PA2 Report

计71 钟闰鑫 2017010306

实现内容

抽象类

在`errors`添上`NotOverrideAllAbstractFunc`和`InstantiateAbstractClass`两个类以及其对应输出

NotOverrideAllAbstractFunc

具体实现中，定义了一个

```
let mut abs_func_set: HashMap<&'a str, HashSet<&'a str>> = HashMap::new();
```

用来存放所有的一个类(用类名作为`key`)及其对应的未被重载的抽象方法(对应一个`HashSet`)

`symbol_pass`中的`class_def`和`func_def`修改为

```
fn class_def(&mut self, c: &'a ClassDef<'a>, checked: &mut HashSet<Ref<'a,
ClassDef<'a>>>, abs_func_map: &mut HashMap<&'a str, HashSet<&'a str>>) ->
HashSet<&'a str>
fn func_def(&mut self, f: &'a FuncDef<'a>, class_abs_func_set: &mut
HashSet<&'a str>)
```

现在`class_def`返回对应解析的class的未重载的抽象方法

在`class_def`内部，首先查看是否有`parent`，有的话则解析它并得到其对应的未重载的抽象方法`HashSet`，

每次调用`func_def`将这个set传入，若当前是一个抽象类则加入进set；若是重载了一个抽象类则从set中删除

- set需要注意的问题主要在于何时传入和何时删除，传入的位置很好确定，判断出来是抽象函数即可，而对于删除的位置，由于原框架有关于函数是否为重载函数的判断逻辑，因此也只需要在这里加入对抽象函数的判断以及删除即可

InstantiateAbstractClass

在`expr`的`NewClass`类型节点处理过程中加入抽象类的判断即可

局部类型推断

在`error`中添加`InferVoid`错误

由于类型推断大多数情况需要首先知道右端表达式的类型，这一步在框架中的`type_pass`阶段完成，因此对于此类情况的类型推断直到`type_pass`才完成

在`symbol_pass`中的`var_def`中对于一般情况进行判断，如果类型为`void`则报错

在`type_pass`中的`stmtKind::LocalVarDef`逻辑中，在处理了之中的`expr`后得知其类型，根据类型是否为`void`报错

函数变量

原来的AST节点VarSel只能对应一个VarDef，现在一个VarSel还有可能是一个FuncDef，因此其结构需要被改变

```
pub struct VarSel<'a> {
    pub owner: Option<Box<Expr<'a>>>,
    pub name: &'a str,
    pub var: Cell<VarSelContent<'a>>,
}
#[derive(Copy, Clone)]
pub enum VarSelContent<'a> {
    Var(&'a VarDef<'a>),
    Func(&'a FuncDef<'a>),
    Empty,
}
```

Void参数

在`errors`中添加`NonVoidArgType`

在`TypeCk`的`ty`函数中添加对于`NonVoidArgType`的判断，遍历lambda表达式的所有参数，若为`Void`则报错

函数调用

重写type_pass的对Call节点的处理

现在将Call节点内的访问权限的判断放到了VarSel节点的处理函数内，而Call节点现在只关注这个expr是不是callable的，以及参数是否匹配

lambda表达式

AST节点结构

更改了原来的AST中lambda表达式的结构，变为

```
pub struct LambdaDef<'a> {
    pub loc: Loc,
    pub finish_loc: Loc,
    pub name: String,
    pub param: Vec<&'a VarDef<'a>>,
    pub ret_param_ty: Cell<Option<&'a [Ty<'a>]>>,
    pub kind: LambdaKind<'a>,
    pub scope: RefCell<Scope<'a>>,

    //if kind is block, then use scope of block instead of this
    pub local_scope: RefCell<Scope<'a>>,
}
impl<'a> LambdaDef<'a> {
    pub fn ret_ty(&self) -> Ty<'a> {
        self.ret_param_ty.get().unwrap()[0]
    }
}
```

```rust
pub enum LambdaKind<'a> {
    Expr(Box<Expr<'a>>),
    Block(Block<'a>),
}
```

在构造AST的过程中设置lambda表达式的名字，整个语句的末尾位置(`finish_loc`)等

- scope域

  这里放lambda表达式的参数的symbol

- local_scope域

  由于原来框架的expr不含有scope，考虑到只有类型为lambda的expr才需要有一个scope，因此选择在这里加

  如果lambda体类型为block，则直接使用block里面的scope(这个scope里放置在block内部定义的symbol)

  如果lambda体类型为expr，则使用这个local_scope

  事实上，上述两种scope里面只会将那些在此处定义的symbol放入，而一个lambda体类型为expr的情况下，里面是不可能存在一般的变量定义的，这个local_scope是完全只是为了支持嵌套的lambda(以及方便打印符号表)，例如

  ```
  func (int x) => x + (func (int y) => y)(1)
  ```

  此时这个内部的lambda(`func (int y) => y`)的symbol则需要被放在local_scope里面

Symbol与Scope

添加新的`Symbol`

```rust
#[derive(Copy, Clone)]
pub enum Symbol<'a> {
    Var(&'a VarDef<'a>),
    Func(&'a FuncDef<'a>),
    Lambda(&'a LambdaDef<'a>), //new
    This(&'a FuncDef<'a>),
    Class(&'a ClassDef<'a>),
}
```

当发现一个lambda表达式定义的时候，则得到其对应的symbol并放入当前scope中

添加新的`ScopeOwner`

```rust
#[derive(Copy, Clone)]
pub enum ScopeOwner<'a> {
    Local(&'a Block<'a>),
    Param(&'a FuncDef<'a>),
    LambdaParam(&'a LambdaDef<'a>), //new
    LambdaExprLocal(&'a LambdaDef<'a>), //new
    Class(&'a ClassDef<'a>),
    Global(&'a Program<'a>),
}
```

其中LambdaParam是类似原本框架中的Param写的，当在symbol_pass中遇到一个lambda时，则将LambdaParam作为scopeowner的kind，对应这个lambda，其中的参数对应的symbol放入到`LambdaDef`中的scope域中

而之后，若这个lambda体为expr类型，则再打开一层scope，以LambdaExprLocal作为scopeowner，在这个lambda体(expr类型)中定义的symbol(其实只有可能是lambda类型)将会被放入到`LambdaDef`中的local_scope域中

之所以这样设计也是为了方便打印符号表

- 在定义好这些符号之后，需要重写symbol_pass阶段，由于lambda的引入，必须遍历所有出现的表达式(原框架忽略了一些表达式)

参数捕获

由于引入lambda表达式后，可能会出现在lambda表达式内部引用正在定义的名字，因此需要修改原框架中查找引用所使用的函数`lookup_before`

参考实验指导书所给的算法，现在AST节点维护了一个finish_loc来标明这条语句的结束位置，`lookup_before`则使用这个finish_loc来判断

```
    pub fn lookup_before(&self, name: &'a str, finish_loc: Loc) ->
(Option<Symbol<'a>>, bool) {
        let mut out_of_lambda = false;
        for &owner in
self.stack.iter().rev().chain(iter::once(&self.global)) {
            if let Some(sym) = owner.scope().get(name).cloned() {
                if !(owner.is_local() && sym.finish_loc() >= finish_loc) {
                    return (Some(sym), out_of_lambda);
                }
            }

            if !out_of_lambda && owner.is_lambda_param() {
                out_of_lambda = true;
            }
        }
        return (None, out_of_lambda);
    }
```

- lookup_before现在也能返回是否穿过了当前的lambda域(out_of_lambda)，也就是说，它找到的符号是不是在当前的lambda的外部，这个在访问权限中被使用

其他的工作主要在于在判断访问权限的时候考虑清楚所有情况，在对应位置插入报错

返回类型计算

由于原框架type_pass中对于各节点的处理函数只返回了是否有返回值(bool)，因此需要被重写，改成能够返回`(最后一条语句的返回值，内部所有return语句的返回值的list)`的二元组

具体的返回值计算方法则采用实验指导书中的算法

```
fn get_upper_ty(&mut self, ty_list: &[Ty<'a>], loc: common::Loc) -> Ty<'a>
fn get_lower_ty(&mut self, ty_list: &[Ty<'a>], loc: common::Loc) -> Ty<'a>
```

符号表打印

原来符号表跳过了一部分expr，现在需要遍历所有的(有可能出现lambda定义的)expr找到其中的lambda表达式打印出来，剩下的工作主要就是根据样例调整打印格式

问题回答

Q1

> 实验框架中如何实现根据符号名在作用域中查找该符号？在符号定义和符号引用时的查找有何不同？

实验框架中的作用域`Scope`被定义为一个`HashMap`，通过符号名(`str`)找到对应的`Symbol`，作用域被组织成了一个栈，每次进入一个新作用域则压栈

在作用域里查找符号采用自顶向下地遍历栈，将最先找到的一个符号以及对应的作用域返回

符号定义时的查找采用的是`lookup`函数，即直接自顶向下遍历栈，找到最开始找到的的`Symbol`(此时查找范围不包括定义在同一作用域位置在后面的变量，因为它们还未被"发现")，对类使用`lookup_class`，直接在全局作用域查找

符号引用时的查找采用的是`lookup_before`，将最先找到的在查找位置之前的一个符号和其作用域返回(因为此时定义的符号都已经放入`HashMap`，需要查找位置来进行先定义后使用的限制)

Q2

> 对AST的两趟遍历分别做了什么事？分别确定了哪些节点的类型？

第一趟遍历收集所有符号定义，将符号插入到对应的作用域中(`HashMap`)，确定了`ClassDef`，`FuncDef`节点的类型，检查重复定义，继承，是否定义Main类，抽象类是否重载完成，函数重载是否正确等问题

第二趟遍历确认各节点的返回类型(Expr lambda等)并进行类型检查，检查参数类型，返回值情况，访问权限等问题

Q3

> 在遍历AST时，是如何实现对不同类型的AST节点分发相应的处理函数的？

使用rust的match和enum，从enum中提取出对应的子节点，根据类型调用对应的函数(如类型为expr则调用expr(..)函数)，由此遍历所有AST节点并对其进行相应的处理