CS 421 Lecture 21 – The OCaml type system

- Polymorphic types, i.e. "type schemes"
- Type rules – polymorphism introduced by "let" expressions
- Examples
- Explaining generalization
- Reference types in OCaml
  - How they work
  - Why they break polymorphism
  - The "value restriction"

T_{OCaml} – the Ocaml type system

Main points about OCaml type system:

- Types contain variables (notated α, β, …)
- Variables can be generalized in some circumstances; types with generalized variables are written $\forall\alpha, \beta, \ldots . \tau$, and called "type schemes"
- If a variable's type is a type scheme, it can be used with any types substituted for the quantified type variables.

Example of polymorphic types (type schemes)

- fst: $\forall\alpha, \beta.\ \alpha * \beta \rightarrow \alpha$.
  When applied to (3, "ab"), it has type int * string → int; when applied to ([3], fun y -> y+1) it has type int list * (int → int) → int list.
- cons: $\forall\alpha.\ \alpha * \alpha$ list → α list

A user-defined function can have a polymorphic type only in the body of a let expression where it is the let-defined name.

Types in T_{OCaml}

Expressions: consts, variables, application, abstraction, let, letrec

Types (notated τ, τ', τ_n, etc.) : int | bool | …
| $\tau \rightarrow \tau'$ (for any types τ and τ') | TypeVar

TypeVar = α, β, … $\vdash$

TypeScheme (σ, σ', etc.) = $\forall \alpha_1, \ldots, \alpha_n.\ \tau$ ($n \geq 0$)
(Note: TypeSchemes include types)

TypeEnv (notated Γ): map from variables to type schemes

Judgments: $\Gamma \vdash e : \tau$

Axioms of T_{OCaml}

T_{OCaml} has just one axiom:

(Var)

$$\frac{\Gamma(x) = \sigma \qquad \tau \leq \sigma}{\Gamma \vdash x : \tau}$$

There are no Const axioms; all predefined names are assumed to be in the initial environment (which we continue to write, by abuse of notation, as $\varnothing$)

Axioms of T_{OCaml}

Understanding the Var axiom:

- If a name has a monomorphic type in Γ, then this works the same as in T_{simp}
- If a name has a polymorphic type, then it can be used at any instance of that type. "$\tau \leq \sigma$" means "τ is an instance of σ" – i.e. τ is obtained from σ by substituting types for type variables.
- The Var rule is an axiom because the assertions above the line are not judgments in the system.

Example: fst (3, true)

Rules of inference of T_{OCaml}

Application and abstraction rules are the same as in T_{simp}. Also add rules for tuples.

(Application)

$$\frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau}$$

(Abstraction)

$$\frac{\Gamma[x : \tau] \vdash e : \tau'}{\Gamma \vdash fun\ x \rightarrow e : \tau \rightarrow \tau'}$$

(Tuple)

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

Rules of inference of T_{OCaml}

let and letrec are new:

(let)

$$\frac{\Gamma \vdash e_1 : \tau' \qquad \Gamma[x : GEN_\Gamma(\tau')] \vdash e_2 : \tau}{\Gamma \vdash let\ x = e_1\ in\ e_2 : \tau}$$

(letrec)

$$\frac{\Gamma[x : \tau'] \vdash e_1 : \tau' \qquad \Gamma[x : GEN_\Gamma(\tau')] \vdash e_2 : \tau}{\Gamma \vdash let\ rec\ x = e_1\ in\ e_2 : \tau}$$

Example: let f = fun x -> x 0
in f (fun y -> y+1): int

Example:

```
let f = fun x -> x 0
in (f (fun y -> y+1),
    f (fun n -> [n])): int * (int list)
```

Notes on T_{OCaml}

(1) As in T_{simp}, the structure of a proof is completely determined by the syntactic structure of the expression

(2) Judgments always assign types to expressions, never type schemes. E.g. $\Gamma \vdash \text{fst} : \forall\alpha, \beta.\ \alpha * \beta \rightarrow \alpha$ is not a valid judgment, even though $\Gamma(\text{fst}) = \forall\alpha, \beta.\ \alpha * \beta \rightarrow \alpha$ (implicitly). Every *use* of a polymorphic name has a specific type.

Generalization in the let rule

In the let rule, $GEN_{\Gamma}(\tau)$ usually means "quantify over all type variables in τ." However, consider this case:

```
let f = fun x -> (let g = fun y -> y x) in g incr,
                  x)
in e
```

We can type-check the body of f giving x type α. Then, g has type $(\alpha\rightarrow\beta)\rightarrow\beta$, which generalizes to $\forall\alpha,\beta.(\alpha\rightarrow\beta)\rightarrow\beta$, so g incr has type int (with α and β both being int), and f types as int * α. Generalizing f, it gets type $\forall\alpha.\ \alpha\rightarrow$ int * α. Now, if e contains the expression "f true", it type checks. However, f actually requires that x be of type int.

Generalization in the let rule (cont.)

For this reason, $GEN_\Gamma(\tau)$ actually means "quantify over all type variables in τ *except those that occur free in* Γ." Then, in this case:

```
let f = fun x -> (let g = fun y -> y x) in g incr,
                  x)
in e
```

if we give x type α, g has type $(\alpha\rightarrow\beta)\rightarrow\beta$, but this generalizes to $\forall\beta.(\alpha\rightarrow\beta)\rightarrow\beta$ (note there is no quantification over α). Now, g incr cannot be typed, because incr has type int$\rightarrow$int, and the closest we can get by instantiating g's type is $\alpha\rightarrow$int. To type-check this term, we would *have* to give x type int, so f would have type int $\rightarrow$ int*int, and the call "f true" would be a type error.

References in OCaml

OCaml has *references*, or assignable variables. Unlike most other languages, *dereferencing* of references has to be done explicitly.

Types: α ref – reference to a value of type α

Operations:

ref: $\alpha \rightarrow \alpha$ ref

!: α ref $\rightarrow \alpha$

:= α ref * $\alpha \rightarrow$ unit

We also have ; : $\alpha * \beta \rightarrow \beta$, which is useful only when doing imperative programming.

Type-checking references

Would like to treat these operators as polymorphic, but consider this example:

```
let i = fun x -> x
in let fp = ref i
    in (fp := not;  (!fp) 5)
```

i gets type $\forall\alpha.\alpha\rightarrow\alpha$, and then fp would have type $\forall\alpha.(\alpha\rightarrow\alpha)$ ref. Since it is polymorphic, fp can be used at type (bool$\rightarrow$bool) ref or (int$\rightarrow$int) ref, making both uses in the last line type-correct. However, the effect is to assign a boolean function to fp and then apply fp to an int.

Type-checking references (cont.)

Treating an expression of type α ref as a normal polymorphic expression has caused a serious error: an expression that type-checks but has a run-time type error.

How can the type system be fixed?

- Easiest method: do not generalize reference expressions at all – make all refs monomorphic
- Method used by OCaml: “value restriction”
  - causes some meaningful polymorphism to fail

The "value restriction"

It turns out that the problem with polymorphic refs can be solved by making this restriction: the type of an expression can be generalized only if the expression is a "syntactic value" – meaning, essentially, that it is either a constant or an abstraction.

Axioms of T_{OCaml}

T_{OCaml} has just one axiom:

(Var)

There is no Const axioms; all predefined names are assumed to be in the initial environment (which we continue to notate, by abuse of notation, $\emptyset$)

Rules of inference of T_{OCaml}

Application and abstraction rules are the same as in T_{simp}. Also add rules for tuples.

(Application)

(Abstraction)

(Tuple)

Rules of inference of T_{OCaml}

let and letrec are new:

(let)

(letrec)

Type-checking references (cont.)

Treating an expression of type α ref as a normal polymorphic expression has caused a serious error: an expression that type-checks but has a run-time type error.

How can the type system be fixed?

- Easiest method: do not generalize reference expressions at all – make all refs monomorphic
- Method used by OCaml: "value restriction"