

Lecture 22 — OCaml type-checking, part 1

● Lecture 22

- *Monomorphic* (i.e. no type variables), *explicitly-typed* (i.e. variable declarations) OCaml
- Type checking (checking whether expression is type correct)
- Type inference (figuring out variable declarations to make the expression type-correct, if possible)
- Unification (an algorithm used in type inference)

● Lecture 23

- “Let-polymorphism” — types with variables
- OCaml type system
- Type inference with polymorphism

Monomorphic, explicitly-typed OCaml

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

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

$\emptyset \vdash \text{fun } x:\text{int} \rightarrow x+1 : \text{int} \rightarrow \text{int}$

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

Examples (cont.)

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

Type-correctness theorem

Theorem If $\emptyset \vdash e: \tau$, then if $e \Downarrow v$, v is a value of type τ (i.e. if $\tau = \text{int}$, then $v = \text{Int } i$, if $\tau = \dots \rightarrow \dots$, then $v = \text{fun } x \rightarrow e$, etc.). Furthermore, although it is possible that there is no v such that $e \Downarrow v$ (since the evaluation of e may not terminate), no sub-evaluation produces a type error.

Type inference

- If variables are not declared, can still infer enough information *from how the variables are used* to give a correct type for any expression (if one exists). This is called *type inference*.
- Basic idea: Start by assigning a distinct *type variable* to every subexpression (which we notate by subscripting the expression with its type variable). Exploit ways in which the type expressions are connected to find information about the type variables. For example, if we have an annotated expression of the form “ $(e_\alpha e'_\beta)_\gamma$ ”, then we know $\alpha = \beta \rightarrow \gamma$.

Informal example of type inference

$\emptyset \vdash \text{fun } x \rightarrow x+1$

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

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

Type inference, formally

- We are given a closed term t . W.l.o.g. assume all lambda-bound variables have different names.
- Some terminology: Following tradition, functions are called “abstractions,” and parameters of functions are called “lambda-bound variables,” or “program variables.”
- Allow type variables α, β , etc., in types. (E.g. $\alpha \rightarrow \text{int}$)
- For each lambda-bound variable, assign a unique type variable.
- To every subexpression of t , assign a distinct type variable (except that every occurrence of a program variable has the type variable assigned to it in the previous step).

Type inference, formally

- Generate a set E of equations, or *constraints*, of the form $\alpha = \tau$, as follows: Start with $E = \emptyset$. 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 $\tau \rightarrow \tau' \rightarrow \tau''$)
$(\text{fun } x_\alpha \rightarrow e_\beta)_\gamma:$	$\{ \gamma = \alpha \rightarrow \beta \}$
$(e_\alpha e'_\beta)_\gamma:$	$\{ \alpha = \beta \rightarrow \gamma \}$

Example of generating E

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

Annotated: $(\text{fun } f_\alpha \rightarrow (f_\alpha (f_\alpha \ 3_\beta)_\gamma)_\delta)_\psi$

Constraints:

Solving E (informally)

- Start from the constraints generated in the previous example:
- *(Note how we had to make two type expressions match up — this is called unification.)*

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 later. These are examples to be solved by inspection.

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

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

Algorithm for solving E

- Repeat the following until E cannot change any more:
 - Pick an equation $\alpha = \tau$ from E .
 - If α occurs in τ , raise an “occurs check” exception.
 - If there is no other equation $\alpha = \tau'$ in E , then replace all occurrences of α in E by τ . (This means that α occurs only once in all of E , namely in the equation $\alpha = \tau$.)
 - If there is another equation $\alpha = \tau'$, unify τ and τ' . If it fails, then E cannot be solved. If it succeeds, it produces a substitution S . Remove $\alpha = \tau$ and $\alpha = \tau'$ from E , and add $\alpha = S\tau$. Then add S itself to E .
- If type variables remain in E (i.e. in τ for some equation $\alpha = \tau$), they can be replaced by any type.

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.**