patterns

- Pattern p is of the form $(s\ p_1 \ldots p_r)$

$$\text{Eval}[\![\ \lambda(s\ p_1 \ldots p_r).E\ ]\!]\ (s\ a_1 \ldots a_r) = \text{Eval}[\![\ \lambda p_1 \ldots p_r.E\ ]\!]\ a_1 \ldots a_r$$

$$\text{Eval}[\![\ \lambda(s\ p_1 \ldots p_r).E\ ]\!]\ (s'\ a_1 \ldots a_r) = \text{FAIL} \quad \text{if } s \neq s'$$

$$\text{Eval}[\![\ \lambda(s\ p_1 \ldots p_r).E\ ]\!]\ \bot = \bot$$

Implementing Sum-constructor Pattern Matching

- To implement the semantics of sum-constructors, we invent a new function, UNPACK_SUM_s, for each sum-constructor s

$$
\begin{array}{lcl}
\text{UNPACK\_SUM\_s}\ \ f\ \ (s\, a_1 \ldots a_r) & = & f\ a_1 \ldots a_r \\
\text{UNPACK\_SUM\_s}\ \ f\ \ (s'\, a_1 \ldots a_r) & = & \text{FAIL},\ \text{if } s \neq s' \\
\text{UNPACK\_SUM\_s}\ \ f\ \ \perp & = & \perp
\end{array}
$$

- Note that

$$(\lambda(s\ p_1 \ldots p_r).E) = \text{UNPACK\_SUM\_s}\ (\lambda p_1 \ldots p_r.E)$$

Exercises

- Consider the data declarations:

```
data Tree a  =  Branch (Tree a) (Tree a)
             |  Leaf a
data List a  =  Cons a (List a)
             |  Nil
```

  Compute the result by applying pattern matching rules to following λ terms

  - `(λ (Branch t1 t2).Branch t2 t1) (Leaf 0)`
  - `(λ (Branch t1 t2).Branch t2 t1)`
    `          (Branch (Leaf 0) (Leaf 1))`
  - `(λ FALSE y.y) FALSE TRUE`
  - `(λ (Cons x Nil).x) (Cons 4 (Cons 3 Nil))`

Product-constructor Patterns

- Pattern p is of the form $(s\ p_1 \ldots p_r)$
- However, there is no alternate pattern to match
- Therefore, type checking $\Rightarrow$ pattern matching

Product-constructor Patterns

- Consider
  `zeroPair (x,y) = 0`
- What is the value of `zeroPair` $\perp$?
- Strict pattern matching results in value $\perp$.
- Lazy pattern matching results in value 0.

Lazy Product-constructor Pattern Matching

$$
\begin{array}{rcll}
\text{Eval}[\![\ \lambda(t\ p_1 \ldots p_r).E\ ]\!]\ a & = & \text{Eval}[\![\ \lambda p_1 \ldots p_r.E\ ]\!] & (\text{SEL}_t^1\ a) \\
 & & & \ldots \\
 & & & (\text{SEL}_t^r\ a)
\end{array}
$$

where

$$
\begin{array}{rcl}
\text{SEL}_t^i\ (t\ a_1 \ldots a_i \ldots a_r) & = & a_i \\
\text{SEL}_t^i\ \perp & = & \perp
\end{array}
$$

Implementing Product-constructor Pattern Matching

- To implement the semantics of product-constructors, we invent a new function, UNPACK_PROD_t, for each product-constructors t

$$\text{UNPACK\_PROD\_t} \;\; f\; a \quad = \quad f\;(\text{SEL}_t^1\, a) \ldots (\text{SEL}_t^r\, a)$$

- Note that

$$(\lambda(t\; p_1 \ldots p_r).E) = \text{UNPACK\_PROD\_t}\; (\lambda p_1 \ldots p_r.E)$$

- SEL *lazily* selects components of an object.

Exercise

- Consider the definition
  `zeroPair` = λ `(PAIR x y).0`
  Show complete evaluation of the expression `zeroPair` $\perp$.

Multi-definition letrec

- Multi-definition letcrec can be translated to a single definition letrec having product-constructor (tuple):

$$\left.\begin{array}{ll} \text{letrec} & p_1 = B_1 \\ & \ldots \\ & p_n = B_n \\ \text{in} & E \end{array}\right\} \equiv \left\{\begin{array}{ll} \text{letrec} & (p_1, \ldots, p_n) = (B_1, \ldots, B_n) \\ \text{in} & E \end{array}\right.$$

Case Expression

- Notation for a describing a simple form of pattern-matching
- patterns are
  - simple: not nested
  - exhaustive: cover all constructors of a type
- General form:

$$
\begin{array}{l}
\text{case } v \text{ of} \\
\quad c_1 \; v_{1,1} \ldots v_{1,r_1} \Rightarrow E_1 \\
\quad \ldots \\
\quad c_n \; v_{n,1} \ldots v_{n,r_n} \Rightarrow E_1
\end{array}
$$

  - v is a variable
  - $E_1 \ldots E_n$ are expressions
  - $v_{i,j}$ are distinct variables
  - $c_1 \ldots c_n$ are a *complete family* of constructors from a structured type declaration

Case Expression Semantics

- To evaluate case expression:
  - v is evaluated first
  - According to what construct v was built with, appropriate E_i is selected, and
  - E_i is evaluated with the $v_{i,j}$ bound to components of v.
- The construct is equivalent to

$$
\begin{array}{ll}
 & ((\lambda(c_1\ v_{1,1} \ldots v_{1,r_1}).E_1)v) \\
[] & \ldots \\
[] & ((\lambda(c_n\ v_{n,1} \ldots v_{n,r_n}).E_n)v)
\end{array}
$$

- However, it is more readable. Also, more efficient implementation is possible.

Translating Expressions: $\mathbb{TE}$

$$
\begin{aligned}
\mathbb{TE}[\![\, k \,]\!] &= k \\
\mathbb{TE}[\![\, v \,]\!] &= v \\
\\
\mathbb{TE}[\![\, E_1 \; E_2 \,]\!] &= \mathbb{TE}[\![\, E_1 \,]\!] \; \mathbb{TE}[\![\, E_2 \,]\!] \\
\mathbb{TE}[\![\, E_1 \text{ infix } E_2 \,]\!] &= \mathbb{TE}[\![\, \text{infix} \,]\!] \; \mathbb{TE}[\![\, E_1 \,]\!] \; \mathbb{TE}[\![\, E_2 \,]\!] \\
\mathbb{TE}[\![\, E_1 \text{ `v` } E_2 \,]\!] &= \mathbb{TE}[\![\, \text{v} \,]\!] \; \mathbb{TE}[\![\, E_1 \,]\!] \; \mathbb{TE}[\![\, E_2 \,]\!] \\
\\
\mathbb{TE}[\![\, : \,]\!] &= \text{CONS} \\
\mathbb{TE}[\![\, [\,] \,]\!] &= \text{NIL} \\
\mathbb{TE}[\![\, [E_1, \ldots, E_n] \,]\!] &= \text{CONS } \mathbb{TE}[\![\, E_1 \,]\!] \ldots \mathbb{TE}[\![\, E_n \,]\!] \\
\mathbb{TE}[\![\, (E_1, E_2) \,]\!] &= \text{PAIR } \mathbb{TE}[\![\, E_1 \,]\!] \; \mathbb{TE}[\![\, E_2 \,]\!]
\end{aligned}
$$

Translating Definitions: $\mathbb{TD}$

Exercise.

Example Translations

- We shall drive the rest of the translation process through a series of examples.
- In the process, we shall introduce new rules and optimizations.
- A thorough treatment of the translation process can be found in the reference (book by S L Peyton Jones)

Translation #1

- Translate the following definition:

```
length []       =  0
length (x:xs)   =  1 + length xs
```

```
length [] = 0
length (x:xs) = 1 + length xs
```

push patterns to the right

```
length l = case l of
                  [] -> 0
                  (x:xs) -> 1 + length xs
```

replace multiple alts by (pattern, _) pairs

```
length l = case l of
                  [] -> 0
                  _ -> case l of
                           (x:xs) - 1 + length xs
                           _ -> error "insufficient patterns"
```

0-ary constructor rule

```
length l = if (l==[]) then 0
           else case l of
                  (x:xs) - 1 + length xs
                  _ -> error "insufficient patterns"
```

refutable pattern matching

```
length l = if (l==[]) then 0
           else (UNPACK_SUM_:  (λxλxs -> 1 + length xs) l)
                [] ERROR
```

remove recursion

```
length = Y (λlenλl.if (l==[]) then 0
                  else (UNPACK_SUM_:  (λxλxs -> 1 + len xs) l)
                       [] ERROR)
```