Design Principles of Programming Languages

Practices in Class

Chap 13-19

Zhenjiang Hu, Haiyan Zhao, Yingfei Xiong
Peking University, Spring Term, 2016

Code packages

- “fullref”
- “fullerror”
- “rcdsub”
- “fullsub”
- “joinsub”

Syntax

We added to $\lambda_{\rightarrow}$ (with Unit) syntactic forms for *creating*, *dereferencing*, and *assigning* reference cells, plus a new type constructor Ref.

t ::=		*terms*
	unit	*unit constant*
	x	*variable*
	λx:T.t	*abstraction*
	t t	*application*
	ref t	*reference creation*
	!t	*dereference*
	t:=t	*assignment*
	l	store location

Evaluation

Evaluation becomes *a* *four-place* relation: $t \mid \mu \longrightarrow t' \mid \mu'$

$$\frac{l \notin dom(\mu)}{\texttt{ref}\ \ \texttt{v}_1 \mid \mu \longrightarrow l \mid (\mu, l \mapsto \texttt{v}_1)} \qquad \text{(E-RefV)}$$

$$\frac{\mu(l) = \texttt{v}}{!l \mid \mu \longrightarrow \texttt{v} \mid \mu} \qquad \text{(E-DerefLoc)}$$

$$l := \texttt{v}_2 \mid \mu \longrightarrow \texttt{unit} \mid [l \mapsto \texttt{v}_2]\mu \qquad \text{(E-Assign)}$$

Typing

Typing becomes *a three-place* relation: $\Gamma \mid \Sigma \vdash \texttt{t} : \texttt{T}$

$$\frac{\Sigma(l) = \texttt{T}_1}{\Gamma \mid \Sigma \vdash l : \texttt{Ref}\ \texttt{T}_1} \qquad \text{(T-Loc)}$$

$$\frac{\Gamma \mid \Sigma \vdash \texttt{t}_1 : \texttt{T}_1}{\Gamma \mid \Sigma \vdash \texttt{ref}\ \texttt{t}_1 : \texttt{Ref}\ \texttt{T}_1} \qquad \text{(T-Ref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \texttt{t}_1 : \texttt{Ref}\ \texttt{T}_{11}}{\Gamma \mid \Sigma \vdash \texttt{!t}_1 : \texttt{T}_{11}} \qquad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \texttt{t}_1 : \texttt{Ref}\ \texttt{T}_{11} \qquad \Gamma \mid \Sigma \vdash \texttt{t}_2 : \texttt{T}_{11}}{\Gamma \mid \Sigma \vdash \texttt{t}_1\texttt{:=}\texttt{t}_2 : \texttt{Unit}} \qquad \text{(T-Assign)}$$

Subtype Relation

$$S <: S \qquad \text{(S-Refl)}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \qquad \text{(S-Trans)}$$

$$\{l_i{:}T_i \ ^{i\in 1..n+k}\} <: \{l_i{:}T_i \ ^{i\in 1..n}\} \qquad \text{(S-RcdWidth)}$$

$$\frac{\text{for each } i \qquad S_i <: T_i}{\{l_i{:}S_i \ ^{i\in 1..n}\} <: \{l_i{:}T_i \ ^{i\in 1..n}\}} \qquad \text{(S-RcdDepth)}$$

$$\frac{\{k_j{:}S_j \ ^{j\in 1..n}\} \text{ is a permutation of } \{l_i{:}T_i \ ^{i\in 1..n}\}}{\{k_j{:}S_j \ ^{j\in 1..n}\} <: \{l_i{:}T_i \ ^{i\in 1..n}\}} \qquad \text{(S-RcdPerm)}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \qquad \text{(S-Arrow)}$$

$$S <: Top \qquad \text{(S-Top)}$$

Syntax

$t ::=$		*terms:*
	x	*variable*
	$\lambda x{:}T.t$	*abstraction*
	$t\ t$	*application*
$v ::=$		*values:*
	$\lambda x{:}T.t$	*abstraction value*
$T ::=$		*types:*
	Top	*maximum type*
	$T{\rightarrow}T$	*type of functions*
$\Gamma ::=$		*contexts:*
	$\varnothing$	*empty context*
	$\Gamma, x{:}T$	*term variable binding*

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-APP2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-APPABS)}$$

Subtyping $\boxed{S <: T}$

$$S <: S \quad \text{(S-REFL)}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \quad \text{(S-TRANS)}$$

$$S <: \mathsf{Top} \quad \text{(S-TOP)}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1{\rightarrow}S_2 <: T_1{\rightarrow}T_2} \quad \text{(S-ARROW)}$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1{\rightarrow}T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11}{\rightarrow}T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \quad \text{(T-SUB)}$$

Records

$\rightarrow \{\}$ *Extends* $\lambda_{\rightarrow}$ *(9-1)*

New syntactic forms

$\mathtt{t}$	::=	...	*terms:*
		$\{\mathtt{l}_i = \mathtt{t}_i\ ^{i \in 1..n}\}$	*record*
		$\mathtt{t.l}$	*projection*
$\mathtt{v}$	::=	...	*values:*
		$\{\mathtt{l}_i = \mathtt{v}_i\ ^{i \in 1..n}\}$	*record value*
$\mathtt{T}$	::=	...	*types:*
		$\{\mathtt{l}_i : \mathtt{T}_i\ ^{i \in 1..n}\}$	*type of records*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\{\mathtt{l}_i = \mathtt{v}_i\ ^{i \in 1..n}\}.\mathtt{l}_j \longrightarrow \mathtt{v}_j \qquad \text{(E-ProjRcd)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1.\mathtt{l} \longrightarrow \mathtt{t}_1'.\mathtt{l}} \qquad \text{(E-Proj)}$$

$$\frac{\mathtt{t}_j \longrightarrow \mathtt{t}_j'}{\{\mathtt{l}_i = \mathtt{v}_i\ ^{i \in 1..j-1}, \mathtt{l}_j = \mathtt{t}_j, \mathtt{l}_k = \mathtt{t}_k\ ^{k \in j+1..n}\} \longrightarrow \{\mathtt{l}_i = \mathtt{v}_i\ ^{i \in 1..j-1}, \mathtt{l}_j = \mathtt{t}_j', \mathtt{l}_k = \mathtt{t}_k\ ^{k \in j+1..n}\}} \qquad \text{(E-Rcd)}$$

New typing rules $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\text{for each } i \quad \Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{l}_i = \mathtt{t}_i\ ^{i \in 1..n}\} : \{\mathtt{l}_i : \mathtt{T}_i\ ^{i \in 1..n}\}} \qquad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{l}_i : \mathtt{T}_i\ ^{i \in 1..n}\}}{\Gamma \vdash \mathtt{t}_1.\mathtt{l}_j : \mathtt{T}_j} \qquad \text{(T-Proj)}$$

"Algorithmic" subtype relation

$$\mapsto S <: \mathtt{Top} \qquad \text{(SA-Top)}$$

$$\frac{\mapsto T_1 <: S_1 \qquad \mapsto S_2 <: T_2}{\mapsto S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \qquad \text{(SA-Arrow)}$$

$$\frac{\{l_i \;^{i\in 1..n}\} \subseteq \{k_j \;^{j\in 1..m}\} \qquad \text{for each } k_j = l_i,\ \mapsto S_j <: T_i}{\mapsto \{k_j{:}S_j \;^{j\in 1..m}\} <: \{l_i{:}T_i \;^{i\in 1..n}\}} \qquad \text{(SA-Rcd)}$$

Subtyping Algorithm

This *recursively defined total function* is a decision procedure for the subtype relation:

$subtype(S, T) =$

if $T = Top$, then *true*

else if $S = S_1 \longrightarrow S_2$ and $T = T_1 \longrightarrow T_2$

then $subtype(T_1, S_1) \wedge subtype(S_2, T_2)$

else if $S = \{k_j : S_j{}^{j \in 1..m}\}$ and $T = \{l_i : T_i{}^{i \in 1..n}\}$

then $\{l_i{}^{i \in 1..n}\} \subseteq \{k_j{}^{j \in 1..m}\}$

$\wedge$ for all $i \in 1..n$ there is some $j \in 1..m$ with $k_j = l_i$

and $subtype(S_j, T_i)$

else *false*.

Algorithmic Typing

The next step is to "build in" the use of subsumption in application rules, by changing the T-App rule to incorporate a subtyping premise.

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_2 \qquad \vdash T_2 <: T_{11}}{\Gamma \vdash t_1 \; t_2 : T_{12}}$$

Given any typing derivation, we can now

1. normalize it, to move all uses of subsumption to either just before applications (in the right-hand premise) or at the very end
2. replace uses of T-App with T-SUB in the right-hand premise by uses of the extended rule above

This yields a derivation in which there is just *one* use of subsumption, at the very end!

What learnt in Chap 18-19

1. Identify some characteristic "core features" of object-oriented programming
2. Develop two different analysis of these features:
   2.1 A *translation* into a lower-level language
   2.2 A *direct*, high-level formalization of a simple object-oriented language ("Featherweight Java")

Object-oriented languages

Most OO languages treats each object as

A *data structure*

- encapsulating some internal states
- offering access to thesse states

via a *collection of methods*.

basic features of object-oriented languages

encapsulation

Inheritance

…….

Modeling features of OO with λ -calculus

How the *basic features* of object-oriented languages

- *encapsulation of state*
- *Inheritance*
- *……*

can be understood as *"derived forms"* in a lower-level language with a rich collection of primitive features:

- *(higher-order) functions*
- *records*
- *references*
- *recursion*
- *subtyping*

Encapsulation

An object is a record of functions, which maintain *common internal state via a shared reference to a record* of mutable instance variables.

This state is inaccessible *outside of the object* because there is no way to name it.

- lexical scoping ensures that instance variables can only be named from inside the methods.

Inheritance

Objects that *share parts of their interfaces* will typically (though not always) *share parts of their behaviors*.

To avoid duplication of code, the way is to write the implementations of these behaviors in *just one place*.

$\Longrightarrow$ *inheritance*

Basic mechanism of inheritance: *classes*

A class *is a data structure* that can be

- *instantiated* to create new objects ("instances")
- *refined* to create new classes ("subclasses")

The essence of objects

- Encapsulation of state with behavior
- Behavior-based subtyping
- Inheritance (incremental definition of behaviors)
- Access of super class
- Open recursion through this

Featherweight Java

A concrete language with core OO features

FJ models "core OO features" and their types and *nothing else*.

History:

- Originally proposed by a Penn visiting student (Atsushi Igarashi) as a tool for analyzing GJ ("Java plus generics"), which later became Java 1.5
- Since then used by many others for studying a wide variety of Java features and proposed extensions

Practice #1

- Do exercise 18.6.1
  - Write a subclass of resetCounterClass with an additional method dec that subtracts one from the current value stored in the counter.
  - Use the fullref checker to test your new class.

Practice #2

- Do exercise 18.7.1
  - Define a subclass of backupCounterClass with two new methods, *reset2* and *backup2*, controlling a second "backup register." This register should be completely separate from the one added by backupCounterClass: calling *reset* should restore the counter to its value at the time of the last call to *backup* (as it does now) and calling *reset2* should restore the counter to its value at the time of the last call to *backup2*.
  - Use the fullref checker to test your new class

Practice #3

- Do exercise 17.3.1
  - The *joinexercise* typechecker is an incomplete implementation of the simply typed lambda-calculus with subtyping, records, and conditionals: basic parsing and printing functions are provided, but the clause for TmIf is missing from the typeof function, as is the join function on which it depends. Add booleans and conditionals (and joins and meets) to this implementation.
  - Refer to: § 16.3 showed how adding booleans and conditionals to a language with subtyping required extra support functions for calculating the least upper bounds of a given pair of types. The proof of Proposition 16.3.2 (see page 522) gave mathematical descriptions of the necessary algorithms

Practice #4

- Do exercise 17.3.3
  - If the subtype check in the application rule fails, the error message that our typechecker prints may not be very helpful to the user. We can improve it by including the expected parameter type and the actual argument type in the error message, but even this may be hard to understand.
  - Reimplement the typeof and subtype functions to make all of the error messages as informative as possible.