OCaml Bindings and Builtin Types

CSC 310 - Programming Languages

Let Expressions

- Syntax: `let x = e1 in e2`
  - `x` is a bound variable
  - `e1` is the binding expression
  - `e2` is the body expression
- Evaluation
  - Evaluate `e1` to `v1`
  - Substitute `v1` for `x` in `e2` yielding `e2'`
  - Evaluate `e2'` to `v2` the final result

Let Expression Type Checking

- Syntax: `let x = e1 in e2`
- Type checking
  - If `e1:t1`
  - and assuming `x:t1` implies `e2:t`
  - then `(let x = e1 in e2):t`

Let Definitions vs. Let Expressions

- At the top-level, we write
  - `let x = e;; (* no in e2 part)`
  - This is called a let definition, not a let expression
- Omitting `in` means "from now on"

"' # let pi = 3.14;; (* pi is now boun in the rest of the top-level scope *)

Top-level Expressions

- We can write any expression at top-level
- Syntax: `e;;`
  - This means evaluate `e` and then ignore the result
  - Equivalent to `let _ = e`
  - Useful when `e` has a side effect, such as reading/writing a file, printing to the screen, etc.

Let Expressions: Scope

- In `let x = e1 in e2`, the variable `x` is not visible outside of `e2`
- Examples

```
# let x = 1 in x + 1;;
- : int = 2
# (let x = 1 in x + 1);;
- : int = 2
# x;;
Error: Unbound value x
# let x = 4 in (let x = x + 1 in x);;
- : int = 5
```

Nested Let Expressions

- Uses of `let` can be nested
- Example

```
let result =
  (let area =
    (let pi = 3.14 in
     let r = 1.0 in
     pi *. r *. r) in
  area /. 2.0);;
```

Nested Let Idiom

- We generally avoid nested let expressions
- Sometimes a nested binding can be rewritten in a linear style
- Example

```
let result =
  let pi = 3.14 in
  let r = 1.0 in
  let area = pi *. r *. r in
  area /. 2.0;;
```

Let Expressions in Functions

- You can use `let` inside of function bodies for local variables
- Example

```
let area r =
  let pi - 3.14 in
  pi *. r *. r
```

Shadowing Names

- Shadowing is rebinding a name in an inner scope to have a different meaning
  - Depends on the language
- C

```
int x;
void f (float x) {
    {
        char *x = NULL;
    }
}
```

- OCaml

```
let x = 3;;
let g x = x + 3;;
```

Shadowing: Semantics

- What if `e2` is also a `let` for `x`?
    - Substitution will stop at the `e2` of a shadowing `x`
- Example
    - `let x = 1+2 in let x = 3*x in x+1`
    - `let x = 3 in let x = 3*x in x+1`
    - `let x = 3*3 in x+1`
    - `let x = 9 in x+1`
    - `9+1`
    - `10`

Shadowing Idiom

- You can use shadowing to simulate mutation

```
let rec f x n =
  if x = 0 then 1
  else
    let x = x - 1 in (* shadowed *)
    n * (f x n)
```

- Avoiding shadowing is clearer, and recommended
  - With no shadowing, when you see a variable `x` you know it has not been "changed" no matter where it appears
  - If you want to "mutate" `x`, use a new name `x1`, `x'`, etc.

`let` and `match`

- The `let` expressions allows patterns
- Syntax: `let p = e1 in e2`
  - `p` is a pattern; if `e1` fails to match the pattern, then an exception is thrown
  - Equivalent to `match e1 with p -> e2`
- Examples
  - `let [x] = [[1]] in 1::x`
  - `let h::_ = [1;2;3] in h`
  - `let () = print_int 1 in 2`

Tuples

- Constructed using `(e1, ..., en)`
- Destructed using pattern matching
- Tuples can be heterogeneous unlike lists
- Tuple types use `*` to separate components

Tuple Examples

```
# (1,2);;
- : int * int = (1, 2)

# (1, "a", 2.14);;
- : int * string * float = (1, "a", 2.14)

# [(1,2)];;
- : (int * int) list = [(1, 2)]

# [(1,2); (1,2,3)];;
Error: This expression has type 'a * 'b * 'c
       but an expression was expected of type int * int
```

Pattern Matching Tuples

```
# let sum t =
  match t with
  | (x, y, z) -> x + y + z;;
val sum : int * int * int -> int = <fun>

# let sum' (x, y, z) = x + y + z;;
val sum' : int * int * int -> int = <fun>

# let addOne (x, y, z) = (x+1, y+1, z+1);;
val addOne : int * int * int -> int * int * int = <fun>

# sum (addOne (1, 2, 3));;
- : int = 9
```

Tuple Size

- Tuples of different size have different types
  - (a, b) has type 'a * 'b
  - (a, b, c) has type 'a * 'b * 'c
  - Patterns in the same match must have the same type
- Example

```
# let f t = match t with
    | (a, b) -> a + b
    | (a, b, c) -> a + b + c;;
Error: This pattern matches values of type 'a * 'b * 'c
       but a pattern was expected which matches values
       of type 'd * 'e
```

Records

- Records identify elements by name whereas tuple elements are identified by position
- Syntax to define a record type:

  ```
  type name = { f1: t1; ... fn: tn }
  ```

  where `f` is a field name
- Syntax to define a record value

  ```
  let variable_name = { f1=v1, ..., fn=vn }
  ```

Destructing Records

- Access by field name or pattern matching
- In record patterns, the fields can be skipped or reordered
- A field name can be used as the bound variable

Record Example

```
type date = { month: string; day: int; year: int };;

let mydate = { day=1; year=2000; month: "jan" };;

print_string mydate.month;;

let { month=_; day=d } = mydate in
let { year } = mydate in
let _ = print_int d in  (* prints 1 *)
print_int year;;        (* prints 2000 *)
```