CSE 130 [Winter 2014] Programming Languages

Records and References

Ravi Chugh

Feb 18
Today’s Plan

• Take a quick look at midterm solutions

• Finish our tour of ML

• Two features widely found in PLs:
  1. “Records” (≈ “Objects”)
  2. “References” (≈ “Pointers”)

... Segue to Scala!
Ways to Build Complex Values

Tuple Type

\[
  \text{type } t = (t1 \,*\, t2) 
\]
Value of \( t \) contains value of \( t1 \) and a value of \( t2 \)

Data Type

\[
  \text{type } t = \text{C1 of } t1 \mid \text{C2 of } t2 
\]
Value of \( t \) contains value of \( t1 \) or a value of \( t2 \)

Record Type

\[
  \text{type } t = \{ \, f1:t1 \,; \, \ldots \,; \, fn:tn \, \} 
\]
Value of \( t \) is an unordered tuple with field names
Records

Syntax of **record type** definitions:

```plaintext
type person = {
    first : string ;
    last  : string ;
    dob   : int*int*int
}
```

Syntax of **record literal** expressions:

```plaintext
{ first = "Ravi" ;
  last  = "Chugh" ;
  dob   = (11,5,1984) }
```
What does `p1` evaluate to?

```plaintext
# type person = { first : string ; last : string } ;;
# let p1 = { first = "Ravi" } ;;
```

(a) Syntax Error

(b) Type Error

(c) `val p1 : person = {first = "Ravi"}`
What does p1 evaluate to?

```ocaml
# type person = { first : string ; last : string } ;;
# let p1 = { first = "Ravi" } ;;
Error: Some record field labels are undefined: last
```

(a) Syntax Error
(b) Type Error
(c) val p1 : person = 
   {first = "Ravi"}

What does p2 evaluate to?

```
# type person = { first : string 
                ; last  : string } ;;
# let p2 = { first = "Ravi"
           ; last  = "Chugh"
           ; age   = 29 } ;;
```

(a) Syntax Error
(b) Type Error
(c) val p2 : person =
    {first = "Ravi"; last = "Chugh"; age = 29}
What does p2 evaluate to?

```ocaml
# type person = { first : string ; last : string } ;;
# let p2 = { first = "Ravi" ; last = "Chugh" ; age = 29 } ;;

Error: Unbound record field label age
```

(a) Syntax Error

(b) Type Error

(c) `val p2 : person = {first = "Ravi"; last = "Chugh"; age = 29}`
Records

- **All fields** must be defined
- **No extra fields** can be defined
- **Upside**: helps automatic type inference
  - By looking at a field read expression, can figure out exactly what type of record
- **Downside**: no overlapping field names in different record types
Records

```plaintext
# type person = { first : string ; last: string } ;;
# let p0 = { first = "Ravi" ; last = "Chugh" } ;;
```

Syntax of “record projection” (field read):

```plaintext
# p0.first ;;
- : string = "Ravi"
```
Records

# type person = { first : string ; last: string } ;;
# let p0 = { first = “Ravi” ; last = “Chugh” } ;;

Syntax of “record update”:

# { p0 with last = “CHUGH” } ;;
- : person = { first = “Ravi”;
  last = “CHUGH” };

This creates a **new** record value

- The immutable record \texttt{p0} is unchanged
What does greet evaluate to?

(a) Syntax Error
(b) Type Error
(c) person -> string
(d) {first:string; last:string} -> string
(e) {first:string} -> string
What does `greet` evaluate to?

```ocaml
# type person = { first : string ; last : string } ;;
# let greet p =
  "Hello " ^ p.first ;;
```

In OCaml, record types identified by name

(c) `person` -> `string`
(d) `{first:string; last:string} -> string`
(e) `{first:string} -> string`
What does greet evaluate to?

```ocaml
# type person = { first : string ; last : string } ;;
# let greet p =
  "Hello " ^ p.first ;;
```

In some other PLs, record types described by “structural types”

(d) `{first:string; last:string} -> string
(e) `{first:string} -> string
What does greet evaluate to?

```plaintext
# type person = { first : string 
                   ; last   : string } ;;
# let greet p = 
   "Hello " ^ p.first ;;
```

In some other PLs, record types augmented by subtyping so that extra fields do not have to be mentioned

(e) `{first: string} -> string`
What does greet evaluate to?

```ocaml
# type person = { first : string ; last : string } ;;
# let greet p =
  "Hello " ^ p.first ;;
```

- Record types in (core) OCaml are nominal, not structural, with no subtyping
- Makes type inference much easier
- We’ll see structural types and subtyping in Scala
No “inexhaustive pattern match warning” because there is only one “shape” for a person value
# type person = { first : string ; last : string } ;;

# let greet p =

match p with {first = f} ->

"Hello " ^ f ;;

Can omit fields that are not needed

No “inexhaustive pattern match warning” because the single field first determines entire type for expression
Like any other pattern, record pattern can appear:

- in formal parameter
- on left-hand side of let-expression
OCaml has some **imperative** features!

**Why?**

- Sometimes even a really smart compiler cannot figure out how to efficiently compile the code
- Sometimes you want to maintain some “state” in your program, and life’s too short to rewrite the program to pass the “state” around throughout every function
  - e.g. think about how to implement a global counter function

**In such situations, a dash of imperative programming can help**
KEEP CALM AND JOIN THE DARK SIDE
# type counter = { mutable curVal : int } ;;
# let c = { curVal = 0 } ;;
# c.curVal ;;
- : int = 0
# c.curVal <- c.curVal + 1 ;;
- : unit = ()
# c.curVal ;;
- : int = 1

Assignment performs in-place update

Beyond this “side effect”, produces a dummy result (i.e. the unit value)
What does `res` evaluate to?

(a) Syntax Error
(b) Type Error

(c) `(0, 0, 1) : (int*int*int)`
(d) `(0, 1, 1) : (int*int*int)`
(e) `(1, 1, 1) : (int*int*int)`
What does res evaluate to?

```haskell
# type counter = { mutable curVal : int } ;;
# let c = { curVal = 0 } ;;
# let c2 = { curVal = 0 } ;;
# let c3 = c2 ;;

c2.curVal and c3.curVal are aliases for the same reference cell

Proceed at your own risk ...
Reasoning with Mutation

```ocaml
# type counter = { mutable curVal : int } ;;
# let c = { curVal = 0 } ;;
# let f x = (* some well-typed expr *) ;;
# let incCounter () =
  let _ = c.curVal <- c.curVal + 1 in
  c.curVal ;;
val incCounter : unit -> int = <fun>
# let _ = incCounter () in c.curVal ;;
```

What does this produce? 0, 1, 2, ... ?

Who knows ?!? Depends on what f does!
Reasoning with Mutation

**Cannot reason about function locally!**

```ocaml
let incCounter () =
  let _ = c.curVal <- c.curVal + 1 in
  c.curVal

val incCounter : unit -> int = <fun>
```

Colbert’s Immutability Principle
Reasoning with Mutation

Cannot reason about function locally!

```ocaml
let incCounter () =
  let _ = c.curVal <- c.curVal + 1 in
  c.curVal

val incCounter : unit -> int = <fun>
```

- At least try to “encapsulate” stateful (imperative) features as much as possible
- Local let-bindings to the rescue!
Reasoning with Mutation

```ml
# type counter = { mutable curVal : int } ;;
# let mkCounter () =
  let c = { curVal = 0 } in
  fun () ->
    let _ = c.curVal <- c.curVal + 1 in
    c.curVal ;;
val mkCounter : unit -> unit -> int = <fun>

# let incCounter = mkCounter () ;;
# incCounter () ;;
- : int = 1
# incCounter () ;;
- : int = 2
```
• Standard library provides a built-in
  “reference type” for a common use-case

```ocaml
type 'a ref = { mutable contents: 'a }
```

Create new reference:

```ocaml
# let c = ref 0 ;;
val c : int ref = {contents = 0}
```

```ocaml
# let c = { contents = 0 } ;;
val c : int ref = {contents = 0}
```
Standard library provides a built-in “reference type” for a common use-case

```ocaml
type 'a ref = { mutable contents: 'a }
```

Dereference:

```ocaml
# !c ;;
- int = 0
```

```ocaml
# c.contents ;;
- int = 0
```
Standard library provides a built-in "reference type" for a common use-case

```ocaml
type 'a ref = { mutable contents: 'a }
```

Set, or update, reference:

```ocaml
# c := !c + 1 ;;
- unit = ()
```

```ocaml
# c.contents <- c.contents + 1 ;;
- unit = ()
```
• Standard library provides a built-in “reference type” for a common use-case

```ocaml
type 'a ref = { mutable contents: 'a }
```

```ocaml
# ref ;;
- : 'a -> 'a ref = <fun>

# (!) ;;
- : 'a ref -> 'a = <fun>

# (:=) ;;
- : 'a ref -> 'a -> unit = <fun>
```
# let mkCounter () =
let c = ref 0 in
fun () ->
  let _ = c := !c + 1 in
!c ;;
val mkCounter : unit -> unit -> int = <fun>

# let incCounter = mkCounter () ;;
# incCounter () ;;
- : int = 1

# incCounter () ;;
- : int = 2
Mutation

- Sometimes imperative features can be tremendously handy ...

- But, use them with discretion!

<table>
<thead>
<tr>
<th>OCaml</th>
<th>C, Java, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immutability by default + Careful use of mutability</td>
<td>Mutability by default + Jump through hoops to enforce immutability</td>
</tr>
</tbody>
</table>
We have studied the core features of ML

If (hopefully *when*!) you continue to use OCaml, you’ll discover loads of others

- Mutually recursive functions and types
- Exception handling
- Labeled function arguments
- Optional function arguments
- Polymorphic variants
- Modules
- Functors (functions from modules to modules)
- Objects (though quite different from most languages)
- ...

Time to Say Goodbye to OCaml
Time to Say Goodbye to OCaml

- But don’t be too sad...
- We’ll find lots of core ML features in Scala!

Image from:
algorithmssinaction.blogspot.com/2013/03/functional-programming-in-scala.html