CSCI-344
Programming Language Concepts (Section 3)

Lecture 17

Typed μScheme

Instructor: Hossein Hojjat

October 14, 2016

Where we are

Done:

- Kinding Rules
- Polymorphic Types

This session:

- Typed μScheme

Classifying Type Expressions

- Instead of having a set of type-formation rules like

$$\frac{\tau \text{ is a type}}{\text{ARRAY}(\tau) \text{ is a type}} \text{ (ArrayType)}$$

- we use **kinds** on top of type systems to classify type expressions
- This is to ensure that types are well-formed

```
- int int , bool×list
```

- Kind environment Δ tracks type constructor names and kinds

Three Environments

Δ	maps names (of tycons and tyvars) to **kinds**
Γ	maps names (of variables) to **types**
ρ	maps names (of variables) to **values** or **locations**

Three Environments

Δ	maps names (of tycons and tyvars) to **kinds**
Γ	maps names (of variables) to **types**
ρ	maps names (of variables) to **values** or **locations**

- New val decl

```
val x = 33
```

- New type decl

```
type 'a transformer = 'a list * 'a list
```

- New datatype decl

```
datatype color = RED | GREEN | BLUE
```

Three Environments

Δ	maps names (of tycons and tyvars) to **kinds**
Γ	maps names (of variables) to **types**
ρ	maps names (of variables) to **values** or **locations**

- New val decl modifies Γ, ρ

  `val` `x = 33` means $\Gamma\{\texttt{x} : \texttt{int}\}$, $\rho\{x \mapsto 33\}$

- New type decl modifies Δ

  `type` `'a transformer = 'a list * 'a list`

  means $\Delta\{\texttt{transformer} :: * \Rightarrow *\}$

- New datatype decl

  `datatype` `color = RED | GREEN | BLUE`

  means $\Delta\{\texttt{color} :: *\}$, $\rho\{\texttt{RED} \mapsto 0, \texttt{GREEN} \mapsto 1, \texttt{BLUE} \mapsto 2\}$,

  $\Gamma\{\texttt{RED:color , GREEN:color , BLUE:color}\}$

Three Environments

Δ	maps names (of tycons and tyvars) to **kinds**
Γ	maps names (of variables) to **types**
ρ	maps names (of variables) to **values** or **locations**

Exercise: How does the following declaration change the environments?

```
datatype 'a tree =
    NODE of 'a tree * 'a * 'a tree
  | EMPTY
```

Three Environments

Δ	maps names (of tycons and tyvars) to **kinds**
Γ	maps names (of variables) to **types**
ρ	maps names (of variables) to **values** or **locations**

Exercise: How does the following declaration change the environments?

```
datatype 'a tree =
    NODE of 'a tree * 'a * 'a tree
  | EMPTY
```

- $\Delta\{\texttt{tree} :: * \Rightarrow *\}$
- $\Gamma\{\texttt{NODE} \rightarrow \forall\texttt{'}a.\ \texttt{'}a\ \texttt{tree} * \texttt{'}a * \texttt{'}a\ \texttt{tree} \rightarrow \texttt{'}a\ \ \texttt{tree},$
  $\texttt{EMPTY} \rightarrow \forall\texttt{'}a.\ \texttt{'}a\ \texttt{tree}\}$
- $\rho\{\texttt{NODE} \rightarrow 0, \texttt{EMPTY} \rightarrow 1\}$

μScheme vs. Typed μScheme

μScheme

```
-> (define compare (x y) (< x y))
compare
```

Typed μScheme

```
-> (define bool compare ([x : int] [y : int])(< x y))
compare : (int int -> bool)
```

μScheme vs. Typed μScheme

μScheme

```
-> (let ((x #f)) (< x 0))
run-time error: in (< x 0), expected an integer,
but got #f
```

Typed μScheme

```
-> (let ((x #f)) (< x 0))
type error: function of type (int int -> bool)
got arguments of types bool int
```

μScheme vs. Typed μScheme

μScheme

```
-> (val f (lambda (x) (+ (< x 0) x)))
f
-> (f 7)
run-time error: in (+ (< x 0) x),expected an integer,
but got #f
```

Typed μScheme

```
-> (val f (lambda ([x : int]) (+ (< x 0) x)))
type error: function of type (int int -> int)
got arguments of types bool int
```

Types for Typed μScheme

```
datatype tyex
= TYCON of name
| CONAPP of tyex * tyex list
| FORALL of name list * tyex
| TYVAR of name
```

Typed μScheme Polymorphic Functions

`length:`	$\forall\alpha.\ \alpha$ `list` $\rightarrow$ `int`
`cons:`	$\forall\alpha.\ \alpha \times \alpha$ `list` $\rightarrow \alpha$ `list`
`car:`	$\forall\alpha.\ \alpha$ `list` $\rightarrow \alpha$
`cdr:`	$\forall\alpha.\ \alpha$ `list` $\rightarrow \alpha$ `list`
`'():`	$\forall\alpha.\ \alpha$ `list`

Typed μScheme Polymorphic Functions

length: $\forall\alpha.\ \alpha$ list $\rightarrow$ int

cons: $\forall\alpha.\ \alpha \times \alpha$ list $\rightarrow \alpha$ list

car: $\forall\alpha.\ \alpha$ list $\rightarrow \alpha$

cdr: $\forall\alpha.\ \alpha$ list $\rightarrow \alpha$ list

'(): $\forall\alpha.\ \alpha$ list

```
-> length
<procedure> : (forall ('a) ((list 'a) -> int))
-> cons
<procedure> : (forall ('a) ('a (list 'a) -> (list 'a)))
-> car
<procedure> : (forall ('a) ((list 'a) -> 'a))
-> cdr
<procedure> : (forall ('a) ((list 'a) -> (list 'a)))
-> '()
() : (forall ('a) (list 'a))
```

Quantified Type Instantiation

To instantiate a value that has a quantified type, we use the `@` operator

```
-> (length '(1 2 3))
type error: function is polymorphic; instantiate
before applying
-> ((@ length int) '(1 2 3))
3 : int
```

```
-> (cons 1 '())
type error: function is polymorphic; instantiate
before applying
-> ((@ cons int) 1 (@ '() int))
(1) : (list int)
```

Quantified Type Instantiation

```
-> (val length-int (@ length int))
length-int : ((list int) -> int)
-> (length-int '(3 4 5))
3 : int
```

```
-> (val cons-sym (@ cons sym))
cons-sym : (sym (list sym) -> (list sym))
-> (val empty-sym (@ '() sym))
() : (list sym)
-> (cons-sym 'x empty-sym)
(x) : (list sym)
```

Quantified Type Instantiation

- Type-instantiation operator `@` lets us **use** a polymorphic value
- Type-abstraction operator `type-lambda` **creates** a polymorphic value

```
-> (val id (type-lambda ['a] (lambda ([x : 'a]) x)))
id : (forall ('a) ('a -> 'a))
```

- `type-lambda` in term becomes `forall` in type
- Inside `(type-lambda ['a] ...)` the type variable 'a is a legitimate type, whose nature is unknown

Type rules for Typed μScheme

The typing rules are much like typed Impcore, but

1. only one type environment Γ
2. a kind environment is needed for type constructors and type variables
3. no special rules for individual type constructors like array

Interesting Type rules

$$\frac{\Delta, \Gamma \vdash e : \forall \alpha_1, \cdots, \alpha_n.\ \tau \qquad \Delta \vdash \tau_i :: *,\ 1 \leq i \leq n}{\Delta, \Gamma \vdash \mathrm{TYAPPLY}(e, \tau_1, \cdots, \tau_n) : \tau[\alpha_1 \mapsto \tau_1, \cdots, \alpha_n \mapsto \tau_n]} \text{ (TyApply)}$$

$$\frac{\alpha_i \notin \mathsf{ftv}(\Gamma),\ 1 \leq i \leq n \qquad \Delta\{\alpha_1 :: *, \cdots, \alpha_n :: *\}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \mathrm{TYLAMBDA}(\alpha_1, \cdots, \alpha_n, e) : \forall \alpha_1, \cdots, \alpha_n.\ \tau} \text{ (TyLambda)}$$

Summary

This Lecture

- Typed μScheme

Next Lecture

- Smalltalk