

Lecture 23 — OCaml type-checking, part 2

- Review of *monomorphic, explicitly-typed* type system for OCaml
 - Typable and untypable terms
 - Relation to OCaml type system
 - Review of type inference
- Let polymorphism
- Unification

Monomorphic, explicitly-typed OCaml

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

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

Type rules (where Γ is a mapping from variables to types, and each binary operation $\oplus$ is assumed to have a given type $\tau \rightarrow \tau' \rightarrow \tau''$):

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

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

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

Examples

$\emptyset \vdash \text{fun } f:(\text{int} \rightarrow \text{int}) \rightarrow f\ 3 : (\text{int} \rightarrow \text{int}) \rightarrow \text{int}$

$\emptyset \vdash \text{fun } f:(\text{int} \rightarrow \text{int}) \rightarrow \text{fun } x:\text{int} \rightarrow f\ (f\ x)$
 $\quad : (\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})$

Examples of untypable terms

- These are terms for which no type declarations can be given that would allow the term to be typed:

$\emptyset \vdash \text{fun } f: ? \rightarrow f \ f$

$\emptyset \vdash \text{fun } f: ? \rightarrow \text{fun } g: ? \rightarrow g \ (f \ 1) \ (f \ \text{true})$

$\emptyset \vdash (\text{fun } f: ? \rightarrow f \ f)(\text{fun } i: ? \rightarrow i)$

Type-checking

- Given an explicitly-typed term t , **tcheck** determines whether t is type-correct and what its correct type is. Γ is a type environment mapping program variables to types.

```
tcheck  $t$   $\Gamma$  = match  $t$  with  
   $i \rightarrow$  int  
| true  $\rightarrow$  bool  
| false  $\rightarrow$  bool  
|  $x \rightarrow \Gamma$   $x$   
| fun  $x : \tau \rightarrow e \rightarrow (\tau \rightarrow$  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
```

Implicitly-typed OCaml

- If we omit the declarations of lambda-bound variables, the language is more similar to OCaml. The type system barely changes: just omit the type declarations:

$$\begin{array}{l} \Gamma \vdash \text{fun } a \rightarrow e : \tau \rightarrow \tau' \\ \Gamma[a:\tau] \vdash e : \tau' \end{array}$$

- These systems are equivalent, since the following transformation converts proofs in one system to proofs in the other:

$$\begin{array}{c} \text{Implicit} \\ \hline \Gamma \vdash \text{fun } a \rightarrow e : \tau \rightarrow \tau' \\ \Gamma[a:\tau] \vdash e : \tau' \end{array} \quad \Leftrightarrow \quad \begin{array}{c} \text{Explicit} \\ \hline \Gamma \vdash \text{fun } a : \tau \rightarrow e : \tau \rightarrow \tau' \\ \Gamma[a:\tau] \vdash e : \tau' \end{array}$$

- So what's the difference? With the explicitly-typed system, we can *check* types — which is really simple — but with the implicitly-typed languages, we have to *infer* types, which is much harder.

Type inference review

- **Allow type variables in types:** `type t = ... | Typevar of string`
- **To infer types for t , assign a unique type variable to each program variable, and to each subexpression of t . (Occurrences of variables get the type assigned to that variable.)**
- **Generate constraint set E :** For every subexpression t' of t , add constraints to E , depending upon the form of t' :

$(\text{Int } i)_{\alpha}:$	$\{ \alpha = \text{int} \}$
$\text{true}_{\alpha} \text{ or } \text{false}_{\alpha}:$	$\{ \alpha = \text{bool} \}$
$(e_{\alpha} \oplus e'_{\beta})_{\gamma}:$	$\{ \alpha = \tau, \beta = \tau', \gamma = \tau'' \}$ (where $\oplus$ has type $\tau \rightarrow \tau' \rightarrow \tau''$)
$(\text{fun } x_{\alpha} \rightarrow e_{\beta})_{\gamma}:$	$\{ \gamma = \alpha \rightarrow \beta \}$
$(e_{\alpha} e'_{\beta})_{\gamma}:$	$\{ \alpha = \beta \rightarrow \gamma \}$

Type inference review (cont.)

- “Solve” E by repeatedly performing the following operation:
 - If E contains a single constraint for α , $\alpha = \tau$, replace all other occurrences of α in E by τ .
 - If E contains two constraints for α , $\alpha = \tau$ and $\alpha = \tau'$, unify τ and τ' and apply the resulting substitution to all types in E .

Example of type inference

$\emptyset \vdash \text{fun } f \rightarrow 1 + (f \ 3)$

Example of type inference

$\emptyset \vdash \text{fun } f \rightarrow f \ 3$

Example of type inference

$\emptyset \vdash \text{fun } f \rightarrow f \ (f \ 3)$

Example of type inference

$\emptyset \vdash \text{fun } f \rightarrow \text{fun } g \rightarrow g (f \ 1) (f \ \text{true})$

Type inference and monomorphic type system

- We started with an assignment of a unique type variable to each program variable and each subterm:

$$(\dots(\text{fun } x_\alpha \rightarrow \dots(\text{fun } y_\beta \rightarrow \dots)\dots)\dots)_\gamma$$

- After solving E — if successful — it contains type variables that have exactly one constraint, and may also contain type variables that have *no* constraints.
- Suppose E contains $\alpha = \tau_\alpha$ and $\beta = \tau_\beta$ We want to simply give x and y declarations:

$$(\dots(\text{fun } x : \tau_\alpha \rightarrow \dots(\text{fun } y : \tau_\beta \rightarrow \dots)\dots)\dots)_\gamma$$

- However, τ_α and τ_β may contain (unconstrained) type variables, so are not types in the monomorphic system. So how is this related to the original type system?

Type inference and monomorphic type system

- Answer: we can fill in any types we want for the unconstrained type variables:
 - Assign *any* monotype (type without variables) to each unconstrained variable in E .
 - Replace every occurrence of such a variable by its assigned monotype. Let $\bar{\tau}_\alpha$, etc., be the resulting types.
 - We can show the following is provable (no matter which monotypes were used):

$$\emptyset \vdash \dots \text{fun } x:\bar{\tau}_\alpha \rightarrow \dots \text{fun } y:\bar{\tau}_\beta \rightarrow \dots : \bar{\tau}_\gamma$$

- Thus, the type inference method and the monomorphic type system are essentially equivalent: type inference simply finds a way to fill in declarations of variables, if possible.

Monomorphic type system and OCaml

- We earlier saw three terms that could not be type-checked:

```
fun f -> f f
```

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

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

It follows that the type inference method will not find a solution.

Monomorphic type system and OCaml

- This is consistent with 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 polymorphism

- On the other hand, OCaml does have polymorphic types, meaning one function definition *can* be used with different types of arguments, under the right circumstances.
- The expression `(fun f -> f f)(fun i -> i)` is not typable in OCaml. However, the following term is typable:

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

- We know these terms are “equivalent”: In general (and, more specifically, in dynamically-typed OCaml):

`let  $x=e_1$  in  $e_2 \equiv (\text{fun } x \rightarrow e_2) e_1$`

let polymorphism

- The difference – as we will discuss in the next lecture — is that terms used in let expressions — and only those — are polymorphic.
- More specifically, we can add this rule to our type system:

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

- For example, this judgment is provable in our system:

$$\emptyset \vdash (\text{fun } i \rightarrow i)(\text{fun } i \rightarrow i) : \text{int} \rightarrow \text{int}$$

- However, this 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.

Unification

- Part of the process of solving constraint sets is answering the following question:

If we have two constraints on α — $\alpha = \tau$ and $\alpha = \tau'$ — are they mutually consistent?

- To put the question more concretely: Can we find a *substitution* S — a map from the the variables in τ and τ' to types — such that if we substitute those types for the variables, τ and τ' become equal?

- Example:
$$\begin{aligned}\tau &= \beta \rightarrow (\text{int} \rightarrow \gamma) \\ \tau' &= \text{int} \rightarrow \delta \\ S &= \{\beta \mapsto \text{int}, \delta \mapsto (\text{int} \rightarrow \gamma)\} \\ S\tau &= S\tau' = \text{int} \rightarrow (\text{int} \rightarrow \gamma)\end{aligned}$$

Unification examples

- Will present algorithm for unification next. These are examples to be solved by inspection.

unify($\alpha \rightarrow \beta$, $\text{int} \rightarrow \gamma$)

unify($\alpha \rightarrow (\text{int} \rightarrow \beta)$, $(\text{int} \rightarrow \text{int}) \rightarrow \gamma$)

Unification, formally

- **$\text{unify}(\tau, \tau')$ returns a substitution S — a map from variables to types — such that $S\tau = S\tau'$.**
 - (Technically, it produces a *most general substitution* that unifies τ and τ' — i.e. a substitution that makes no unnecessary assumptions.)
- **$\text{unify}(\tau, \tau')$ can fail in two ways:**
 - If τ is a proper subexpression of τ' (or vice versa), they cannot be unified.
 - If there is a place in the two terms where they have a different constructor, they cannot be unified. For example, `int` and `bool` are not unifiable, and `int  $\rightarrow$   $\tau_1$` and `bool  $\rightarrow$   $\tau_2$` are not unifiable (no matter what τ_1 and τ_2 are).

Unification, formally

- **Unification algorithm:**

$\text{unify}(\alpha, \tau) = \{\alpha \mapsto \tau\}$

(if α is a variable that does not occur in τ)

$\text{unify}(\tau, \alpha) = \{\alpha \mapsto \tau\}$

(if α is a variable that does not occur in τ)

$\text{unify}(\text{int}, \text{int}) = \{\}$

$\text{unify}(\text{bool}, \text{bool}) = \{\}$

$\text{unify}(\tau_1 \rightarrow \tau_2, \tau'_1 \rightarrow \tau'_2) =$

let $S_1 = \text{unify}(\tau_1, \tau'_1)$

in let $S_2 = \text{unify}(S_1\tau_2, S_1\tau'_2)$

in $S_1 \circ S_2$ ($S_1 \circ S_2 = \text{fun } x \rightarrow S_2(S_1x).$)

- **In all other cases, report error.**