

Functional Programming and ML

[part 4]

In part based on slides from Gerardo Schneider, which where
in turn based on John C. Mitchell's

Abstraction

1. Hide details (encapsulation)
 2. Tame complexity
 3. Allow for code reuse
 4. Communicate relative importance of components
- etc

Language support for abstraction can be broken down into categories:

- Control flow abstraction
 - `if-then-else` as opposed to `goto`
 - exceptions
 - continuations
 - evaluation order
- Data abstraction
 - data types, abstract data types, modules
- Syntactic abstraction
 - macro systems and meta-programming features

Control flow abstraction

1968 Go To Statement Considered Harmful (Dijkstra)

```
if B then S else S` end
```

```
addr0 cmpl $0x0, B  
addr1 je    addr4  
addr2 S  
addr3 jmp   addr5  
addr4 S'  
addr5
```

Similar for:

```
while B do S end  
for B do S end
```

Exceptions

Terminate part of computation

- Jump out of a construct (as opposed to into another)
 - “Wait... but aren't jumps bad?”
- Pass data as part of jump
- Return to most recent site set up to handle exception

Memory management needed

- Unnecessary activation records need to be deallocated

Two main language constructs

- raise exception: `throw` Java, `raise` Python
- handle exception: `catch` Java, `try...except` Python

Exceptions in ML

Exceptions do not affect a function's type signature

```
- fun f () = 1;  
- fun g () = if false then raise Div else 1;  
- fun h () = if true then raise Div else 1;
```

Exceptions must be declared before use

Exceptions are dynamically scoped

- Control jumps to the handler most recently established (run-time stack)
- ML is otherwise statically scoped

Pattern matching to determine the appropriate handler
(C++/Java uses type matching)

Example in ML

```
- exception outOfBounds;  
exception outOfBounds  
- fun nth (n, nil) = raise outOfBounds  
  | nth (0, h::t) = h  
  | nth (n, h::t) = nth (n-1, t);  
val nth = fn : int * 'a list -> 'a
```

```
- val lst = ["bob", "stuart", "kevin"];  
val lst = ["bob", "stuart", "kevin"] : string list  
- nth(0, lst);  
val it = "bob" : string  
- nth(3, lst);  
uncaught exception outOfBounds
```

```
- fun safeNth(n,xs) = nth(n,xs)
    handle X => "minion";
val safeNth = fn : int * string list -> string
- safeNth(3, lst);
val it = "minion" : string
```

Dynamic scoping of handlers

Exception propagates up the call stack

Who handles the exception?

- depends on runtime information

Motivation:

- Users know better how to handle errors
- Author of library function does not

Example: Dynamic scoping of handlers

```
- exception X;  
- (let fun f(y) = raise X  
    and g(h) = h(1) handle X => 2  
    in  
    g(f) handle X => 4  
  end) handle X => 6;
```

Question. What is the value of `g(f)`?

Question. Which handler will be used? The `=>2`, `4`, or `6`?

Question. What will be the final value when executing the outer let-expression?

```
- val x = 6;  
- (let fun f(y) = x  
      and g(h) = (let val x=2 in h(1) end)  
  in  
    (let val x=4 in g(f) end)  
  end);
```

Handlers with pattern matching

```
- exception Signal of int;  
- fun f(x) = if x=0 then raise Signal(0)  
             else if x=1 then raise Signal(1)  
             else if x=10 then raise Signal(x-8)  
             else (x-2) mod 4;
```

```
f(10) handle Signal(0) => 0  
          | Signal(1) => 1  
          | Signal(x) => x;
```

Exceptions and resource allocation

```
- exception X;  
- (let val x = ref [1,2] in      (* [1,2] heap, x stack *)  
    let val y = ref [3,4] in    (* [3,4] heap, y stack *)  
        ... raise X  (* Control to outside scope *)  
    end  
end) (* x, y off stack  
      [1,2] [3,4] garbage collected *)  
handle X => ...;
```

In Java, use the `finally` construct to dealloc resources

Typing of exceptions

Recall that:

$$\frac{e \rightsquigarrow v \quad v : T}{e : T}$$

What happens when exceptions occur along the way?

Question. What is the type of the expression below?

```
- 42 + (1 div 0 handle X => 1);
```

Question. What is the type of the expression below?

```
- 42 + ((raise Div) handle X => 1);
```

Note: `handle X` converts exception to normal termination

In general,

- the type of `raise e` is a type variable 'a
- the type of `handle e => e2` is the type of `e2`

Question. What must be the types of `e1` and `e2`? Why?

```
1 + (e1 handle X => e2)
```

Exception for efficiency

Not just for error conditions

Consider the following definition of tree

```
datatype tree =  
  leaf of int  
  | node of tree * tree;
```

```
leaf 10;  
node (leaf 10, leaf 2);  
node ((node (leaf 10, leaf 2)), leaf 5);
```

Function to multiply values of tree leaves:

```
- fun prod(leaf x)      = x: int
  | prod(node(x,y)) = prod(x) * prod(y);

- prod( node ((node (leaf 10, leaf 2)), leaf 5) );
```

Optimized using exception:

```
- fun prodExep(tree) =
  let exception Zero
    fun p(leaf x) = if x=0 then (raise Zero) else x
      | p(node (x,y)) = p(x) * p(y)
  in
    p(tree) handle X => 0
  end;
```

Even though you can use exceptions for efficiency,
it doesn't mean you should.

Better to use continuations!

Continuations

“A continuation is an abstract representation of the control state of a computer program.”

“The continuation is a data structure that represents the computational process at a given point in the process’s execution;

the created data structure can be accessed by the programming language, instead of being hidden in the runtime environment.”

<https://en.wikipedia.org/wiki/Continuation>

“A continuation is *something which waits for a value* in order to perform some calculations with it.

With every intermediate value in a computation, there is a continuation associated, which represents the future of the computation once that value is known.

A continuation is not something, like a function, which takes a value and returns another: it just takes a value and does everything that follows to it, and never returns.”

<http://www.madore.org/~david/computers/callcc.html>

$$5 * 3 + 2$$

Continuations in practice

Scheme (derived-from/dialect-of Lisp): first to implement first-class continuations

Continuations in ML

Cont module inside SMLofNJ module.

```
- open SMLofNJ.Cont;
```

```
- 3 + callcc (fn k => 2 + 1);
```

```
- 3 + callcc (fn k => 2 + throw k 1);
```

```
- callcc;
```

```
val it = fn : ('a cont -> 'a) -> 'a
```

Continuations

“One can think of a first-class continuation as saving the execution state of the program.”

“It is important to note that true first-class continuations do not save program data – unlike a process image.”

<https://en.wikipedia.org/wiki/Continuation>

Revisiting prod

```
- fun prodExep(tree) =  
  let exception Zero  
    fun p(leaf x) = if x=0 then (raise Zero) else x  
      | p(node (x,y)) = p(x) * p(y)  
  in  
    p(tree) handle X => 0  
  end;
```

```
- fun prodCC(tree) =  
  callcc (fn k =>  
    let fun p(leaf x) = if x=0 then (throw k 0) else x  
      | p(node (x,y)) = p(x) * p(y)  
    in p(tree) end);
```

Continuation-Passing Style (CPS)

Functions don't return; they send their result to the next

```
- fun plus x y = x + y;  
val plus = fn : int -> int -> int
```

```
- plus 2 5;  
val it = 7 : int
```

Continuation-Passing Style (CPS)

Functions don't return; they send their result to the next

```
- fun plus0t x y k      = let val v = x + y
                          in v end;
val plus0t = fn : int -> int -> 'a -> int
```

```
- plus0t 2 5 ();
val it = 7 : int
```

Continuation-Passing Style (CPS)

Functions don't return; they send their result to the next

```
- fun pluscc x y k : int = let val v = x + y
                           in throw k v end;
val pluscc = fn : int -> int -> int cont -> int
```

```
- callcc( pluscc 2 5 );
val it = 7 : int
```

Continuation-Passing Style (CPS)

Functions don't return; they send their result to the next

```
- fun pluscc x y k : int = let val v = x + y
                           in throw k v end;
val pluscc = fn : int -> int -> int cont -> int
```

```
- callcc( pluscc 2 5 );
val it = 7 : int
```

Question. How does the expression reduce to 7?

Continuation-Passing Style (CPS)

Programs can be automatically transformed CPS

Functional and logic compilers often use CPS as an intermediate representation (IR)

- Compilers for imperative langs often use static single assignment form (SSA)
- SSA is equivalent to a subset of CPS

Continuations, conclusion

Continuations are a powerful construct;
they can be used to implement other control mechanisms
such as exceptions, generators, coroutines, and so on.

Continuations have been used in practice;
for example, to implement web servers.

However, continuation are often not well understood;
they can add complexity;
some call it “the go-to of functional programming.”

Evaluation order

Eager/strict evaluation: Arguments are evaluated before function is called

Call-by-value
Applicative-Order Evaluation
Ex: C, ML, etc

Non eager: Arguments are not evaluated unless they are used during the evaluation of the function body

Call-by-name	Call-by-need
Normal order reduction	Lazy evaluation
Ex: ALGOL 60	Ex: Haskell

Example: $\text{sq}(3+4)$

Eager (3 steps)

$\text{sq}(3+4) \leadsto \text{sq}(7) \leadsto 7*7 \leadsto 49$

Lazy (4 steps)

$\text{sq}(3+4) \leadsto (3+4)*(3+4) \leadsto 7*(3+4) \leadsto 7*7 \leadsto 49$

Example: `fst(sq(4), sq(2))`

Example: `fst(sq(4), sq(2))`

Eager (5 steps)

```
fst(sq(4), sq(2))  
  ~> fst(4*4, sq(2)) ~> fst(16, sq(2))  
  ~> fst(16, 2*2) ~> fst(16, 4) ~> 16
```

Lazy (3 steps)

```
fst(sq(4), sq(2))  
  ~> sq(4) ~> 4*4 ~> 16
```

Example: `fst(sq(4), diverge)`

Question. Why not always use “call-by-need”?

Question. Why not always use “call-by-need”?

From a language design perspective, lazy evaluation is very hard to get right in the presence of side effects.

- Haskell and monads (a more advanced course)

From a programmer perspective, monads can be hard to understand.

Delay

Question. How to delay the evaluation of an expression?

```
- 1+1;
```

Delay

Question. How to delay the evaluation of an expression?

```
- 1+1;
```

```
- val e = fn () => 1+1;  
val e = fn : unit -> int  
- e();  
val it = 2 : int
```

Delay

Question. How to delay the evaluation of an expression?

```
- 1+1;
```

```
- val e = fn () => 1+1;  
val e = fn : unit -> int  
- e();  
val it = 2 : int
```

Question. How about delaying arbitrary expression?

Delay

```
- fun delay e = fn () => e;  
val delay = fn : 'a -> unit -> 'a  
- val e = delay (1+1);
```

Question. Does `delay` above work? Does it delay the evaluation of `e`?

Summary: Abstraction at language level

- Control flow abstraction
 - `if-then-else` as opposed to `goto`
 - functions
 - exceptions
 - continuations
 - evaluation order
- Data abstraction
 - data types, abstract data types, modules
- Syntactic abstraction
 - macro systems and meta-programming features.

Summary: ML

Covered

- Basic ML constructs
- Recursion, tail recursion
- Higher order functions
- Modules (data abstraction)
- Polymorphism, type system, type inference
- Exceptions, continuation (control flow abstraction)

Not covered

- Input/Output
- Files
- Network programming
- Concurrency (see *Concurrent ML*) ...