Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha
Terms

- A function is in **Direct Style** when it returns its result back to the caller.

- A **Tail Call** occurs when a function returns the result of another function call without any more computations (eg tail recursion)

- A function is in **Continuation Passing Style** when it, and every function call in it, passes its result to another function.

- Instead of returning the result to the caller, we pass it forward to another function.
Terminology

- Tail Position: A subexpression $s$ of expressions $e$, such that if evaluated, will be taken as the value of $e$
  - if $(x>3)$ then $x + 2$ else $x - 4$
  - let $x = 5$ in $x + 4$

- Tail Call: A function call that occurs in tail position
  - if $(h \, x)$ then $f \, x$ else $(x + g \, x)$
Terminology

- **Available**: A function call that can be executed by the current expression.

- The fastest way to be unavailable is to be guarded by an abstraction (anonymous function, lambda lifted).
  
  - if \((h \ x)\) then \(f \ x\) else \((x + g \ x)\)
  
  - if \((h \ x)\) then (fun \(x\) -> \(f \ x\)) else \((g \ (x + x))\)

  Not available
CPS Transformation

- **Step 1**: Add continuation argument to any function definition:
  - let f arg = e \(\Rightarrow\) let f arg k = e
  - Idea: Every function takes an extra parameter saying where the result goes

- **Step 2**: A simple expression in tail position should be passed to a continuation instead of returned:
  - return a \(\Rightarrow\) k a
  - Assuming a is a constant or variable.
  - “Simple” = “No available function calls.”
CPS Transformation

- Step 3: Pass the current continuation to every function call in tail position
  - return f arg ⇒ f arg k
  - The function “isn’t going to return,” so we need to tell it where to put the result.
CPS Transformation

- Step 4: Each function call not in tail position needs to be converted to take a new continuation (containing the old continuation as appropriate)
  - return $\text{op (f arg)} \Rightarrow f \text{ arg (fun r -> k(op r))}$
  - $\text{op}$ represents a primitive operation

- return $\text{f(g arg)} \Rightarrow g \text{ arg (fun r-> f r k)}$
Example

Before:
let rec add_list lst =
  match lst with
  | [] -> 0
  | 0 :: xs -> add_list xs
  | x :: xs -> (+) x
    (add_list xs);;

After:
let rec add_listk lst k =
  (* rule 1 *)
  match lst with
  | [] -> k 0 (* rule 2 *)
  | 0 :: xs -> add_listk xs k
  | x :: xs -> add_listk xs
    (fun r -> k ((+) x r));;
  (* rule 4 *)
Variants - Syntax (slightly simplified)

- type \( \text{Name} = C_1 [\text{of ty}_1] | \ldots | C_n [\text{of ty}_n] \)
- Introduce a type called \( \text{Name} \)
- \((\text{fun } x \rightarrow C_i x) : \text{ty}_1 \rightarrow \text{Name}\)
- \(C_i\) is called a \textit{constructor}; if the optional type argument is omitted, it is called a \textit{constant}
- Constructors are the basis of almost all pattern matching
Enumeration Types as Variants

An enumeration type is a collection of distinct values.

In C and Ocaml they have an order structure; order by order of input.
# type weekday = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday

```plaintext
type weekday =
  Monday
| Tuesday
| Wednesday
| Thursday
| Friday
| Saturday
| Sunday
```

Enumeration Types as Variants

9/22/15
Functions over Enumerations

```ocaml
# let day_after day = match day with
    | Monday -> Tuesday
    | Tuesday -> Wednesday
    | Wednesday -> Thursday
    | Thursday -> Friday
    | Friday -> Saturday
    | Saturday -> Sunday
    | Sunday -> Monday

val day_after : weekday -> weekday = <fun>
```
Functions over Enumerations

# let rec days_later n day =
   match n with 0 -> day
   | _  -> if n > 0 then day_after (days_later (n - 1) day)
       else days_later (n + 7) day;;

val days_later : int -> weekday -> weekday = <fun>
Functions over Enumerations

```
# days_later 2 Tuesday;;
- : weekday = Thursday
# days_later (-1) Wednesday;;
- : weekday = Tuesday
# days_later (-4) Monday;;
- : weekday = Thursday
```
Problem:

```ocaml
# type weekday = Monday | Tuesday | Wednesday
    | Thursday | Friday | Saturday | Sunday;;

- Write function is_weekend : weekday -> bool
  let is_weekend day =
```
Problem:

```ocaml
# type weekday = Monday | Tuesday | Wednesday
| Thursday | Friday | Saturday | Sunday

- Write function `is_weekend : weekday -> bool`
- let is_weekend day =
  match day with Saturday -> true
  | Sunday -> true
  | _ -> false
```
Example Enumeration Types

```haskell
# type bin_op = IntPlusOp | IntMinusOp | EqOp | CommaOp | ConsOp

# type mon_op = HdOp | TlOp | FstOp | SndOp
```
Disjoint Union Types

- Disjoint union of types, with some possibly occurring more than once

- We can also add in some new singleton elements
Disjoint Union Types

# type id = DriversLicense of int | SocialSecurity of int | Name of string;;

type id = DriversLicense of int | SocialSecurity of int | Name of string

# let check_id id = match id with
  DriversLicense num ->
    not (List.mem num [13570; 99999])
  | SocialSecurity num -> num < 900000000
  | Name str -> not (str = "John Doe");;

val check_id : id -> bool = <fun>
Problem

- Create a type to represent the currencies for US, UK, Europe and Japan
Problem

Create a type to represent the currencies for US, UK, Europe and Japan

type currency =
    Dollar of int
| Pound of int
| Euro of int
| Yen of int
Example Disjoint Union Type

# type const =

  BoolConst of bool

  | IntConst of int

  | FloatConst of float

  | StringConst of string

  | NilConst

  | UnitConst
Example Disjoint Union Type

```haskell
# type const = BoolConst of bool 
| IntConst of int | FloatConst of float 
| StringConst of string | NilConst 
| UnitConst 
```

- How to represent 7 as a const?
- Answer: IntConst 7
Polymorphism in Variants

- The type 'a option is gives us something to represent non-existence or failure

```ocaml
# type 'a option = Some of 'a | None;;
type 'a option = Some of 'a | None
```

- Used to encode partial functions
- Often can replace the raising of an exception
Functions producing option

```ocaml
# let rec first p list =
    match list with [ ] -> None
    | (x::xs) -> if p x then Some x else first p xs;;
val first : ('a -> bool) -> 'a list -> 'a option = <fun>
```

```ocaml
# first (fun x -> x > 3) [1;3;4;2;5];;
- : int option = Some 4
```

```ocaml
# first (fun x -> x > 5) [1;3;4;2;5];;
- : int option = None
```
# let result_ok r =
  match r with None -> false
  | Some _ -> true;;
val result_ok : 'a option -> bool = <fun>
# result_ok (first (fun x -> x > 3) [1;3;4;2;5]);;
- : bool = true
# result_ok (first (fun x -> x > 5) [1;3;4;2;5]);;
- : bool = false
Problem

- Write a `hd` and `tl` on lists that doesn’t raise an exception and works at all types of lists.
Problem

- Write a hd and tl on lists that doesn’t raise an exception and works at all types of lists.

- \( \text{let hd list =} \)
  
  \[
  \text{match list with } [] \rightarrow \text{None} \\
  \mid (x::xs) \rightarrow \text{Some x}
  \]

- \( \text{let tl list =} \)
  
  \[
  \text{match list with } [] \rightarrow \text{None} \\
  \mid (x::xs) \rightarrow \text{Some xs}
  \]
Mapping over Variants

```ocaml
# let optionMap f opt =
  match opt with
  | None -> None
  | Some x -> Some (f x);

val optionMap : ('a -> 'b) -> 'a option -> 'b option = <fun>

# optionMap
  (fun x -> x - 2)
  (first (fun x -> x > 3) [1;3;4;2;5]);
  : int option = Some 2
```
Folding over Variants

# let optionFold someFun noneVal opt =
  match opt with None -> noneVal
  | Some x -> someFun x;;
val optionFold : ('a -> 'b) -> 'b -> 'a option -> 'b = <fun>

# let optionMap f opt =
  optionFold (fun x -> Some (f x)) None opt;;
val optionMap : ('a -> 'b) -> 'a option -> 'b option = <fun>
Recursive Types

- The type being defined may be a component of itself
Recursive Data Types

# type exp =

    VarExp of string
| ConstExp of const
| MonOpAppExp of mon_op * exp
| BinOpAppExp of bin_op * exp * exp
| IfExp of exp* exp * exp
| AppExp of exp * exp
| FunExp of string * exp
Recursive Data Types

# type bin_op = IntPlusOp | IntMinusOp | EqOp | CommaOp | ConsOp | ... 
# type const = BoolConst of bool | IntConst of int | ... 
# type exp = VarExp of string | ConstExp of const | BinOpAppExp of bin_op * exp * exp | ... 

- How to represent 6 as an exp?
Recursive Data Types

# type bin_op = IntPlusOp | IntMinusOp | EqOp | CommaOp | ConsOp | ...
# type const = BoolConst of bool | IntConst of int | ...
# type exp = VarExp of string | ConstExp of const | BinOpAppExp of bin_op * exp * exp | ...

- How to represent 6 as an exp?
- Answer: ConstExp (IntConst 6)
Recursive Data Types

```haskell
# type bin_op = IntPlusOp | IntMinusOp
    | EqOp | CommaOp | ConsOp | ...
# type const = BoolConst of bool | IntConst of int |
...
# type exp = VarExp of string | ConstExp of const
    | BinOpAppExp of bin_op * exp * exp | ...
```

- How to represent \((6, 3)\) as an exp?
Recursive Data Types

```haskell
# type bin_op = IntPlusOp | IntMinusOp | EqOp | CommaOp | ConsOp | ...
# type const = BoolConst of bool | IntConst of int |
...
# type exp = VarExp of string | ConstExp of const | BinOpAppExp of bin_op * exp * exp | ...
```

- How to represent \((6, 3)\) as an `exp`?
- `BinOpAppExp (CommaOp, ConstExp (IntConst 6), ConstExp (IntConst 3))`
Recursive Data Types

```haskell
# type bin_op = IntPlusOp | IntMinusOp
    | EqOp | CommaOp | ConsOp | ...

# type const = BoolConst of bool | IntConst of int |
    ...

# type exp = VarExp of string | ConstExp of const
    | BinOpAppExp of bin_op * exp * exp | ...
```

- How to represent \([(6, 3)]\) as an exp?
Your turn now

Try Problem 1 on MP3
Recursive Data Types

# type int_Bin_Tree =
Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree);

type int_Bin_Tree = Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)
Recursive Data Type Values

# let bin_tree =
  Node(Node(Leaf 3, Leaf 6), Leaf (-7));;

val bin_tree : int_Bin_Tree = Node (Node (Leaf 3, Leaf 6), Leaf (-7))
Recursive Data Type Values

```
bin_tree = Node
         /     \
        /       \
  Node    Leaf (-7)
 /       /     \
Leaf 3   Leaf 6
```
Recursive Functions

```ocaml
# let rec first_leaf_value tree =
    match tree with (Leaf n) -> n |
    Node (left_tree, right_tree) ->
    first_leaf_value left_tree;;
val first_leaf_value : int_Bin_Tree -> int = <fun>
# let left = first_leaf_value bin_tree;;
val left : int = 3
```
Problem

type int_Bin_Tree = Leaf of int
| Node of (int_Bin_Tree * int_Bin_Tree);

- Write sum_tree : int_Bin_Tree -> int
- Adds all ints in tree

let rec sum_tree t =
type int_Bin_Tree = Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree);;

- Write sum_tree : int_Bin_Tree -> int
- Adds all ints in tree

let rec sum_tree t =
  match t with
  Leaf n -> n
  | Node(t1,t2) -> sum_tree t1 + sum_tree t2
Mapping over Recursive Types

# let rec ibtreeMap f tree =

  match tree with
  (Leaf n) -> Leaf (f n)
  | Node (left_tree, right_tree) ->
    Node (ibtreeMap f left_tree, ibtreeMap f right_tree);

val ibtreeMap : (int -> int) -> int_Bin_Tree ->
  int_Bin_Tree = <fun>
Mapping over Recursive Types

```haskell
# ibtreeMap ((+) 2) bin_tree;;

- : int_Bin_Tree = Node (Node (Leaf 5, Leaf 8), Leaf (-5))
```
# let rec ibtreeFoldRight leafFun nodeFun tree =
  match tree with Leaf n -> leafFun n |
  | Node (left_tree, right_tree) ->
  nodeFun
  (ibtreeFoldRight leafFun nodeFun left_tree)
  (ibtreeFoldRight leafFun nodeFun right_tree);

val ibtreeFoldRight : (int -> 'a) -> ('a -> 'a -> 'a) ->
  int_Bin_Tree -> 'a = <fun>
Folding over Recursive Types

# let tree_sum =
    ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum bin_tree;;
- : int = 2