UNIVERSITETET
I OSLO

Control in Sequential Languages

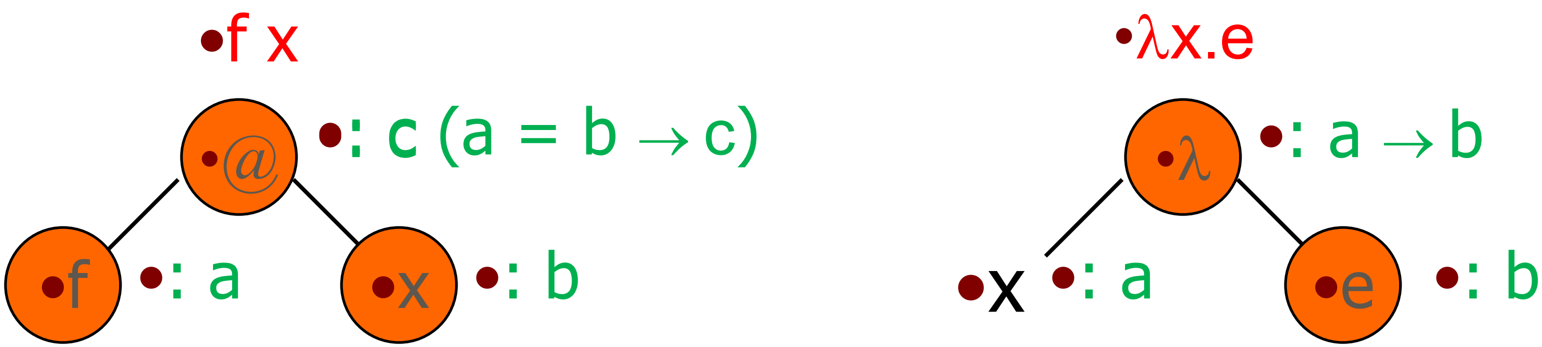
Volker Stolz
stolz@ifi.uio.no

Department of Informatics – University of Oslo

Initially by Gerardo Schneider & Arild Torjusen
Based on John C. Mitchell's slides (Stanford U.)

INF 3110 – 2016

Application and Abstraction



- **Application** f x
  - f must have function type domain→ range
  - domain of f must be type of argument x (b)
  - the range of f is the result type (c)
  - thus we know that a = b→ c

- **Abstraction** λx.e (fn x => e)
  - The type of λx.e is a function type domain→ range
  - the domain is the type of the variable x (a)
  - the range is the type of the function body e (b)

The type inference algorithm

- f(x) = 2+x equiv f = λx. (2+x) equiv f = λx. ((plus 2) x)

◆ Example

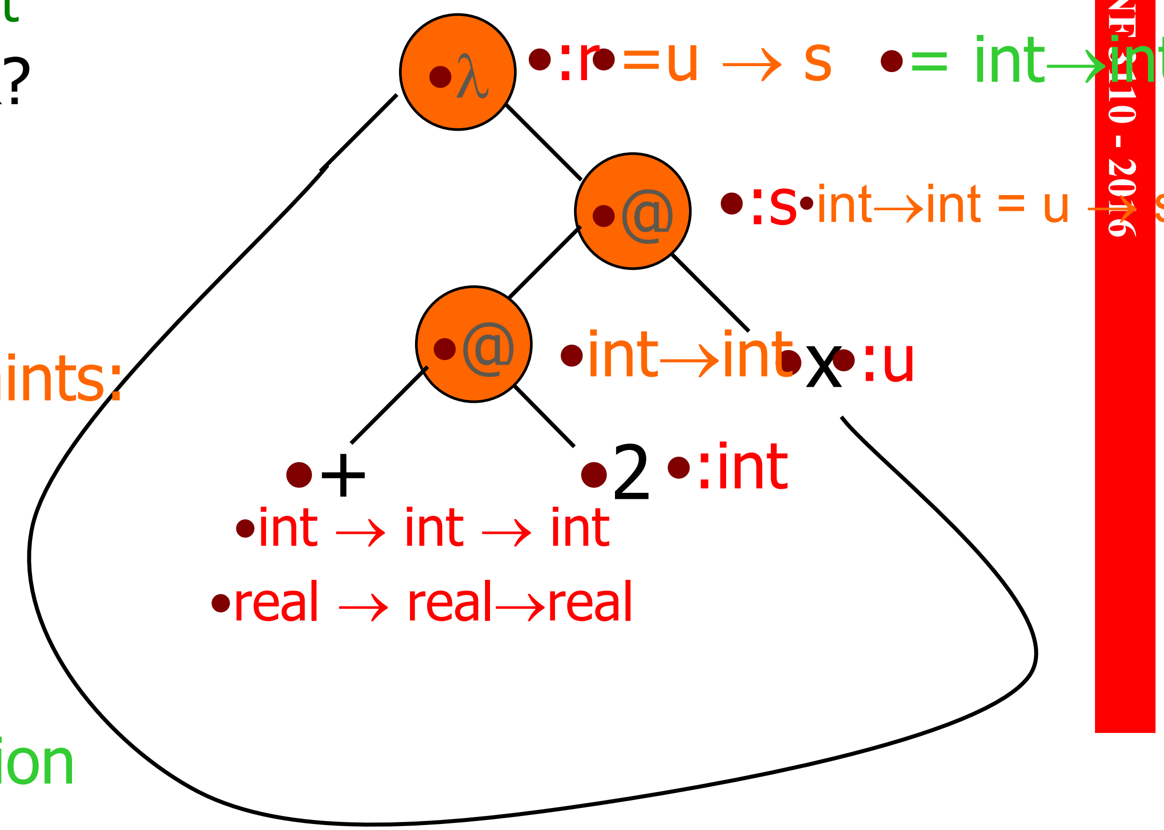
- fun f(x) = 2+x;
- (val f = fn x => 2+x ;)

> val f = fn : int → int

◆ How does this work?

- 1. Assign types to expressions
- 2. Generate constraints:
  - int→int = u → s
  - r = u → s
- 3. Solve by unification/substitution

- Graph for λx. ((+ 2) x)



Types with type variables

- ’a is syntax for “type variable” (t in the graph)

◆ Example

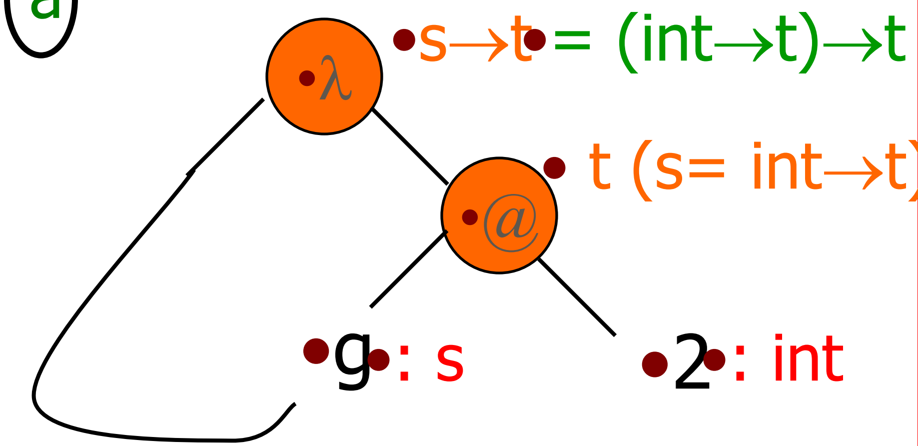
- fun f(g) = g(2);

> val f = fn : (int→’a)→’a

◆ How does this work?

- 1. Assign types to leaves
- 2. Propagate to internal nodes and generate constraints
- 3. Solve by substitution

- Graph for λg. (g 2)



Use of Polymorphic Function

- Function

  - fun f(g) = g(2);

  > val f = fn : (int→'a)→'a

- Possible applications

g may be the function:

- fun add(x) = 2+x;

> val add = fn : int → int

Then:

- f(add);

> val it = 4 : int

- g may be the function:
- \- fun isEven(x) = ...;
- \> val it = fn : int → bool
- Then:
- \- f(isEven);
- \> val it = true : bool

INF 3110 - 2016

Recognizing type errors

- Function
  - fun f(g) = g(2);
  - > val f = fn : (int→'a)→'a
- Incorrect use
  - fun not(x) = if x then false else true;
  - > val not = fn : bool → bool
  - f(not);

  Why?

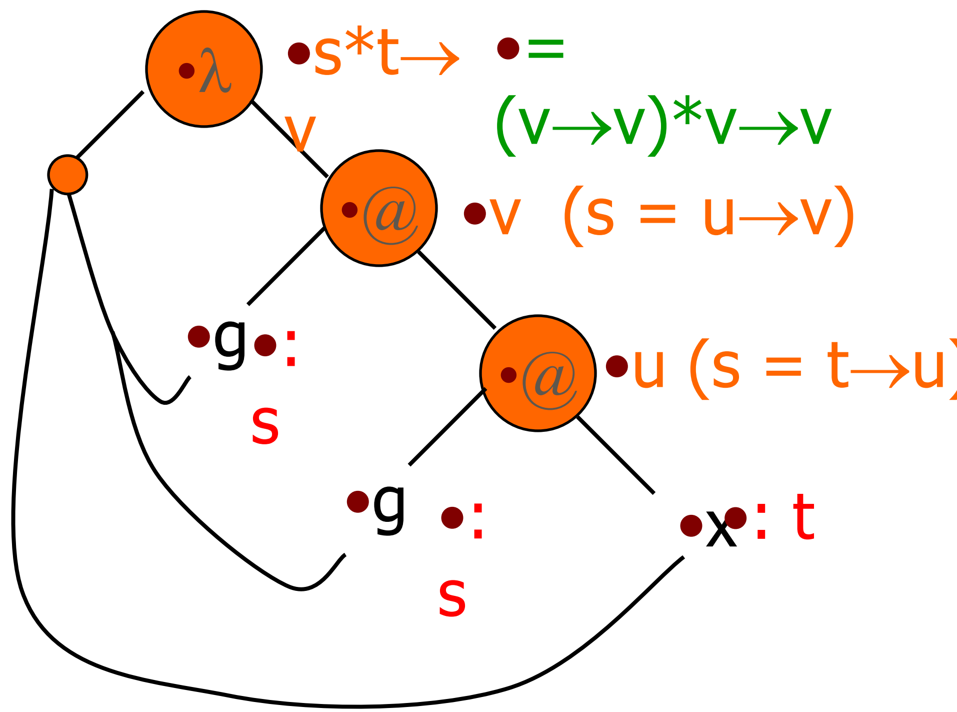
  Type error: cannot make bool → bool = int → 'a

INF 3110 - 2016

Another type inference example

- Function Definition
  - fun f(g,x) = g(g(x));

•Graph for λ⟨g,x⟩. g(g x)



- Assign types to leaves
- Propagate to internal nodes and generate constraints:
  - s = t→u, s = u→v
  - t=u,u=v
  - t=v
- Solve by substitution

INF 3110 - 2016

Multiple clause function

- Datatype with type variable

```
- datatype ’a list = nil | cons of ’a*(’a list);
> nil : ’a list
> cons : ’a*(’a list) → ’a list
```

- Polymorphic function

```
- fun append(nil,l) = l
    |   append (x::xs,l) = x:: append(xs,l);
> val append= fn: ‘a list * ‘a list → ’a list
```

- Type inference
  - Infer separate type for each clause
    append: ‘a list * ‘b -> ‘b
    append: ‘a list * ‘b -> ‘a list
  - Combine by making the two types equal (if necessary) ‘b = ‘a list

Main points about type inference

- Compute type of expression
  - Does not require type declarations for variables
  - Find *most general type* by solving constraints
  - Leads to polymorphism
- Static type checking without type specifications
- May lead to better error detection than ordinary type checking
  - Type may indicate a programming error even if there is no type error (example following slide).

Type inference and recursion

- Function definition
  - fun sum(x) = x + sum(x-1);

  > val sum= fn : ' int→' int

  sum = λx .((+ x) (sum((- x) 1)))

Outline

- Polymorphism*s*
- Type inference
- **Type declaration**

INF 3110 - 2016

Type declaration

- Transparent: alternative name to a type that can be expressed without this name

- Opaque: new type introduced into the program, different to any other

ML has both forms of type declaration

Type declaration: Examples

- Transparent ("type" declaration)

  - - type Celsius = real;
  - - type Fahrenheit = real;
  - - fun toCelsius(x) = ((x-32.0)*0.5556);
  - > val toCelsius = fn : real → real
  - More information:
  - - fun toCelsius(x: Fahrenheit) = ((x-32.0)*0.5556): Celsius;
  - > val toCelsius = fn : Fahrenheit → Celsius

  - Since Fahrenheit and Celsius are synonyms for real, the function may be applied to a real:

    - - toCelsius(60.4);
    - > val it = 15.77904 : Celsius

INF 3110 - 2016

Type declaration: Examples

- Opaque ("datatype" declaration)
  - - datatype A = C of int;
  - - datatype B = C of int;
  - A and B are different types
  - Since B declaration follows A decl.: C has type int→B

  Hence:

  - fun f(x:A) = x: B;

  > Error: expression doesn't match constraint [tycon mismatch]

  expression: A constraint: B

  in expression: x: B

INF 3110 - 2016

Equality on Types

Two forms of type equality:

- Name type equality: Two type names are equal in type checking only if they are the same name

- Structural type equality: Two type names are equal if the types they name are the same

Example: Celsius and Fahrenheit are structurally equal although their names are different

Outline

- Structured Programming
  - *go to* considered harmful
- Exceptions
  - “Structured” jumps that may return a value
  - Dynamic scoping of exception handler
- Continuations

Control flow in sequential programs

- The execution of a (sequential) program is done by following a certain control flow
- The end-of-line (or semi-colon) terminates a statement
- What is the next instruction to be executed?
  - The flow of control goes top-down in general
  - Jumps (loops, conditionals, etc)
- It is not easy, in general to ”see” whether a given instruction is reachable from another (Program Analysis)

Fortran Control Structure

```
10 IF (X .GT. 0.000001) GO TO 20
11 X = -X
   IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.00001) GO TO 30
   X = X-Y-Y
30  X = X+Y
   ...
50 CONTINUE
   X = A
   Y = B-A
   GO TO 11
   …
```

Just a label

INF 3110 - 2016

Similar structure may occur in assembly code

Historical Debate

- Dijkstra: “Go To Statement Considered Harmful” (1968)
  - “… the **go to** statement should be abolished from all ‘higher level’ programming languages…”
- Knuth: “Structured Programming with go to Statements” (1974)
  - You can use goto, but do so in structured way …
- General questions
  - Do syntactic rules force good programming style?
  - Can they help?

Advance in Computer Science

- Standard constructs that structure jumps

  if … then … else … end

  while … do … end

  for … { … }

  case …

- Modern style
  - Group code in logical blocks
  - Avoid explicit jumps except for function return
  - Cannot jump *into* middle of block or function body
  - Exceptions and continuations (?!)

Jumps into Blocks – Why not?

- Label in the body of a function
- Should an activation record be created?
- If not, what about local variables?
  - They are meaningless
- If so, how to set function parameters?
  - There are no parameter values

```
fun bizarre(pars);
  local vars;
   ...
  a: ....
   ...
end;

Program P;
  ....
  goto a;
  ....
end;
```

No clear answers! Better to reject these programs!

Outline

- Structured Programming
  - *go to* considered harmful
- Exceptions
  - "Structured" jumps that may return a value
  - Dynamic scoping of exception handler
- Continuations
- Evaluation order

INF 3110 - 2016

Exceptions: Structured Exit

- Terminate part of computation
  - Jump *out* of construct, not *into* some part of the program.
  - Pass data as part of jump
  - Return to most recent site set up to handle exception
- Memory management needed
  - Unnecessary activation records may be deallocated
- Two main language constructs
  - Statement or expression to ***raise*** exception (*throw, Java*)
  - Exception ***handler*** (*catch,* Java)
- Possible to have more than one handler

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 isZero(Det )
          then raise Determinant  (* exit if Det is zero*)
            else …
 end;
...
invert (myMatrix) handle Determinant => … ;
```

Value for expression if determinant of myMatrix is zero

INF 3110 - 2016

Java Example

```
class DetException extends Exception { ... } ;  //Exception declaration

public static Matrix invert(Matrix m) throws DetException {
    ...
    if (isZero(det)) {throw new DetException(“Determinant is zero”); }
    //throw statement
    else {  ... }
}

public static void main(String[] args) throws Exception {
    Matrix M, inverted ;
    ...
    try { inverted = invert (M);}
    catch(DetException de) { //code to handle exception
       log("An exception occured:"+de.toString());
       inverted = ....
    }
```

ML Exceptions

- Exceptions are a different kind of entity than types
- Declare exceptions before use
- Exceptions are dynamically scoped
  - Control jumps to the handler most recently established (run-time stack) (more later…)
  - ML is otherwise statically scoped
- Pattern matching is used to determine the appropriate handler (C++/Java uses type matching)

ML Exceptions

- Declaration

  exception ⟨name⟩ of ⟨type⟩

  gives name of exception and type of data passed when raised

  - exception Overflow;
  - exception Signal of int;

- Raise

  raise ⟨name⟩ ⟨parameters⟩

  expression form to raise an exception and pass data

  - raise Overflow;
  - raise Signal(x+4);

- Handler

  ⟨exp1⟩ handle ⟨pattern⟩ => ⟨exp2⟩

  evaluate first expression exp1

  if exception that matches pattern is raised,

  then evaluate second expression exp2 instead

  (General form allows multiple patterns)

- Compare

  try {res:=exp1} catch (OvflException oe) {res:=exp2}

ML Exceptions - example

```
- exception noSuchElement ;
- fun nth (n,nil) = raise noSuchElement
    | nth (0,s::ss) = s
    | nth (n,s::ss) = nth((n-1),ss) ;
 > val nth = fn : int * 'a list -> 'a

 - nth(2,[1,2,3]) ;
  > val it = 3 : int
 - nth(4,[1,2,3]) ;
 > uncaught exception noSuchElement
      raised at: stdIn:10.25-10.38

 - fun safeNth(n,xs) = nth(n,xs) handle noSuchElement => 0 ;
  > val safeNth = fn : int * int list -> int
 -   safeNth(4,[1,2,3]) ;
  > val it = 0 : int
```

INF 3110 - 2016

Which Handler is Used?

```
- exception Ovflw;
- fun reciprocal(x) =
      if x<=min  then raise Ovflw  else 1.0/x;
- (reciprocal(x) handle Ovflw=>0.0)  /  (reciprocal(x) handle Ovflw=>1.0);
```

- 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

Handlers with pattern matching

- Handler

  ⟨exp⟩ handle ⟨pattern1⟩ => ⟨exp1⟩

  | ⟨pattern2⟩ => ⟨exp2⟩

  ...

  | ⟨pattern3⟩ => ⟨exp3⟩

  evaluate first expression exp

  if exception that matches one of the patterns is raised,

  then evaluate the corresponding expression

INF 3110 - 2016

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+8;
> val it = 10 : int
```

- The expression to the left of the handler is evaluated
- If it terminates normally the handler is not invoked
- If the handler is invoked, pattern matching works as usual in ML

Exception for Error Condition

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

- This function raises an exception when there is no reasonable value to return

```
- lsub(Leaf(3));
 > uncaught exception No_Subtree raised at:…

 - lsub(Node (Leaf(3),Leaf(5)));
 > val it = Leaf 3 : int tree
```

INF 3110 - 2016

Exception for Efficiency

- Function to multiply values of tree leaves

```
- fun prod(LF x)      = x: int
  |   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;
```

Dynamic Scope of Handler

```
- 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

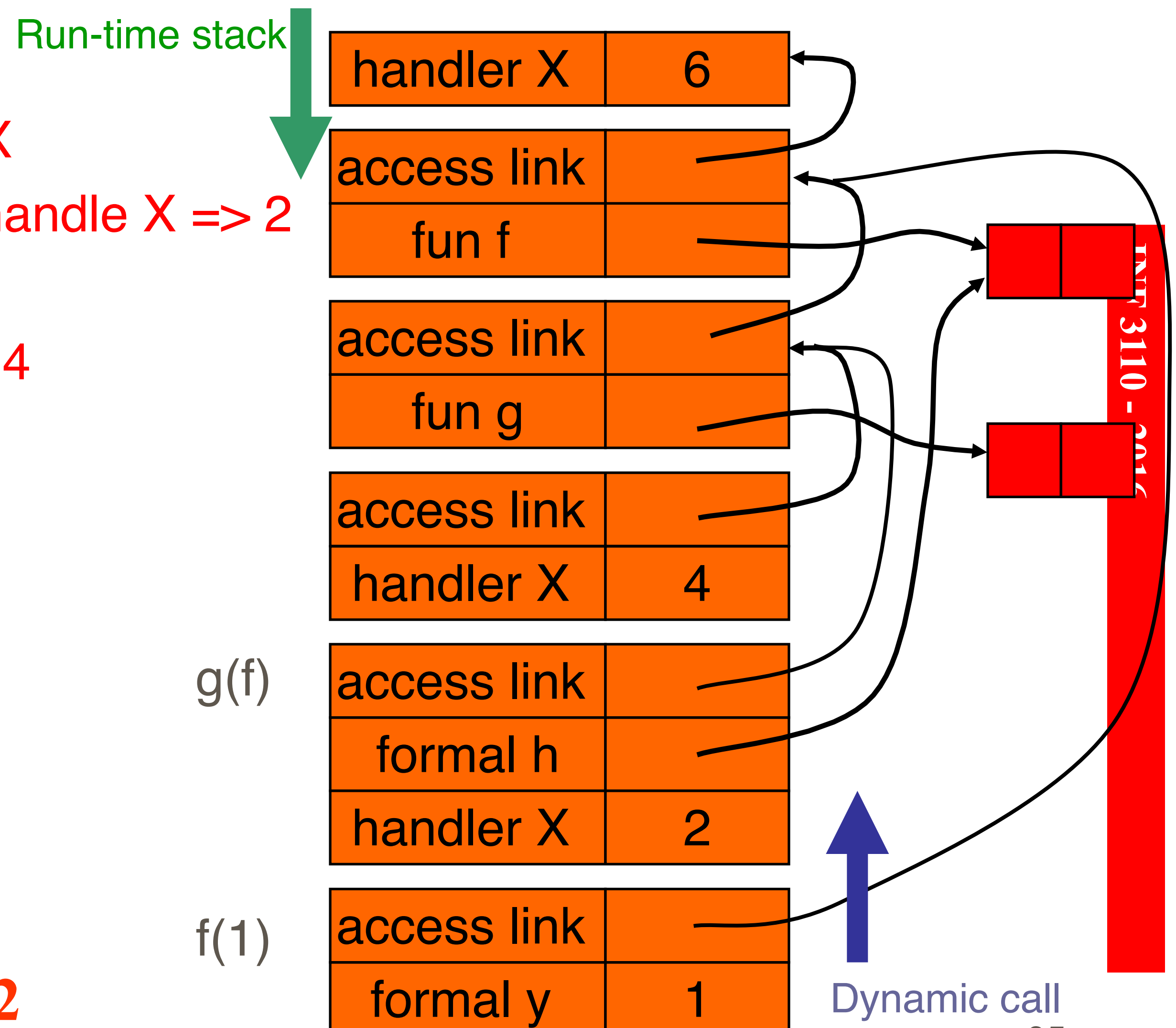
handler

What is the value of g(f)?

It depends on which handler is used!

Dynamic Scope of Handler

```
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.

Answer: g(f) = 2

Compare to Static Scope of Variables

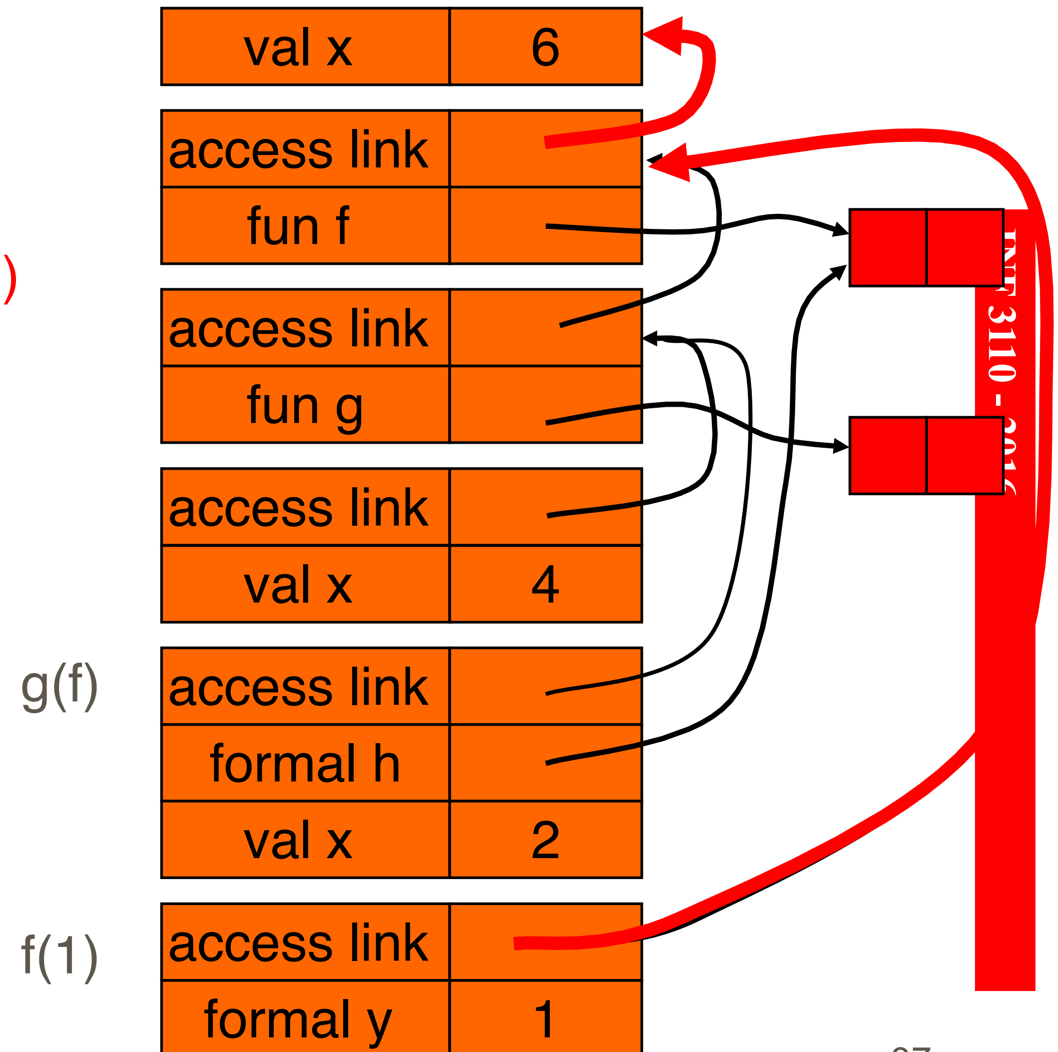
```
exception X;
(let fun f(y) = raise X
     and g(h) = h(1)
                  handle X => 2
in
     g(f) handle X => 4
end) handle X => 6;
```

```
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);
```

INF 3110 - 2016

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.

Answer: g(f) = 6

Typing of Exceptions

- Typing of raise ⟨exc⟩
  - 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
    - 1 + raise No_value (the sum will not be performed)
  - Type of raise ⟨exc⟩ is a type variable ‘a
- Typing of handle ⟨exc⟩ => ⟨value⟩
  - Converts exception to normal termination
  - Need type agreement
  - Examples
    - 1 + ((raise X) handle X => e) Type of e must be int
    - 1 + (e_1 handle X => e_2) Type of e_1, e_2 must be int

INF 3110 - 2016

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 => ...
```

[1,2,3] built in the heap, ref x pushed into stack

[4,5,6] built in the heap, ref y pushed into stack

Control is transferred outside the scope

x and y popped off the stack

[1,2,3] and [4,5,6] garbage collected

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 allocated between handler and raise may be "garbage" after exception
- Open files might not be closed

General problem: no obvious solution

Exceptions and Resource Allocation

```
try {
    ...
    fOut = new PrintWriter(
      new FileWriter("OutFile.txt"));
    ...
}
catch (Exception e) {
    ...
}
finally {
      if (fOut != null) {
     fOut.close();
      }  else { ... }
}
```

- Resources allocated between handler and raise may be "garbage" after exception
- Open files might not be closed
- In Java you would use the "finally" construct

ML summary

- Is ML unpractical?, what about
  - Input/Output, using files
  - Interacting with underlying OS
  - Making executable applications
  - etc. etc.
- We have focused on the basics
  - Basic ML constructs
  - Learning to think ”functional”, recursion
  - Higher order functions
  - Type system and type inference
  - Exceptions

Basic I/O example

```
val infile = TextIO.openIn("somefile.txt");
TextIO.lookahead infile ;
TextIO.inputN(infile,10);
TextIO.inputLine infile ;
TextIO.inputAll infile ;
print it ;
TextIO.lookahead infile ;
TextIO.closeIn infile ;
```