Curried/uncurried functions

Multiple arguments can be handled in two different ways

- Curried function (from Haskell Curry) with n arguments:
  a higher-order function returning a "chain" of (higher order) functions

  $$\texttt{fun}\ \mathit{pat}_1\ \texttt{->}\ \texttt{fun}\ \mathit{pat}_2\ \texttt{->}\ \ldots\ \texttt{fun}\ \mathit{pat}_n\ \texttt{->}\ \mathit{exp}$$

- Uncurried function with n arguments:
  a function taking as argument a tuple of size n

  $$\texttt{fun}\ (\mathit{pat}_1, \mathit{pat}_2, \ldots, \mathit{pat}_n)\ \texttt{->}\ \mathit{exp}$$

Correspondence between curried and uncurried function

- an uncurried function can be transformed in the equivalent curried version
- a curried function can be transformed in the equivalent uncurried version
- isomorphism between type $t_1\ \texttt{->}\ t_2\ \texttt{->}\ \ldots\ \texttt{->}\ t_n\ \texttt{->}\ t$ and type $t_1 \star t_2 \star \ldots \star t_n\ \texttt{->}\ t$

Curried/uncurried functions

Examples

```
(* addition of two integers *)
# fun x y->x+y;; (* curried version *)
- : int -> int -> int = <fun>

# fun (x,y)->x+y;; (* uncurried version *)
- : int * int -> int = <fun>

(* multiplication of three integers *)
# fun x y z->x*y*z;; (* curried version *)
- : int -> int -> int -> int = <fun>

# fun (x,y,z)->x*y*z;; (* uncurried version *)
- : int * int * int -> int = <fun>
```

Partial application

Curried functions and partial application

- curried functions allow partial application:

  arguments can be passed once at time

- uncurried functions do not allow partial application:

  arguments must be passed altogether

Partial application

Example

```
# let uncurried_add(x,y)=x+y;;
val uncurried_add : int * int -> int = <fun>

# uncurried_add(1,2);; (* both arguments must be passed *)
- : int = 3

# let curried_add x y=x+y;;
val curried_add : int -> int -> int = <fun>

# let inc=curried_add 1;; (* only argument 1 is passed *)
val inc : int -> int = <fun>

# inc 2;; (* argument 2 is passed to compute the final result *)
- : int = 3
```

Remark: the result of the partial evaluation is saved in `inc` as a useful by-product, 1+2 can always be computed with the single expression `curried_add 1 2`

Partial application

Partial application promotes generic programming

Partial application allows function specialization: from a generic function it is possible to generate more specific ones with no code duplication.

- software reuse and maintenance are favored
- interesting examples will be shown later

Equality on functions

Function extensionality

Two functions are equal if and only if they return the same result for all possible arguments.

Limitations of functions as first class values

Functions are more complex than other kinds of values: they cannot be compared for theoretical limitations

Example

```
let curried_add x y=x+y;; (* curried_add : int -> int -> int *)
let f1=curried_add;; (* f1 : int -> int -> int *)
let f2 x=curried_add x;; (* f2 : int -> int -> int *)
let f3 x y=curried_add x y;; (* f3 : int -> int -> int *)
```

We cannot test that `curried_add`, `f1`, `f2`, `f3` are equal:

```
# curried_add = f1;; (* '=' is test equality in OCaml *)
Exception: (Invalid_argument "compare: functional value")
```

Declarations of global variables

A more detailed syntax for declarations

```
Dec ::= 'let' Def ('and' Def)*
Def ::= Pat '=' Exp | ID Pat+ '=' Exp
Pat ::= ID | '_' |  '(' Pat? ')' | Pat (',' Pat)+
```

Declarations of global variables

A simple example

```
# let x=2;;
val x : int = 2
# let y=x+40;;
val y : int = 42
# x+y;;
- : int = 44
```

Remarks

- variables are global because they are declared at the top level
- local variables can be declared as well at inner levels (see later)
- the content of variables cannot be changed, variables are constant
- there is no variable assignment

Declarations of global variables

More elaborate examples

```
# let x=2 and y=42;;
val x : int = 2
val y : int = 42
# x+y;;
- : int = 44
# let x=3;; (* previous declaration is shadowed *)
val x : int = 3
# x+y;;
- : int = 45
# let pair = 4,2;;
val pair : int * int = (4,2)
# let a,b = pair;;
val a : int = 4
val b : int = 2
```

Remark

a new declaration with the same name shadows the previous declaration

Declarations of global variables

Examples of global function declarations

```
# let inc x = x+1 and add (x,y) = x+y and add2 x y = x+y;;
val inc : int -> int = <fun>
val add : int * int -> int = <fun>
val add2 : int -> int -> int = <fun>
```

A useful syntactic abbreviation

`let inc x = x+1` abbreviates `let inc = fun x->x+1`

`let add (x,y)= x+y` abbreviates `let add = fun (x,y)->x+y`

`let add2 x y = x+y` abbreviates `let add2 = fun x->fun y->x+y`

More in general:

let id $pat_1\ pat_2 \ldots pat_n$ = exp abbreviates

let id = **fun** $pat_1\ pat_2 \ldots pat_n$ -> exp which abbreviates

let id = **fun** pat_1 -> **fun** pat_2 -> ... **fun** pat_n -> exp

Boolean values

Syntax

```
Exp ::= BOOL | 'not' Exp | Exp '&&' Exp | Exp '||' Exp
Type ::= 'bool'
```

`BOOL` defined by the regular expression **`false|true`**

Standard syntactic rules

- left syntactic associativity for `&&` and `||`
- **`not`** higher precedence than `&&`
- `&&` higher precedence than `||`

Boolean values

Static semantics

- **`false`** and **`true`** are type correct and have type `bool`
- **`not`** e is type correct and has type `bool` if and only if e is type correct and has type `bool`
- e_1 `&&` e_2 and e_1 `||` e_2 are type correct and have type `bool` if and only if e_1 and e_2 are type correct and have type `bool`

Boolean values

Dynamic semantics

- operands of `&&` and `||` evaluated left-to-right with short circuit
- short circuit means that not always the second operand is evaluated
- if e_1 evaluates to `false` then e_1`&&`e_2 evaluates to `false` and e_2 is not evaluated
- if e_1 evaluates to `true` then e_1`&&`e_2 evaluates to the value of e_2
- if e_1 evaluates to `true` then e_1`||`e_2 evaluates to `true` and e_2 is not evaluated
- if e_1 evaluates to `false` then e_1`||`e_2 evaluates to the value of e_2

Boolean values

Conditional expression

```
Exp ::= 'if' Exp 'then' Exp 'else' Exp
```

Conditional expression has precedence lower than all other operators

Static semantics

`if` e **`then`** e_1 **`else`** e_2 is type correct and has type t if and only if

- e is type correct and has type `bool`
- e_1 and e_2 are type correct and have the same type t

Dynamic semantics

- if e evaluates to `true`, then **`if`** e **`then`** e_1 **`else`** e_2 evaluates to the value of e_1; hence, e_2 is not evaluated
- if e evaluates to `false`, then **`if`** e **`then`** e_1 **`else`** e_2 evaluates to the value of e_2; hence, e_1 is not evaluated

Recursive declarations

Syntax for recursive declarations

```
Dec ::= 'let' 'rec'? Def ('and' Def)*
Def ::= Pat '=' Exp | ID Pat+ '=' Exp
Pat ::= ID | '_' |  '(' Pat? ')' | Pat (',' Pat)+
```

Remark

- the optional `'rec'` keyword means that the declaration is allowed to be recursive
- the use of `'rec'`, `'and'` keywords supports mutually recursive declarations
- recursive declarations allowed only for function types and other particular types
- for simplicity we consider only recursive declarations of functions

Recursive declarations of functions

Examples

```
(* addition of square numbers *)
let sumsquare n = (*sumsquare cannot be used on the right-hand side*)
    if n<0 then 0 else n*n+sumsquare(n-1);;
Error:  Unbound value sumsquare

let rec sumsquare n = (*sumsquare can be used on the right-hand side*)
    if n<0 then 0 else n*n+sumsquare(n-1);;
```

Curried functions and generic programming

Example 1: addition of square numbers

```
let rec sumsquare n =
    if n<0 then 0 else n*n+sumsquare(n-1);;
```

Example 2: addition of cube numbers

```
let rec sumcube n =
    if n<0 then 0 else n*n*n+sumcube(n-1);;
```

Remarks

- the two declarations above are almost identical!
- can we improve code reuse and maintenance?

Solution: use a curried function with an argument of type function

Curried functions and generic programming

Solution

```
(* computes f 0 + f 1 + ... + f n *)
let rec gen_sum f n = (* (int -> int) -> int -> int *)
if n<0 then 0 else f n+gen_sum f (n-1);;

let sumsquare = gen_sum (fun x->x*x);; (* int -> int *)
let sumcube = gen_sum (fun x->x*x*x);; (* int -> int *)
```

Remarks

`gen_sum` can be specialized because

- it is curried
- the “first” argument is the function `f` rather than the number `n`

Declarations of local variables

Syntax for declarations of local variables

```
Dec ::= 'let' 'rec'? Def ('and' Def)* 'in' Exp
Def ::= Pat '=' Exp | ID Pat+ '=' Exp
Pat ::= ID | '_' |  '(' Pat? ')' | Pat (',' Pat)+
```

Example

```
# let f x=x+1 and v=41 in f v;; (* f and v can only be used here *)
- : int = 42
# let x=1 in let x=x*2 in x*x (* nested declarations *)
- : int = 4
```

Remark

Nested declarations shadow outer declarations with the same ID

Static scope of declarations

Example

```
let v=40;;

let f x = x*v;; (* v refers to the declaration above *)

f 3;; (* evaluates to 120 *)

let v=4;; (* previous declaration of v shadowed *)

(* f still refers to the shadowed variable v *)
f 3;; (* evaluates to 120 *)
```

Curried functions and generic programming (revisited)

A slightly better solution

```
let gen_sum f = (* (int -> int) -> int -> int *)
    let rec aux n = if n<0 then 0 else f n+aux (n-1) (* int -> int *)
    in aux;;
```

Remarks

We do not have to pass argument `f` to the recursive function `aux`

Lists

List constructors

- Syntax:

```
Exp ::= '['  ']'| Exp '::'Exp
```

- `[]` is the empty list constructor; it is a constant
- `::` is the non-empty list constructor; it is a binary operator
  $h::t$ is the list with head h and tail t
  Remark: h is an element (the first one), while t is a list

The usual properties of constructors hold

- $[\,] \neq h::t \qquad h \neq h::t \qquad t \neq h::t$
- $h_1::t_1 = h_2::t_2$ if and only if $h_1 = h_2$ and $t_1 = t_2$

Syntactic rules for lists

Non-empty list constructor `::`

- right syntactic associativity holds
  $h_1 :: h_2 :: t$ is equivalent to $h_1 :: (h_2 :: t)$
  this is the only sensible choice (see later on)
- `::` has lower precedence than unary and binary infix operators
- `::` has higher precedence than
  - the tuple constructor
  - anonymous function expression (**`fun`** `... -> ...`)
  - conditional expression (**`if`** `...` **`then`** `...` **`else`** `...`)

Syntactic rules for lists

A useful shorthand notation

`[`e_1`;`e_2`;...;`e_n`]` is equivalent to e_1`::`e_2`::...::`e_n`::[]`

Examples

- `[1] = 1::[]`
- `[1;2;3] = 1::2::3::[]`
- `[1,true] = (1,true)::[]`
- `1,[true] = 1,true::[]`

Warning

- the operator `;` inside square brackets has its own precedence rules!
- `;` has lower precedence than the tuple constructor
  `[1,true;2,false]=[(1,true);(2,false)]=(1,true)::(2,false)::[]`
- advice: use parentheses in this case!