Lecture 24 — OCaml type-checking, part 3; given by Susannah Johnson

- **Imperative features**
  - **Problems with polymorphism**
  - **The value restriction**

Imperative operations in OCaml

- **OCaml variables are not assignable — once a variable gets its value, that value does not change.**
- **However, there is a type for pointer-like values that are assignable. These are called *references*.**
- **The type of pointers to values of type τ is "τ `ref`".**
- **Operations on ref types are:**

$$
\begin{aligned}
\texttt{ref} &: \forall\tau.\tau \rightarrow \tau\ \texttt{ref} \\
\texttt{!} &: \forall\tau.\tau\ \texttt{ref} \rightarrow \tau \\
\texttt{:=} &: \forall\tau.\tau\ \texttt{ref}\ \texttt{*}\ \tau \rightarrow \texttt{unit} \\
\texttt{;} &: \forall\tau.\ \texttt{unit}\ \texttt{*}\ \tau \rightarrow \tau
\end{aligned}
$$

Imperative operations in OCaml

- **With ref types, OCaml users can use ordinary imperative functions. OCaml also has a while loop:**

```
let r = ref 1
in while !r < 11  do
      print_int !r ;
      print_string " " ;
      r := !r+1
   done ;;
```

Using ref values in higher-order functions

- The combination of higher-order functions and imperative values allows for some interesting examples. This function produces a random number generator, generating a number between 1 and 10 each time it's called:

```
let rand i = let r = ref i
             in fun () -> (r := (!r) * 7 mod 11; !r);;
let gen = rand 1;;
gen();;
gen();;
```

Semantics of imperative operations

- The expression above *cannot* be understood using the substitution model. It requires the environment model.
  - The value of $!r$ changes over time, so any substitution of a static value for $!r$ would be incorrect
  - Further, the location referenced by r could also be changed, so substituting a static (location) value for r would be incorrect, as well!
- More generally, since this allows aliasing, just like MiniJava, it requires a two-level state.
  - We place values that have been referenced on the heap

Evaluation rules

$(\text{Ref}) \quad \texttt{ref}\ e,\ (\rho,\ \omega) \Downarrow \texttt{Loc}\ l, \omega'[l \mapsto v]\ (l \text{ fresh})$

$e,\ (\rho,\ \omega) \Downarrow v, \omega'$

$(\text{Deref}) \quad !e,\ (\rho,\ \omega) \Downarrow \omega'(l), \omega'$

$e,\ (\rho,\ \omega) \Downarrow \texttt{Loc}\ l, \omega'$

Evaluation rules (cont.)

$$(\textsc{Assign}) \quad e_1 := e_2,\ (\rho,\ \omega) \Downarrow (), \omega''[l \mapsto v]$$

$$e_1,\ (\rho,\ \omega) \Downarrow \texttt{Loc}\ l, \omega'$$

$$e_2,\ (\rho,\ \omega') \Downarrow v, \omega''$$

$$(\textsc{Seq}) \quad e_1; e_2,\ (\rho,\ \omega) \Downarrow v, \omega''$$

$$e_1,\ (\rho,\ \omega) \Downarrow (), \omega'$$

$$e_2,\ (\rho,\ \omega') \Downarrow v, \omega''$$

Explicit polymorphic type system

- Γ **is a map from variables to type schemes.** τ**,** τ'**,** τ'' **are types.**

(Const) $\Gamma \vdash$ **Int i : int**

(Var) $\Gamma \vdash a : \Gamma(a)$
($\Gamma(a)$ **a type)**

(Fun) $\Gamma \vdash$ **fun** $a{:}\tau$ **->** $e : \tau \to \tau'$
$\Gamma[a{:}\tau] \vdash e : \tau'$

(δ**)** $\Gamma \vdash e \oplus e' : \tau''$
$\Gamma \vdash e : \tau$
$\Gamma \vdash e' : \tau'$

(App) $\Gamma \vdash e\ e' : \tau'$
$\Gamma \vdash e : \tau \to \tau'$
$\Gamma \vdash e' : \tau$

(True) $\Gamma \vdash$ **true : bool**

(False) $\Gamma \vdash$ **false : bool**

(PolyVar) $\Gamma \vdash a[\tau] : \tau$
where $\tau \leq \Gamma(a)$
($\Gamma(a)$ **a type scheme)**

(Let) $\Gamma \vdash$ **let** $a{:}\tau$ **=** e `in` $e' : \tau'$
$\Gamma \vdash e : \tau$
$\Gamma[a{:}\mathbf{GEN}_\Gamma(\tau)] \vdash e' : \tau'$

Type-checking references

- **How about references? How should they be typed?**

(Ref) $\Gamma \vdash \textsf{ref}\ x : \tau\ \textsf{ref}$

$\quad\Gamma \vdash x : \tau$

(Deref) $\Gamma \vdash !x : \tau$

$\quad\Gamma \vdash x : \tau\ \textsf{ref}$

(Assign) $\Gamma \vdash x := e : \textsf{unit}$

$\quad\Gamma \vdash x : \tau\ \textsf{ref}$

$\quad\Gamma \vdash e : \tau$

(Seq) $\Gamma \vdash e_1; e_2 : \tau$

$\quad\Gamma \vdash e_1 : \textsf{unit}$

$\quad\Gamma \vdash e_2 : \tau$

Polymorphism and references

- **Prove the following judgment:**

$\emptyset \vdash$ `let i = fun x -> x`
`in let fp = ref i in (fp := not;  (!fp) 5)` : **int**

$\emptyset \vdash$ `fun x -> x` : $\alpha \to \alpha$

$\{x{:}\alpha\} \vdash x$: α

$\{i{:}\ \forall\alpha.\alpha \to \alpha\ \} \vdash$ `let fp = ref i in (fp := not; (!fp) 5)` : **int**

$\{i{:}\ \forall\alpha.\alpha \to \alpha\ \} \vdash$ `ref i` : **ref** $\forall\alpha.\alpha \to \alpha$

$\{i{:}\ \forall\alpha.\alpha \to \alpha,\ fp{:}\textbf{ref}\ \forall\alpha.\alpha \to \alpha\ \} \vdash$ `fp := not; (!fp) 5` : **int**

$\{i{:}\ \forall\alpha.\alpha \to \alpha,\ fp{:}\textbf{ref}\ \forall\alpha.\alpha \to \alpha\ \} \vdash$ `fp := not` : **unit**

$\{i{:}\ \forall\alpha.\alpha \to \alpha,\ fp{:}\textbf{ref}\ \forall\alpha.\alpha \to \alpha\ \} \vdash$ `fp` : **ref bool** $\to$ **bool**

$\{i{:}\ \forall\alpha.\alpha \to \alpha,\ fp{:}\textbf{ref}\ \forall\alpha.\alpha \to \alpha\ \} \vdash$ `not` : **bool** $\to$ **bool**

$\{i{:}\ \forall\alpha.\alpha \to \alpha,\ fp{:}\textbf{ref}\ \forall\alpha.\alpha \to \alpha\ \} \vdash$ `(!fp) 5` : **int**

$\{i{:}\ \forall\alpha.\alpha \to \alpha,\ fp{:}\textbf{ref}\ \forall\alpha.\alpha \to \alpha\ \} \vdash$ `!fp` : **int** $\to$ **int**

$\{i{:}\ \forall\alpha.\alpha \to \alpha,\ fp{:}\textbf{ref}\ \forall\alpha.\alpha \to \alpha\ \} \vdash$ `fp` : **ref (int** $\to$ **int)**

$\{i{:}\ \forall\alpha.\alpha \to \alpha,\ fp{:}\textbf{ref}\ \forall\alpha.\alpha \to \alpha\ \} \vdash$ `5` : **int**

Polymorphism and references

- **The above term type-checks in the polymorphic type system, but it has a serious run-time type error: it applies a boolean function (`not`) to an integer argument.**
- **Treating imperative operations as having normal polymorphic types causes a problem. How can the type system be fixed?**
- **Easiest method: do not generalize reference expressions at all, i.e. make all reference types monomorphic.**
- **Method used by OCaml: "value restriction"**

The value restriction

- It turns out that the problem typified by the example above can be eliminated if the let-bound expression cannot create references when it is evaluated.
- However, it is difficult to determine statically whether an expression will create a reference.
- So the rule used is (roughly): a let-bound expression can be polymorphic only if it *does no computation*.
- This sounds worse than it is. Recall the notion of a "value" from the substitution model.
- *Value restriction*: The type of an expression in a let can be generalized only if the expression is a syntactic value — a constant or abstraction (function definition).

The value restriction (cont.)

Which of the following are disallowed under value restriction? (starred are disallowed)

```
let f = List.map (fun x->x);; **
```

```
let f = fun lis -> List.map (fun x->x) lis;;
```

```
let f = ref (fun x -> x + 2);;
```

```
let f = ref (fun x -> x);; **
```

```
let f = ref (fun x -> 2);; ??
```

The value restriction (cont.)

- **The good:**
  - **Polymorphic expressions almost always define functions. This means the value restriction is not that severe, because**

    `let` x `=` $e\,e'$ `in` e''

    can just be changed to

    `let` x `= fun` z `->` $(e\,e')z$ `in` e''.

  - **On the other hand, the example above cannot be changed in this way (since `ref i` is not a function). This is good — that expression shouldn't type-check!**

The value restriction (cont.)

- **The bad:**
  - **The value restriction can be very annoying, especially when using a programming style that uses use of higher-order functions.**
  - **For example, this is illegal:**

```
let f = List.map (fun x->x)
in (f [1], f [true]);;
```

even though this is legal:

```
let f = fun lis -> List.map (fun x->x) lis
in (f [1], f [true]);;
```

The value restriction (cont.)

- **OCaml uses a modified version of the value restriction that is a little less restrictive. (It is too complicated to explain here.) It makes it legal to write** `let f = List.map (fun x->x);;`**. But note that we lose polymorphic behavior in this case:**

```
# let mapid = List.map (fun x -> x);;
val mapid : '_a list -> '_a list = <fun>
# mapid [1;2];;
- : int list = [1; 2]
# mapid [true;false];;
Characters 7-11:
  mapid [true;false];;
This expression has type bool but is here used with type int
```

Type-checking summary

- Two major trends in programming in recent years are the increasing use of dynamically-typed languages (e.g. JavaScript, Python), and the increasing sophistication of static type systems (OCaml, Scala, Java generics, C++).
  - Dynamically-typed languages are (1) more flexible, and (2) easier to implement.
  - Statically-typed languages are (1) safer to use (since the types provide a form of "sanity check"), and (2) more efficient.
- Continuing research is attempting to combine the advantages of these two classes of languages in a single language, or at least simplify the transition from one to the other. But for now, there is still a wide gulf between these two worlds.

Wrap-up

- **Today we discussed:**
  - **Imperative features of OCaml**
  - **Value restriction**
- **We discussed it because:**
  - **References introduce a level of indirection that makes naive type-checking unsafe**
  - **Value restriction is an examples of how we cannot entirely "fix" type-checking to accept every (otherwise correct) program**
- **What to do now:**
  - **MP12**