Lecture 7:Type Checking (II)

1 Typing Lists

Let's add typed lists to the language. To do so, we introduce five new kinds of expressions:

1. empty creates an empty list,
2. e_1 :: e_2 constructs a non-empty list with e_1 as the head and e_2 as the tail,
3. head(e) produces the first element of the list e, or signals an error if e is the empty list,
4. tail (e) produces the rest of the list e and also signals an error if e is the empty list, and
5. empty?(e) produces true if e is the empty list.

Notice that list is not a type but a *type constructor* that can be applied to a type to produce an actual type. For example, int list , (int list) list , and so on are valid types, but list by itself is not.

The typing rules for head(e), tail (e), and empty?(e) are straightforward:

$$\text{T-Head}\ \frac{\Gamma \vdash e : \tau\,\mathsf{list}}{\Gamma \vdash \mathsf{head}(e) : \tau} \qquad \text{T-Tail}\ \frac{\Gamma \vdash e : \tau\,\mathsf{list}}{\Gamma \vdash \mathsf{tail}\ (e) : \tau\,\mathsf{list}} \qquad \text{T-IsEmpty}\ \frac{\Gamma \vdash e : \tau\,\mathsf{list}}{\Gamma \vdash \mathsf{empty?}(e) : \mathsf{bool}}$$

The typing rule for e_1 :: e_2 forces the added element to have the same type as the elements in the rest of the list. In other words, this rule ensures that all lists are *homogenous*:

$$\text{T-Cons}\ \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\,\mathsf{list}}{\Gamma \vdash e_1 :: e_2 : \tau\,\mathsf{list}}$$

However, the typing rule for empty gets us into trouble:

$$\text{T-Empty}\ \Gamma \vdash \mathsf{empty} : ???\mathsf{list}$$

Given the type language that we have, we need to say that empty is a list of some actual type, but we don't have enough information to determine what that type should be. If we had type identifiers or generics, we could write the

$\boxed{e}$

Types

$\tau ::= \cdots$

|	τ list	List of τ elements

Expressions

$e ::= \cdots$

|	e_1 :: e_2	List constructor
|	empty[τ]	Empty list
|	head(e)	Head of list
|	tail (e)	Tail of list
|	empty?(e)	Test if list is empty

(a) Syntax.

$\boxed{\Gamma \vdash e : T}$

$$\text{T-Empty}\ \Gamma \vdash \mathsf{empty}[\tau] : \tau\,\mathsf{list}$$

$$\text{T-Cons}\ \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau\,\mathsf{list}}{\Gamma \vdash e_1 :: e_2 : \tau\,\mathsf{list}}$$

$$\text{T-Head}\ \frac{\Gamma \vdash e : \tau\,\mathsf{list}}{\Gamma \vdash \mathsf{head}(e) : \tau}$$

$$\text{T-Tail}\ \frac{\Gamma \vdash e : \tau\,\mathsf{list}}{\Gamma \vdash \mathsf{tail}\ (e) : \tau\,\mathsf{list}}$$

$$\text{T-IsEmpty}\ \frac{\Gamma \vdash e : \tau\,\mathsf{list}}{\Gamma \vdash \mathsf{empty?}(e) : \mathsf{bool}}$$

(b) Type Checking.

Figure 10.1: Type Checking Lists (extends fig. 9.1)

```
let int_length = fix (self : int list →int) →
  fun (lst : int list) →
    if empty? lst then
      0
    else
     1 + self (tail lst ) in
...
```

(a) Length for integer lists.

```
let bool_length = fix (self : bool list →int) →
  fun (lst : bool list) →
    if empty? lst then
      0
    else
     1 + self (tail lst ) in
...
```

(b) Length for boolean lists.

Figure 10.2: The only different is the type of the list element.

Type Identifiers
$\alpha, \beta, \gamma ::= \cdots$
Types

$\tau ::=$	$\tau_1 \to \tau_2$	Type of functions
$\mid$	α	Type identifiers
$\mid$	$\forall \alpha.\tau$	Type of type abstractions

Identifiers
$x, y, z ::= \cdots$
Expressions

$e ::=$	x	Identifiers
$\mid$	fun $(x{:}\tau)$ e	Functions
$\mid$	$e_1 e_2$	Function applications
$\mid$	$\Lambda\, \alpha\, .\, e$	Type abstractions
$\mid$	$e[\tau]$	Type applications

Environments
$\Gamma \in x \rightharpoonup \tau$
Type Environments
$\Delta \subseteq 2^{\alpha}$

(a) Syntax.

$\boxed{\Delta \vdash \tau \text{ ok}}$

$$\text{OK-TID}\ \frac{\alpha \in \Delta}{\Delta \vdash \alpha \text{ ok}}$$

$$\text{OK-TFORALL}\ \frac{\Delta, \alpha \vdash \tau \text{ ok}}{\Delta \vdash \forall \alpha.\tau \text{ ok}}$$

$$\text{OK-TFUN}\ \frac{\Delta \vdash \tau_1 \text{ ok} \qquad \Delta \vdash \tau_2 \text{ ok}}{\Delta \vdash \tau_1 \to \tau_2 \text{ ok}}$$

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\text{T-ID}\ \frac{\Gamma(x) = \tau}{\Delta; \Gamma \vdash x : \tau}$$

$$\text{T-FUN}\ \frac{\Delta; \Gamma, x : \tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash \text{fun}\ (x{:}\tau_1).e : \tau_1 \to \tau_2}$$

$$\text{T-APP}\ \frac{\Delta; \Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Delta; \Gamma \vdash e_2 : \tau_1}{\Delta; \Gamma \vdash e_1 e_2 : \tau_2}$$

$$\text{T-TYPFUN}\ \frac{\Delta, \alpha; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda\alpha.e : \forall \alpha.\tau}$$

$$\text{T-TYPAPP}\ \frac{\Delta; \Gamma \vdash e : \forall \alpha.\tau_2 \qquad \Delta \vdash \tau_1 \text{ ok}}{\Delta; \Gamma \vdash e[\tau_1] : \tau_2[\alpha/\tau_1]}$$

(b) Type Checking.

Figure 10.3: Type Checking Polymorphism

type α list. We'll get to generics shortly. For now, we modify the syntax of empty lists to explicitly state the type of the element: we will always write empty[τ] and never just empty. The complete type system for lists along with this fix, is shown in fig. 10.1.

Using the technique described in this section, it is straightforward to develop type systems for other kinds of collections, such as arrays and tuples.

2 Polymorphism

Writing empty[int], empty[bool], etc. seems annoying. But, there is a deeper problem with our programming language. The types that we have so far do not let us write any type-generic functions, such as a length function that works on any list. Instead, we have to specialize length to a particular type, such as the int_length and bool_length functions shown in fig. 10.2. The only difference between the two functions is the type of the list element.

Just as we use functions to abstract over values, we now introduce *type abstractions* to abstract over types: the type abstraction $\Lambda\, \alpha\, .\, e$ is a new kind of function that takes a type as an argument and can be applied by writing $(\Lambda\, \alpha\, .\, e)[\tau]$. The supplied type τ is bound to the *type identifier* α. Any types that occur within e (e.g., an annotation on a function) can refer to α. When applied, α gets substituted with the actual type τ.

With these new features, we can write a generic length function:

```
let length = Λα .
```

```
fix (self : α list →int) .
  fun (lst : α list) .
    if empty?(lst) then
      0
    else
     1 + self (tail (lst )) in
...
```

To actually apply length, we have to provide the type of the elements as an argument:

```
length[int] (1 :: 2 :: 3 :: empty[int])
length[bool] (true :: false :: true :: empty[bool])
```

If we simply wrote length (1 :: 2 :: 3 :: empty[int]), we'd get a type error because length expect a type as its first argument.

We cannot add support for polymorphism to a type system by simply adding new rules. We need to also augment the environment to keep track of the set of bound type identifiers. If we did not do so, the programmer might mistakenly write: et ength = typfun α -> fix (self : β list -> int) -> or even just: et ength = fix (self : β list -> int) -> And we wouldn't be able to tell that β is an unbound type identifier.

Therefore, all the typing rules need to be lightly modified to track the set of bound type identifiers. Let Δ to denote a set of type identifiers. The environment now has two components: Δ, Γ. The existing typing rules leave Δ unmodified and simply propogate it to their subexpressions.

To type-check a type abstraction $\Lambda\,\alpha\,.\,e$, we type-check e in an environment where the set of type identifiers is augmented with α.

Type-checking type application, $e[\tau]$ requires several ingredients:

1. We must ensure that e is a type function, which is easy to do. A type function has a type of the form $\forall\alpha.\tau_1$.

2. We must ensure that τ_1 does not have any free identifiers (apart from τ_1). This requires another relation $\Delta \vdash \tau_1$ ok.

3. Given these, we can calculate the type of $e[\tau_1]$ as the type of the body, τ_1, with τ_2 substituted for α.

For example, if e has type $\forall\alpha.\alpha$ list →int then e[bool] is bool list →int.

Evaluting these new expressions is straightforward. Type functions are values, so no further evaluation is necessary. Type application can be evaluated in a manner analogous to function application.

Figure 10.3 shows the full syntax and type system for a language with type abstraction and functions. The language in the figure does not have any other kind of values, such as constants or data structures, but these are straightforward to add. The bare-bones language shown in the figure is known as System F, and it captures the essence of polymorphism.