Lecture 23 — OCaml type-checking, part 2

- We will look at the actual OCaml type system, which is polymorphic, in the same way as we did for the monomorphic version. We will define explicitly- and implicitly-typed versions (OCaml$_{ep}$ and OCaml$_{ip}$), and discuss type-checking and type inference.

- It turns out that the key to polymorphism in OCaml is the treatment of `let`.

Monomorphic type system and OCaml

- **In lecture 22, we saw several untypable terms:**

```
fun f -> f f

fun f -> fun g -> g (f 1) (f true)

(fun f -> f f)(fun i -> i)

let f = fun i -> i in f f
```

- **This is consistent with actual OCaml, except for one case...**

Monomorphic type system and OCaml

```
# fun f -> f f;;
Characters 11-12:
  fun f -> f f;;
This expression has type 'a -> 'b but is here used with type 'a

# fun f -> fun g -> g (f 1) (f true);;
Characters 29-33:
  fun f -> fun g -> g (f 1) (f true);;
                               ^^^^
This expression has type bool but is here used with type int

# (fun f -> f f)(fun i -> i);;
Characters 12-13:
  (fun f -> f f)(fun i -> i);;
This expression has type 'a -> 'b but is here used with type 'a

# let f = fun i -> i in f f;;
- : '_a -> '_a = <fun>
```

let polymorphism

- The expression `(fun f -> f f)(fun i -> i)` is not typable in OCaml. However, the following "equivalent" term is typable:

```
let f = fun i -> i in f f
```

- In OCaml, `let` expressions are treated as if their type rule were:

(Let (*proposed*)) $$\frac{\Gamma \vdash e'[e/x] : \tau}{\Gamma \vdash \textbf{let } x = e \texttt{ in } e' : \tau}$$

- Then, we obtain the above judgement because we can type this term (in the monomorphic system):

```
(fun i -> i)(fun i -> i)
```

let polymorphism (cont.)

- **Note that there is no way we can use an analogous rule with abstractions, because we don't normally know** *statically* **what their argument is.**

- **The rule just given works well, but it would be very expensive to implement. Instead, we will assign** *polymorphic* **types to let-bound variables, and type-check to make sure they are used consistently.**

Polymorphic types

- **We now add *quantified* variables to types. These are type variables that can be replaced by any (monomorphic) type. E.g.**
  - `hd`: $\forall\alpha.\alpha$ **list** $\rightarrow \alpha$
  - `map`: $\forall\alpha,\beta.(\alpha \rightarrow \beta) \rightarrow \alpha$ **list** $\rightarrow \beta$ **list**
  - `nth`: $\forall\alpha.\alpha$ **list** $\rightarrow$ **int** $\rightarrow \alpha$
- **A type with variables quantified (not necessarily *all* of its variables) is called a *type scheme*. (Normal types are a subset of type schemes.)**

Explicitly-typed, polymorphic OCaml (Ocaml$_{ep}$)

- **We declare types of variables bound in `let` and `fun` expressions:**
  - **`let`-bound variables' types can contain *new* type variables (which we should think of as being quantified);**
  - **`fun`-bound variables' types *can only contain type variables if those were introduced by an enclosing* `let` *expression*.**
- **Variables introduced in `let` expressions are polymorphic. *When they are used*, we have to say what specific type they are used at; this can be a monotype, or it can contain variables that are used in the types of enclosing declarations.**

Ocaml$_{ep}$

Concrete syntax:

$typeterm \rightarrow$ `int` $|$ `bool` $|$ $typeterm$ `->` $typeterm$ $|$ id

$exp \rightarrow int$ $|$ `true` $|$ `false` $|$ id $|$ $exp\ exp$ $|$ `fun` id `:` $typeterm$ `->` exp
$\quad |$ `let` id `:` $typeterm$ `=` exp `in` exp $|$ id`[`$typeterm$`]`

Abstract syntax:

```
type typeterm = IntType | BoolType | Typevar of string
              | FunType of typeterm * typeterm

type exp = Int of int | True | False | Var of string
         | Operation of exp * binary_operation * exp
         | App of exp * exp | Fun of string * typeterm * exp
         | Let of id * typeterm * exp * exp
         | Polyvar of string * typeterm
```

Ocaml$_{ep}$ examples

Ocaml: `let id = fun x -> x in id 4`

Ocaml$_{ep}$:

```
let id:(alpha->alpha) = fun x:alpha -> x
in id[int->int] 4
```

OCaml:

```
let g = fun a -> fun b -> a
in let id = fun x -> x
   in g (id 4) (id true)
```

Ocaml$_{ep}$:

```
let g:alpha->beta->alpha = fun a:alpha -> fun b:beta -> a
in let id:(gamma->gamma) = fun x:gamma -> x
   in g[int->(bool->int)] (id[int->int] 4) (id[bool->bool] true)
```

Ocaml$_{ep}$ examples (cont.)

OCaml:
```
let incr = fun i:int -> i+1
in let double = fun g -> fun x -> g (g x)
   in double incr 4
```

Ocaml$_{ep}$:

```
let incr:(int->int) = fun i:int -> i+1
in let double:((alpha->alpha)->(alpha->alpha)) =
           fun g:(alpha->alpha) -> fun x:alpha -> g (g x)
   in double[(int->int)->(int->int)] incr 4
```

OCaml:
```
let sub = fun i -> fun j -> i-j
in let reverse = fun f = fun x -> fun y -> f y x
   in reverse sub 3 5
```

Ocaml$_{ep}$:

```
let sub:(int->int->int) = fun i:int -> fun j:int -> i-j
in let reverse:((alpha->(beta->gamma))->(beta->(alpha->gamma))) =
      fun f:(alpha->(beta->gamma)) =
              fun x:alpha -> fun y:alpha -> f y x
   in reverse[(int->(int->int))->(int->(int->int))] sub 3 5
```

Explicitly-typed, polymorphic OCaml exercises

```
(let id = fun x -> x in id id) 5
```

```
let g = fun a -> fun b -> a
in let incr = fun x -> x+1 in g incr (incr 5)
```

```
let apply = fun g -> fun x -> g x
in apply (fun x -> x) 3
```

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:τ **->** e : $\tau \rightarrow \tau'$
$\Gamma[a{:}\tau] \vdash e : \tau'$

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

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

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

Generalization and instantiation

- In the PolyVar rule, we write: $\tau \leq \Gamma(a)$. This means τ is an *instance* of $\Gamma(a)$, i.e. obtained from $\Gamma(a)$ by replacing its quantified variables by types. E.g.
  - $\textbf{int}\rightarrow\textbf{int} \leq \forall\alpha.\alpha \rightarrow \alpha$
  - $\textbf{int}\rightarrow\textbf{int} \leq \forall\alpha.\textbf{int} \rightarrow \alpha$
  - $\textbf{int}\rightarrow\textbf{int} \leq \textbf{int}\rightarrow\textbf{int}$
  - $\textbf{int}\rightarrow \beta \leq \forall\alpha.\alpha \rightarrow \beta$
  - $\textbf{int}\rightarrow \gamma \leq \forall\alpha.\alpha \rightarrow \gamma$
  - $\textbf{int}\rightarrow \beta \not\leq \forall\alpha.\alpha \rightarrow \gamma$

Note: nothing in the type scheme can change except the replacement of bound type variables.

Generalization and instantiation (cont.)

- **GEN$_\Gamma(\tau)$ is *usually* just τ with all its variable generalized, e.g. GEN$_\Gamma(\alpha \to (\beta \to \textbf{int}))$ is *usually* $\forall\alpha,\beta.\ \alpha \to (\beta \to \textbf{int})$**
  - **We will see a more accurate definition later.**

Example

```
let id:(alpha->alpha) = fun x:alpha -> x
in id[int->int] 4
```

Example

```
let g:alpha->beta->alpha = fun a:alpha -> fun b:beta -> a
in let id:(gamma->gamma) = fun x:gamma -> x
   in g[int->bool->int] (id[int->int] 4) (id[bool->bool] true)
```

Example

```
let g:(int->beta)->beta = fun f:(int->beta) -> f 0
   in g[(int->bool)->bool] (fun x:int -> x>0)
```

Type-checking (MP 12)

- As with the explicitly-typed system from lecture 22, type-*checking* (that is, checking whether the type declarations in an expression in the explicitly-typed system are valid) is fairly simple.
- The algorithm is nearly identical to the one from lecture 22, except for handling let expressions and polymorphic variables. Γ is, as above, a map from program variables to type schemes.

Type-checking (MP 12)

```
tcheck $t$ $\Gamma$ = match $t$ with
        $i \rightarrow$ int
      | true $\rightarrow$ bool
      | false $\rightarrow$ bool
      | $x \rightarrow \Gamma\ x$        (non-polymorphic $x$)
      | fun $x$ : $\tau$ -> $e \rightarrow (\tau \rightarrow$ tcheck $e\ \Gamma[\tau/x])$
      | $e_1\ e_2 \rightarrow$ let $\tau_1 =$ tcheck $e_1\ \Gamma$
                  and $\tau_2 =$ tcheck $e_2\ \Gamma$
                  in if $\tau_1 = \tau_1' \rightarrow \tau_1''$ and $\tau_2 = \tau_1'$
                       then $\tau_1''$ else error
      | $x[\tau] \rightarrow \tau$, if $\tau \leq \Gamma(x)$        (polymorphic $x$)
      | let $x$ : $\tau$ = $e$ in $e' \rightarrow$ if $\tau =$ tcheck $e\ \Gamma$
                  then tcheck $e'\ \Gamma[\text{GEN}_\Gamma(\tau)/x]$
                  else type error
```

Technicality #1: Generalization

- **Ex: Prove this judgment (where `incr` has type int→int):**

```
∅ ⊢ let f:(a->a) = fun x:a ->
              let g:((a->b)->b) = fun y:(a->b) -> y x
              in g[(int->int)->int] incr
    in f[bool->bool] true                       : bool
```

Technicality #1: Generalization (cont.)

- One way to look at the problem is that we should not generalize type variables that were introduced in enclosing `lets` — we should just generalize the ones added in this `let`.
- So, $\text{GEN}_\Gamma(\tau)$ is actually defined as: τ with all variables generalized *except* those that occur (free) in Γ.
- How does this make our proof of the previous type judgment wrong?

Implicitly-typed, polymorphic OCaml (aka Ocaml$_{ip}$) (aka OCaml)

- **As with the monomorphically-typed system, we can obtain an implicitly-typed polymorphic system easily: just remove all the type declarations. The typing rules remain the same.**

Explicit		**Implicit**	
(Fun)	$\Gamma \vdash$ **fun** a:τ `->` e **:** $\tau \rightarrow \tau'$ $\Gamma[a{:}\tau] \vdash e$ **:** τ'	**(Fun)**	$\Gamma \vdash$ **fun** a `->` e **:** $\tau \rightarrow \tau'$ $\Gamma[a{:}\tau] \vdash e$ **:** τ'
(Let)	$\Gamma \vdash$ **let** a:τ = e `in` e' **:** τ' $\Gamma \vdash e$ **:** τ $\Gamma[a{:}\mathbf{GEN}_\Gamma(\tau)] \vdash e'$ **:** τ'	**(Let)**	$\Gamma \vdash$ **let** a = e `in` e' **:** τ' $\Gamma \vdash e$ **:** τ $\Gamma[a{:}\mathbf{GEN}_\Gamma(\tau)] \vdash e'$ **:** τ'
(PolyVar)	$\Gamma \vdash a[\tau]$ **:** τ **where** $\tau \leq \Gamma(a)$ **(**$\Gamma(a)$ **a type scheme)**	**(PolyVar)**	$\Gamma \vdash a$ **:** τ **where** $\tau \leq \Gamma(a)$ **(**$\Gamma(a)$ **a type scheme)**

Type inference

- As with the monomorphic system, we an view type inference either as finding a proof in the type system for Ocaml$_{ip}$, or as finding a way to fill in variable declarations so as to obtain a proof in the type system for Ocaml$_{ep}$.
- Again, type inference is a matter of writing down all the constraints on the types of the variables that are implied by the use of the variables in the expression. If they are contradictory, then there is a type error.
  - Major technicality: need to take into account that names introduced by `let` are polymorphic. E.g. `f f` does not necessarily represent a contradiction, as in the monomorphic system.

Type inference

- Robin Milner presented a (relatively) efficient algorithm for type inference, called “algorithm W.”
- Algorithm W is actually exponential, because the type of a term can be exponential in its size. Here is an example:

```
let compose f g = fun x -> f (g x);;
let pair x = (x, x);;
val pair : 'a -> 'a * 'a = <fun>
compose pair pair;;
- : 'a -> ('a * 'a) * ('a * 'a) = <fun>
compose pair (compose pair pair);;
- : 'a -> (('a * 'a) * ('a * 'a)) * (('a * 'a) * ('a * 'a)) = <fun>
```

- In practice, as you know, type inference in OCaml is quite efficient.

Wrap-up

- Today we discussed polymorphically-typed OCaml, both with and without explicit type declarations. The key difference from the monomorphic system is that `let` introduces quantified types, which may be instantiated differently at each use within the `let`.

- In the next class, we will discuss side effecting operations in OCaml and the notion of `ref` types. We are discussing this now because it has a major impact on the type system.