Terminology: Review

- A function is in **Direct Style** when it returns its result back to the caller.
- A function is in **Continuation Passing Style** when it, and every function call in it, passes its result to another function.
- A **Tail Call** occurs when a function returns the result of another function call without any more computations (e.g., tail recursion)
- Instead of returning the result to the caller, we pass it forward to another function giving the computation after the call.

CPS Transformation

**Step 1:** Add continuation argument to any function definition:
- \( \text{let } f \arg = e \Rightarrow \text{let } f \arg k = e \)
- 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:
- \( \text{return } a \Rightarrow k \ a \)
- Assuming \( a \) is a constant or variable.
- “Simple” = “No available function calls.”

**Step 3:** Pass the current continuation to every function call in tail position
- \( \text{return } f \arg \Rightarrow f \arg k \)
- The function “isn’t going to return,” so we need to tell it where to put the result.

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

Example

**Before:**

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

**After:**

\[
\text{let rec add_listk lst k =}
\text{match lst with}
\text{[ ] -> 0 (* rule 1 *)}
\text{| 0 :: xs -> add_listk xs k (* rule 2 *)}
\text{| x :: xs -> add_listk xs (fun r -> k ((+) x r)) ; ; (* rule 4 *)}
\]
Example

Before:
let rec mem (y, lst) =
match lst with
| [] -> false
| x :: xs ->
  if (x = y)
    then true
    else mem(y, xs);;

After:
let rec memk (y, lst) k =
(* rule 1 *)
match lst with
| [] -> false
| x :: xs ->
  eqk (x, y)
  (fun b ->
    if b (* rule 4 *)
    then k true (* rule 2 *)
    else memk (y, xs) k (* rule 3 *)
  )
  memk (y, xs) k (* rule 3 *)

Example

Before:
let rec mem (y, lst) =
match lst with
| [] -> false
| x :: xs ->
  if (x = y)
    then true
    else mem(y, xs);;

After:
let rec memk (y, lst) k =
(* rule 1 *)
match lst with
| [] -> false
| x :: xs ->
  eqk (x, y)
  (fun b ->
    if b (* rule 4 *)
    then k true (* rule 2 *)
    else memk (y, xs) k (* rule 3 *)
  )
  memk (y, xs) k (* rule 3 *)

Example

Before:
let rec mem (y, lst) =
match lst with
| [] -> false
| x :: xs ->
  if (x = y)
    then true
    else mem(y, xs);;

After:
let rec memk (y, lst) k =
(* rule 1 *)
match lst with
| [] -> false
| x :: xs ->
  eqk (x, y)
  (fun b ->
    if b (* rule 4 *)
    then k true (* rule 2 *)
    else memk (y, xs) k (* rule 3 *)
  )
  memk (y, xs) k (* rule 3 *)

Example

Before:
let rec mem (y, lst) =
match lst with
| [] -> false
| x :: xs ->
  if (x = y)
    then true
    else mem(y, xs);;

After:
let rec memk (y, lst) k =
(* rule 1 *)
match lst with
| [] -> false
| x :: xs ->
  eqk (x, y)
  (fun b ->
    if b (* rule 4 *)
    then k true (* rule 2 *)
    else memk (y, xs) k (* rule 3 *)
  )
  memk (y, xs) k (* rule 3 *)

Example

Before:
let rec mem (y, lst) =
match lst with
| [] -> false
| x :: xs ->
  if (x = y)
    then true
    else mem(y, xs);;

After:
let rec memk (y, lst) k =
(* rule 1 *)
match lst with
| [] -> false
| x :: xs ->
  eqk (x, y)
  (fun b ->
    if b (* rule 4 *)
    then k true (* rule 2 *)
    else memk (y, xs) k (* rule 3 *)
  )
  memk (y, xs) k (* rule 3 *)
Example

Before:
let rec mem (y, lst) =
  match lst with
  | [] -> false
  | x :: xs ->
    if (x = y) then true
    else mem(y, xs);

After:
let rec memk (y, lst) k =
  match lst with
  | [] -> k false
  | x :: xs ->
    eqk (x, y)
    (fun b ->
      if b
      then k true
      else memk (y, xs) k)

Data type in Ocaml: lists
Frequently used lists in recursive program
Matched over two structural cases
- [] - the empty list
- (x :: xs) a non-empty list
Covers all possible lists
- type 'a list = [] | (::) of 'a * 'a list
  - Not quite legitimate declaration because of special syntax

Variants - Syntax (slightly simplified)
- type name = C_i [of ty_i] | . . . | C_n [of ty_n]
- Introduce a type called name
  - (fun x -> C_i x) : ty_i -> name
  - C_i is called a constructor; if the optional type argument is omitted, it is called a 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

Enumeration Types as Variants
# type weekday = Monday | Tuesday | Wednesday
  | Thursday | Friday | Saturday | Sunday;;

Functions over Enumerations
# 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

```ocaml
# 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>
```

Problem:

```ocaml
# type weekday = Monday | Tuesday | Wednesday
  | Thursday | Friday | Saturday | Sunday;;
- Write function is_weekend : weekday -> bool
  let is_weekend day =
```

Example Enumeration Types

```ocaml
# 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

```haskell
# 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

```haskell
type currency =
  Dollar of int
| Pound of int
| Euro of int
| Yen of int
```

Example Disjoint Union Type

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

Polymorphism in Variants

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

```haskell
# 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

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

Example Disjoint Union Type

- How to represent 7 as a const?
- Answer: IntConst 7
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>
# first (fun x -> x > 3) [1;3;4;2;5];;
- : int option = Some 4
# first (fun x -> x > 5) [1;3;4;2;5];;
- : int option = None
```

Functions over option

```ocaml
# 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.

```ocaml
# let hd list =    
  match list with [ ] -> None| (x::xs) -> Some x
- : int option = Some 2
# let tl list =    
  match list with [ ] -> None| (x::xs) -> 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

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

\[ \text{ty} \rightarrow \text{ty} \]

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 Functions

```
# 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
```

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

```plaintext
# 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)

- How to represent (6, 3) as an exp?
- BinOpAppExp (CommaOp, ConstExp (IntConst 6), ConstExp (IntConst 3))

Problem

```plaintext
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 =
```
Problem

\[
\text{int}_\text{Bin}\_\text{Tree} = \text{Leaf of int} \\
\text{Node of (int}_\text{Bin}\_\text{Tree} \times \text{int}_\text{Bin}\_\text{Tree});
\]

- Write \text{sum_tree} : \text{int}_\text{Bin}\_\text{Tree} \to \text{int}
- Adds all ints in tree

Let rec \text{sum_tree} t =
  match t with
    \text{Leaf n} \to n
    \text{Node(t1,t2)} \to \text{sum_tree t1} + \text{sum_tree t2}

Recursion over Recursive Data Types

# type \text{exp} = \text{VarExp of string} | \text{ConstExp of const} \\
| \text{BinOpAppExp of bin_op \times exp \times exp} \\
| \text{FunExp of string \times exp} | \text{AppExp of exp \times exp}

- How to count the number of variables in an \text{exp}?

Let rec \text{varCnt} \text{exp} =
  match \text{exp} with
    \text{VarExp x} \to 1
    \text{ConstExp c} \to 0
    \text{BinOpAppExp (b, e1, e2)} \to \text{varCnt e1} + \text{varCnt e2}
    \text{FunExp (x,e)} \to 1 + \text{varCnt e}
    \text{AppExp (e1, e2)} \to \text{varCnt e1} + \text{varCnt e2}

Your turn now

Try Problem 3 on MP5

Mapping over Recursive Types

# let rec \text{ibtreeMap} f \text{tree} =
  match \text{tree} with
    \text{Leaf n} \to \text{Leaf (f n)}
    \text{Node (left\_tree, right\_tree)} \to
      \text{ibtreeMap f left\_tree,}
      \text{ibtreeMap f right\_tree});

val \text{ibtreeMap} : (\text{int} \to \text{int}) \to \text{int}_\text{Bin}\_\text{Tree} \to \text{int}_\text{Bin}\_\text{Tree} = <\text{fun}>
Mapping over Recursive Types

# ibtreeMap ((+) 2) bin_tree;;
- : int_Bin_Tree = Node (Node (Leaf 5, Leaf 8), Leaf (-5))

Folding over Recursive Types

# 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>

600 minutes

Mutually Recursive Types

# type 'a tree = TreeLeaf of 'a
  | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree
  | More of ('a tree * 'a treeList);
val tree : 'a tree =
  TreeNode
    (More (TreeLeaf 5,
      (More (TreeNode (More (TreeLeaf 3,
        Last (TreeLeaf 2)))))
      (More (TreeLeaf 7))));
Mutually Recursive Types - Values

val tree : int tree =
TreeNode
  (More
    (TreeLeaf 5, 
     More
       (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))), Last (TreeLeaf 7))))

Mutually Recursive Functions

# let rec fringe tree =
  match tree with (TreeLeaf x) -> [x]
  | (TreeNode list) -> list_fringe list
and list_fringe tree_list =
  match tree_list with (Last tree) -> fringe tree
  | (More (tree,list)) ->
    (fringe tree) @ (list_fringe list);;

val fringe : 'a tree -> 'a list = <fun>
val list_fringe : 'a treeList -> 'a list = <fun>

Problem

# type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);
Define tree_size
Problem

# type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList);

Define tree_size
let rec tree_size t =
  match t with TreeLeaf _ -> 1
  | TreeNode ts -> treeList_size ts

and treeList_size ts =
  match ts with Last t -> tree_size t
  | More t ts' -> tree_size t + treeList_size ts'
Nested Recursive Types

# type 'a labeled_tree =
    TreeNode of ('a * 'a labeled_tree list);;

  type 'a labeled_tree = TreeNode of ('a * 'a labeled_tree list)

Nested Recursive Type Values

val ltree : int labeled_tree =
    TreeNode
        (5, [TreeNode (3, []); TreeNode (2, [TreeNode (1, []); TreeNode (7, [])]);
         TreeNode (5, [])])

Mutually Recursive Functions

# let rec flatten_tree labtree =
    match labtree with TreeNode (x, treelist)
        -> x::flatten_tree_list treelist
    and flatten_tree_list treelist =
        match treelist with [] -> []
        | labtree::labtrees
            -> flatten_tree labtree
            @ flatten_tree_list labtrees;;
Mutually Recursive Functions

val flatten_tree : 'a labeled_tree -> 'a list = <fun>
val flatten_tree_list : 'a labeled_tree list -> 'a list = <fun>
# flatten_tree ltree;;
-: int list = [5; 3; 2; 1; 7; 5]

Nested recursive types lead to mutually recursive functions

Why Data Types?

- Data types play a key role in:
  - **Data abstraction** in the design of programs
  - **Type checking** in the analysis of programs
  - **Compile-time code generation** in the translation and execution of programs
- Data layout (how many words; which are data and which are pointers) dictated by type

Terminology

- **Type**: A type \( t \) defines a set of possible data values
  - E.g. `short` in C is \( \{ x | 2^{15} - 1 \geq x \geq -2^{15} \} \)
  - A value in this set is said to have type \( t \)
- **Type system**: rules of a language assigning types to expressions

Types as Specifications

- Types describe properties
- Different type systems describe different properties, eg
  - Data is read-write versus read-only
  - Operation has authority to access data
  - Data came from “right” source
  - Operation might or could not raise an exception
- Common type systems focus on types describing same data layout and access methods

Sound Type System

- If an expression is assigned type \( t \) and it evaluates to a value \( v \), then \( v \) is in the set of values defined by \( t \)
- SML, OCAML, Scheme and Ada have sound type systems
- Most implementations of C and C++ do not

Strongly Typed Language

- When no application of an operator to arguments can lead to a run-time type error, language is **strongly typed**
  - Eg: `1 + 2.3;;`
- Depends on definition of “type error”
Strongly Typed Language

- C++ claimed to be “strongly typed”, but
- Union types allow creating a value at one type and using it at another
- Type coercions may cause unexpected (undesirable) effects
- No array bounds check (in fact, no runtime checks at all)
- SML, OCAML “strongly typed” but still must do dynamic array bounds checks, runtime type case analysis, and other checks

Static vs Dynamic Types

- **Static type**: type assigned to an expression at compile time
- **Dynamic type**: type assigned to a storage location at run time
- **Singly typed language**: static type assigned to every expression at compile time
- **Dynamically typed language**: type of an expression determined at run time

Type Checking

- When is op(arg1,…,argn) allowed?
- **Type checking** assures that operations are applied to the right number of arguments of the right types
  - Right type may mean same type as was specified, or may mean that there is a predefined implicit coercion that will be applied
  - Used to resolve overloaded operations

Dynamic Type Checking

- Performed at run-time before each operation is applied
- Types of variables and operations left unspecified until run-time
  - Same variable may be used at different types

Data object must contain type information

- Errors aren’t detected until violating application is executed (maybe years after the code was written)
Static Type Checking

- Performed after parsing, before code generation
- Type of every variable and signature of every operator must be known at compile time

Static Type Checking

- Can eliminate need to store type information in data object if no dynamic type checking is needed
- Catches many programming errors at earliest point
- Can’t check types that depend on dynamically computed values
  - Eg: array bounds

Static Type Checking

- Typically places restrictions on languages
  - Garbage collection
  - References instead of pointers
  - All variables initialized when created
  - Variable only used at one type
    - Union types allow for work-arounds, but effectively introduce dynamic type checks