Lecture 22 — OCaml type-checking

- In the next three lectures, will discuss type-checking in OCaml (which is much more interesting than type-checking in Java). The three lectures will concern: *non*-polymorphic OCaml; polymorphic OCaml; polymorphic OCaml with side effects.

- Today we will cover *monomorphic* OCaml:
  - Typing rules
  - Examples
  - Type-checking algorithm
  - Type inference

Monomorphic OCaml

- **Today we will discuss two simplified versions of OCaml:**
  - **OCaml$_{em}$ — explicit type declarations; monomorphic types (no type variables). E.g.**

    ```
    fun x:int -> fun f:(int->string) -> f x
    ```

  - **OCaml$_{im}$ — no type declarations; no polymorphism. E.g.**

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

- **For us, the interesting question about expressions in OCaml$_{im}$ is this: can we *infer* the types of variables — i.e. transform the expression to an expression in OCaml$_{em}$?**

OCaml$_{em}$

```
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
         | Let of string * exp * 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) $\Gamma \vdash$ **Int i : int**

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

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

$\quad \Gamma[a{:}\tau] \vdash e : \tau'$

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

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

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

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

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

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

(Let) $\Gamma \vdash \texttt{let}\ a{:}\tau\texttt{=}e\ \texttt{in}\ e' : \tau'$

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

$\quad \Gamma[a{:}\tau] \vdash e' : \tau'$

Examples

$\emptyset \vdash$ `fun x:int -> x+1` : **int** $\rightarrow$ **int**

$\emptyset \vdash$ `fun f:(int->int) -> f 3` : **(int**$\rightarrow$ **int)** $\rightarrow$ **int**

Examples (cont.)

$\emptyset \vdash$ `fun f:(int->int) -> fun x:int -> f (f x)`

$: (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int})$

Type-correctness theorem

- **As we mentioned w.r.t. the MiniJava type system, we can in principle prove that this type system is "correct" — that is, it is consistent with the operational semantics of MiniOCaml:**

Theorem If $\emptyset \vdash e$: τ, then if $e \Downarrow v$, v is a value of type τ (i.e. if τ = int, then v = Int i, if $\tau = \cdots \rightarrow \cdots$, then v = `fun x ->` 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-checking algorithm

- **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.**

$$
\begin{array}{l}
\textbf{tcheck}\ t\ \Gamma = \textbf{match}\ t\ \textbf{with} \\
\quad\ i \rightarrow \textbf{int} \\
\quad |\ \texttt{true} \rightarrow \textbf{bool} \\
\quad |\ \texttt{false} \rightarrow \textbf{bool} \\
\quad |\ x \rightarrow \Gamma\ x \\
\quad |\ \texttt{fun}\ x\ \texttt{:}\ \tau\ \texttt{->}\ e \rightarrow (\tau \rightarrow \textbf{tcheck}\ e\ \Gamma[\tau/x]) \\
\quad |\ e_1\ e_2 \rightarrow \textbf{let}\ \tau_1 = \textbf{tcheck}\ e_1\ \Gamma \\
\qquad\qquad\qquad \textbf{and}\ \tau_2 = \textbf{tcheck}\ e_2\ \Gamma \\
\qquad\qquad\qquad \textbf{in if}\ \tau_1 = \tau_1' \rightarrow \tau_1''\ \textbf{and}\ \tau_2 = \tau_1' \\
\qquad\qquad\qquad\qquad \textbf{then}\ \tau_1''\ \textbf{else error}
\end{array}
$$

Implicitly-typed monomorphic OCaml (OCaml$_{im}$)

- If we omit the declarations of lambda-bound variables, the language is more similar to OCaml.
- Can view this in two ways (that turn out to be equivalent). The first is:
  - Given an expression e in OCaml$_{im}$, can view it as an incomplete expression in OCaml$_{em}$, and ask: can we add type declarations to all variables so that this expression type checks? This is called *type inference*.
- If it is impossible to fill in type declarations in e, the term is said to be *untypable*. (Note that there may be more than one way to fill in types.)

Examples of typable and untypable terms in OCaml OCaml$_{im}$

```
fun f: ____________ -> fun x:________  -> f (f x)


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
```

OCaml$_{im}$ type system

- **The other way to look at OCaml$_{im}$ is as its own language with its own type system. In fact, its type system is identical to the type system of OCaml$_{em}$, except for the omission of type declarations. I.e. leave all rules the same except these two:**

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

(Let) $\Gamma \vdash \texttt{let } a\texttt{=}e \texttt{ in } e' : \tau'$
$\quad \Gamma \vdash e : \tau$
$\quad \Gamma[a{:}\tau] \vdash e' : \tau'$

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

Implicit		Explicit
$\Gamma \vdash \textbf{fun } a \texttt{ -> } e : \tau \to \tau'$ $\Gamma[a{:}\tau] \vdash e : \tau'$	$<=>$	$\Gamma \vdash \textbf{fun } a : \tau \texttt{ -> } e : \tau \to \tau'$ $\Gamma[a{:}\tau] \vdash e : \tau'$

OCaml$_{im}$ type system (cont.)

(We leave the corresponding transformation for `let` expressions to you.)

- So, an expression in OCaml$_{im}$ is typable *iff* it can be proven to have a type in the OCaml$_{im}$ type system *iff* it can be completed with type declarations and then be proven in the OCaml$_{em}$ type system. So what's the difference? With the explicitly-typed system, we can *check* types — which is really simple — but with the implicitly-typed langauges, we have to *infer* types, which is much harder.

Type inference

- We will not discuss type inference formally. But we will discuss it informally. The basic idea is this:
  - Given a term with no type declarations, start to write *constraints* on the types of the variables; these constraints are implied by what appears in the term:
    - If we see a subterm `f e`, then we know `f` is a function, i.e. it has a type of the form $\alpha \to \beta$ for some α and β.
    - If we see a subterm `if e then ...`, we know `e` has type `bool`.
    - If we see `f (g e)`, then in addition to knowing (from above) that `f`'s type has the form $\alpha \to \beta$ and `g`'s type has the form $\gamma \to \delta$, we know that $\alpha = \delta$.

Type inference (cont.)

  - If we see `e1 + e2`, we know `e1` and `e2` have type `int`; if we see `e1 +. e2`, we know `e1` and `e2` have type `float`.
  - ... and so on.
- Continuing in this way, we either find all constraints, or find a contradiction (e.g. our constraints show that `int` = `float`, or a term of the form $\alpha \to \beta$ also has to have the form $\gamma * \delta$, or a term has both type $\alpha \to \beta$ and type α, or something).
- If we don't find a contradiction, the term is typable. If we still have some Greek letters that are unconstrained, we can replace them by any types we want (uniformly, of course). (In that case, the term has more than one type.)

Informal examples of type inference

```
fun x -> x+1
```

```
fun f -> 1 + (f 3)
```

```
fun f -> f 3
```

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

Informal examples of type inference (cont.)

```
fun x  -> x +. 1
```

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

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

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

Wrap-up

- **Today we discussed monomorphically-typed OCaml, both with and without explicit type declarations. The basis of our discussion was the type systems for the two languages, mainly the explicitly-typed one. We showed examples, discussed type-checking, and viewed type inference as the problem of adding declarations to an implicitly-typed term.**
- **We did this primarily as a prelude to the actual OCaml type system, which is polymorphic.**
- **What to do now:**
  - **MP11**