Exceptions

Vitaly Shmatikov

Reading Assignment

- Mitchell, Chapter 8.2

Exceptions: Structured Exit

- Terminate part of computation
  - Jump out of construct
  - Pass data as part of jump
  - Return to most recent site set up to handle exception
  - Unnecessary activation records may be deallocated
    - May need to free heap space, other resources
- Two main language constructs
  - Declaration to establish exception handler
  - Statement or expression to raise or throw exception

Often used for unusual or exceptional condition, but not necessarily

ML Example

```
exception Determinant;  (* declare exception name *)
fun invert (M) =              (* function to invert matrix *)
        ...
        if ...
              then raise Determinant    (* exit if Det=0 *)
              else ...
 end;
...
invert (myMatrix) handle Determinant => ... ;
```

Value for expression if determinant of myMatrix is 0

C++ Example

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

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

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 what C++ only recommends

ML Exceptions

- Declaration: exception ⟨name⟩ of ⟨type⟩
  - Gives name of exception and type of data passed when this exception is raised
- Raise: raise ⟨name⟩ ⟨parameters⟩
- Handler: ⟨exp1⟩ handle ⟨pattern⟩ => ⟨exp2⟩
  - Evaluate first expression
  - If exception that matches pattern is raised, then evaluate second expression instead

General form allows multiple patterns

Dynamic Scoping of Handlers

```
exception Ovflw;
fun reciprocal(x) = if x<min  then  raise Ovflw  else  1/x;
(reciprocal(x) handle Ovflw=>0)  /  (reciprocal(y) handle Ovflw=>1);
```

  - First call to reciprocal() handles exception one way, second call handles it another way

- Dynamic scoping of handlers: in case of exception, jump to most recently established handler on run-time stack
- Dynamic scoping is not an accident
  - User knows how to handle error
  - Author of library function does not

Exceptions for Error Conditions

```
- datatype ‘a tree = LF of  ‘a | ND of  (‘a tree)*(‘a tree)
- exception No_Subtree;
- fun lsub (LF x) = raise No_Subtree
  |     lsub (ND(x,y)) = x;
> val lsub = fn : ‘a tree -> ‘a tree
```

- This function raises an exception when there is no reasonable value to return
  - What is its type?

Exceptions for Efficiency

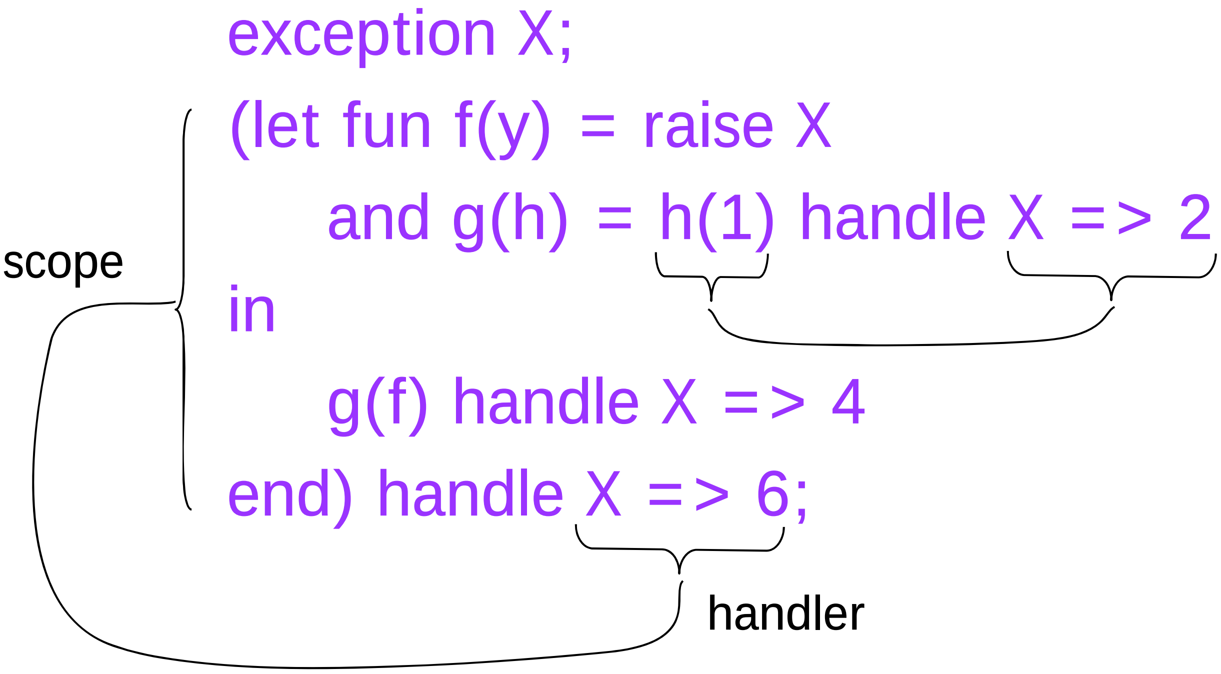
- Function to multiply values of tree leaves

```
fun prod(LF x) = x
|    prod(ND(x,y)) = prod(x) * prod(y);
```

- Optimize using exception

```
fun prod(tree) =
    let exception Zero
        fun p(LF x) = if x=0 then (raise Zero) else x
        |    p(ND(x,y)) = p(x) * p(y)
    in
        p(tree) handle Zero=>0
    end;
```

Scope of Exception 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;
```

scope

handler

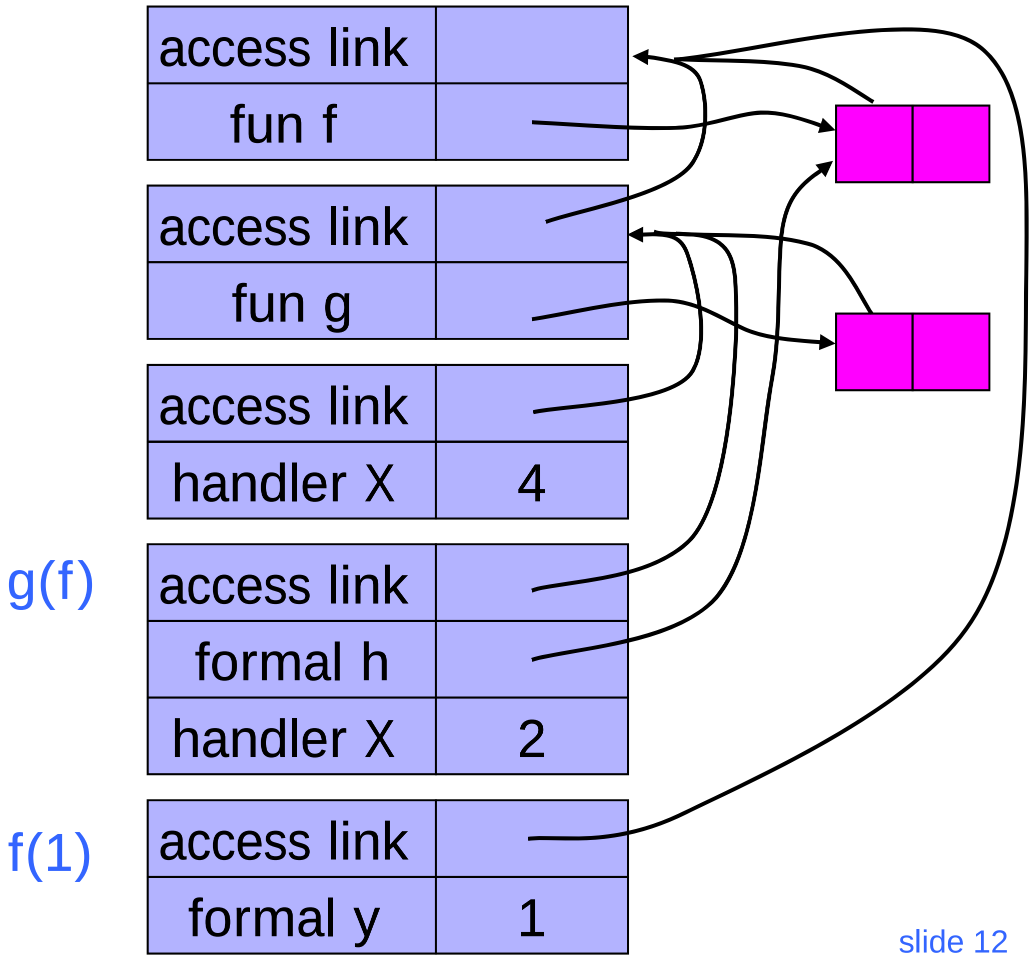
Which handler is used?

Dynamic Scope of Handlers (1)

```
exception X;
fun f(y) = raise X
fun g(h) = h(1) handle X => 2
g(f) handle X => 4
```

Dynamic scope:

find first X handler, going up the dynamic call chain leading to "raise X"

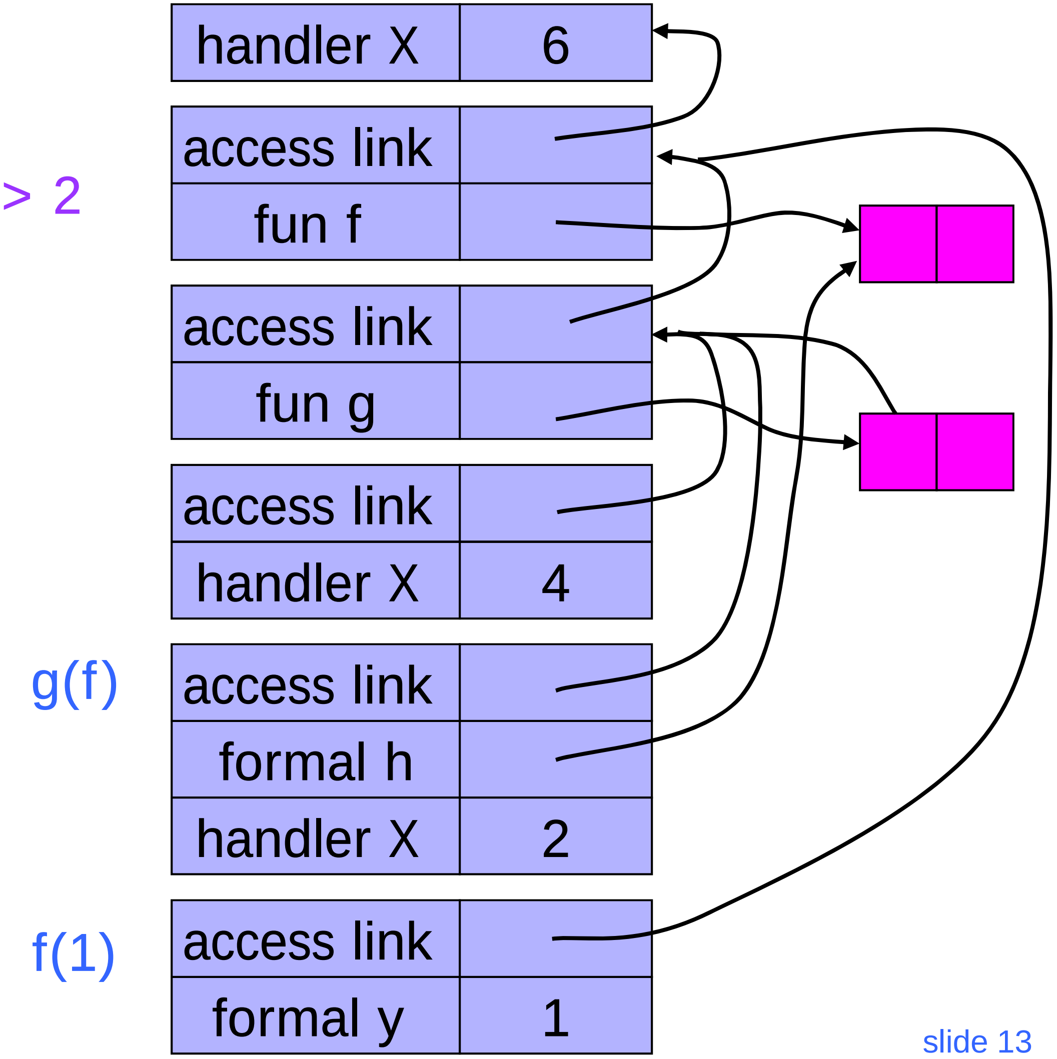


Dynamic Scope of Handlers (2)

```
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;
```

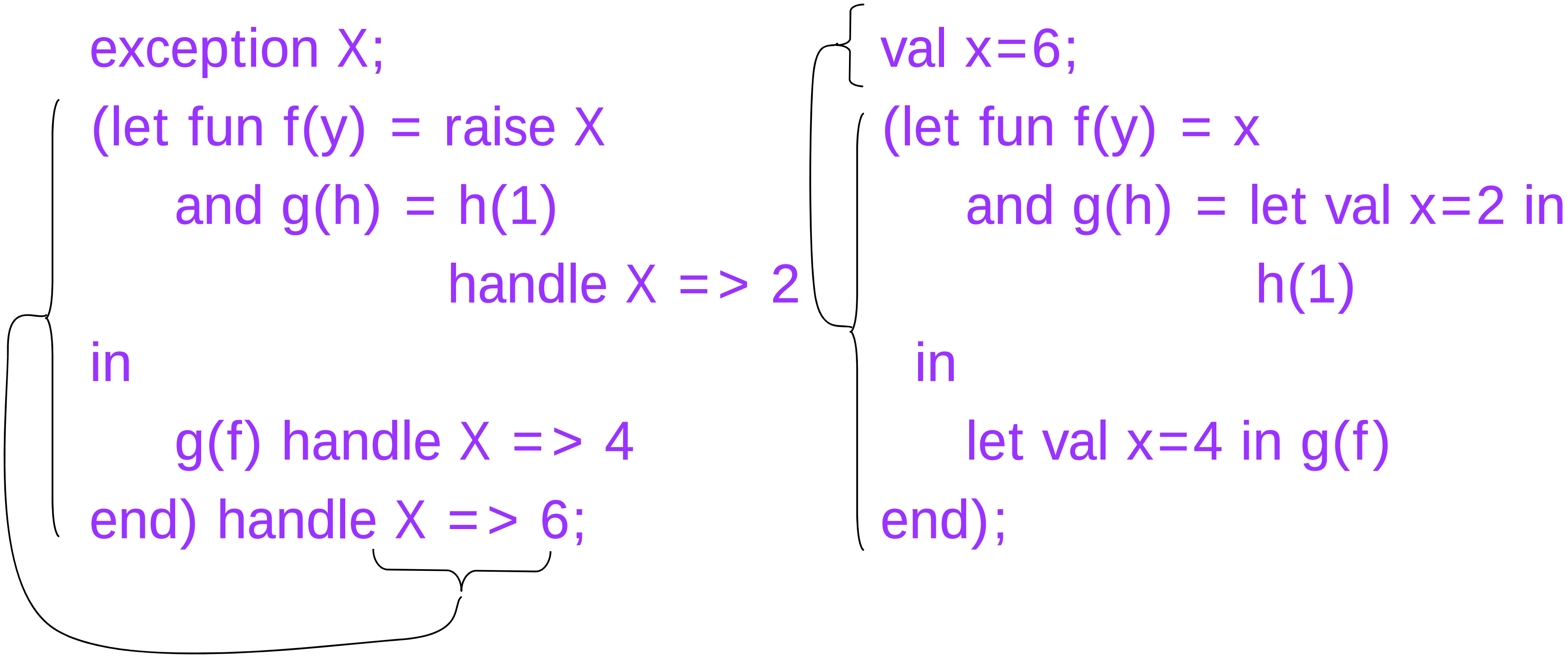
Dynamic scope:

find first X handler, going up the dynamic call chain leading to “raise X”



	handler X	6
	access link	
	fun f	
	access link	
	fun g	
	access link	
	handler X	4
g(f)	access link	
	formal h	
	handler X	2
f(1)	access link	
	formal y	1

Scoping: Exceptions vs. Variables



```
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;
```

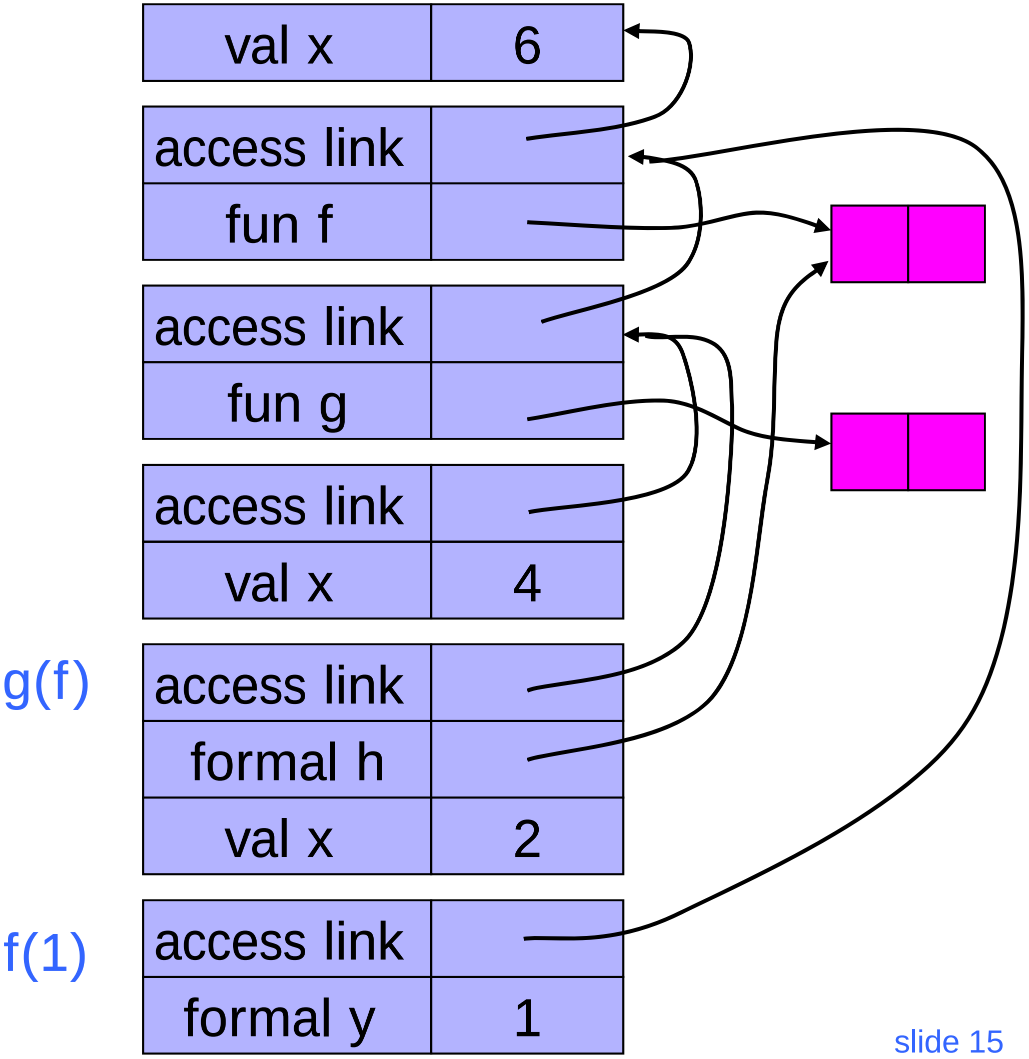
```
val x=6;
(let fun f(y) = x
     and g(h) = let val x=2 in
                    h(1)
  in
     let val x=4 in g(f)
end);
```

Static Scope of Declarations

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

Static scope:

find first x, following access links from the reference to X



Typing of Exceptions

- Typing of raise ⟨exn⟩
  - Definition of typing: expression e has type t if normal termination of e produces value of type t
  - Raising an exception is not normal termination
    - Example: 1 + raise X
- Typing of handle ⟨exception⟩ => ⟨value⟩
  - Converts exception to normal termination
  - Need type agreement
  - Examples
    - 1 + ((raise X) handle X => e) Type of e must be int (why?)
    - 1 + (e_1 handle X => e_2) Type of e_1,e_2 must be int (why?)

Exceptions and Resource Allocation

```
exception X;
(let
    val x = ref [1,2,3]
 in
    let
        val y = ref [4,5,6]
    in
        ... raise X
     end
end);  handle X => ...
```

- Resources may be allocated between handler and raise
  - Memory, locks on database, threads ...
- May be “garbage” after exception

General problem,
no obvious solution