Simple-Typed Lambda Calculus

为 lambda calculus 添加形式化的类型系统

Type System

Basic

基本尝试：使用函数的参数和返回类型来定义函数类型，函数体内的 free variable 的类型需要依据上下文而定，即 context

对于 STLC (Simple-Typed Lambda Calculus) 来说，定义类型为

$$\tau, \sigma ::= T \mid \sigma \to \tau$$

T 为基本类型（如 int），而 $\tau \to \sigma$ 为函数，其中 $\to$ 是右结合的

同时也需要扩充 lambda term 原本的定义

$$M, N ::= x \mid \lambda x : \tau . M \mid M \ N$$

Reduction rules 同理

Typing judgment 是对 term 分配类型的语句，形如

$$\Gamma \vdash M : \tau$$

语义为：在上下文 Γ 下，M 是一个类型为 τ 的**良类型** term

其中 Γ 是 typing context，其定义为

$$\Gamma ::= \cdot \mid \Gamma, x : \tau$$

对于有 free variable 的 term 来说，Γ 中包含了所有 free variable 的信息，而对于 closed term 来说，Γ 为空（即 $\cdot$）

$\vdash$ 表示推导，即根据 context 推导类型信息

则有以下基本的 Typing rules

$$\overline{\Gamma, x : \tau \vdash x : \tau}$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma.\, M) : \sigma \to \tau}$$

Soundness and Completeness

定义 sound 和 complete 为

- sound：系统不会接受错误的程序，没有假阴性，sound type system 称为 type-safe
- complete：系统不会拒绝正确的程序，没有假阳性

但是对于任何图灵完备的编程语言来说，会出错的程序的集合是不可判定的，因此类型系统不能做到同时满足 sound 和 complete

因此实践中的做法是选择 soundness，同时尽可能减少 false positive

soundness 满足

$$\cdot \vdash M : \tau \land M \to^* M' \implies \cdot \vdash M' : \tau \land (M' \in \text{Values} \lor \exists M''.\, M' \to M'')$$

即对于一个良类型的 term 进行 reduction，结果只有两种可能：或者其能继续推导，或者停止在一个值上（值的定义是语言的语义决定的，对于 lambda calculus 来说 lambda abstraction 和 constant 都是值），且推导过程中类型不变

soundness 的定义基于两个基本的引理

- preservation：良类型的 term 只会推导到同样类型的良类型 term，即 $\cdot \vdash M : \tau \land M \to M' \implies \cdot \vdash M' : \tau$
- progress：良类型的 term 或是值或是能继续推导，即 $\cdot \vdash M : \tau \implies M \in \text{Values} \lor \exists M'.\, M \to M'$

由于良类型的 term 推导一定会停止，故形如 (λx. x x) (λx. x x) 的 term 无法得出其类型

Extend STLC

可以扩展 STLC，步骤为

- 扩展语法（type 和 term）
- 扩展操作语义（reduction rule）
- 扩展类型系统（typing rule）
- 扩展 soundness 证明

Product

扩展类型

$$\tau, \sigma ::= \cdots \mid \sigma \times \tau$$

以及 term

$$M, N ::= \cdots \mid < M, N > \mid \text{proj1} < M, N > \mid \text{proj2} < M, N >$$

其中 proj1 获取 product 中的第一个 term，而 proj2 获取第二个 term

由此可以扩展 reduction rule 和 typing rule

Reduction rules

$$\frac{}{\text{proj1} <M, N> \to M} \qquad \frac{}{\text{proj2} <M, N> \to N}$$

$$\frac{M \to M'}{<M, N> \to <M', N>} \qquad \frac{N \to N'}{<M, N> \to <M, N'>}$$

$$\frac{M \to M'}{\text{proj1}\ M \to \text{proj1}\ M'} \qquad \frac{M \to M'}{\text{proj2}\ M \to \text{proj2}\ M'}$$

Typing rules

$$\frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash <M, N> : \sigma \times \tau}\ (\text{pair})$$

$$\frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \text{proj1}\ M : \sigma}\ (\text{proj1}) \qquad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \text{proj2}\ M : \tau}\ (\text{proj2})$$

Sum

扩展类型

$$\tau, \sigma ::= \cdots \mid \sigma + \tau$$

以及 term

$$M, N ::= \cdots \mid \text{left}\ M \mid \text{right}\ M \mid \text{case}\ M\ \text{do}\ M1\ M2$$

sum 类型有点类似 java 的子类型或是 c 的 union

由此可以扩展 reduction rule 和 typing rule

Sum type: reduction rules

$$\frac{}{\text{case (left M) do M1 M2} \rightarrow \text{M1 M}}$$

$$\frac{}{\text{case (right M) do M1 M2} \rightarrow \text{M2 M}}$$

$$\frac{\text{M} \rightarrow \text{M}'}{\text{case M do M1 M2} \rightarrow \text{case M}' \text{ do M1 M2}}$$

$$\frac{\text{M1} \rightarrow \text{M1}'}{\text{case M do M1 M2} \rightarrow \text{case M do M1}' \text{ M2}}$$

$$\frac{\text{M2} \rightarrow \text{M2}'}{\text{case M do M1 M2} \rightarrow \text{case M do M1 M2}'}$$

$$\frac{\text{M} \rightarrow \text{M}'}{\text{left M} \rightarrow \text{left M}'}$$

$$\frac{\text{M} \rightarrow \text{M}'}{\text{right M} \rightarrow \text{right M}'}$$

Sum type: typing rules

$$\frac{\Gamma \vdash \text{M}: \sigma}{\Gamma \vdash \text{left M} : \sigma + \tau} \text{(left)} \qquad \frac{\Gamma \vdash \text{M}: \tau}{\Gamma \vdash \text{right M} : \sigma + \tau} \text{(right)}$$

$$\frac{\Gamma \vdash \text{M}: \sigma + \tau \quad \Gamma \vdash \text{M1}: \sigma \rightarrow \rho \quad \Gamma \vdash \text{M2}: \tau \rightarrow \rho}{\Gamma \vdash \text{case M do } M1\ M2 : \rho} \text{(case)}$$

Recursion

在无类型的 lambda calculus 中，递归是通过 fixpoint 实现的，而 fixpoint 是通过 combinator 得到的

由于良类型的 term 一定会终止，故不能为 combinator 定义类型，于是为了兼容，定义一个 term 来显式表示递归：$\text{fix } M$

由此可以得出 rule

$$\frac{}{\text{fix } \lambda x.\, M \to M[\text{fix } \lambda x.\, M/x]}$$

$$\frac{M \to M'}{\text{fix } M \to \text{fix } M'}$$

以及类型推导

$$\frac{\Gamma \vdash M : \tau \to \tau}{\Gamma \vdash \text{fix } M : \tau}$$

Curry-Howard Isomorphism

类型系统和逻辑系统的联系

- propositions are types
- proofs are programs

逻辑命题

$$p, q ::= B \mid p \implies q \mid p \wedge q \mid p \vee q$$

和类型

$$\tau, \sigma ::= T \mid \sigma \to \tau \mid \sigma \times \tau \mid \sigma + \tau$$

是同构的

type 可以分为非空和空

- nonempty：存在该类型的 close term
- empty：不存在该类型的 close term

而将类型替换成逻辑命题，能被证明的命题就是 nonempty type

同理，提供一个 well-typed closed term，类型推导的过程就是逻辑证明的过程

但是 STLC 同构的只是 constructive 的命题逻辑，与传统命题逻辑区别在于不支持排中律

$$\frac{}{\Gamma \vdash p \vee (p \implies q)}$$

需要显式表明排中律

$$(p \vee (p \implies q)) \wedge (p \implies r) \wedge ((p \implies q) \implies r) \implies r$$

而对于递归来说，fix 的类型规则可以用来证明任何命题，故逻辑系统是不一致的

不同的逻辑系统均有对应的类型系统