Control in Sequential Languages

Mitchell Chapter 8

Topics

- Exceptions
  - "structured" jumps that may return a value
  - dynamic scoping of exception handler
- Continuations
  - Function representing the rest of the program
  - Generalized form of tail recursion
- Control of evaluation order (force and delay)
  - Can increase efficiency
  - *Call-by-need* parameter passing.

Exceptions: Structured Exit

- Historically, `goto` statements were used, which can jump out of anywhere or into anywhere
- Some languages have `break` statements
- Exceptions provide a *clean* way to jump out of or abort a function call.

  – Their effects would not be easy to achieve with other forms of controlled jumps.

- Main language constructs:
  – Statement or expression to *raise* or *throw* exception
  – Statement or expression to *handle* or *catch* exceptions, called a *handler*

Exceptions: Structured Exit

Terminate part of computation, achieving the following effects:

- Jump out of construct
- Pass data as part of jump
  - This data can be used, for example, to recover from an error.

- Return to most recent site set up to handle exception

  - The correct handler is determined according to dynamic scoping rules

- Unnecessary activation records may be deallocated
  - May need to free heap space, other resources

C++ vs ML Exceptions

- C++ exceptions
  - Can throw any type
  - Stroustrup: "I prefer to define types with no other purpose than exception handling. This minimizes confusion about their purpose. In particular, I never use a built-in type, such as int, as an exception." -
    - The C++ Programming Language, 3rd ed.

- ML exceptions
  - Exceptions are a different kind of entity than types.
  - Declare exceptions before use

Similar, but ML requires the recommended C++ style.

OCaml Exceptions

- Declaration

  exception ⟨name⟩ of ⟨type⟩

  – gives name of exception and type of data passed when raised

- Raise

  raise (⟨name⟩ ⟨parameters⟩)

  – expression form to raise an exception and pass data

- Handler

  try ⟨exp1⟩ with | ⟨pattern⟩ -> ⟨exp2⟩

  – Evaluate first expression.

  – If exception that matches pattern is raised, then evaluate second expression instead.

  – General form allows multiple patterns.

Examples

```
exception Ovflw.
exception Signal of int
```

```
raise Ovflw
raise (Signal (x+4))
```

```
let f x = if x<min then raise Ovflw else 1/x
(try f x with | Ovflw -> 0) / (try f 0 with | Ovflw -> 1)
```



```
let g 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
try g 10 with | Signal 0 -> 0
              | Signal 1 -> 1
              | Signal x -> x+8
```

Which Handler is Used?

```
let f x = if x<min then raise Ovflw else 1/x
(try f x with | Ovflw -> 0) / (try f 0 with | Ovflw -> 1)
```

- Dynamic scoping of handlers
  - First call handles exception one way
  - Second call handles exception another
  - General dynamic scoping rule

    Jump to most recently established handler on run-time stack
- Dynamic scoping is not an accident
  - User knows how to handler error
  - Author of library function does not

General Form of Handler Expressions

try <exp> with

| <pattern$_1$> -> <exp$_1$>

| <pattern$_2$> -> <exp$_2$>

…

| <patern$_n$> -> <exp$_n$>

- First, <exp> is evaluated.
- If the evaluation terminates normally, the value of the whole try expression is the value of this expression; the handler is never invoked.
- If the evaluation raises an exception that matches <pattern$_i$> (and there is no matching handler declared in <exp>), then the corresponding handler is invoked.
- Pattern matching works just as in ordinary OCaml.

Exception for Error Condition

```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
exception No_Subtree
let lsub (t:'a tree) =
  match t with
  | Leaf x -> raise No_Subtree
  | Node (x,y) -> x
```


- This function raises an exception when there is no reasonable value to return
- We’ll look at typing later.

Exception for Efficiency

- Function to multiply values of tree leaves

```
let rec prod (t:int tree) : int =
  match t with
  | Leaf x -> x
  | Node (x,y) -> (prod x) * (prod y)
```


- Optimize using exception


```
let exception Zero in
  let rec prod (t:int tree) : int =
    match t with
    | Leaf x -> if x=0 then (raise Zero) else x
    | Node (x,y) -> (prod x) * (prod y)
  in
    try (prod t) with Zero -> 0
```

Dynamic Scope of Handler

```
exception X
try (let f y = raise X in
     let g h = try h 1 with X -> 2
     in
     try g f with X -> 4)
with X -> 6
```

scope

handler

Which handler is used?

Dynamic Scope of Handler

```
exception X
try (let f y = raise X in
    let g h = try (h 1) with X -> 2
    in
    try (g f) with X -> 4)
with X -> 6
```

- When a handler is in a nested block, the handler expression goes on the stack first and is treated like a declaration.
- A handler in a function definition is treated like a local variable.

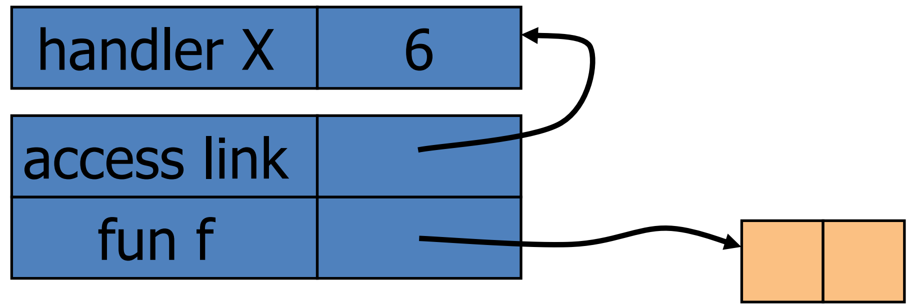
Dynamic Scope of Handler

```
exception X
try (let f y = raise X in
     let g h = try (h 1) with X -> 2
     in
     try (g f) with X -> 4)
with X -> 6
```

handler X	6

Dynamic Scope of Handler

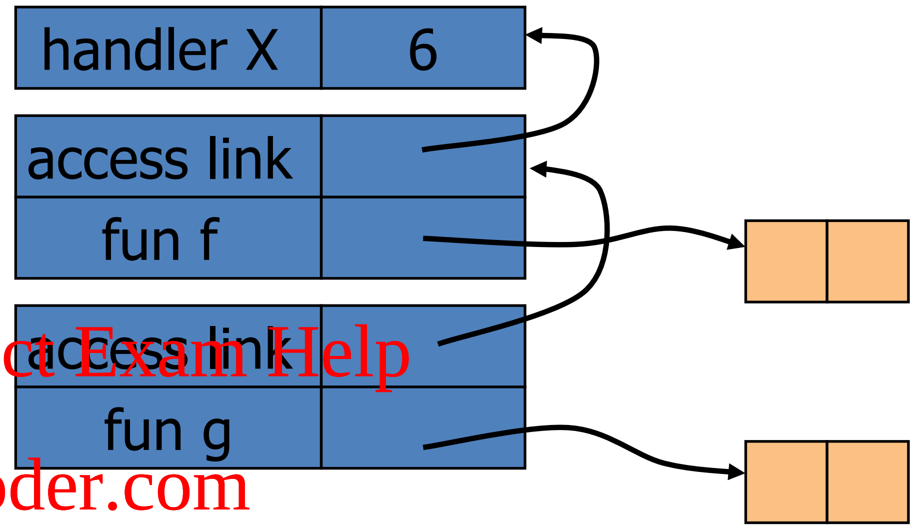
```
exception X
try (let f y = raise X in
    let g h = try (h 1) with X -> 2
    in
    try (g f) with X -> 4)
with X -> 6
```



Note: pointers in closures left out of diagram, but can be deduced.

Dynamic Scope of Handler

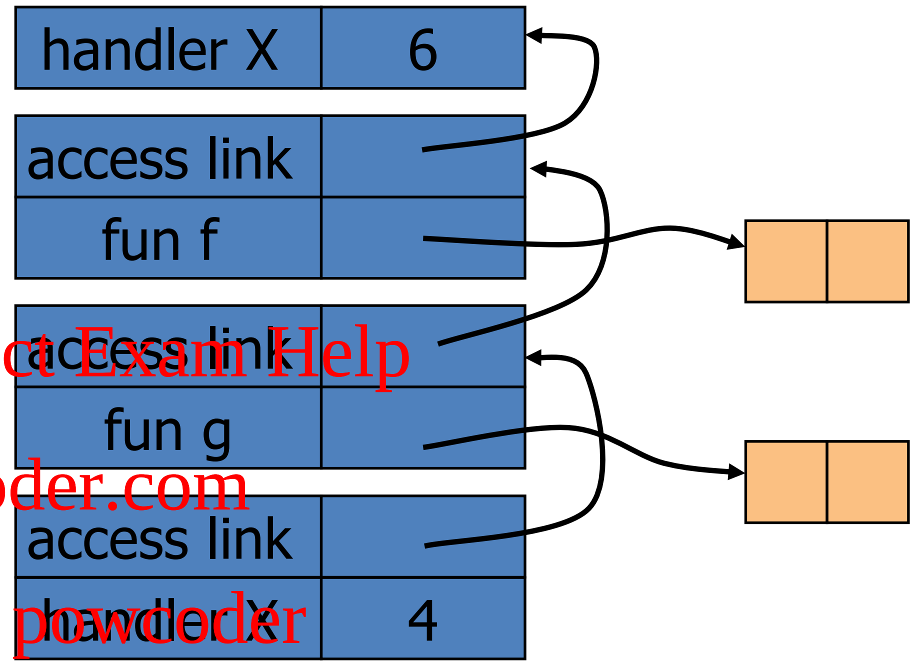
```
exception X
try (let f y = raise X in
    let g h = try (h 1) with X -> 2
    in
    try (g f) with X -> 4)
with X -> 6
```



Note: pointers in closures left out of diagram, but can be deduced.

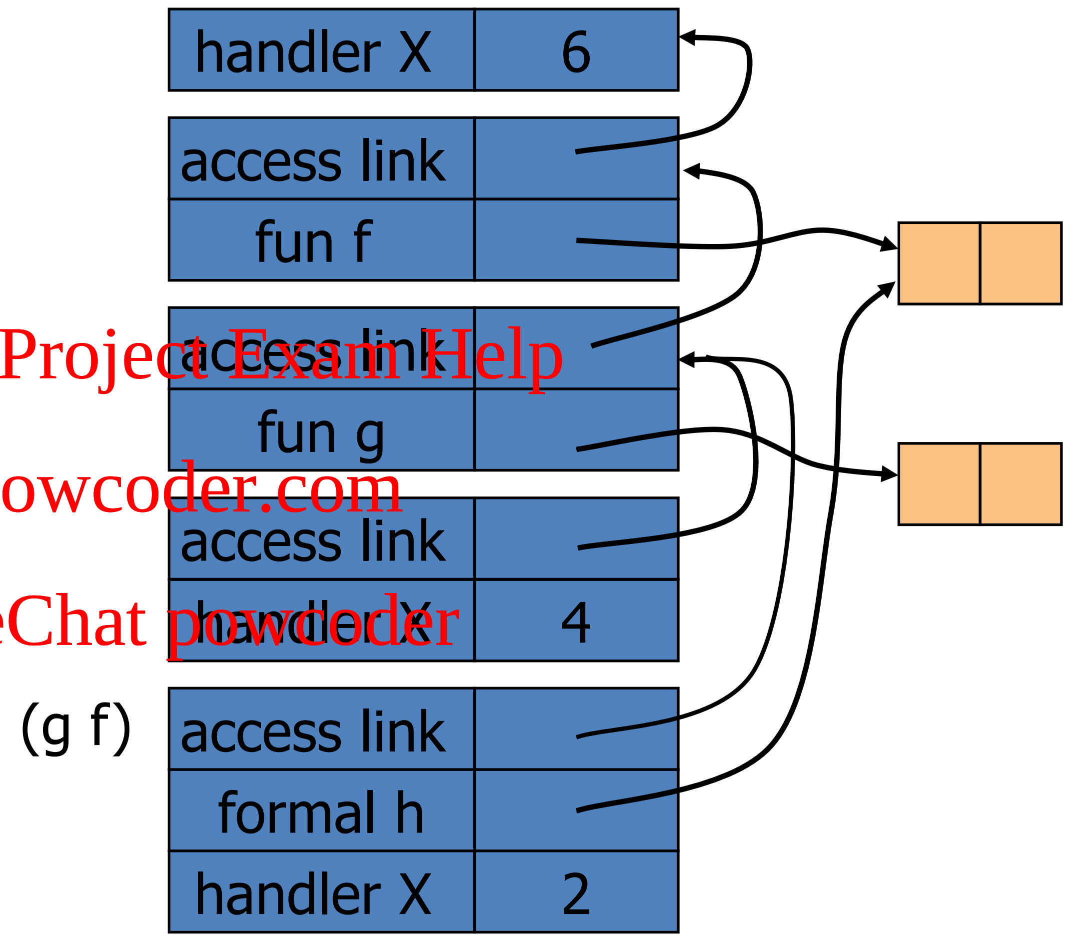
Dynamic Scope of Handler

```
exception X
try (let f y = raise X in
    let g h = try (h 1) with X -> 2
    in
    try (g f) with X -> 4)
with X -> 6
```



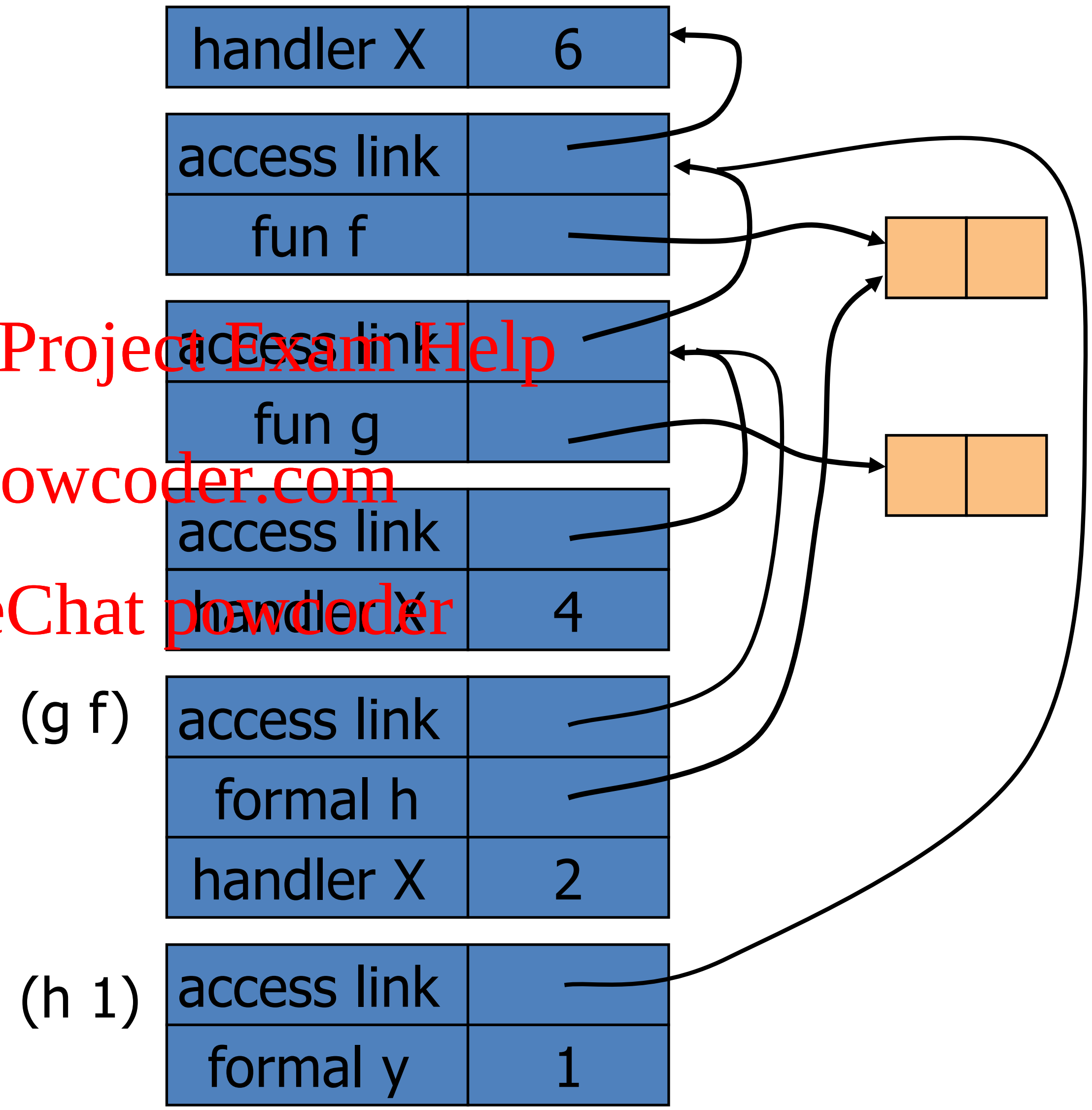
Dynamic Scope of Handler

```
exception X
try (let f y = raise X in
    let g h = try (h 1) with X -> 2
    in
    try (g f) with X -> 4)
with X -> 6
```



Dynamic Scope of Handler

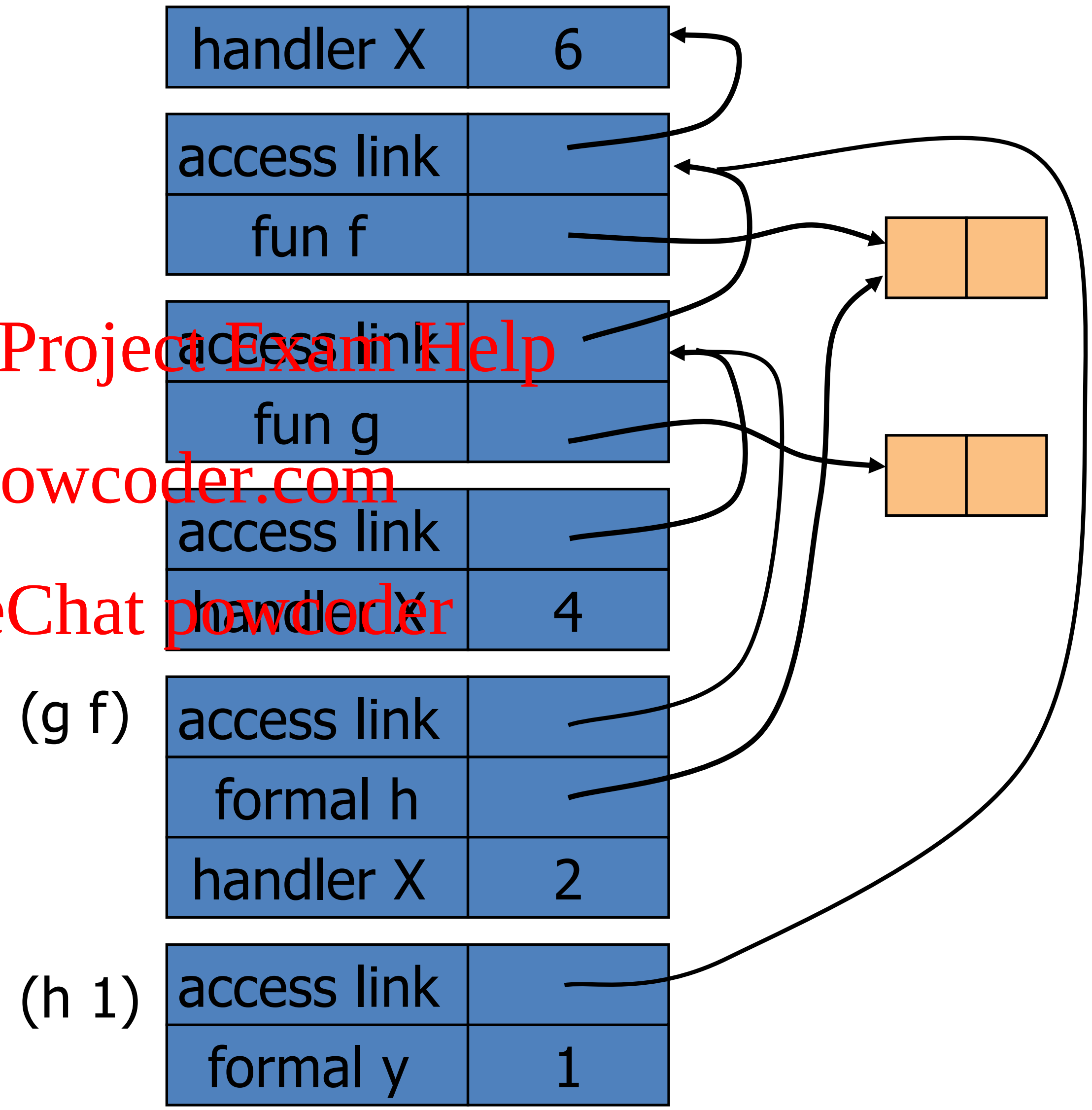
```
exception X
try (let f y = raise X in
     let g h = try (h 1) with X -> 2
     in
     try (g f) with X -> 4)
with X -> 6
```



Dynamic Scope of Handler

```
exception X
try (let f y = raise X in
     let g h = try (h 1) with X -> 2
     in
     try (g f) with X -> 4)
with X -> 6
```


- Dynamic scope: find first X handler, going up the dynamic call chain at the point raise X is executed.
- Result is 2.
- After the handler returns 2, the computation is done, so all activation blocks are popped.



Comparison to Static Scope of Variables

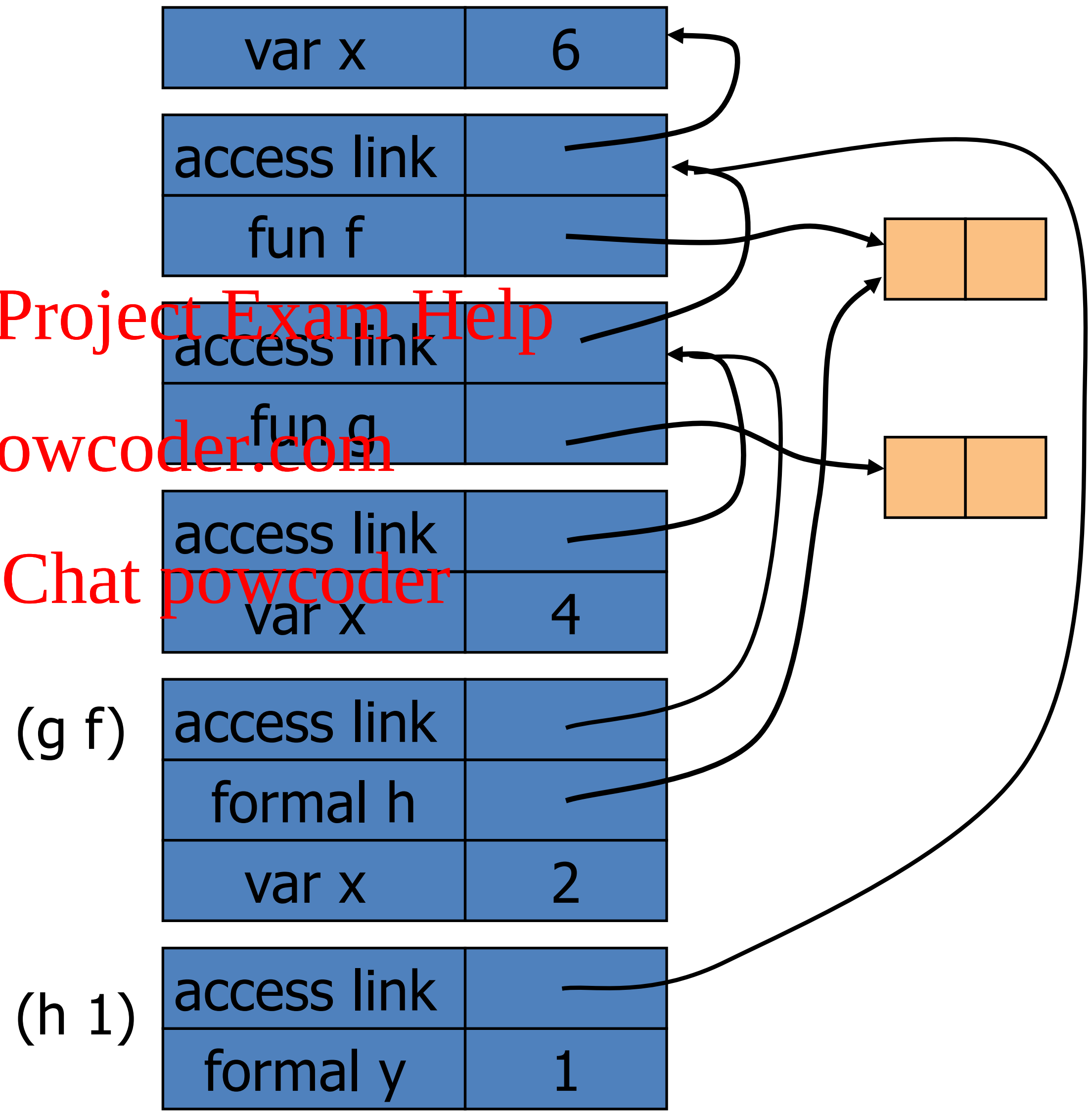
```
exception X
try let f y = raise X in
    let g h = try h 1 with X -> 2
    in
    try g f with X -> 4
with X -> 6
```

```
let x = 6 in
let f y = x in
let g h = let x = 2 in
          h 1
in
let x = 4 in
g f
```

Static Scope of Declarations

```
let x = 6 in
let f y = x in
let g h = let x = 2 in
          h 1
in
let x = 4 in
g f
```

Static scope: find first x, following access links from the reference to x.



Typing of Exceptions

- Typing of raise ⟨exn⟩
  - Recall definition of typing
    - Expression e has type t if (normal termination of) e produces value of type t
  - Raising exception is not normal termination

    - Example: 1 + raise X
- Typing of with | ⟨exn⟩ -> ⟨value⟩

  - Converts exception to normal termination

  - Need type agreement
  - Examples
    - 1 + (try raise X with X -> e) Type of e must be int
    - 1 + (try e_1 with X -> e_2) Type of e_1, e_2 must be int

Exceptions and Resource Allocation

```
exception X
try
  (let x = ref [1,2,3]
   in
   let y = ref [4,5,6]
   in
     … raise X
   ) with X -> …
```

- Resources may be allocated between handler and raise
- May be "garbage" after exception
- Examples
  - Memory
  - Lock on database
  - Threads
  - …

General problem: no obvious solution

Comparison: ML Example

- Exception used to handle a condition that makes it impossible to continue the computation

```
exception Determinant;  (* declare exception name *)
let invert M =          (* function to invert matrix *)
      …
      if …
          then raise Determinant    (* exit if Det=0 *)
          else …
…
in
try invert myMatrix with | Determinant -> …
```


Value for expression if determinant of myMatrix is 0

Comparison: C++ Example

```
Matrix invert(Matrix m) {
    if … throw Determinant;
    …
};

try { … invert(myMatrix); …
}
catch (Determinant) { …
    // recover from error
}
```

- Note:
  - raise instead of throw
  - catch instead of with
  - try as in ML
- A more significant difference:
  - exceptions are types

Continuations

- The main idea:
  - Stop execution, and then later continue

- More precisely:

  - The continuation of an expression in a program is the remaining actions to perform *after* evaluating the expression

- Important:
  - does not depend on the expression, only the program that contains it.

Continuations

- Idea:
  - The continuation of an expression is “the remaining work to be done after evaluating the expression”
  - Continuation of *e* is a function applied to *e*
- General programming technique

  - Capture the continuation at some point in a program

  - Use it later: “jump” or “exit” by function call
  - A continuation with only a unit argument is like a simple jump.

  - A continuation with arguments is like a jump or exit with data.
- Useful in
  - Compiler optimization: make control flow explicit
  - Operating system scheduling, multiprogramming
  - Web site design

Example of Continuation Concept

- Expression
  - 2*x + 3*y + 1/x + 2/y
- What is continuation of 1/x?
  - Remaining computation after division:


```
let before = 2*x + 3*y in
let continue d = before + d + 2/y
in
continue (1/x)
```


  - before is not essential, alternative is:

```
let continue d = 2*x + 3*y + d + 2/y
in
continue (1/x)
```

Example: Error-Avoiding Division using Continuations

```
let divide (numer:float) (denom:float)
           (normal_cont: float -> float)
           (error_cont: unit -> float) : float =
  if denom > 0.0001
  then normal_cont (numer /. denom)
  else error_cont ()

let f (x:float) (y:float) : float =
  let before = 2.0 *. x +. 3.0 *. y in
  let continue (quotient: float) =
     before +. quotient +. 2.0 /. y in
  let error_continue () = before /. 5.2 in
  divide 1.0 x continue error_continue
```

Example: Error-Avoiding Division using Exceptions

```
exception Div
let f (x:float) (y:float) : float =
  try (2.0 *. x +. 3.0 *. y +.
       1.0 /. (if x > 0.0001
               then x
               else raise Div) +.
       2.0 /. y)
  with Div ->
    (2.0 *. x +. 3.0 *. y) /. 5.2
```


- Same behaviour, simpler with exceptions
- In general, continuations are more flexible than exceptions, but may require more programming effort.

Continuation-Passing Form and Tail Recursion

- *continuation-passing form* (CPS): each function or operation is passed a continuation
  - Functions terminate by calling a continuation
  - Thus, no function needs to return to the point from where it was called.
  - Like tail calls…
  - There are systematic rules for transforming an expression or program to CPS.

Example: Tail Recursive Factorial

- Standard recursive function

  fact n = if n=0 then 1 else n*(fact (n-1))

- Tail recursive

  f n k = if n=0 then k else f (n-1) (n*k)

  fact n = f n 1

- How could we derive this?
  - Transform to continuation-passing form
  - Optimize continuation functions to single integer

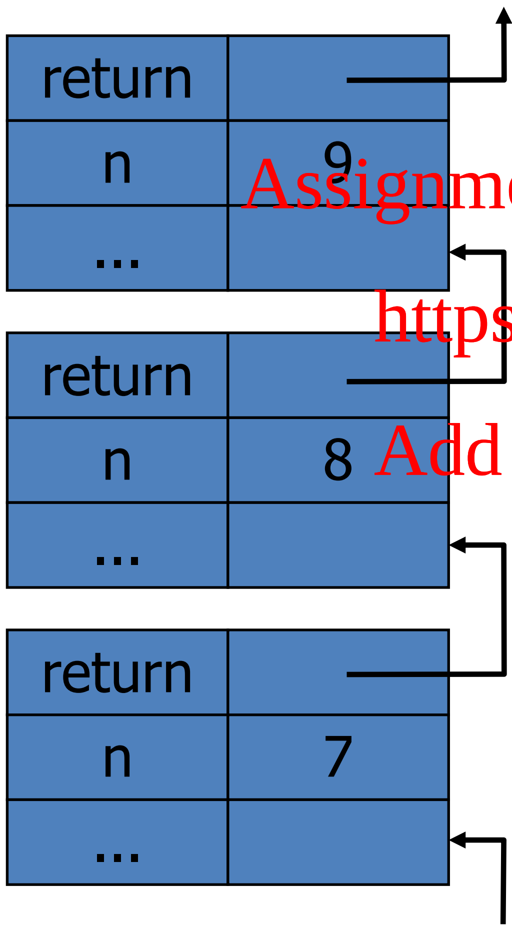
Continuation View of Factorial

fact n = if n=0 then 1 else n* (fact (n-1))

(fact 9)

(fact 8)

(fact 7)



- This invocation multiplies by 9 and returns
- Continuation of (fact 8) is $\lambda x.\ 9*x$

- Multiplies by 8 and returns
- Continuation of (fact 7) is

  $\lambda y.\ (\lambda x.\ 9*x)\ (8*y)$

- Multiplies by 7 and returns
- Continuation of (fact 6) is

  $\lambda z.\ (\lambda y.\ (\lambda x.\ 9*x)\ (8*y))\ (7*z)$

Derivation of Tail Recursive Form

- Standard function

  fact n = if n=0 then 1 else n*(fact (n-1))

- Continuation form

  fact n k = if n=0 then (k 1)

  else (fact (n-1)) (λx.k (n*x))

  fact n (λx.x) computes n!

Computation to do after calculating fact(n)

Computation to do after calculating fact(n-1)

Derivation of Tail Recursive Form

- Standard function

  fact n = if n=0 then 1 else n*(fact (n-1))

- Continuation form

  fact n k = if n=0 then (k 1)

  else (fact (n-1)) (λx.k (n*x))

  fact n (λx.x) computes n!

- Example computation

  fact 3 (λx.x) = fact 2 (λy.((λx.x) (3*y)))

  = fact 1 (λx.((λy.3*y)(2*x)))

  = fact 0 (λy.((λx.3*(2*x))(1*y)))

  = λy.(3*(2*(1*y))) 1 = 6

Derivation of Tail Recursive Form

- Continuation-passing form

  fact n k = if n=0 then k 1 else fact (n-1) (λx.k (n*x))

- Tail Recursive Form as Optimization of CPS

  fact n a = if n=0 then a else fact (n-1) (n*a)

  Each continuation is effectively λx.(a*x) for some a

- Example computation

  fact 3 1 = fact 2 3 was fact 2 (λy.3*y)

  = fact 1 6 was fact 1 (λx.6*x)

  = fact 0 6 = 6

Summary and Other Uses for Continuations

- Derivation of Tail Recursive Form (Optimization)
- Explicit Control
  - Normal termination -- call continuation
  - Abnormal termination -- do something else

- Compilation Techniques

  - Call to continuation is functional form of `goto`
  - Continuation-passing style makes control flow explicit

- Web Applications and Services (next page)

Web Applications and Services

- Web Applications, Web Services, Message-Oriented Middleware (MOM) and Service-Oriented Architecture (SOA) services
  - Handle long running workflows
  - Workflow may take 1 year to complete
  - Progress of subtasks is asynchronous
- Sequential programming is simpler than asynchronous
- Continuations provide
  - An easy way to suspend workflow execution at a wait state
  - Thread of control can be resumed when the next message/event occurs, maybe some long time ahead

Continuations supported in some versions of Java JVM

Control of Evaluation Order (Force and Delay)

Example: controlling the order for efficiency

let f x y = … x … y … in

f e_1 e_2

- Suppose the value of y is needed only if the value of x has some property.
- Suppose the evaluation of e_2 is expensive.
- We would like:

let f x y = … x … Force y … in

f e_1 (Delay e_2)

- where Delay e_2 causes the evaluation of e to be delayed until we call Force (Delay e_2)

Control of Evaluation Order (Force and Delay)

- Delay and Force are explicit program constructs in Scheme
- They can be "programmed" in ML.
- Delay e is an abbreviation for (fun () -> e)
  - Example: Delay (3+4) is (fun () -> 3+4)

- Force e is an abbreviation for e()

  - Force (Delay (3+4)) is ((fun () -> 3+4) ()) = 7

Example

```
let time_consuming (n:int) =
  let rec tak x y z =
    if x <= y then y
    else tak (tak (x-1) y z) (tak (y-1) z x) (tak (z-1) x y) in
  tak (3*n) (2*n) n
let rec fib (n:int) =
  if n=0 || n=1 then 1 else fib (n-1) + fib (n-2)
let odd (n:int) = (n mod 2) = 1
let f (x:int) (y:int) = if (odd x) then 1 else (fib y)
in
f (fib 9) (time_consuming 9)
```


- tak runs for a very long time (and is used by time_consuming)
- Function f has 2 arguments and the second is used only if the first is not odd.

Example (Continued)

```
let f (x:int) (y:int) = if (odd x) then 1 else (fib y)
in
f (fib 9) (time_consuming 9)
```

- f (fib 9) (time_consuming 9) runs for a very long time
- A version that uses Delay and Force to only evaluate the second argument if needed.


```
let lazy_f (x:int) (y:unit -> int) =
  if odd x then 1 else fib (y())
in
lazy_f (fib 9) (fun () -> time_consuming 9)
```

- Because (fib 9) is odd, this expression terminates much more quickly than the one without Delay

Using a Delayed Value More than Once

- The version of Delay and Force described so far:
  - Requires static scoping
  - Saves time only if the delayed argument is used at most once.
- A version that works when the delayed argument is used more than once can also be programmed in ML.

- Main idea: store a flag that indicates whether the expression has been evaluated once or not.
  - If not, then evaluate when needed and store the result.
  - If so, retrieve the stored result.
  - This is *call-by-need* parameter passing.

Implementation and Example

```
type 'a delay =
  | EV of 'a
  | UN of (unit -> 'a)

let ev (d:'a delay) =
  match d with
  | EV x -> x
  | UN f -> f()

let force (d:'a delay ref) =
  let v = ev !d in
  (d := EV v; v)
```


- A delayed value is a reference cell containing an "unevaluated delay"

```
let d = ref (UN (fun () -> fib 9)

let d =
  ref (UN (fun () ->
                     time_consuming 9))
```

- Forcing evaluation evaluates and stores

```
force d = 55
```

- After the call to force:

```
d = ref (EV 55)
```

Summary

- Exceptions
  - "structured" jumps that may return a value
  - dynamic scoping of exception handler
- Continuations
  - Function representing the rest of the program
  - Generalized form of tail recursion
  - Used in Lisp and ML compilation, some OS projects, web application development, …
- Delay and Force
  - For controlling evaluation order
  - Can be used to (greatly) improve efficiency
  - Can be used to implement call-by-need parameter passing