基于函数式程序的元编程理论

上海交通大学第 38 期 PRP 项目答辩

刘涵之[1]
指导老师：曹钦翔

[1] 电子信息与电气工程学院
上海交通大学

2021 年 3 月 24 日

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

总览

元编程定义

元编程

- 编写程序视为数据的计算机程序的过程
- 用程序来编写、修改另一个程序

元编程性质很有用

- 越来越多的语言开始对元编程提供支持
- 元编程代码的开发数量呈指数增长
- 灵活性高，相比设计全新的程序语言而言学习门槛低

目前对元程序与对象程序的行为之间关系缺乏针对性的工作

函数式编程与元编程

常见的函数式语言都支持元编程，支持基于构造子的类型系统

- OCaml
- Haskell
- Standard ML
- ...

只要选取一个很小的函数式程序语言片段就可以写出一个自身语言的解释器

无类型 λ 演算

语法：

$$\begin{array}{lll} \langle term \rangle & ::= & \text{<x>} \quad \text{(变元)} \\ & | & \lambda\ \text{<x>.<term>} \quad \text{(函数抽象)} \\ & | & \text{<term> <term>} \quad \text{(应用)} \end{array}$$

β-归约小步语义：

$$\frac{value\ t}{app\ (abs\ x\ t')\ t \rightarrow t'[t/x]} \quad (step\_beta)$$

语法拓展

$$
\begin{aligned}
\langle term \rangle \quad ::= \quad & \langle x \rangle \quad \text{(变元)} \\
| \quad & \lambda\, \langle x \rangle . \langle term \rangle \quad \text{(函数抽象)} \\
| \quad & \langle term \rangle\ \langle term \rangle \quad \text{(应用)} \\
| \quad & \langle c \rangle \quad \text{(构造子)} \\
| \quad & \textbf{match}\ \langle term \rangle\ \langle c \rangle\ \langle term \rangle\ \langle term \rangle \quad \text{(模式匹配)} \\
| \quad & \textbf{rec}\ \langle term \rangle \quad \text{(递归算子)}
\end{aligned}
$$

```
Inductive tm : Type :=
  | const: string → tm
  | var : string → tm
  | app : tm → tm → tm
  | abs : string → tm → tm
  | mat : tm → string → tm → tm → tm
  | rec : tm → tm.
```

小步操作语义

模式匹配：

$$\frac{Forall\ value\ xts}{mat\ (fold\_left\ app\ xts\ (const\ s)\ s\ t1\ \ t2 \to fold\_left\ app\ xts\ t1} \quad (step\_match)$$

$$\frac{s1 <> s2}{mat\ (fold\_left\ app\ ts\ (const\ s1)\ s2\ t1\ \ t2 \to t2} \quad (step\_unmatch)$$

递归算子：

$$\frac{t1 \to t2}{rec\ \ t1 \to rec\ \ t2} \quad (step\_rec)$$

$$\frac{value\ t2}{app\ (rec\ (abs\ x\ t1))\ \ t2 \to app\ \ t1[rec(abs\ x\ t1)/x]\ \ t2} \quad (step\_rec\_reduce)$$

类型种类

可以定义元语言的项的类型

```
Inductive typ: Type :=
| T_nat
| T_bool
| T_tm
| T_list (t: typ)
| T_ascii
| T_string
| T_option (t: typ)
| T_prod (t1 t2: typ)
| T_arrow (t1 t2: typ).
```

类型规则

以及对应的类型规则

```
Inductive has_type: (string → typ → Prop) → tm → typ → Prop :=
| HT_rec: forall G a A B,
    has_type G a (T_arrow (T_arrow A B) (T_arrow A B)) →
    has_type G (rec a) (T_arrow A B)
| HT_mat1: forall G e1 e2 e3 T1 T2 s,
    has_type G e1 T1 →
    has_type G e2 T2 →
    has_type G e3 T2 →
    has_type G (mat e1 s e2 e3) T2
| HT_mat2: forall G e1 e2 e3 T1 T2 s A,
    has_type G (const s) (T_arrow A T1) →
    has_type G e1 T1 →
    has_type G e2 (T_arrow A T2) →
    has_type G e3 T2 →
    has_type G (mat e1 s e2 e3) T2
```

指称语义的形式化

形式化元语言的指称语义

```
Inductive typed_sim :
(string → typ → Prop) → (string → forall A : Type, A → Prop)
→ tm → typ → forall A : Type, A → Prop :=
  | typed_sim_app :
      forall G rho_var (t1 t2 : tm) (t_a t_b : typ) A B a b,
        type_inj t_a A → type_inj t_b B →
        typed_sim G rho_var t1 (T_arrow t_a t_b) (A→ B) a →
        typed_sim G rho_var t2 (t_a) A b →
        typed_sim G rho_var (app t1 t2) t_b B (a b)
  | typed_sim_var :
      forall G (rho_var : (string → forall A : Type, A → Prop))
      (t_s : typ) (s : string) t,
        has_type G (var s) t_s → rho_var s (type_inj_f t_s) t →
        typed_sim G rho_var (var s) t_s (type_inj_f t_s) t
```

指称语义的形式化

```
| typed_sim_abs :
    forall s t1 G rho_var t_t1 t_X a,
      has_type G (abs s t1) (T_arrow t_X t_t1) →
      (forall x : (type_inj_f t_X),
        typed_sim (var_type_subst G s t_X)
          (var_subst rho_var s (type_inj_f t_X) x)
          t1 t_t1 (type_inj_f t_t1) (a x)) →
        typed_sim G rho_var (abs s t1)
          (T_arrow t_X t_t1) (type_inj_f (T_arrow t_X t_t1)) a
```

元编程性质

在 Coq 中定义：

- 对象语言的项和执行程序（项 t，项 t_1，执行程序 $next\_state$）
- 以及它们在元语言中对应的表示形式（t^{reify}，t_1^{reify}，$next\_state^{reify}$）

元编程性质：

- 如果一个对象语言的项 t 经过一步运行 ($next\_state$) 后等于项 t_1，那么元语言中的项 t^{reify} 应用（app）到 $next\_state^{reify}$ 的多步执行结果等于 t_1^{reify}。

```
Definition reify_correctness_statement run:=
  forall t ,
  (next_state t = None →
    multi_step (app run (reify t)) _None) ∧
  (forall t',
    next_state t = Some t' →
     multi_step (app run (reify t)) (app _Some (reify t'))).
```

证明路径

- 使用 Ltac 自动化地生成用元语言编写的自身执行程序并指称到在 Coq 中的元语言的执行程序 *next_state*
- 证明有关指称语义和小步操作语义相关的一般性质（项经过小步语义后，指称的保持性）
- 利用相关性质证明最终结论

保持性引理

指称在 β-归约上的保持性

```
Lemma sim_beta_reduction:
    forall G rho_var (t1 t2 : tm) (tx : typ) (X : Type) x s,
        empty_context_gamma G → empty_context_rho rho_var →
        typed_sim G rho_var (app (abs s t1) t2) tx X x →
        typed_sim G rho_var (tm_subst [(s,t2)] t1) tx X x.
```

指称在小步语义上的保持性

```
Lemma sim_step:
  forall G (rho_var : (string → forall A : Type, A → Prop))
  (A: Type) (a: A) t t' t_t,
  empty_context_gamma G → empty_context_rho rho_var →
  typed_sim G rho_var t t_t A a →
  step t t' →
  typed_sim G rho_var t' t_t A a.
```

使用 Ltac 实现 Coq 到元语言的自动化转换

实现了自动化策略 *xy*，例子 my_func：

```
Inductive Pair: Type := | pair: nat→ nat→ Pair.
Definition my_func (n1 n2: nat): Pair := (pair (S n1) (S (S (S O)))).
```

```
Definition myfunc_sim:
forall (rho_const: (string → forall A : Type, A → Prop))
(rho_var: (string → forall A : Type, A → Prop)), ·sig tm (fun t: tm ⇒ (
  rho_const "pair" _ pair →
  rho_const "S" _ S →
  rho_const "O" _ O →
  sim rho_const rho_var t _ my_func
  )).
Proof.
  intros. eexists. intros. unfold my_func. repeat xy.
Qed.
```

使用保持性证明元编程理论：引理

Lemma 1 (sim_multistep)

元语言的项 t 的指称是 a，同时项 t 经过多步后变为项 t'，那么元语言的项 t' 的指称也是 a。

Lemma 2 (typ_preservation)

元语言的项 t 具有类型 T，同时项 t 经过一步后变为项 t'，那么元语言的项 t' 也有类型 T。

使用保持性证明元编程理论：引理

Lemma 3 (app_Some_only_if)

如果 *app t1 t2* 是 *const_hd_val*，同时它具有类型 *T_option T_tm*，那么 *t1* = *_Some*

Lemma 4 (val_None)

如果 *t* 是 *val*，并且 *t* 具有类型 *T_option T_tm*，如果 *t* 的指称是 *None*，那么 *t* = *_None*。

Lemma 5 (val_Some)

如果 *t* 是 *val*，并且 *t* 具有类型 *T_option T_tm*，如果 *t* 的指称是 *Some a*，那么 *t* = *app _Some (reify a)*。

使用保持性证明元编程理论：证明

根据类型的保持性（引理2），已知 (app run (reify t)) 的类型是 T_option T_tm，那么多步归约后，它的类型也是 T_option T_tm。已知 (app run (reify t)) 的指称是 (next_state t)。首先存在一个 a，使得 (app run (reify t)) 可以多部归约到 a，a 此时是个 val，a 的类型也是 T_option T_tm，根据指称在小步语义上的保持性定理，可以得到 a 的指称是 (next_state t)。考虑到 next_state t 的结果要么是 None，要么是 Some b。

- 对于 None 的情况，可以得到 a 的指称是 None，根据引理4，可以得到 a = _None，得证。
- 对于 Some b 的情况，可以得到 a 的指称是 Some b，根据引理5，可以得到 a = app _Some (reify b)，得证。

证明结论

- 选取了含构造子的 λ 演算作为元语言与目标语言
- 在 Coq 证明助手中形式化了元语言的小步语义与指称语义
- 通过 Ltac 实现了指称（Coq 程序）到元语言的自动化转换

本项目为证明函数式程序的元编程性质提供了一条切实可行的证明路径，展现了指称语义在元编程理论证明中的应用前景