Excepciones

Paradigmas de la Programación
FaMAF 2015
basado en filminas de Vitaly Shmatikov
capítulo 8.2 de Mitchell

excepciones: salida estructurada

terminar una parte de la computación

- saltar fuera de una construcción
- pasar datos como parte del salto
- retornar al lugar más reciente donde tratar la excepción
- en el proceso de retorno se pueden desalojar los active records innecesarios

compuestas de **dos construcciones** lingüísticas

- un **manjeador** de excepciones (*exception handler*)
- sentencia o expresión que **levanta** (*raise*) o tira (*throw*) la excepción

uso: normalmente para una condición excepcional, pero no necesariamente

ejemplo en ML

```
exception Determinant;   (*declarar el nombre de la excepción*)
fun invert (M) =                    (*función para invertir una matriz*)
       …
       if …
          then raise Determinant    (* salir si Det=0 *)
       else …
 end;
…
invert (myMatrix) handle Determinant => … ;
```

valor para la expresión si el determinante de myMatrix es 0

excepciones en ML

- Declaración: `exception ⟨name⟩ of ⟨type⟩`
  - dá el nombre de la excepción y el tipo de dato que se pasa cuando se levanta
- Levantado: `raise ⟨name⟩ ⟨parameters⟩`
- Manejador: `⟨exp1⟩ handle ⟨pattern⟩ => ⟨exp2⟩`
  - evaluar la primera expresión
  - si la excepción levantada se corresponde con el patrón (*pattern-matching*), se evalúa la segunda expresión

ejemplo C++

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

try { … invert(myMatrix); …
}
catch (Determinant) { …
  // recuperarse del error
}
```

excepciones en C++ vs ML

- C++
  - pueden tirar cualquier tipo
  - 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
  - las excepciones son un tipo diferente de entidades, distintas a los tipos
  - se declaran antes de usarse

ML requiere, C++ recomienda

los manejadores tienen alcance dinámico!

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

la primera llamada a `reciprocal()` maneja la excepción de una forma, la segunda la maneja de otra forma

- manejo dinámico de los manejadores: si se levanta una excepción, se salta al manejador más cercano en la pila de ejecución
- no es una casualidad: el autor del programa sabe cómo manejar una excepción, pero el autor de una biblioteca no sabe

excepciones para condiciones de error

```
- 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
```

esta función levanta una excepción cuando no hay ningún valor retornable para devolver

- cuál es su tipo?

excepciones para eficiencia

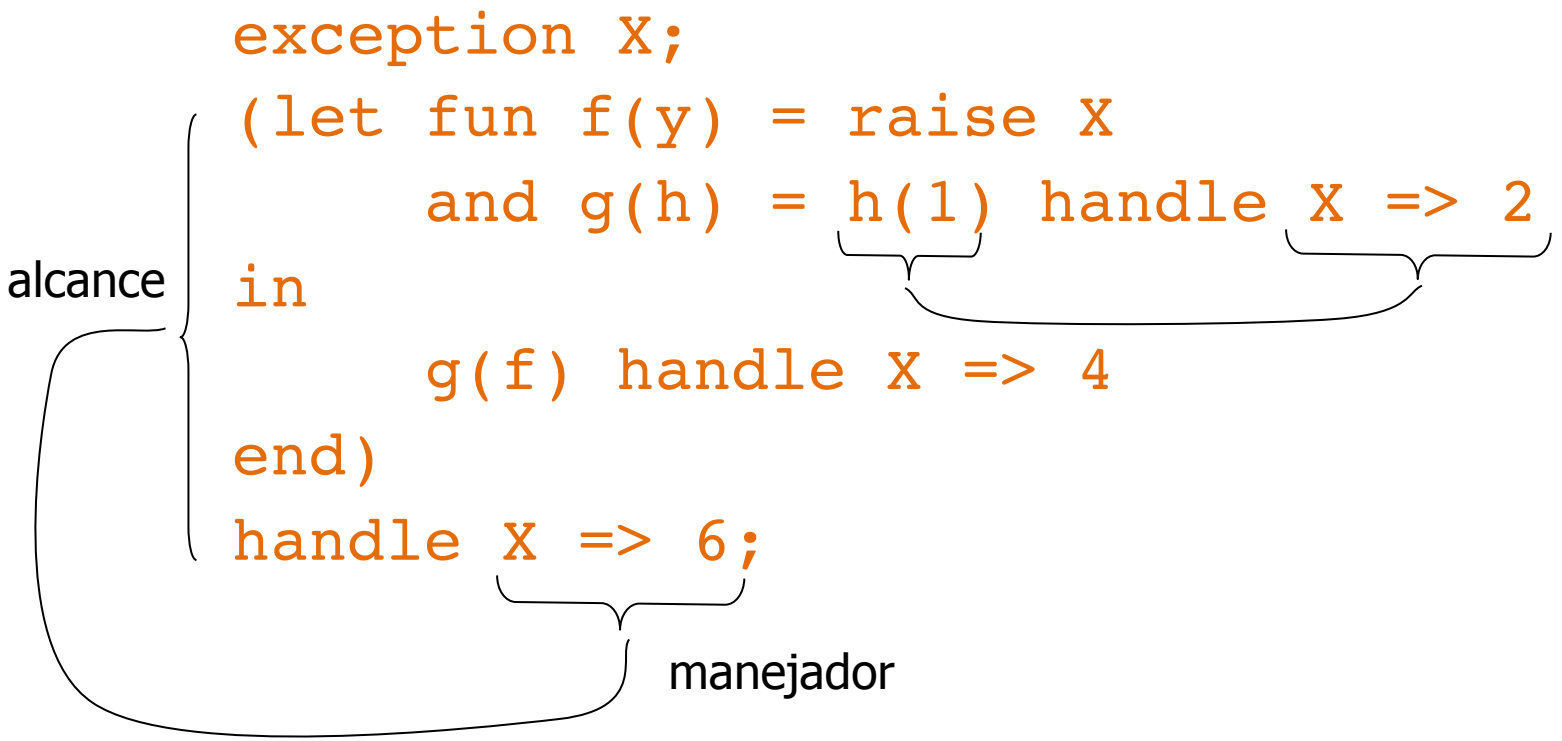
- function para multiplicar los valores de las hojas de los árboles

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

- optimización usando excepciones

```
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;
```

alcance de los manejadores de excepciones



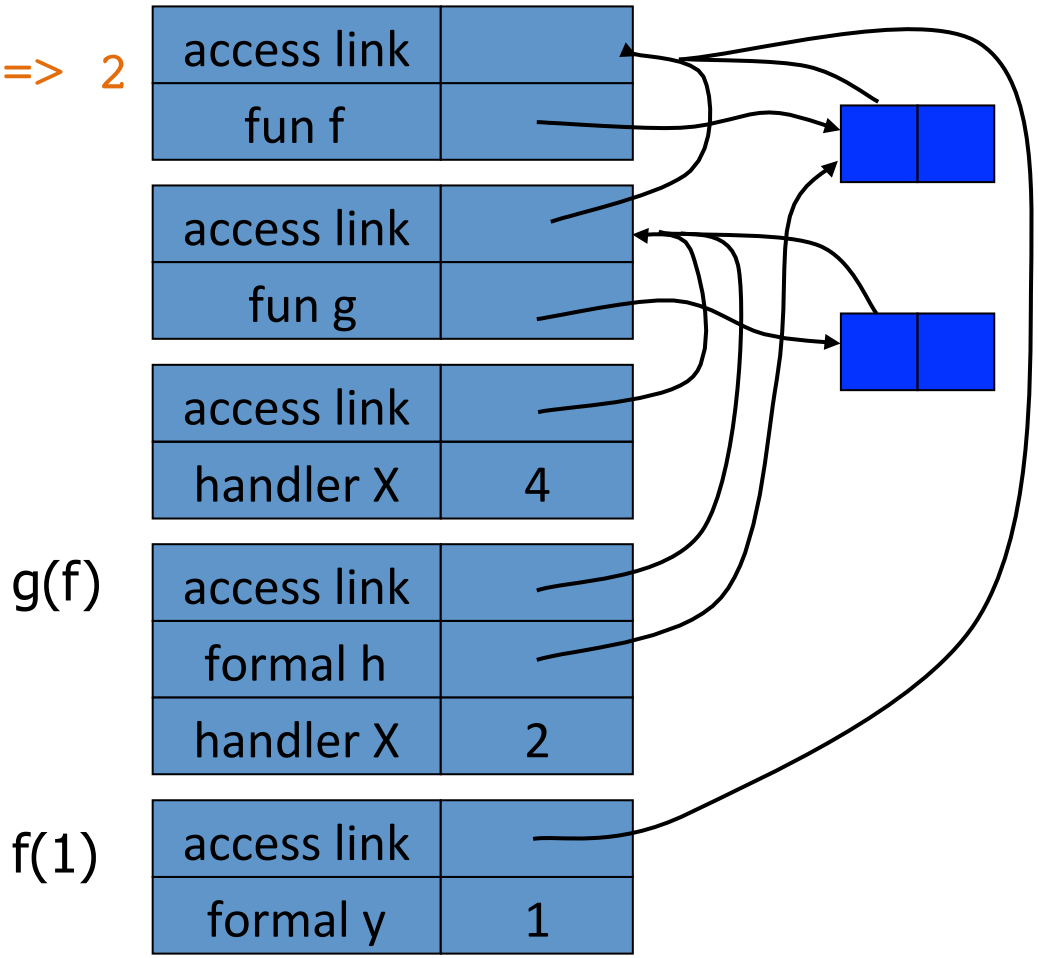
qué manejador se usa?

alcance dinámico de los manejadores (1)

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

alcance dinámico:

encontrar el primer manejador X, subiendo por la cadena dinámica de llamados a función que nos lleva a “raise X”

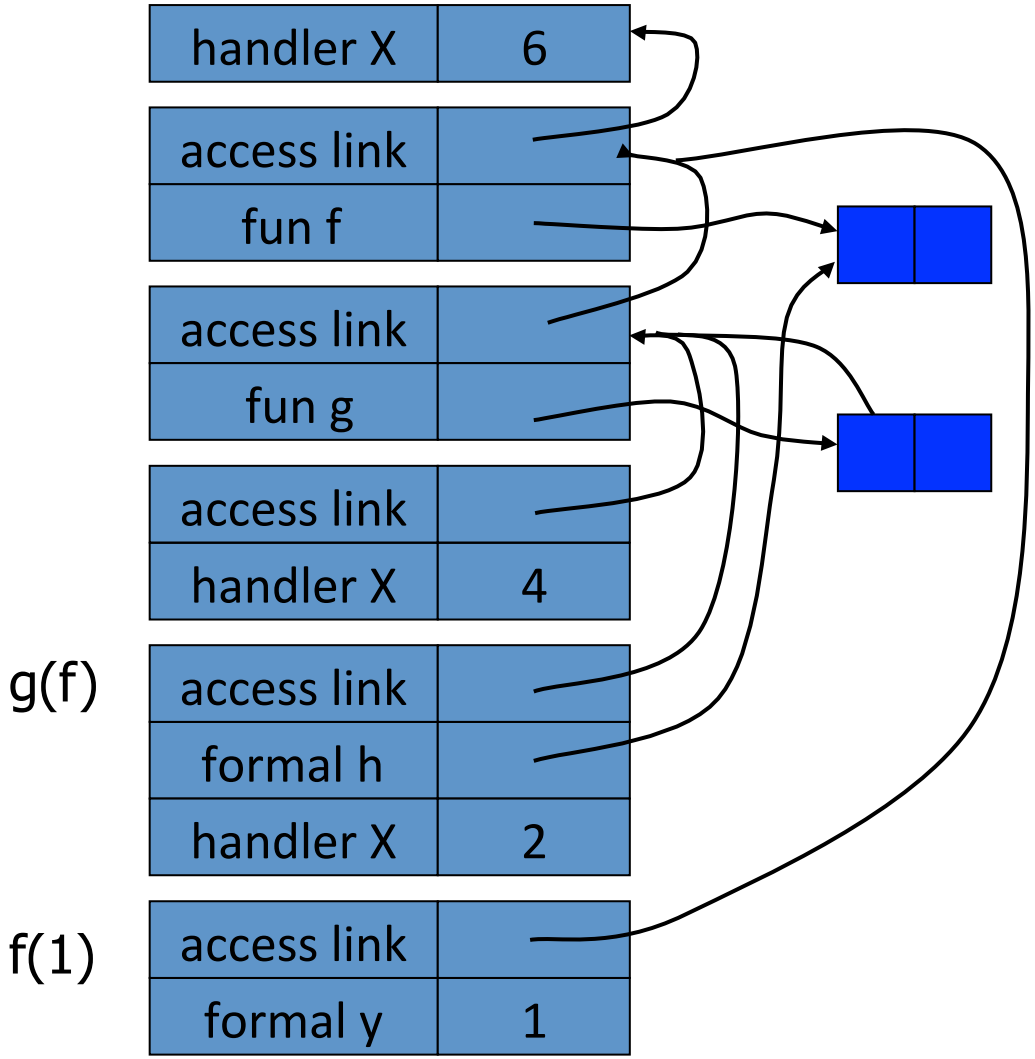


alcance dinámico de los manejadores (2)

```
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;
```

alcance dinámico:

encontrar el primer manejador X, subiendo por la cadena dinámica de llamados a función que nos lleva a “raise X”



alcance: excepciones 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;
```

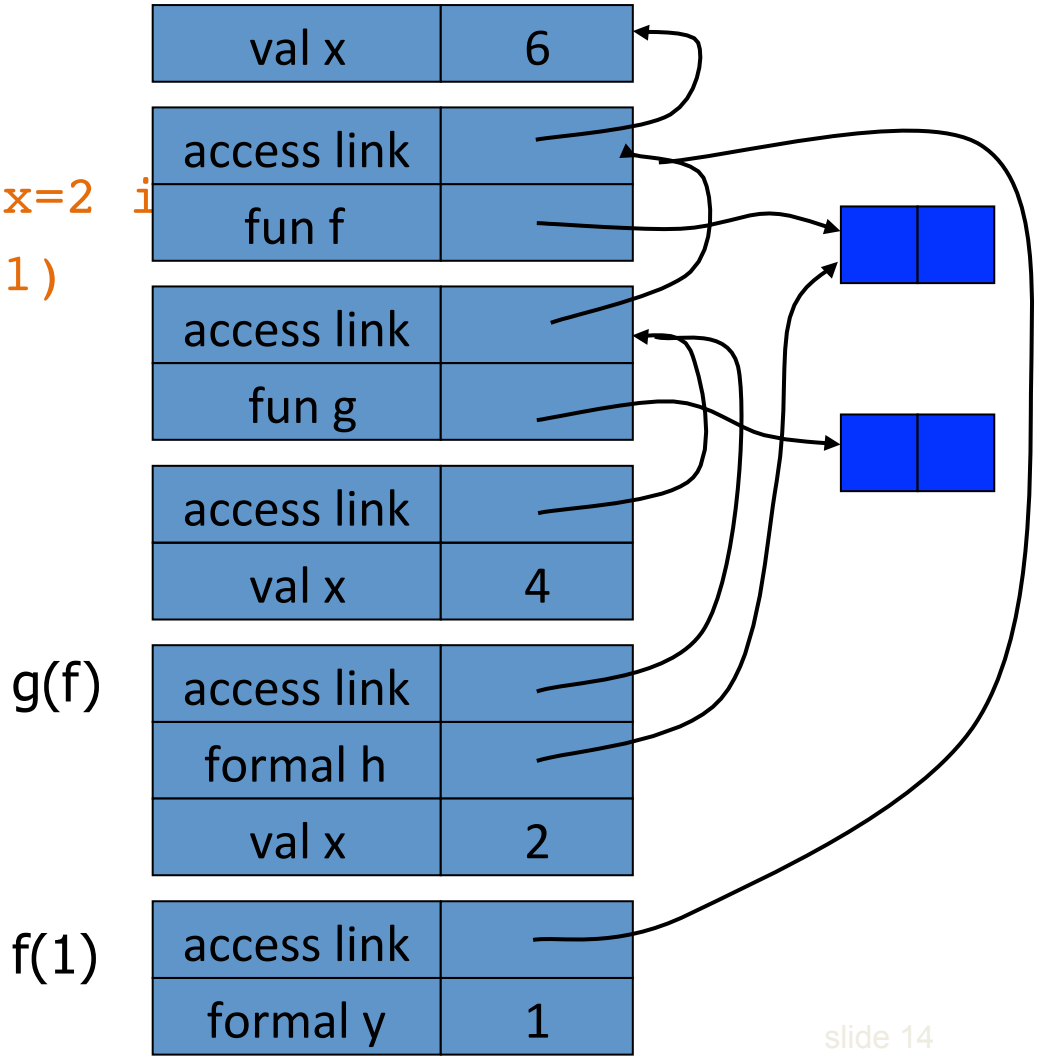
```
val x=6;
(let fun f(y) = x
    and g(h) = let val x=2 in
                   h(1)
  in
    let val x=4 in g(f)
end);
```

alcance estático de las declaraciones

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

alcance estático:

encontrar x, siguiendo los access links desde la referencia hasta X



tipado de las excepciones

- tipado de `raise ⟨exn⟩`
  - definición de tipado: la expresión `e` tiene el tipo `t` si la terminación normal de `e` produce un valor de tipo `t`
  - levantar una excepción no es una terminación normal

    ```
    1 + raise X
    ```
- tipado de `handle ⟨exception⟩ => ⟨value⟩`
  - convierte una excepción a terminación normal
  - implementa acuerdo de tipos
    - `1 + ((raise X) handle X => e)`

      el tipo de `e` tiene que ser int
    - `1 + (e1 handle X => e2)`

      el tipo de `e1`, `e2` tiene que ser int

excepciones y *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 => ...
```

pueden haberse alojado recursos entre el handler y el raise: memoria, locks, threads... que quizás deberían ser basura después de la excepción

no está claro cómo habría que tratarlo