Curried/uncurried functions

Definition

- Curried function (from Haskell Curry): a higher-order function with a single argument returning a chain of functions with a single argument
- Uncurried function: a function with multiple arguments

Facts

- an uncurried function can be transformed in the equivalent curried version
- a curried function can be transformed in the equivalent uncurried version

Examples

```
(* addition of two integers *)
fun x y->x+y;;        (* curried version int->int->int *)
fun (x,y)->x+y;;      (* uncurried version int*int->int *)
(* multiplication of three integers *)
fun x y z->x*y*z;;    (* curried version int->int->int->int *)
fun (x,y,z)->x*y*z;;  (* uncurried version int*int*int->int *)
```

Curried/uncurried functions

Partial application

- curried functions allow *partial* application: arguments can be passed once at time
- uncurried functions do not allow partial application: all arguments must be passed altogether

Example

```
let curried_add x y=x+y;;
let uncurried_add(x,y)=x+y;;
(* computes 1+2 with the uncurried version *)
uncurried_add(1,2);;
(* computes 1+2 by partial application *)
let inc=curried_add 1;; (* passes argument 1 and saves the result *)
inc 2;; (* passes argument 2 and computes the final result *)
```

Curried/uncurried functions

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 on

Boolean values

Syntax

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

`BOOL` boolean values **`false`**`|`**`true`**

Standard syntactic rules

- `&&` and `||` are left-associative
- **`not`** higher precedence than `&&`
- `&&` higher precedence than `||`

Boolean values

Static semantics

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

Boolean values

Standard semantics

- operands of `&&` and `||` evaluated left-to-right with “short circuit”
- if e_1 evaluates to `false` then e_1`&&`e_2 evaluates to `false`, else it evaluates to the value of e_2
- if e_1 evaluates to `true` then e_1`||`e_2 evaluates to `true`, else it 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 has type t if and only if
  e has type `bool` and e_1 and e_2 have type t
- **`if`** e **`then`** e_1 **`else`** e_2 is **not** type correct if
  - e has type $\neq$ `bool`
  - or there is no type t such that e_1 and e_2 have type t
  - or e or e_1 or e_2 is **not** type correct

More on declarations of global “variables”

Grammar

```
Dec ::= 'let' Def ('and' Def)*
    | 'let' 'rec' FunDef ('and' FunDef)*
Def ::=  Pat '=' Exp |  FunDef
FunDef ::= ID Pat* '=' Exp
```

Remark

- recursive declarations allowed only for function types and other types
- for simplicity we consider only recursive declarations of functions

Example

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

Example 2: addition of cube numbers

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

Remarks

- the two examples 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

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

let der_sumsquare = gen_sum (fun x->x*x);; (* int -> int *)
let der_sumcube = gen_sum (fun x->x*x*x);; (* int -> int *)
```

Remarks

`gen_sum` can be specialized because

- it is curried
- the "first" argument is `f` rather than `n`

Declarations of local “variables”

Syntax

```
Dec ::= 'let' Def ('and' Def)* 'in' Exp
     | 'let' 'rec' FunDef ('and' FunDef)* 'in' Exp
Def ::=  Pat '=' Exp |  FunDef
FunDef ::= ID Pat* '=' Exp
```

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 overrides 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;; (* declaration of v overridden *)
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`