Design Principles of Programming Languages

Universal Types

Zhenjiang Hu, Haiyan Zhao, Yingfei Xiong

Peking University, Spring Term, 2014

Project Deadlines

- Report and code submission: May 27th
- Final presentation: May 28th, Jun 4th
  - Presentation: 30 mins
  - Discussion: 10 mins
  - Do introduce your individual responsibility

Presentation Schedule

- May 28th
  - 网络协议编程语言
    - 徐泽骅、刘晨昊、包新启
  - 没有停机问题的编程语言
    - 杨嘉骐 李屹 王译梧
  - 嵌入时间复杂度表示的类型系统
    - 林舒 刘智猷 苏暐恩
  - 无死锁、无隐私泄露的pi演算
    - 杨纬坤 侯嘉琦 汪成龙
- Jun 4th
  - 可执行伪码
    - 郭嘉琦 窦笑添 王晓阳
  - Race-Free Imperative Language
    - 王诗君 赵玮泽 齐荣嵘 米亚晴
  - 浮点数精度判定类型系统
    - 吴逸鸣 邹达明 胡天翔 郑淇木

Key to homework

- Change the constraint typing rule and the unification algorithm so that the following term can be typed
  - fix (λh. λx:Nat. h)
- Generate constraints for “fix”
  - $$\frac{\Gamma \vdash \mathrm{t}:\mathrm{T} \mid \mathrm{C} \qquad X \text{ is a fresh variable}}{\Gamma \vdash \mathrm{fix}\ \mathrm{t}:\mathrm{X} \mid \mathrm{C} \cup \{\mathrm{T}=\mathrm{X}\rightarrow\mathrm{X}\}}$$
- Unification Algorithm: adding two rules

```
else if S is X and X∈FV(T)
  then unify([X↦ μX.T]C’)∘[X↦ μX.T]
else if T is X and X∈FV(S)
  then unify([X↦ μX.S]C’)∘[X↦ μX.S]
```

Not a general algorithm but works for hungry

System F

- The foundation for polymorphism in modern languages
  - C++, Java, C#, Modern Haskell
- Discovered by
  - Jean-Yves Girard (1972)
  - John Reynolds (1974)
- Also known as
  - Polymorphic λ-calculus
  - Second-order λ-calculus
    - (Curry-Howard) Corresponds to second-order intuitionistic logic
  - Impredicative polymorphism (for the polymorphism mechanism)

Review

- What is the limitation of Hindley-Milner system?

System F by Examples

```
  id = λX. λx:X. x;
▸ id : ∀X. X → X
  id [Nat];
▸ <fun> : Nat → Nat
  id [Nat] 0;
▸ 0 : Nat
```

Exercise

- What are the types of the following terms?
  - double= $\lambda X.\ \lambda f{:}X\to X.\ \lambda a{:}X.f\ (f\ a)$
  - double [Nat]
  - double [Nat→Nat]

Key to Exercise

- What are the types of the following terms?
  - double=λX. λf:X→X. λa:X.f (f a)
    - ∀X. (X→X) → X →X
  - double [Nat]
    - (Nat→ Nat) →Nat→ Nat
  - double [Nat→Nat]
    - ((Nat→ Nat) → Nat→ Nat) → (Nat→ Nat) → Nat→ Nat

Syntax

$\mathtt{t}$	$::=$		*terms:*
		$\mathtt{x}$	*variable*
		$\lambda\mathtt{x}{:}\mathtt{T}.\mathtt{t}$	*abstraction*
		$\mathtt{t}\ \mathtt{t}$	*application*
		$\lambda\mathtt{X}.\mathtt{t}$	*type abstraction*
		$\mathtt{t}\ [\mathtt{T}]$	*type application*
$\mathtt{v}$	$::=$		*values:*
		$\lambda\mathtt{x}{:}\mathtt{T}.\mathtt{t}$	*abstraction value*
		$\lambda\mathtt{X}.\mathtt{t}$	*type abstraction value*
$\mathtt{T}$	$::=$		*types:*
		$\mathtt{X}$	*type variable*
		$\mathtt{T}\to\mathtt{T}$	*type of functions*
		$\forall\mathtt{X}.\mathtt{T}$	*universal type*
Γ	$::=$		*contexts:*
		$\varnothing$	*empty context*
		$\Gamma, \mathtt{x}{:}\mathtt{T}$	*term variable binding*
		$\Gamma, \mathtt{X}$	*type variable binding*

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

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(E-App2)}$$

$$(\lambda\mathtt{x}{:}\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \quad \text{(E-AppAbs)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ [\mathtt{T}_2] \longrightarrow \mathtt{t}_1'\ [\mathtt{T}_2]} \quad \text{(E-TApp)}$$

$$(\lambda\mathtt{X}.\mathtt{t}_{12})\ [\mathtt{T}_2] \longrightarrow [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{t}_{12} \quad \text{(E-TappTabs)}$$

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

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

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1\to\mathtt{T}_2} \quad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11}\to\mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \quad \text{(T-App)}$$

$$\frac{\Gamma, \mathtt{X} \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{X}.\mathtt{t}_2 : \forall\mathtt{X}.\mathtt{T}_2} \quad \text{(T-TAbs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \forall\mathtt{X}.\mathtt{T}_{12}}{\Gamma \vdash \mathtt{t}_1\ [\mathtt{T}_2] : [\mathtt{X} \mapsto \mathtt{T}_2]\mathtt{T}_{12}} \quad \text{(T-TApp)}$$

Exercise

- Can we type this term in simple typed λ-calculus?
  - $\lambda x. x\ x$

Exercise

- Can we type this term in system F?
  - $\lambda x. x\ x$

Exercise

- Can we type this term in system F?
  - $\lambda x. x\ x$

- $\lambda x\colon \forall X. X \to X.$ x $[\forall X. X \to X]$ x

- quadruple = λX. double [X→X] (double [X])

Exercise

- Implment csucc for CNat so that c_i = csucc c_{i-1}

```
CNat = ∀X. (X→X) → X → X;

c0  =  λX. λs:X→X. λz:X. z;
▸ c0 : CNat

c1   =  λX. λs:X→X. λz:X. s z;
▸ c1 : CNat

c2   =  λX. λs:X→X. λz:X. s (s z);
▸ c2 : CNat
```

Exercise

- Implment csucc for CNat so that c_i = csucc c_{i-1}

```
CNat = ∀X. (X→X) → X → X;
c₀  =  λX. λs:X→X. λz:X. z;
▸ c₀ : CNat
c₁   =  λX. λs:X→X. λz:X. s z;
▸ c₁ : CNat
c₂   =  λX. λs:X→X. λz:X. s (s z);
▸ c₂ : CNat

scc = λn. λs. λz. s (n s z);
```

Exercise

- Implment csucc for CNat so that c_i = csucc c_{i-1}

```
CNat = ∀X. (X→X) → X → X;
 c0  =  λX. λs:X→X. λz:X. z;
▸ c0 : CNat
 c1   =  λX. λs:X→X. λz:X. s z;
▸ c1 : CNat
 c2   =  λX. λs:X→X. λz:X. s (s z);
▸ c2 : CNat
csucc = λn:CNat. λX. λs:X→X. λz:X. s (n [X] s z);
▸ csucc : CNat → CNat
```

Extending System F

- Introducing advanced types by directly copying the extra rules
  - Tuples, Records, Variants, References, Recursive types

- PolyPair = $\forall X.\ \forall Y.\ \{X, Y\}$

Can you define list in System F?

- List =…
- nil = …
- cons = …

Can you define list in System F?

- List = ∀X. μA. <nil:Unit, cons:{X, A}>;
- nil = λX. <nil:Unit> as μA. <nil:Unit, cons:{X, A}>
- cons = λX. λn:X.λl:List.<cons={n, l [X]}> as μA. <nil:Unit, cons:{X, A}>

- What is the problem of the above list?

Can you define list in System F?

- List = ∀X. μA. <nil:Unit, cons:{X, A}>;
- nil = λX. <nil:Unit> as μA. <nil:Unit, cons:{X, A}>
- cons = λX. λn:X.λl:List.<cons={n, l [X]}> as μA. <nil:Unit, cons:{X, A}>

- What is the problem of the above list?
  - cons 1 (cons 2 nil) is not well typed
- Full polymorphism list requires System Fω

A pseudo solution

- List X = μA. <nil:Unit, cons:{X, A}>
- nil = λX.<nil:Unit> as List X
- cons = λX.λn:X.λl:List X.<cons={n, l [X]}> as List X

Church Encoding

- Read the book

Basic Properties

- Preservation
- Progress
- Normalization
  - Every typable term halts.
  - Y Combinator cannot be written in System F.

Efficiency Issue

- Additional evaluation rule adds runtime overhead.

$$(\lambda X.t_{12})\ [T_2] \longrightarrow [X \mapsto T_2]t_{12} \quad \text{(E-TAPPTABS)}$$

- Solution:
  - Only use types in type checking
  - Erase types during compilation

Removing types

$$
\begin{array}{lcl}
\mathit{erase}(\mathtt{x}) & = & \mathtt{x} \\
\mathit{erase}(\lambda\mathtt{x}\mathtt{:}\mathtt{T}_1\mathtt{.}\ \mathtt{t}_2) & = & \lambda\mathtt{x}\mathtt{.}\ \mathit{erase}(\mathtt{t}_2) \\
\mathit{erase}(\mathtt{t}_1\ \mathtt{t}_2) & = & \mathit{erase}(\mathtt{t}_1)\ \mathit{erase}(\mathtt{t}_2) \\
\mathit{erase}(\lambda\mathtt{X}\mathtt{.}\ \mathtt{t}_2) & = & \mathit{erase}(\mathtt{t}_2) \\
\mathit{erase}(\mathtt{t}_1\ [\mathtt{T}_2]) & = & \mathit{erase}(\mathtt{t}_1)
\end{array}
$$

t reduces to t' $\Rightarrow$ erase(t) reduces to erase(t')

A Problem in Extended System F

- Do the following two terms the same?
  - let f=(λX.error) in 0;
  - let f=error in 0;

A Problem in Extended System F

- Do the following two terms the same?
  - let f=(λX.error) in 0;
  - let f=error in 0;

- A new erase function

$$
\begin{array}{lcl}
erase_v(\mathtt{x}) & = & \mathtt{x} \\
erase_v(\lambda\mathtt{x}:\mathtt{T}_1.\ \mathtt{t}_2) & = & \lambda\mathtt{x}.\ erase_v(\mathtt{t}_2) \\
erase_v(\mathtt{t}_1\ \mathtt{t}_2) & = & erase_v(\mathtt{t}_1)\ erase_v(\mathtt{t}_2) \\
erase_v(\lambda\mathtt{X}.\ \mathtt{t}_2) & = & \lambda\_.\ erase_v(\mathtt{t}_2) \\
erase_v(\mathtt{t}_1\ [\mathtt{T}_2]) & = & erase_v(\mathtt{t}_1)\ \mathrm{dummyv}
\end{array}
$$

Wells' Theorem

- Can we construct types in System F?
  - One of the longest-standing problems in programming languages
  - 1970s – 1990s
- [Wells94] It is undecidable whether, given a closed term m of the untyped λ-calculus, there is some well-typed term t in System F such that $erase(t) = m$.

Rank-N Polymorphism

- In AST, any path from the root to an ∀ passes the left of no more than N-1 arrows
  - $\forall X. X \to X$: Rank 1
  - $(\forall X. X \to X) \to Nat$: Rank 2
  - $((\forall X. X \to X) \to Nat) \to Nat$: Rank 3
  - $Nat \to (\forall X. X \to X) \to Nat \to Nat$: Rank 2
  - $Nat \to (\forall X. X \to X) \to Nat$: Rank 2

- Rank-1 is HM-system
- Type inference for rank-2 is decidable
- Type inference for rank-3 or more is undecidable

Term Impredicative

- A term in logic

- A quantifier whose domain includes the very thing being defined