双向类型检查

基于归一化的类型推导的缺点

- 无法提供更加精确的错误信息
- 无法利用类型信息支持字面量重载和歧义消除
- 对子类型(subtyping)等类型系统的特性支持比较复杂

例子：函数调用的报错信息

idea
基础软件中心

```
1. fn f(s: String) -> Int {
2.   s.length()
3. };
4.
5. fn main {
6.   let a = f(42);
7.   ...
8. }
```

类型推导的过程：

- 通过推导得到 `f: (String) -> Int`，参数 `42` 的类型是 `Int`
- 对返回值类型生成类型变量 `TVar("a")`
- 将 `(String) -> Int` 和 `(Int) -> TVar("a")` 进行归一化，对函数调用进行报错
- 更精确的错误信息：`f` 的参数类型不匹配

例子：字面量重载

```
1. fn f() -> Double {
2.   ...
3.   return 1;
4. }
```

类型推导的过程：

- 通过类型推导，`1` 的类型是 `Int`
- 因为返回值的期望类型是 `Double`，所以会出现类型报错
- 字面量重载：在类型明确的情况下，可以将 `1` 的类型推导为 `Double`

例子：类型导向(type-directed)的歧义消除

```
1. enum S {
2.   Value(Int)
3. };
4. enum T {
5.   Value(Int)
6. };
7.
8. fn f(s: S) -> Unit { .. };
9. fn main {
10.   f(Value(42));
11.   ...
12. };
```

- 进行类型推导的情况下，`Value(42)` 的类型是 `S` 还是 `T`？
- 通过上下文中的函数调用 `f(Value(42))`，可以确定参数的类型是 `S`

例子：子类型(subtyping)的支持

假设类型 `Dog` 是 `Animal` 的子类型：

```
1. fn f(x: Animal) -> Unit { ... };
2. fn main {
3.   fn g(x) { f(x) };
4.   let a: Dog = ...;
5.   g(a);
6. }
```

- 在类型推导的过程中，`g` 的参数类型会被归一化为 `Animal`
- 但实际上，`g` 的参数类型应该是 `Animal` 的子类型的集合 `{ x | x <: Animal}`

类型推导和类型检查

- 类型推导：根据变量的使用情况，推导出变量的类型

```
1. fn infer(ctx: Ctx, e: Expr) -> Type!TyErr { .. }
```

- 类型检查：根据类型规则，检查表达式的类型是否符合期望的类型

```
1. fn check(ctx: Ctx, e: Expr, expect_ty: Type) -> Unit!TyErr { .. }
```

- `expect_ty` 是期望的类型，用于类型检查，并且提供额外的信息

类型推导和类型检查的转化

idea
基础软件中心

以推导函数调用的过程为例：

```
fn infer(ctx: Ctx, e: Expr) -> Type!TyErr {
  match e {
    Apply(f, args) => {
      let f_ty = infer!(ctx, f)
      match f_ty {
        Fun(params_ty, ret_ty) => {
          guard (params_ty.len() == args.len()) else { .. }
          for i in 0..<params_ty.len() {
            check!(ctx, args[i], params_ty[i])
          }
          ret_ty
        }
        _ => .. // infer and unify
      }
    }
    ..
  }
}
```

例子：函数调用的报错信息

```
1. fn f(s: String) -> Int {
2.   s.length()
3. };
4.
5. fn main {
6.   let a = f(42);
7.   ...
8. }
```

类型推导的过程：

- 通过推导得到 `f: (String) -> Int`，之后进入类型检查
- 用类型 `String` 对参数 `42` 进行类型检查，得到错误信息
- 更精确的错误信息： `f` 的参数类型不匹配

例子：字面量重载

```
1. fn f() -> Double {
2.   ...
3.   return 1;
4. }
```

在检查模式下：

- 当返回值的期望类型是 `Double` 时，`1` 的类型是 `Double`

例子：类型导向(type-directed)的歧义消除

```
 1. enum S {
 2.   Value(Int)
 3. };
 4. enum T {
 5.   Value(Int)
 6. };
 7.
 8. fn f(s: S) -> Unit { .. };
 9. fn main {
10.   f(Value(42));
11.   ...
12. };
```

在类型检查模式下，期望的参数类型是 `S`，从而可以确定参数的类型是 `S`

类型信息的流动方向

- 普通的函数调用：类型信息从函数类型流动到参数类型
- 方法的调用：类型信息从对象类型流动到方法类型

```
1. fn to_string(self: Int) -> String { .. };
2. fn to_string(self: Double) -> String { .. };
3.
4. fn main {
5.   let a = 42;
6.   let s = a.to_string();
7.   ..
8. }
```

双向类型检查

- 更加灵活并且高效的类型检查方法
- 减少甚至避免类型变量的生成，利于拓展类型系统以支持更多的高阶特性
- 便于提供更加友好的错误信息

参考资料

- [1] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. ACM Trans. Program. Lang. Syst. 22, 1 (Jan. 2000), 1–44.