Lecture 22 — OCaml type-check

- In the next three lectures, will discuss type-check OCaml (which is much more interesting than type-ch in Java). The three lectures will concern: *non*-polym OCaml; polymorphic OCaml; polymorphic OCaml wit 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 OCam
  - OCaml$_{em}$ — explicit type declarations; monomorphic (no type variables). E.g.

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

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

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

- For us, the interesting question about expressions in OC is this: can we *infer* the types of variables — i.e. tra 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 ope is assumed to have a given type $\tau \to \tau' \to \tau''$):

(Const) $\Gamma \vdash$ **Int i : int**

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

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

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

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

$\Gamma \vdash e : \tau$

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

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

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

$\Gamma \vdash e' : \tau$

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

$\Gamma \vdash e : \tau$

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

Examples

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

$\emptyset$[x:int] $\vdash$ x+1 : int

$\emptyset$[x:int] $\vdash$ x : int

$\emptyset$[x:int] $\vdash$ 1 : int

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

$\emptyset$[f : int -> int] $\vdash$ f 3 : int

$\emptyset$[f : int -> int] $\vdash$ f : int -> int

$\emptyset$[f : int -> int] $\vdash$ 3 : int

Examples (cont.)

```
∅ ⊢ fun f:(int->int) -> fun x:int -> f (f x)
          : (int→ int) → (int→ int)
```

```
∅[f:int->int] ⊢ fun x:int -> f(f x) : int->int
    ∅[f:int->int; x:int] ⊢ f (f x) : int
        ∅[f:int->int; x:int] ⊢ f : int->int
        ∅[...] ⊢ f x : int
            ∅[...] ⊢ f : int->int
            ∅[...] ⊢ x : int
```

Type-correctness theorem

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

Theorem If $\emptyset \vdash e \colon \tau$, then if $e \Downarrow v$, v is a value of type τ (i = int, then v = Int i, if $\tau = \cdots \to \cdots$, then v = `fun x` etc.). Furthermore, although it is possible that there i such that $e \Downarrow v$ (since the evaluation of e may not term no sub-evaluation produces a type error.

Type-checking algorithm

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

tcheck t Γ = match t with
  i → int
| `true` → bool
| `false` → bool
| x → Γ x
| `fun` x `:` τ `->` e → (τ → tcheck e $\Gamma[\tau/x]$)
| e_1 e_2 → let τ_1 = tcheck e_1 Γ
        and τ_2 = tcheck e_2 Γ
        in if $\tau_1 = \tau_1' \rightarrow \tau_1''$ and $\tau_2 = \tau_1'$
          then τ_1'' else error

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

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

Examples of typable and untypa 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 lar with its own type system. In fact, its type system is id to the type system of OCaml$_{em}$, except for the omiss type declarations. I.e. leave all rules the same except two:

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

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

- These systems are equivalent, since the following tra mation converts proofs in one system to proofs in the

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

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

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

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

Type inference

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

Type inference (cont.)

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

Informal examples of type infere

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

$x : int$

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

$f : int \to int$

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

$f : int \to \alpha$

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

$f : \alpha \to \beta$

$x : \alpha$

Informal examples of type infere (cont.)

```
fun x  -> x +. 1.0
```

x : float

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

f : int -> α

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

X

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

X

Wrap-up

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