类型系统

类型系统

- 类型系统的作用：
  - 类型系统可以静态地检查源程序的语义错误
  - 类型系统可以为后续的代码生成以及优化提供信息
- MoonBit 中更多的语义分析
  - 死代码分析
  - 逃逸分析(escape analysis)
  - ...

类型系统

- MiniMoonbit 中类型的基本定义：
  - 基本类型：Unit, Bool, Int, Double
  - 复合类型：(T1, T2, ..), Array[T], (T1, T2, ..) -> T
- 顶层函数要求标注参数类型和返回值类型
  - 可以独立地对每个顶层函数进行类型检查
- 局部函数和变量可以省略类型标注
  - 可以让程序更加简洁
  - 类型推导：根据变量的使用情况，推导出变量的类型
  - 类型变量：TVar("a"), TVar("b")

类型变量的归一化(unification)


```
1. // pseudo-minimoonbit
2. fn main {
3.   fn fact(n) {
4.     if n <= 0 { 1 } else { n * fact(n - 1) }
5.   };
6.   ..
7. }
```

先对未进行类型标注的函数添加类型变量作为标注：

```
1. // pseudo-minimoonbit
2. fn main {
3.   fn fact(n: TVar("a")) -> TVar("b") {
4.     if n <= 0 { 1 } else { n * fact(n - 1) }
5.   };
6.   ..
7. }
```

类型变量的归一化(unification)

idea
基础软件中心

```
1. // pseudo-minimoonbit
2. fn main {
3.   fn fact(n: TVar("a")) -> TVar("b") {
4.     if n <= 0 { 1 } else { n * fact(n - 1) }
5.   };
6.   ..
7. }
```

- 因为 `<=` 的两个操作数的类型一致，所以 `n` 的类型被归一化到 `Int`
- 同理，因为 `*` 的两个操作数的类型一致，所以 `fact(n - 1)` 的类型被归一化到 `Int`
- 通过类型归一化，我们可以得到 `f: Int -> Int`
- 归一化的过程中，可能会出现类型不一致的情况，此时需要报错

类型变量的表示方式

- 使用可变数据类型来表示一个类型变量，并且在归一化的过程中，更新这个类型变量的值
  - 类型变量是 `None` 时，表示这个类型变量尚未被归一化到其他类型
  - 类型变量是 `Some(t)` 时，表示这个类型变量被归一化到了类型 `t`

```
1. enum Type {
2.   Var(mut ~t: Type?)
3.   ..
4. }
```

类型变量的表示方式

idea
基础软件中心

- 在生成语法树的过程中，当出现类型缺失的情况时，生成一个新的类型变量：

```
fn new_tvar() -> Type {
  Var(~t = None)
}
```

- 找到类型变量代表的实际类型：

```
fn repr(self: Type) -> Type {
  match self {
    Var(~t = Some(ty)) as tvar => {
      let actual_ty = ty.repr()
      tvar.t = Some(actual_ty)
      actual_ty
    }
    ty => ty
  }
}
```

类型变量的归一化(unification)

```
 1. fn unify(t1: Ty, t2: Ty) -> Unit!TyErr {
 2.   let t1 = t1.repr()
 3.   let t2 = t2.repr()
 4.   if physical_equal(t1, t2) { return }
 5.   match (t1, t2) {
 6.     (Int, Int) | (Bool, Bool) => ()
 7.     (TVar(~t=None) as tvar, ty) | (ty, TVar(~t=None) as tvar) => {
 8.       check_occur!(tvar, ty)
 9.       tvar.t = Some(ty)
10.     }
11.     .. // handle the function, tuple, and array type
12.     _ => raise TyErr
13.   }
14. }
```

类型变量的自引用检查

```
1. fn main {
2.   fn f(x) {
3.     x[0] = x
4.   }
5. }
```

- 一开始 `x` 的类型是 `TVar("a")`
- 在检查 `x[0] = x` 的过程中，因为 `x` 的类型是 `TVar("a")`，所以 `x[0]` 的类型被统一到 `TVar("a")`
- 因为 `x[0]` 的类型是 `TVar("a")`，所以需要检查 `x` 的类型是 `Array[TVar("a")]`
- 于是会产生 `TVar("a") = Array[TVar("a")]`

类型推导的上下文

- 需要一个全局的表来记录外部函数的类型
- 在检查函数体时，需要一个局部的表来记录局部变量的类型

```
1. pub let extenv : Map[String, @types.Type] = {
2.   "print_int": Fun([Int], Unit),
3.   "print_char": Fun([Int], Unit),
4.   ..
5. }
6. struct LocalCtx @immut/hashmap.T[String, Ty]
```

类型推导的实现

idea
基础软件中心

比较简单的几种情况：

```
fn infer(ctx : LocalCtx, e : Syntax) -> Type!TyErr {
  match e {
    Int(_) => Int; Bool(_) => Bool
    Var(x) =>
      match ctx._[x] {
        Some(t) => t
        None =>
          match extenv[x] {
            Some(t) => t
            None => {
              let t = new_tvar()
              extenv[x] = t
              t
            }
          }
      }
    ..
  }
}
```

递归函数的类型推导

```
 1. fn infer(ctx : LocalCtx, e : Syntax) -> Type!TyErr {
 2.   match e {
 3.     ...
 4.     LetRec({ name: (f, t), params, body }, rest) => {
 5.       let env_with_f = ctx._.insert(f, t)
 6.       let params_ty = params.map(fn { (_, t) => t })
 7.       let mut env_with_params = env_with_f
 8.       for p in params {
 9.         env_with_params = env_with_params.insert(p.0, p.1)
10.       }
11.       let body_ty = infer!(env_with_params, body)
12.       unify!(t, Fun(params_ty, body_ty))
13.       infer!(env_with_f, rest)
14.     }
15.   }
16. }
```

函数调用的类型推导

```
 1. fn infer(ctx : LocalCtx, e : Syntax) -> Type!TyErr {
 2.   match e {
 3.     ..
 4.     App(f, args) => {
 5.       let ret_ty = new_tvar()
 6.       let f_ty = infer!(ctx, f)
 7.       let args_ty = []
 8.       for a in args {
 9.         args_ty.push(infer!(ctx, a))
10.       }
11.       unify!(f_ty, Fun(args_ty, ret_ty))
12.       ret_ty
13.     }
14.   }
15. }
```

运算符的默认类型

- 和 MoonBit 一样，MiniMoonbit 中的 `+`, `-` 等算术运算符同时支持 `Int` 和 `Double` 两种类型
- 当运算符两侧的表达式类型均为类型变量时，我们使用 `Int` 作为默认类型

```
1. fn main {
2.   fn f(x, y) { x + y }
3.   f(1.0, 2.0) // type error!
4. }
```

清理类型变量

idea
基础软件中心

```
fn deref_type(t : Type) -> Type {
  match t {
    Fun(params, result) =>
      Fun(params.map(fn { t => deref_type(t) }), deref_type(result))
    Tuple(types) => Tuple(types.map(fn { t => deref_type(t) }))
    Array(t) => Array(deref_type(t))
    Var(~t = Some(t)) as tvar => {
      let t = deref_type(t)
      tvar.t = Some(t)
      t
    }
    Var(~t = None) as tvar => {
      tvar.t = Some(Unit)
      Unit
    }
    t => t
  }
}

fn deref_term(syntax : Syntax) -> Syntax { .. }
```

类型推导的实现

- 类型推导的函数入口

```
1. pub fn typing(e : Syntax) -> Syntax!TyErr {
2.   unify!(Unit, infer!(@immut/sorted_map.empty(), e))
3.   for ext_f, ext_t in extenv {
4.     extenv[ext_f] = deref_type(ext_t)
5.   }
6.   deref_term(e)
7. }
```

思考

- 如何在类型检查出现错误的情况下提供更友好的报错信息?