

# Type Systems for Programming Languages (15-814)

## Lecture Notes, Fall 2006, Week 7

Hormoz Zarnani

October 26, 2006 \*

### 1 Variable Types

In the  $\lambda$ -calculus every well-typed expression must have a unique type. As a consequence, a function expression must have unique domain and range types. This limitation keeps us from writing functions of generic type. For example, we cannot write a function that composes the two functions of appropriate type. This motivates *variable types* (a.k.a. *polymorphism* or *generics*), which are the means by which we can capture type uniformities in programming.

Jean-Yves Girard and John Reynolds independently worked on developing a logic with variable types. Girard developed his logic for the consistency of the second-order arithmetic and called it *System F*. Reynolds developed his logic for polymorphic programming and called it the *polymorphic typed  $\lambda$ -calculus* or the *second-order  $\lambda$ -calculus*.

### 2 Polymorphic $\lambda$ -calculus

#### 2.1 Abstract Syntax

The abstract syntax for the polymorphic  $\lambda$ -calculus is as follows:

$$\tau ::= X \mid \tau_1 \rightarrow \tau_2 \mid \forall X. \tau \tag{1}$$

$$e ::= x \mid \lambda x:\tau. e \mid e_1 e_2 \mid \lambda X. e \mid e[\tau] \tag{2}$$

The static and dynamic semantics of the polymorphic  $\lambda$ -calculus are as follows.

#### 2.2 Static Semantics

Before we can give the judgments for the static semantics of the polymorphic  $\lambda$ -calculus, we need to give the rules for type formation and typing expressions:

##### Type Formation

$$\frac{}{\Delta, X \text{ type} \vdash X \text{ type}} \tag{3}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \text{ type}} \tag{4}$$

---

\*Class was cancelled on October 24.

$$\frac{\Delta, X \text{ type} \vdash \tau \text{ type} \quad x \# \Delta}{\Delta \vdash \forall X. \tau \text{ type}} \quad (5)$$

## Typing Expressions

$$\frac{}{\Delta; \Gamma, x:\tau \vdash x : \tau} \quad (6)$$

$$\frac{\Delta; \Gamma, x:\tau_1 \vdash e : \tau_2 \quad x \# \Gamma}{\Delta; \Gamma \vdash \lambda x:\tau_1. e : \tau_1 \rightarrow \tau_2} \quad (7)$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash e_1 e_2 : \tau} \quad (8)$$

$$\frac{\Delta, X \text{ type}; \Gamma \vdash e : \tau \quad X \# \Delta}{\Delta; \Gamma \vdash \Lambda X. e : \forall X. \tau} \quad (9)$$

$$\frac{\Delta; \Gamma \vdash e : \forall X. \tau \quad \Delta \vdash \sigma}{\Delta; \Gamma \vdash e[\sigma] : [\sigma/X] \tau} \quad (10)$$

Rule (10) states that we can instantiate a polymorphic function at any type. If we let  $e$  to be the identity function, *i.e.* let  $e:\forall X. X \rightarrow X$ , then we would have  $e[\forall X. X \rightarrow X]:\forall X. X \rightarrow X \rightarrow \forall X. X \rightarrow X$ . Note that the type of  $e[\forall X. X \rightarrow X]$  is larger than that of  $e$ . At the first glance, it appears that using this rule we could write an infinite loop in the polymorphic  $\lambda$ -calculus, thus threatening the consistency of the logic. But applying  $e[\forall X. X \rightarrow X]$  to  $e$  again results in an expression of type  $\forall X. X \rightarrow X$ , which is the same as the type of  $e$  itself. This property of the polymorphic  $\lambda$ -calculus— that we can instantiate polymorphic types with other polymorphic types — is called *impredicativity* and is the source of the expressive power of this language.

We will now turn our attention back to the static semantics. The statics consist of two hypothetical judgments:

$$\Delta \vdash \tau \text{ type} \quad (11)$$

$$\Delta; \Gamma \vdash e : \tau \quad (12)$$

where meta-variable  $\Delta$  ranges over finite sets type variable formation hypotheses, and  $\Gamma$  ranges over finite sets of expression variable typing hypotheses.

## 2.3 Dynamic Semantics

The dynamic semantics of the polymorphic  $\lambda$ -calculus is as follows (note that it extends the dynamic semantics of the  $\lambda$ -calculus by just adding two structural operational semantics rules):

$$\frac{e_2 \text{ value}}{(\lambda x:\tau. e)e_2 \mapsto [e_2/x]e} \quad (13)$$

$$\frac{}{(\Lambda X. e)[\tau] \mapsto [\tau/X]e} \quad (14)$$

$$\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} \quad (15)$$

$$\frac{e_1 \text{ value} \quad e_2 \mapsto e'_2}{e_1 e_2 \mapsto e_1 e'_2} \quad (16)$$

$$\frac{e \mapsto e'}{e[\tau] \mapsto e'[\tau]} \quad (17)$$

### 3 Hacking in System F

We define the unit, empty, product, sum, and natural types as follows:

- *Unit Type*:  $1 := \forall X. X \rightarrow X$ 
  - **Intro**:  $\langle \rangle := \Lambda X. \lambda x:X. x$
  - **Elim**: None.
- *Empty Type*:  $0 := \forall X. X$ 
  - **Intro**: None.
  - **Elim**:  $abort := \Lambda X. \lambda x:0. x[X]$
- *Product Type*:  $\tau_1 \times \tau_2 := \forall X. (\tau_1 \rightarrow \tau_2 \rightarrow X) \rightarrow X$ 
  - **Intro**:  $\langle e_1, e_2 \rangle := \Lambda X. \lambda p:\tau_1 \rightarrow \tau_2 \rightarrow X. p e_1 e_2.$
  - **Elim**:
    1. **fst**  $e := e[\tau_1](\lambda x:\tau_1. \lambda y:\tau_2. x)$
    2. **snd**  $e := e[\tau_2](\lambda x:\tau_1. \lambda y:\tau_2. y)$
- *Sum Type*:  $\tau_1 + \tau_2 := \forall X. (\tau_1 \rightarrow X) \rightarrow (\tau_2 \rightarrow X) \rightarrow X$ 
  - **Intro**:
    1.  $inl(e) := \Lambda X. \lambda l:\tau_1 \rightarrow X. \lambda r:\tau_2 \rightarrow X. l(e)$
    2.  $inr(e) := \Lambda X. \lambda l:\tau_1 \rightarrow X. \lambda r:\tau_2 \rightarrow X. r(e)$
  - **Elim**:  $case_\tau e \{inl(x:\tau_1) \Rightarrow e_1 \mid inr(y:\tau_2) \Rightarrow e_2\} := e[\tau](\lambda x:\tau_1. e_1)(\lambda y:\tau_2. e_2)$
- *Nat Type*:  $nat := \forall X. X \rightarrow (X \rightarrow X) \rightarrow X$ 
  - **Intro**:
    1.  $zero := \Lambda X. \lambda b:X. \lambda s:X \rightarrow X. b$
    2.  $succ e := \Lambda X. \lambda b:X. \lambda s:X \rightarrow X. s(e[X](b)(s))$
  - **Elim**:  $rec(e, e_0, x:\tau. e_1) := e[\tau](e_0)(\lambda x:\tau. e_1)$