# Closures (9.9)

## Function Declarations, Global Names and Static Binding

| | |
|---|---|
| **val** m= 1;<br>**fun** F n= n+m<br>**fun** G s= 2∗ (F s)<br>*> val m = 1 : int*<br>*> val F = fn : int -> int*<br>*> val G = fn : int -> int* | F is bound to the function, which adds 1 to its argument n.<br>F and G are bound to two specific *function values*,<br>A value can not be changed,<br>but we may later *redefine*<br>F and/or G to denote some other value. |
| - **val** m= 10;<br>*> val m = 10 : int* | redefine m. Does that affect the function, which F denotes? |
| - F 5;<br>*> val it = 6 : int* | No!, F still denotes the function, which adds 1 to its argument |
| - G 5;<br>*> val it = 12 : int* | and the G-function is also unaffected |
| - **fun** F j= 10∗j;<br>*> val F = fn : int -> int* | Now, redefine F to denote the function which multiplies its argument with 10 |
| - G 5;<br>*> val it = 12 : int* | But G is still bound to the function, which returns 2∗(a+1), where a is the argument |

The way SML handles global identifiers in function declarations and fn-expressions is called *static binding*:

| | |
|---|---|
| val (a,b,c)= …<br>fun F(..)= ..<br>fun G(x,y)= ( a x b F( ) c y)<br>val b= ….<br>fun F n= … | In the internal representation of the function g bound to G the actual values of the global a, b, c and F must be catched, such that later redefinitions of a, b, c and F will not affect the g-function |



# Closures (9.9)

A *closure* is the form, which SML uses for the internal representation of a function f

A fn-expression

**fn** $pat_1$=> $exp_1$ | … | $pat_n$=> $exp_n$

is equivalent to the fn-expression below

**fn** x=> **case** x **of** $pat_1$=> $exp_1$ | … | $pat_n$=> $exp_n$

Internally the value of such a fn-expression is represented by the *closure*:

(env, x, case x of $pat_1$=> $exp_1$ | … | $pat_n$=> $exp_n$)

- x is a new identifier
- env is a value environment, which is the part of the actual environment, binding the global identifiers occurring in $expr_i$ ,$1 \le i \le n$

Recall that the execution of the function declaration

**fun** f *$apat_1$ = $exp_1$*
| f *$apat_2$ = $exp_2$*
**…**
| f *$apat_n$ = $exp_n$*

is the same as executing the following value declaration:

**val rec** f = **fn** x => **case** x **of** *$apat_1$ => $exp_1$* | … | *$apat_n$ => $exp_n$*

which binds f to the value of the fn-expression.

Hence, the fun-declaration results in the following binding:

f → (env, x , **case** x **of** *$apat_1$ => $exp_1$* | … | *$apat_n$ => $exp_n$*)

# Closures (9.9)

Example:

| | value environment |
|---|---|
| **val** m= 1; | [m → 1] |
| **fun** F n= n+m | [ m → 1, F → ([m → 1], x , case x of n => n+m) ] |
| **fun** G s= 2∗ (F s) | [ m → 1, F → ([m → 1], x , case x of n => n+m), G → ([F → ([m → 1], x , case x of n => n+m)], y, case y of s=> 2∗(F s)) ] |
| - **val** m= 10; > *val m = 10 : int* | [ m → 10, F → ([m → 1], x , case x of n => n+m), G → ([F → ([m → 1], x , case x of n => n+m)], y, case y of s=> 2∗(F s)) ] |
| - F 5; > *val it = 6 : int* | [ m → 10, F →… ,G →…, it → 6 ] |
| - G 5; > *val it = 12 : int* | [ m → 10, F →… ,G →…, it → 12 ] |
| - **fun** F j= 10∗j; > *val F = fn : int -> int* | [ m → 10, G → ([F → ([m → 1], x , case x of n => n+m)], y, case y of s=> 2∗(F s)) it → 12, F → ([ ], x , case x of j => 10∗j), ] |
| - G 5; > *val it = 12 : int* | |



# Expression Evaluation with Environments

An expression *expr* is evaluated in a value-environment *env* to get its value *v*

Notation: ( *expr*, *env*) ~→ *v*

The evaluation takes place in a finite number of steps:

$( expr_1, env_1) \sim\to ( expr_2, env_2) \sim\to \ldots \sim\to ( expr_n, env_n) \sim\to v$

The *env* part is omitted when no identifiers in the expression.

# Evaluating Function Applications

**Non-recursive functions**:

Consider a function application for a non-recursive function f

f v, where f → ($env_f$, x, $e_f$)

This function application results in the evaluation of $e_f$ in the environment

$env_f$ + [x→v]

So we have

f v ~→ ( $e_f$, $env_f$+ [x→v] )

Example:

F 5 , where F → ([m → 1], x , case x of n => n+m)
~→ (case x of n => n+m, [m → 1, x→ 5] )
~→ (case 5 of n => n+m, [m → 1] )
~→ (n+m, [m → 1, n→ 5] )
~→ 5+1
~→ 6

# Evaluating Function Applications

## Recursive functions

Consider a function application for a recursive function f

f v, where f → ($env_f$, x, $e_f$)

This function application results in the evaluation of $e_f$ in the environment

$env_f$ + [x→v, f → ($env_f$, x, $e_f$)]

So we have

f v ~→ ( $e_f$, $env_f$+ [x→v, f → ($env_f$, x, $e_f$)] )

## Example

| | value environment |
|---|---|
| val c= 10<br>fun R 0= c<br>\| R n= n∗ R(n-1) | [c → 10]<br>[c → 10,<br>R→ ([c→ 10], x, case x of 0=> c \| n=> n∗R(n-1)) |

R 1

~→ (case x of 0=> c | n=> n∗R(n-1), [c → 10, x→ 1, R → ( )] )

~→ (n∗R(n-1), [n→ 1, R → ( )] )

~→ (1∗R(1-1), [R → ( )] )

~→ (R(0), [R → ( )] )

~→ (case x of 0=> c | n=> n∗R(n-1), [c → 10, x→ 0, R → ( )])

~→ ( c , [c → 10] )

~→ 10



# Type Inference

Consider the higher order function

```
fun  foldr f b [ ]          = b
   | foldr f b (x :: xs )   = f(x, foldr f b xs)
```

foldr is a higher order function and the argument pattern shows that:

foldr: $\tau_1 \to \tau_2 \to \tau_3$ list $\to \tau_4$

f: $\tau_1$, b: $\tau_2$, (x :: xs ): $\tau_3$ list,

x: $\tau_3$, xs : $\tau_3$ list

The function body has type $\tau_4$ so

b: $\tau_4$, f(x, foldr f b xs): $\tau_4$

hence $\tau_2 = \tau_4$

foldr f b xs: $\tau_2$

f( x , foldr f b xs ) — x: $\tau_3$, foldr f b xs: $\tau_2$

f: $\tau_3 * \tau_2 \to \tau_2$ , hence $\tau_1 = \tau_3 * \tau_2 \to \tau_2$

Consequently we have

foldr: $(\tau_3 * \tau_2 \to \tau_2) \to \tau_2 \to \tau_3$ list$\to \tau_2$

or

foldr: ('a ∗ 'b → 'b) → 'b → 'a list -> 'b

# Eager and Lazy Evaluation

SML evaluates function applications *eagerly*: In

$f(e_1, e_2, \dots, e_n)$

first evaluate *all* the argument expressions to

$(v_1, v_2, \dots, v_n)$

and then apply the function to the evaluated argument value

$f(v_1, v_2, \dots, v_n)$

Consider:

```
fun ifthenelse(x,y,z)= if x then y else z;
> val 'a ifthenelse = fn : bool * 'a * 'a -> 'a
```

```
val r= if true then 2.0 else 3.1/0.0;
> val r = 2.0 : rea

ifthenelse(true, 2.0, 3.1/0.0);
! Uncaught exception:  Divl
```

works differently

## Getting Lazy Evaluation in SML

The function application (**fn** true=> e2 | false=> e3) e1

works exactly like **if** e1 **then** e2 **else** e3

Using this idea we might declare an ifthenelse function like this:

```
fun ifthenelse(x,y,z)= if x then y() else z();
> val 'a ifthenelse = fn : bool *(unit -> 'a) *(unit -> 'a) -> 'a
```

```
ifthenelse(true, fn()=> 2.0, fn()=> 3.1/0.0);
> val it = 2.0 : real
```

now the function application behaves like
**if** true **then** 2.0 **else** 3.1/0.0