实现类型论中的高端操作

∞-type Café 暑期学校讲座

Tesla Zhang

May 29, 2023

前置知识

- 熟悉如何实现 Martin-Löf 类型论中的简单部分.
  - 可以通过 Mini-TT 的代码学习.
- 阅读过类似 Agda 的语言.

内容总览

- 隐式参数
- 双向的表达式判等
- 三向类型检查
- (逆)归纳类型的模式匹配
- 叠加态求值

代码中的术语表

- `TCM`: 类型检查器的 Monad, 需要支持抛出异常和全局变量.
- `Term`: 已经通过类型检查的表达式.
- `unify`: 表达式判等的函数.
- `spine`: 参数列表. 可以视为一组 `Term`.
- `Meta`: 未知的表达式, 用于表示自动插入的隐式参数.
- `WHNF`: 只知道最外层构造子的表达式 (弱头骑士异闻录).

隐式参数

本质上是 elaboration-zoo 这个项目的扩展讲解.

何为隐式参数?

`id : {A : Type} → A → A`

调用时可以写 `id 114514`.

如何阅读隐式参数?

- 从后往前读, 因为编译器也是这样进行推导的.

如何实现隐式参数?

- 将 `id 114514` 展开成 `id {?} 114514`.
- 于是 `id {?} : ? → ?`, 进一步检查 `114514 : ?`.
- 编译器发现 `? ≡ Nat`, 遂补全程序中省略的内容.

隐式参数的技术挑战

求解程序中省略的内容.

求解很简单吗?

```
test : (a : ?) (B : Type) (b : B) → a ≡ b
test a B b = refl
```

若将 `?` 求解为 `B`, 那就出大事了.

变量求解的作用域问题

- 对于一个省略的程序, 记录它所在的语境.
- 求解时, 要求它的解必须在对应的语境里合法才行.

大致实现:

- 用一个 `Meta(ref, [a, b, c, ...])` 表示 `? a b c ...` 这样的函数应用,
- 创建 `Meta` 的时候, 首先把所有的局部变量加上去.

大致实现:

```
unify(Meta(ref, spine), u) = do
  确保 spine ⊢ u 合法
  修改 ref, 指向 u
```

其中 `Meta` 是省略的表达式, `spine` 是它出现的位置能引用的变量.

大致实现:

- `spine` 中的变量必须唯一, 若重复, 则不能让它被用到.
- 否则不能确定解出来的程序到底使用的是哪一个变量.

大致实现:

- 在化简代码时, 若 `Meta(ref, spine)` 中的 `ref` 已经有解, 那么化简为这个解.

其它情况:

- 确保解出来的程序没有递归, 即它没有引用自己. 很好做.
- 遇到 `unify(Meta(ref1, spine1), Meta(ref2, spine2))` 怎么办?

`Meta` 之间的交互问题:

- 遇到 `unify(Meta(ref1, spine1), Meta(ref2, spine2))` 怎么办?
- 若 `unify(Meta(ref1, spine1), u)` 中, `u` 的子表达式中有 `Meta(ref2, spine2)` 怎么办?

对 `Meta` 的分析:

- `Meta(ref, spine)` 的解不一定会用到 `spine` 中的全部变量.
- 在求解之前, 无法知道到底有哪些会被用到.

粗暴的解决方法:

- 遇到这类情况, 将这些等式存起来,
- 等到相关的 `Meta` 被求解后再进行检查.

其它需求:

- 需要有一个可以修改 `ref` 指向的值的机制,
- 适合有可变引用的语言实现, 或者用 `ST` Monad.

新问题:

- 是否需要检查 `Meta` 的类型?

类型检查:

- 若我生成 `Meta` 的时候, 有 `? : A`, 那么它求解出来的结果也必须确保是 `A` 的实例才行.

类型检查:

- 需要有 `check : Term → Term → TCM ()`,
- 若 `Term` 中没有足够的类型注解, 需要实现成双向的类型检查.

结果:

- 维护一组未求解的等式,
- 额外一套双向类型检查的代码, 下称「二次检查器」.

Agda 的 `lossy-unification`

```
unify(FnCall(f, a), FnCall(f, b)) = do
  unify a b
```

函数调用直接化简到参数.

若同一个函数对不同输入可能返回相同结果, 则这种情况会过度求解.

```
f true = 114514
f false = 114514
```

此时 `f true ≡ f ?` 不应该产生任何关于 `?` 的信息, 但这种情况下会被求解 `? = true`.

若已知函数是单射, 那么就可以这样化简. 需要在编译器内部特判一个函数是否为单射.

表达式判等

- 在 Mini-TT 里面, 直接比较 `reify . eval` 的结果.
- 不支持 `Meta` 求解.

如何实现两个表达式的判等?

```
unify(Ref(var1), Ref(var2)) = do
  if var1 == var2 then return () else fail
```

这是正义吗?

考虑单位类型 (零元元组):

$$\frac{(x : \top) \in \Gamma}{\Gamma \vdash x \equiv \star : \top}$$

根据这一规则, $\top$ 类型的任何实例都相等, 因此该类型两个不同的变量也相等.

表达式判等需要知道变量的类型, 才能做出公正的判断!

这也符合 Martin-Löf 类型论中的判断的写法:

$$\Gamma \vdash u \equiv v : A$$

类型和语境都在那里!

表达式判等其实也是一个双向的过程:

- `compareTyped : Term → Term → Term → TCM ()`
- `compareUntyped : Term → Term → TCM Term`

双向类型检查

- `inherit`: 完全知道类型, 检查表达式
- `synthesize`: 完全不知道类型, 检查表达式

被忽略的需求

考虑如下判断:

$$u : A$$

对于 A, 我们需要什么?

$$u : A$$

A 必须是合法的类型! 怎么检查这一点?

第一反应: 检查

$$A : \mathcal{U}$$

如果我有多个宇宙怎么办?

- 双层类型论: 分为纤类型的宇宙和外类型的宇宙
- 一致的类型论: 宇宙分层

用什么替代 u?

- 实际上哪个都有可能!
- 总不可能一个一个试吧?

另一个思路:

- 使用 `synthesize`, 然后看它返回的类型是不是个宇宙.
- 如果这个类型是个 `Meta`, 它的类型也只能是 `Meta` 而不是某个我们已知的宇宙.
- 无法支持推导 `λ x. x + 1` 的参数.

答案: 引入第三个类型检查的方向

- `universe`: 知道我需要一个类型, 返回这个类型和它的宇宙层级.
- 将 `synthesize` 中的一部分代理过来.
- 不需要判断某个东西是不是宇宙.

第三个方向:

```
universe : Expr → TCM (Term, Int)
```

新需求:

- 要是能支持推导 `λ x. x + 1` 的参数类型就好了!
- 在 `universe` 里面生成 `Meta` 时特殊处理.
  - 使得求解时我再保证一下它解出来的东西是个类型.

等等!

- 「求解时我再保证一下它解出来的东西是个类型」?
- 二次检查器也变成三向类型检查!
- 表达式判等也变成三向!

```
compareType : Term → Term → TCM (Term, Int)
```

我整个人都是三个方向的!

归纳类型:

```
f : Nat → Nat
f zero = zero
f (suc a) = a
```

检查的步骤:

- 确保每一条分支没有类型错误
- 确保所有分支包含了所有的情况

逆归纳类型

用解构函数而不是构造函数来定义:

```
codata Stream A where
  head : Stream A → A
  tail : Stream A → Stream A
```

调用: 若 `a : Stream A` 则允许 `a.head`.

模式匹配例子:

```
zipWith : (A → B → C) → Stream A → Stream B → Stream C
zipWith f xs ys .head = f (xs.head) (ys.head)
zipWith f xs ys .tail = zipWith f (xs.tail) (ys.tail)
```

优雅的对偶性:

- 归纳类型: 对构造子模式匹配
- 逆归纳类型: 对解构子模式匹配

模式匹配的类型检查:

检查某个模式是否符合类型, 并对目标类型进行消去.

```
f : (x : D) → C
f (con a b ...) = ?
```

```
f : (x : D) → C
f (con a b ...) = ?
```

- 检查 `con` 是否是 `D` 的构造子,
- 获取该构造子的类型列表, 并依次检查 `a` 是否属于第一个类型, `b` 是否属于第二个类型, ...
- 若成功, 将 `C` 代换: `C[x := con a b ...]`

逆模式匹配的类型检查:

```
prod : R
prod .decon a b ... = ?
```

```
prod : R
prod .decon a b ... = ?
```

- 检查 `decon` 是否是 `R` 的解构子,
- 获取该解构子的类型列表, 并依次检查 `a` 是否属于第一个类型, `b` 是否属于第二个类型, ...
- 若成功, 将 `R` 代换为 `decon` 的返回类型

混合例子:

```
f : (x : D) → R
f (con a b) .decon c d = ?
```

混合例子:

```
f : (x : D) → R
f (con a b) .decon c (con2 a b) .decon2 e f = ?
```

两种等号

```
data _=_ {A : Type} (x : A) : A → Type where
  refl : x = x

codata _=_ {A : Type} (x : A) : A → Type where
  corefl : x = x → Unit
```

表达式求值中的重复计算

$(\lambda x.\langle x, x, x, x, x, x, x\rangle)(f(u))$

假设: $f(u)$ 的计算很昂贵.

- 将它代换进左边再整个求值, 会导致它在之后可能被计算 7 次.
- 若我们不需要取出左边的 7 元组中的成员, 那么直接代入 $f(u)$ 反而更好.

你永远无法知道到底先算哪个更好.

等到你确定下你需要的信息, 已经晚了.

解决方法

- 用一个「叠加态」来保存表达式的求值结果.

```
data Val = ...
  | VCall FnName Spine ~Val
```

- 函数名, 参数表, 求值结果 (惰性).

- 若需要输出该表达式, 就输出函数名和参数表.
- 若需要进行表达式判等, 就用后一个.

对 `WHNF` 进行函数应用:

```
vapp :: Val -> Val -> Val
vapp (VLam _ t)      ~u = t u
vapp (VLoc x sp)     ~u = VLoc x (SApp sp u)
vapp (VCall x sp t) ~u = VCall x (SApp sp u) (vapp t u)
```

- 在最后一条规则中, `u` 被写了两遍, 这意味着 `u` 至多被计算一次.
- `Call x (SApp sp u)` 是严格求值, 此处不触发任何计算.