

Lecture 24 — OCaml type-checking, part 3

- Let-polymorphism
- Explicitly-typed, polymorphic OCaml
 - Types and type schemes
 - Type system
 - Type-checking (MP9)
- Generalization

Let polymorphism (from lecture 23)

- The monomorphic type system from lectures 22 and 23 is correct w.r.t. OCaml, but doesn't not have let expressions.
- In OCaml, the names defined in let expressions — *and only those* — are polymorphic.
- The idea is to type let expressions by performing substitution first:

$$\begin{array}{l} \text{(Let (proposed))} \quad \Gamma \vdash \mathbf{let} \ x = e \ \mathbf{in} \ e' : \tau \\ \Gamma \vdash e'[e/x] : \tau \end{array}$$

- The only problem is that this is a very expensive rule to implement. We need to find a type system that gets us the same results without the expense.

Let polymorphism (cont.)

- For example, consider this term:

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

- Use the substitution model to see what it does:
- The “equivalent” expression `(fun f -> f f)(fun x -> x)` is not typeable in OCaml (i.e. it is not typable in the monomorphic type system of lecture 22–23).
- However, the equivalent expression `(f f) [(fun x->x)/f]` is typable:

```
(fun x -> x)(fun x -> x)
```

Polymorphic types

- We now add *quantified* variables to types. These are type variables that can be replaced by any (monomorphic) type. E.g.
 - $\text{hd}: \forall \alpha. \alpha \text{ list} \rightarrow \alpha$
 - $\text{map}: \forall \alpha, \beta. (\alpha \rightarrow \beta) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}$
 - $\text{nth}: \forall \alpha. \alpha \text{ list} \rightarrow \text{int} \rightarrow \alpha$
- 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

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

Explicitly-typed, polymorphic OCaml

Concrete syntax:

typeterm $\rightarrow$ int | bool | *typeterm* $\rightarrow$ *typeterm* | *id*

exp $\rightarrow$ *int* | true | false | *id* | *exp exp* | fun *id* : *typeterm* $\rightarrow$ *exp*
| 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*

Explicitly-typed, polymorphic OCaml examples

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

With explicit types:

```
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)
```

With explicit types:

```
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)
```

Explicitly-typed, polymorphic OCaml examples

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

```
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
```

```
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) $\Gamma \vdash \text{Int } i : \text{int}$

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

(Fun) $\Gamma \vdash \text{fun } a:\tau \rightarrow e : \tau \rightarrow \tau'$
 $\Gamma[a:\tau] \vdash e : \tau'$

(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) $\Gamma \vdash \text{true} : \text{bool}$

(False) $\Gamma \vdash \text{false} : \text{bool}$

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

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

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.

- $\text{int} \rightarrow \text{int} \leq \forall \alpha. \alpha \rightarrow \alpha$
- $\text{int} \rightarrow \text{int} \leq \forall \alpha. \text{int} \rightarrow \alpha$
- $\text{int} \rightarrow \text{int} \leq \text{int} \rightarrow \text{int}$
- $\text{int} \rightarrow \beta \leq \forall \alpha. \alpha \rightarrow \beta$
- $\text{int} \rightarrow \gamma \leq \forall \alpha. \alpha \rightarrow \gamma$
- $\text{int} \rightarrow \beta \not\leq \forall \alpha. \alpha \rightarrow \gamma$

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

Generalization and instantiation (cont.)

- $\text{GEN}_{\Gamma}(\tau)$ is *usually* just τ with all its variable generalized, e.g. $\text{GEN}_{\Gamma}(\alpha \rightarrow (\beta \rightarrow \text{int}))$ is *usually* $\forall \alpha, \beta. \alpha \rightarrow (\beta \rightarrow \text{int})$
- 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 9)

- As with the explicitly-typed system from lecture 23, *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 23, except for handling let expressions and polymorphic variables. Γ is, as above, a map from program variables to type schemes.

Type-checking (MP 9)

$\text{tcheck } t \ \Gamma = \text{match } t \text{ with}$

- $i \rightarrow \text{int}$
- | $\text{true} \rightarrow \text{bool}$
- | $\text{false} \rightarrow \text{bool}$
- | $x \rightarrow \Gamma \ x \quad (\text{non-polymorphic } x)$
- | $\text{fun } x : \tau \rightarrow e \rightarrow (\tau \rightarrow \text{tcheck } e \ \Gamma[\tau/x])$
- | $e_1 \ e_2 \rightarrow \text{let } \tau_1 = \text{tcheck } e_1 \ \Gamma$
 - and $\tau_2 = \text{tcheck } e_2 \ \Gamma$
 - in if $\tau_1 = \tau'_1 \rightarrow \tau''_1$ and $\tau_2 = \tau'_1$
 - then τ''_1 else error
- | $x[\tau] \rightarrow \tau, \text{ if } \tau \leq \Gamma(x) \quad (\text{polymorphic } x)$
- | $\text{let } x : \tau = e \text{ in } e' \rightarrow \text{if } \tau = \text{tcheck } e \ \Gamma$
 - then $\text{tcheck } e' \ \Gamma[\text{GEN}_\Gamma(\tau)/x]$
 - 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 we added ourselves.
- 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 (i.e. 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.
- As with the monomorphic system, the implicitly-typed and explicitly-typed systems have identical power: If a type judgment can be proven in the explicitly-typed system, the same proof works in the implicitly-typed system (after removing type declarations from all expressions), and if a type judgement can be proven in the implicitly-typed system, the proof provides types that can be inserted as declarations to turn the expression into an explicitly-typed expression.
- As with the monomorphic system, the differences are: (1) the implicit system is much easier to use; and (2) simple type-checking (for the explicit system) is exchanged for quite complex type-inference (involving creating and solving sets of constraints).