北京大学

本科生毕业论文

题目：基于 Hindley-Milner 类型系统的控制流分析算法

姓　　名：罗翔宇

学　　号：1200012779

院　　系：北京大学

专　　业：信息科学技术学院

研究方向：计算机方向

导　　师：熊英飞

2016.05.09

版权声明

摘要

程序的静态分析在现在软件开发的过程中扮演者越来越重要的角色，而控制流分析则是程序静态分析中最为关键基础的一个环节。对于控制流分析算法，有两个角度来衡量其优劣：精确度和运行时间。在实际的分析过程中这两者往往不能兼得，追求高的精确度意味着运行时间的增长，而在较短时间内得出的结果往往意味着不精确的分析结果。

Hindley-Milner 类型系统被广泛应用于函数式编程语言中，但是由于其复杂的参数重建、多态等特性，目前的基于类型的控制流分析都不能很好地应用于 Hindley-Milner 类型系统中，而传统的基于抽象解释的控制流分析又有着求值顺序相关、效率低下等缺点。本文提出的 Hindley-Milner 类型系统的控制流分析算法可以很好地运行于 Hindley-Milner 类型系统之上，而且可以在编译器的类型推导时完成分析，不需要多次扫描程序。除此之外，本文还在 Haskell 语言的子集上实现了该算法，提供了用户友好的编程接口，可以直接用于 IDE 开发等实际项目中。

关键词：控制流分析，Hindley-Milner 类型系统，流属性

A control flow analysis algorithm based on Hindley-Milner type system

Luo Xiangyu (Computer Science)
Directed by Prof. Yingfei Xiong

Abstract

Nowadays static analysis of program is becoming increasingly significant in modern software development, and the control flow analysis is more fundamental compared to other analysis. To evaluate a control flow analysis algorithm, there are two aspects: accuracy and efficiency. It's impossible to come up with an algorithm which is better at both them. A higher accuracy always leads to a long run time while the results obtained in a short time always is inaccurate.

The Hindley-Milner type system is widely used in functional programming, but the existing control flow analysis algorithm based on type system is not suitable for Hindley-Milner type system due to its parameters reconstruction, polymorphism features. And the traditional control flow analysis based on abstract interpretation is evaluation order dependent and inefficiency. The algorithm in this thesis can be used in Hindley-Milner type system while it runs with type inference and only needs to pass through source code one time. Besides, the implementation of the algorithm can runs on a subset of Haskell providing a user friendly interface and benefits IDE development.

Keywords: Control Flow Analysis, Hindley-Milner Type System, Flow Property

目录

背景介绍

在实际软件工程项目的开发过程中，如果能够仅通过审查代码的方式（而非真正使用计算机运行程序）来获取程序的一些性质，从而进行漏洞排查、冗余代码移除或项目的其它优化，是一项非常有意义的事情。而这种只审查代码来推导程序性质的方式，就称之为程序的静态分析。由于其在项目开发中的重要作用，程序静态分析一直是软件工程研究领域的热点之一。

近些年来，基于 λ 演算的带有静态类型系统的函数式编程语言逐渐开始流行开来。函数式编程语言相对于传统的命令式编程语言，更加强调程序执行的结果而非程序执行的过程，倡导利用若干简单的执行单元让计算结果不断渐进，逐层推导复杂的运算，而不是设计一个复杂的执行过程。此外，语言中带有的静态类型系统也强化了编译器在编译期程序检查的功能，使得大部分程序漏洞可以在编译时期而不是实际运行时发现。静态类型的函数式编程语的高度抽象化、强大的表现力以及复杂的类型系统和语义定义，为在其上进行程序的静态分析带来了巨大的挑战。

在众多函数式编程语言的类型系统中，使用最为广泛的就是由 J. Roger Hindley 最先提出 [4]、后来经由 Robin Milner 改进 [5] 的 Hindley-Milner 类型系统，目前主流的函数式编程语言，诸如 Haskell、Ocaml 等都采用了 Hindley-Milner 类型系统。在 Hindley-Milner 类型中，编程人员可以不用声明任何类型声明和类型提示，而由类型推导算法推导出所有变量的类型。同时在使用该类型系统中的语言中，所有的 let 语句绑定的 λ 函数都默认是一个多态函数，即可应用于不同的类型参数，同时根据输入参数而返回不同类型的返回值。let 多态增加了 Hindley-Milner 类型系统的抽象能力，使得编程人员免于重复编写功能类似、只是参数类型不同的函数。

一般情况下，函数式编程语言上的程序静态分析通常以控制流分析为主，即

确定每一个函数的调用目标，同时控制流分析也是函数式编程语言上其他的静态分析的基础。举一个例子，考虑如下程序片段：

```
1  let f = fun g -> g True
2  in f (fun x -> if x then False else True)
```

通过控制流分析，我们可以知道调用 f 时对应的目标为在 let 绑定语句中的 fun g → g True。一旦有了函数的调用目标的结果，我们就可以执行后续的程序切片、数据依赖等分析。

事实上，为了解决函数式语言的控制流分析，程序分析的研究学者们已经提出了许多优秀的算法。Shivers 在他的论文中提出基于抽象解释的 1-CFA 算法[2]，但是该算法只是针对通用的无类型 λ 演算，没有利用静态类型系统的信息进行优化，同时该算法需要在运行时进行抽象解释，依赖于语言的求值顺序，也降低了算法的运行效率。而 Mossin 在他的论文中提出的基于类型系统的控制流分析算法只支持简单的带类型的 λ 演算，支持的类型系统的特性非常有限，像 Hindley-Milner 类型系统中的参数重建和 let 多态等都不能很好地处理。

设计基于 Hindley-Milner 类型系统的控制流分析算法，相比于简单的带类型 λ 演算的控制流算法，有以下几个难点：

- **参数的重建之前无法获取类型系统信息**。由于 Hindley-Milner 类型系统中所有的函数参数都没有类型信息，我们需要在类型推导的过程中维护一个类型约束集合，在推导过程结束之后求解该类型约束集合，再根据约束集合的解替换掉类型变量后才能获取每一个参数的真实类型。而控制流分析过程依赖于每一个参数的类型，在参数重建之前我们是无法得知变量的类型信息的，从而无法在类型推导的同时进行控制流分析。
- **let 多态导致的流属性变量多态以及约束集合指数爆炸**。基于类型系统的控制流分析往往基于修改现有类型系统，为其添加流属性来维护控制流信息。但是 let 绑定的变量由于其多态性，可以应用于不同的参数，这就要求我们的流变量也要至少支持同样的多态。同时由于流属性的多态不能使用类似类型多态的主类型来表示，所以每一个 let 绑定的变量的类型中都会保存当前 let 的绑定语句中的流属性变量约束集合，而这些集合随着 let 的嵌套会呈现指数级增长。

- **let 多态中流变量的实例化**。由于 let 多态的存在，很多类型的流属性都是以变量的形式来保存的，这就需要我们找到所有应用该类型的位置，但是这就需要在类型推导过程结束之后，再次扫描整个程序来实例化所有多态的流属性变量，降低了分析的效率

本文基于 Mossin 的简单类型系统的控制流分析算法，将其推广到了 Hindley-Milner 类型系统，完美地解决了上述三个问题，并在 Haskell 语言的子集中进行了实现。在接下来的章节中，我们会先具体说明我们控制流分析的目标并且简单概述如何解决在 Hindley-Milner 类型中解决上述问题（第二章），然后形式化定义一个分析的语言用于本论文的算法阐述（第二章），接着形式化描述我们的分析算法（第三章），并给算法在 Haskell 的子集上的实现细节（第四章），最后给出算法在不同数据集下的分析结果（第五章）。

第一章　算法简介

1.1　控制流分析目标

对于一个静态类型系统的函数式编程语言，一般用语法定义、类型推导规则和求值规则来描述。其中类型推导规则描述在给定类型系统下，对于一个源程序，如何推导出每一个子表达和变量的类型。而求值规则则描述在给定的求值规则下，如何将一个表达式转化成一个值。在函数式编程语言中，值一般指函数。

我们的控制流分析的目标即为对于源程序中的每一个函数调用，确定其调用的目标，即对于形如 e_1 e_2 的函数调用，确定表达式 e_1 最终求值的结果。本文的中的算法不仅适用于函数调用的表达式，还可以用于语言中其他的表达式，即对于源程序中任意表达式，控制流分析可以推导出其最终可能的求值结果的集合。

比如对于程序:

```
1    let f x = if x then fun x -> x + 1
2              else fun x -> x - 1
3    in (f True) 1
```

我们所期望的控制流分析结果为: 表达式 (f True) 最终可能被求值为 fun x -> x + 1 或者 fun x -> x - 1，表达式 (f True) 1 最终可能被求值为 1 + 1 或者 1 - 1, 这里我们定义 + 和 -为语言中的元函数，其语义不在求值规则中。

1.2　算法思路

我们的算法基于扩展原生的 Hindley-Milner 类型系统，使得每一个类型都绑定一个称之为流属性的元素，其中流属性是一个值的集合，表示拥有该类型的表达式最终可能的求值结果集合。

在类型推导的过程中，我们需要根据表达式，对其子表达式的流属性做某些集合的操作或者对其加以集合的约束，以更新其流属性。考虑代码：

```
1    if x then e1 else e2
```

这里表达式 if x then e1 else e2 的类型中的流属性就是表达式 e1 和 e2 类型中的流属性的并集。而考虑代码：

```
1    (fun x -> x 1) (fun y -> y + 1)
```

这里表达式 x 1 中 x 类型中的流属性就应该是 (fun y -> y + 1) 类型中的流属性的超集，因为 (fun x -> x + 1) 可能不止在这里进行调用，所以需要包含所有其可能的参数的类型中的流属性。

于是我们可以在原有的 Hindley-Milner 类型系统基础上，添加子类型系统。一个类型 A 是另外一个类型 B 的子类型当且仅当 A 除去所有的流属性之后和 B 除去流属性的类型相同，且 A 中的流属性是 B 的流属性的子集。有了子类型系统之后，我们就可以沿用传统的子类型推导算法，而不用考虑流属性的集合关系了。比如对于函数调用 $e_1\ e_2$，其中 e_1 有类型 $t_1 \to t_2$，e_2 有类型 t_3，我们只需要保证 t_3 是 t_1 的子类型就可以了。这样我们基于流属性的类型系统就和传统的子类型系统一致。

但是对于 Hindley-Milner 类型系统，所有的函数参数的定义都没有类型标识，即在类型推导过程中是无法得知参数的真实类型的。所以我们需要模仿类型变量，引入流属性变量，然后为所有没有类型信息的参数分配一个流属性变量。同时在推导过程中需要维护一个子类型约束集合，用于建立流属性变量之间的约束条件。在类型推导结束之后，现对于 Hindley-Milner 类型推导算法中类型变量的约束集合求解，把所有的类型变量替换为真实类型，然后再将子类型约束集合重新转化

为流属性之间集合的关系，最后求解这些集合关系方程并把所有的流属性变量替换为流属性即可。比如考虑代码

```
1        (fun x -> if True then False else x)
```

这里我们不知道参数 x 的类型和流属性，也不知道 (fun x -> if True then False else x) 的类型。于是可以新建类型变量 α 和 β，令 x 为类型 α，(fun x -> if True then False else x) 为类型 $\alpha \to \beta$。同时新建流属性变量 κ_1 和 κ_2，令 κ_1 是 α 绑定的流属性，κ_2 是 β 绑定的流属性。由前面的子类型系统的定义可知，False 是 β 的子类型，且 α 也是 β 的子类型，于是可以将其添加到子类型约束集合中。在类型推导过程结束之后转换为 κ_1 是 κ_2 的子集，且 False 所代表的流属性也是 κ_2 的子集。这样就成功地将类型重建过程缺失的流属性信息，转化为一系列的流属性变量和这些流属性变量之间的集合关系。我们只需要求解这些集合关系即可，求解流属性集合关系所用的算法会在后文详细描述。

对于 let 多态的问题，Hindley-Milner 解决的方式为先对 let 绑定的表达式进行类型推导，求解出其一个形如 $\forall X.X \to X$ 的主类型和一个类型约束集合 S，然后对于类型约束集合 S 进行求解，并根据 S 的解把当前类型推导环境中的类型变量先进行替换，然后再替换过的环境中继续进行类型推导。当在环境中查询变量时，如果遇变量有形如 $\forall X.X \to X$ 的主类型，我们需要先使用一些未出现过的类型变量将主类型中的约束类型变量替换掉并去掉这些约束变量，再将替换后的类型返回。这样就可以实现 let 多态。

但是对于流属性的多态，我们并不能这么照搬 Hindley-Milner 类型系统的方案，这是因为流属性的约束都是形如 $\kappa_1 \subseteq \kappa_2$ 的子集约束，和 Hindley-Milner 类型系统中的 $X = Y \to Z$ 的等价约束不同，子集约束不可以部分求解。考虑约束 $\{1,2\} \subseteq \kappa$，如果只看这一条约束，我们可以解出 $\kappa = \{1,2\}$，但是如果后续又添加了一个约束 $\{3\} \subseteq \kappa$，那么其实正确的解应当是 $\kappa = \{1,2,3\}$，如果我们部分求解，在后面就会导出 $\{3\} \subseteq \{1,2\}$ 的矛盾。

我们这里采取的方法是对于所有的流属性多态类型，我们不仅创建一个流属性变量来表示主流属性类型，同时也将推导出的子类型约束集合放在类型中，构造成形如 $\forall \kappa.C \Rightarrow T$ 的形式，其中 κ 使我们构造的流属性变量，C 是子类型约束集合中，T 是真实的类型，其含义为对于所有满足约束 C 的流属性变量 ℓ，其表

达式有类型 $T[\kappa/\ell]$，即将 T 所有自由出现的 κ 替换为 ℓ. 而每次在环境中查询一个变量的类型的时候，则需要将所有的约束流属性变量替换为全新的流属性变量，并将约束集合 C 中的流属性变量也替换为新的流属性变量，然后把的得到的新的子类型约束集合中和当前约束集合合并。

但是这样会造成一个问题，如果 let 语句嵌套的层数过多，会使得每一层的 let 都创造一个带有约束集合的流属性类型，这样就造成很多约束在不同的约束集合中重复出现，从而造成空间的极大浪费。这里可以使用的一个优化是：我们可以发现很多约束其实是可以合并的，比如如果有两个约束为 $\{\ell_1 \subseteq \kappa, \ell_2 \subseteq \kappa\}$, 那么其实可以将其合并为 $\{\ell_1 \cup \ell_2 \subseteq \kappa\}$, 另外如果一个流属性变量在类型中没有自由出现，那么其实关于它的约束是没有用的，因为它在以后也不会被替换，那么我们就可以简单地去掉这些约束。经过如上处理之后，每一个流属性变量最多只会在一个约束集合中出现一次，所以约束的个数最多只有平方级别的个数。

由于流属性变量的存在，let 绑定的表达式的类型实际上是包含流属性变量的。比如考虑程序：

```
1    let f = fun x -> if x then False else True
2    in (f True, f False)
```

这里经过类型推导，可以得到 if x then False else True 中 x 的类型实际上是 $Bool^{\kappa}$，其中 κ 是一个流属性变量。而由于在约束中的流属性变量在后续的推导过程会进行实例化，所有添加进约束集合的流属性变量实际上不是 κ，而是一个新的流属性变量 κ'。为了解决这个问题，我们需要在构造子类型约束集合的时使用一些技巧，使得可以正确处理这个情况。详细会在第三章描述。

第二章　语言定义

出于方便起见，我们首先定义自己的一个带有 Hindley-Milner 类型系统的简单编程语言。为了更好地揭露算法的本质，避免无关紧要的细节对于读者理解算法的影响，我们希望这个语言的语法尽可能简洁，同时又能保留静态类型系统、多态等特性。注意本章的语言仅用于说明算法，真正的代码在 Haskell 的一个子集上实现。

首先令 V 是一个可列举的变量集合，同时令 x, y, z 为 V 中的元素。一个程序即为如下一个表达式：

$$
\begin{aligned}
\langle term \rangle \quad ::= \; & \langle variable \rangle \\
| \; & \lambda \, \langle variable \rangle . \langle term \rangle \\
| \; & \langle term \rangle \; \langle term \rangle \\
| \; & \text{fix} \, \langle variable \rangle . \langle term \rangle \\
| \; & \text{let} \, (\langle variable \rangle, \langle variable \rangle) \, \text{be} \, \langle term \rangle \, \text{in} \, \langle term \rangle \\
| \; & \text{let} \, \langle variable \rangle = \langle term \rangle \, \text{in} \, \langle term \rangle \\
| \; & \text{if} \, \langle term \rangle \, \text{then} \, \langle term \rangle \, \text{else} \, \langle term \rangle \\
| \; & \text{True} \\
| \; & \text{False} \\
| \; & \{0, 1, 2, \cdots\}
\end{aligned}
$$

他们的含义分别为:

- x 即为一个单独的变量
- $\lambda x.e$ 定义一个函数接受一个参数 x，函数体为 e. 多参函数可以使用嵌套的 λ 定义实现。注意这里没有声明 x 的类型，即 x 的类型由 e 中对于 x 的类

型约束而重建出。

- $e_1\ e_2$ 函数应用，将 e1 作用于 e2
- fix $x.e$ 利用 fix 组合子来定义一个递归函数
- (e, e') 一个元组，可以使用 let be 模式匹配来分别绑定到不同的变量
- let (x, y) be e in e' 元组的模式匹配。其中 e 必须为一个元组，而在 e' 中 x 被绑定为 e 的第一项，y 被绑定为 e 的第二项。
- let $x = e$ in e' 变量的 let 绑定，注意这里有 let 多态，即 x 在 e' 中只有一个 type schema, 具体的类型会在 x 使用时进行实例化。
- if e then e' else e'' if 条件语句，这里 e 的类型必须是 Bool 型，e' 和 e'' 必须有相同的类型。
- $\text{True}, \text{False}, \{0, 1, 2\ldots\}$ 字面量，分别对应 Bool 真假常量和自然数。

在我们的编程语言中，定义如下的类型系统 [4]：

$$\frac{x : t \in A}{A \vdash x : t \mid \varnothing}\ \text{Id} \qquad \frac{\text{X is a fresh type variable} \quad A, x : X \vdash e : T \mid C}{A \vdash \lambda x.e : X \to T \mid C}\ \text{Abs}$$

$$\frac{A \vdash t_1 : T_1 \mid C_1 \quad A \vdash t_2 : T_2 \mid C_2 \quad \text{X is a fresh type variable}}{A \vdash t_1\ t_2 : X \mid C_1 \cup C_2 \cup \{T_1 = T_2 \to X\}}\ \text{App}$$

$$\frac{\text{X is a fresh type variable} \quad A, x : X_1 \vdash e : X_2 \mid C}{A \vdash \text{fix}\, x.e : X_1 \mid C \cup \{X_1 = X_2\}}\ \text{Fix}$$

$$\frac{A \vdash e_1 : T_1 \mid C_1 \quad A \vdash e_2 : T_2 \mid C_2}{A \vdash (e_1, e_2) : (T_1, T_2) \mid C_1 \cup C_2}\ \text{Tuple}$$

$$\frac{A \vdash e : (T_1, T_2) \mid C_1 \quad A, x : T_1, y : T_2 \vdash e' : T \mid C_2}{A \vdash \text{let } (x, y) \text{ be } e \text{ in } e' : T \mid C_1 \cup C_2}\ \text{Tuple Deconstruct}$$

$$\frac{A \vdash [x \mapsto t_1]t_2 : T_2 \mid C}{A \vdash \text{let } x = t_1 \text{ in } t_2 : T_2 \mid C}\ \text{Let}$$

$$\frac{A \vdash e_1 : \text{Bool} \mid C_1 \quad A \vdash e_2 : t \mid C_2 \quad A \vdash e_3 : t \mid C}{A \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \mid C_1 \cup C_2 \cup C_3} \text{ If}$$

$$\text{True} : \text{Bool} \mid \varnothing \quad \text{True} \qquad \text{False} : \text{Bool} \mid \varnothing \quad \text{False} \qquad \frac{x \in \{0, 1, 2, \cdots\}}{A \vdash x : \text{Int} \mid \varnothing} \quad \text{Int}$$

其中 A 为类型的上下文，C 为 HM 类型系统中产生的约束集合，最终使用联合算法解所有的约束从而得到一个从类型变量到实际类型的替换函数，将这个替换函数应用到表达式所推导的类型即可得到表达式真实的类型。每一条类型规则的解释如下：

- **Id** 如果一个变量的已经在类型上下文中有记录，直接取出记录即可，约束为空。
- **Abs** 对于一个定义的 λ 函数，如果给其参数 x 一个类型变量 X，同时 A 并上 x 是 X 类型的前提下可以推导出 e 是类型 T 和约束集合 C，那么定义的函数即为 X $\to$ T 类型，同时给出约束 C。
- **App** 给定一个函数应用，如果第一个表达式是 T_1 类型，第二个表达式是 T_2 类型，那么可以知道 T_1 必然是是 $T_2 \to X$ 的形式，于是可以在约束集合中添加约束 $\{T_1 = T_2 \to X\}$，且整个表达最终返回的类型为 X
- **Fix** Fix 即为 λ 演算中的不动点组合子，用来解决递归函数的情况。由于 fix 组合子的定义中，传入参数的类型实际上是和函数体的类型等价，所以直接在约束集合中添加约束 $\{X_1 = X_2\}$ 即可。
- **Tuple** 如果一个元组第一分量的类型是 T_1，第二分量的类型是 T_2，那么整个元组的类型即为 (T_1, T_2)
- **Tuple Deconstruct** 元组的模式匹配本质上是将元组的第一分量和第二分量都绑定到一个变量, 然后将绑定过后的变量添加进类型环境，最终在新的类型环境下推导出 e 的类型。
- **Let** [5] 为了实现 let 多态，我们不能简单地推导出 t1 的类型并将 x 添加进类型环境中，而是应在 e 中将所有的 x 出现的地方替换为 t1 后推导出 e 的类型。
- **If** 首先需要保证用于条件判断的表达式 e1 是 Bool 类型，另外一个条件就是 if 语句的 then 分支和 else 分治的类型应当一致。所以将这两个约束添加

进约束集合即可

- **True, False, Int** 对于所有的字面量，直接返回该字面量对应的类型

第三章　算法的形式化描述

有了语法和类型系统的定义之后，我们接下来就可以形式化地定义在该语言上的控制流分析。

3.1　标号和流属性

令 V 为一个无穷可列集合，对于任意 V 中元素 l，称 l 为一个标号。定义一个标号集合 ℓ 为流属性，即 $\ell \subset V$. 对于任意一个表达式 e, 为 e 中所有形如 $\lambda x.e$, $True$, $False$, $\{1, 2, 3, \cdots\}$ 的子表达式分配一个独一无二的标号。如表达式

$$\text{if } True \text{ then } \lambda x.x \text{ else } \lambda y.0$$

分配过标号之后为

$$\text{if } True^{l_1} \text{ then } \lambda^{l_2} x.x \text{ else } \lambda^{l_3} y.0^{l_4}$$

3.2　流属性变量和流属性约束

我们定义形如 α, β, γ 的变量为一个流属性变量，可以指代任一流属性。一个流属性约束为形如 $\alpha \subset \beta$ 的约束，表示 α 是 β 的子集。我们用 C 来代指一个流属性约束组成的集合。

3.3　流属性类型

我们基于 Hindey-Milner 类型系统定义如下流属性类型：

$$\frac{\kappa \in \mathcal{K}(t) \quad \kappa' \in \mathcal{K}(t')}{\kappa \xrightarrow{\ell} \kappa' \in \mathcal{K}(t \to t')} \text{ Arrow} \qquad \frac{\kappa \in \mathcal{K}(t) \quad \kappa' \in \mathcal{K}(t')}{(\kappa, \kappa')^{\ell} \in \mathcal{K}((\kappa, \kappa'))} \text{ Tuple}$$

$$\frac{\kappa \in \mathcal{K}(t)}{\forall \vec{X}.\kappa \in \mathcal{K}(\forall \vec{X}.t)} \text{ TypeForall}$$

$$\frac{\forall \vec{X}.\kappa \in \mathcal{K}(\forall \vec{X}.t)}{\forall \vec{\alpha}.C \Rightarrow \forall \vec{X}.\kappa \in \mathcal{S}(\forall \vec{X}.t)} \text{ FlowPropertyForall} \qquad \text{Bool}^{\ell} \in \mathcal{K}(\text{Bool}) \text{ Bool}$$

$$\text{Int}^{\ell} \in \mathcal{K}(\text{Int}) \text{ Int} \qquad \kappa\,[\vec{\alpha}/\vec{l}]$$

对于原语言中任意一个类型 t，$\mathcal{K}(t)$ 是由类型 t 扩展出的流属性类型的集合，这个集合中任一元素 κ 在删除掉其绑定的流属性之后，就得到原类型 t. 定义函数 $\mathcal{L}$ 为从流属性类型到流属性的函数，即提取出最外层绑定的流属性，定义如下：

$$\begin{aligned}
&\mathcal{L}(Bool^{\ell}) = \ell \\
&\mathcal{L}(Int^{\ell}) = \ell \\
&\mathcal{L}(\kappa_1 \xrightarrow{\ell} \kappa_2) = \ell \\
&\mathcal{L}((\kappa_1, \kappa_2)^{\ell}) = \ell
\end{aligned}$$

对于形如 $\forall \vec{\alpha}.C \Rightarrow \forall \vec{X}.\kappa$ 的流属性类型，我们称其为一个流属性类型模式，其中 C 为一个约束集合，$\vec{\alpha}$ 是一系列的流属性变量，$\vec{X}$ 为一系列类型变量。$\forall \vec{\alpha}.C \Rightarrow \forall \vec{X}\kappa$ 的意义是一个表达式对于所有使得 C 成立的流属性 $\vec{l}$，有类型 $\forall \vec{X}.\kappa[\vec{\alpha}/\vec{l}]$，即将 κ 中所有的 $\vec{\alpha}$ 替换为 $\vec{l}$。我们用 τ 来表示流属性模式，$\mathcal{S}(t)$ 是一个流属性类型模式集合。

3.4 流属性类型的子类型关系

对于流属性类型，我们有如下流属性子类型关系：

$$\frac{\ell_1 \subseteq \ell_2}{\mathrm{Bool}^{\ell_1} \preceq \mathrm{Bool}^{\ell_2}} \quad \text{SubBool} \qquad \frac{\ell_1 \subseteq \ell_2}{\mathrm{Int}^{\ell_1} \preceq \mathrm{Int}^{\ell_2}} \quad \text{SubInt}$$

$$\frac{\kappa_1 \preceq \kappa_1' \quad \kappa_2 \preceq \kappa_2' \quad \ell_1 \subseteq \ell_2}{\kappa_1' \xrightarrow{\ell_1} \kappa_2 \preceq \kappa_1 \xrightarrow{\ell_2} \kappa_2'} \quad \text{SubArrow}$$

$$\frac{\kappa_1 \preceq \kappa_1' \quad \kappa_2 \preceq \kappa_2' \quad \ell_1 \subseteq \ell_2}{(\kappa_1, \kappa_2)^{\ell_1} \preceq (\kappa_1', \kappa_2')^{\ell_2}} \quad \text{SubTuple}$$

3.5 流属性类型系统

定义 1 (替换). 对于任意流属性类型 κ，定义 $\kappa[\vec{\alpha}/\vec{l}]$ 为将 κ 中出现的所有流属性变量 $\vec{\alpha}$ 替换为 $\vec{l}$，定义 $\kappa[\vec{X}/\vec{T}]$ 为将 κ 中出现的所有类型变量 $\vec{X}$ 替换为 $\vec{T}$

定义 2 (环境). 定义由形如 $x : \tau$ 组成的集合为一个环境，表示变量 x 有类型 τ，其中流属性约束变量向量 $\vec{\alpha}$、流属性约束集合 C 和类型约束变量 $\vec{X}$ 均可以是空集。定义 $Type(A)$ 为环境中所有流属性类型模板组成的集合，$Var(A)$ 为环境中所有变量组成的集合.

定义 3 (FV_t). 对于一个流属性类型 κ_i，定义其中所有出现的类型变量的集合为 $FV_t(\kappa)$. 对于一个环境 A，定义 $FV_t(A) = \bigcup FV_t(\kappa_i), \quad \forall \kappa_i \in Type(A)$

定义 4 (FV_f). 对于一个流属性类型 κ_i，定义其中所有出现的流属性变量的集合记为 $FV_f(\kappa)$. 对于一个环境 A，定义 $FV_f(A) = \bigcup FV_f(\kappa_i), \quad \forall \kappa_i \in A$. 对于一个流属性约束集合 C, 定义 $FV_f(C)$ 为所有出现在此约束集合中的流属性变量.

定义 5 ($free_t$). 对于一个形如 $\forall \vec{\alpha}.C \Rightarrow \forall \vec{X}.\kappa$ 的流属性类型模式 τ 和一个环境 A，定义

$$free_t(A, \tau) = FV_t(\kappa) - FV_t(A) - \vec{X}$$

定义 6 ($free_f$)**.** 对于一个形如 $\forall\vec{\alpha}.C \Rightarrow \forall\vec{X}.\kappa$ 的流属性类型模式 τ 和一个环境 A，定义

$$free_f(A,\tau) = FV_f(\kappa) - FV_f(A) - FV_f(C) - \vec{\alpha}$$

定义 7 ($generalize_t$)**.** 对于一个流属性类型 κ 和一个环境 A，定义

$$generalize_t(A,\kappa) = \forall free_t(A,\kappa).\kappa$$

. 称形如 $\forall free_t(A,\kappa).\kappa$ 的类型为类型模式，记为 φ

定义 8 ($generalize_f$)**.** 对于一个形如 $\forall\vec{X}.\kappa$ 的类型模式 φ, 一个环境 A 和一个流属性约束集合 C，定义

$$generalize_f(A,C,\varphi) = \forall free_f(A,C,\kappa).C \Rightarrow \forall\vec{X}.\kappa$$

定义 9 ($instantiate_f$ 和 $instantiate_c$)**.** 对于一个形如 $\forall\vec{\alpha}.C \Rightarrow \forall\vec{X}.\kappa$ 流属性类型模式 τ 和一个环境 A，定义

$$instantiate_f(A,\tau) = \forall\vec{X}.\kappa[\vec{\alpha}/\vec{\beta}]$$

$$instantiate_c(A,\tau) = C[\vec{\alpha}/\vec{\beta}]$$

其中 β 为未出现过的流属性变量

定义 10 ($instantiate_t$)**.** 对于一个形如 $\forall\vec{X}.\kappa$ 的类型模式 φ 和一个环境 A，定义

$$generalize_t(A,\varphi) = \kappa[\vec{X}/\vec{Y}]$$

其中 $\vec{Y}$ 为未出现过的类型变量

定义 11 (类型约束的解)**.** 给定一个类型约束集合 $D = \{\kappa_1 = \kappa_2, \kappa_3 = \kappa_4, \cdots\}$, 定

义 f 是一个由类型变量到类型的映射，定义替换

$$S(\kappa) = \begin{cases} S(\kappa_1) \xrightarrow{\ell} S(\kappa_2) & if\ \kappa = \kappa_1 \xrightarrow{\ell} \kappa_2 \\ X & if\ X = Bool\ or\ X = Int\ or\ X \notin dom(f) \\ f(X) & if\ X \in dom(f) \end{cases}$$

为由 f 生成的替换。如果 $\forall \kappa_i = \kappa_j \in D$ 都有 $S(\kappa_i) = S(\kappa_j)$, 称 f 为约束集合 D 的解函数，称 S 为约束集合 D 的解。对于替换 S 我们可以将其推广到形如 $\forall \vec{\alpha}.C \Rightarrow \forall \vec{X}.\kappa$ 流属性模板 τ 和环境 A 上，即

$$S(\tau) = \tau[free_t(\tau)/S(free_t(\tau))]$$

$$S(A) = \{x_i : S(\kappa_i)\} \quad \forall x_i : \kappa_i \in A$$

基于以上定义，我们给出如下类型推导规则：

$$\frac{x : \tau \in A \quad \text{let } \varphi \text{ be } instantiate_f(\tau) \text{ and } C \text{ be } generalize_c(\tau)}{A \vdash x : instantiate_t(\varphi) \mid \varnothing \mid C} \quad \text{Id}$$

$$\frac{X \text{ is a fresh type variable} \quad A, x : X \vdash e : \kappa \mid D \mid C}{A \vdash \lambda^l x.e : X \xrightarrow{\{l\}} \kappa \mid D \mid C} \quad \text{Abs}$$

$$\frac{\begin{array}{c} A \vdash e_1 : \kappa_1 \mid D_1 \mid C_1 \quad A \vdash e_2 : \kappa_2 \mid D_1 \mid C_1 \\ \text{X is a fresh type variable} \quad \alpha \text{ is a fresh flow properties variable} \end{array}}{A \vdash e_1\ e_2 : X \mid D_1 \cup D_2 \cup \{\kappa_1 = \kappa_2 \xrightarrow{\alpha} X\} \mid C_1 \cup C_2 \cup \{\kappa 1 \preceq \kappa_2 \xrightarrow{\alpha} X\}} \quad \text{App}$$

$$\frac{\text{X is a fresh type variable} \quad A, x : \kappa \vdash e : \kappa \mid D \mid C}{A \vdash \text{fix}\, x.e : \kappa \mid C \cup \{\kappa = X\} \mid D \cup \{\kappa \preceq X\}} \quad \text{Fix}$$

$$\frac{A \vdash e_1 : \kappa_1 \mid D_1 \mid C_1 \quad A \vdash e_2 : \kappa_2 \mid D_2 \mid C_2}{A \vdash (e_1, e_2)^l : (\kappa_1, \kappa_2)^{\{l\}} \mid D_1 \cup D_2 \mid C_1 \cup C_2} \quad \text{Tuple}$$

$$\frac{A \vdash e : (\kappa_1, \kappa_2)^\alpha \mid D_1 \mid C_1 \quad A, x : \kappa_1, y : \kappa_2 \vdash e' : \kappa_3 \mid D_2 \mid C_2}{A \vdash \text{let } (x, y) = e \text{ in } e' : \kappa_3 \mid C_1 \cup C_2 \mid D_1 \cup D_2} \quad \text{Tuple Deconstruct}$$

$$\frac{\begin{array}{c} A \vdash e : \kappa \mid D \mid C \quad \text{let } S \text{ be the solution of } D \\ S(A), x : S(generalize_f(S(A), C, generalize_t(S(A), \kappa))) \vdash e' : \kappa' \mid D' \mid C' \end{array}}{A \vdash \text{let } x = e \text{ in } e' : \kappa' \mid D \cup D' \mid C \cup C'} \quad \text{Let}$$

$$\frac{\begin{array}{c} \alpha \text{ is a fresh flow properties variable} \quad X \text{ is a fresh type variable} \\ A \vdash e_1 : \kappa_1 \mid D_1 \mid C_1 \quad A \vdash e_2 : \kappa_2 \mid D_2 \mid C_2 \quad A \vdash e_3 : \kappa_3 \mid D_3 \mid C_3 \end{array}}{\begin{array}{c} A \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : X \mid D_1 \cup \\ D_2 \cup D_3 \cup \{\kappa_1 = Bool^\alpha, \kappa_2 = \kappa_3\} \mid C_1 \cup C_2 \cup C_3 \cup \{\kappa_2 \preceq X, \kappa_3 \preceq X\} \end{array}} \quad \text{If}$$

$$A \vdash True^l : Bool^{\{l\}} \mid \varnothing \mid \varnothing \quad \text{True} \qquad A \vdash False^l : Bool^{\{l\}} \mid \varnothing \mid \varnothing \quad \text{False}$$

$$\frac{x \in \{1, 2, \cdots\}}{A \vdash x^l : Int^{\{l\}} \mid \varnothing \mid \varnothing} \quad \text{Int}$$

其中 D 为 Hindley-Milner 系统的类型约束集合，C 是子类型约束的集合，用于推导出流属性约束集合。

- **Id** 对于一个变量 x，如果 x 在当前的环境中有记录为 τ 流属性类型模式，那么使用未出现的流属性变量和类型变量进行实例化即可
- **Abs** 对于一个 lambda 函数的定义，如果将 x:X 添加到当前环境中可以推导出 e 是 κ 类型，且得到类型约束和子类型约束 D 和 C，那么 $\lambda x.e$ 的类型即为 $X \to \kappa$，该类型绑定的流属性即为当前 λ 表达式标号。
- **App** 对于函数调用，如果第一个表达式是 κ_1 类型，第二个表达式是 κ_2 类型，那么可以知道 κ_1 必然有形式 $\kappa_2 \to X$, 于是将 $\{\kappa_1 = \kappa_2 \to X\}$ 其添加到类型约束集合中，将 $\{\kappa_1 \preceq \kappa_2 \to X\}$ 添加到子类型约束集合中即可。
- **Fix** 根据 Fix 组合子的定义，函数体 e 的类型必须和参数 x 的类型相同，且其绑定的流属性是参数 x 绑定的流属性的子集，所以分别往类型约束集合和子类型约束集合中添加 $\{\kappa = X\}$ 和 $\{\kappa \preceq X\}$ 即可。
- **Tuple** 类似于之前的类型推导中 Tuple 规则，不同的一点是一个 Tuple 值所绑定的流属性即为这个值的标号本身。

- **Tuple Deconstruct** 类似于之前的类型推导中的 Tuple Deconstruct 规则，不同的一点是将类型替换为流属性类型即可。
- **If** 在环境 A 中如果可以推出 e_1 为 T_1 类型，e_2 为 T_2 类型，e_3 为 T_3 类型，可以得到类型的约束为 $\{T_1 = Bool\}$，且 T_2 和 T_3 去掉流属性之后的类型应当一致。除此之外，还应当存在一个流属性类型 X，使得 T_2 为 X 的子类型同时 X 也是 T_3 的子类型。将这四条条约束添加进相应的约束集合即可。
- **True, False, Int** 对于字面量，直接得到其对应的类型即可，绑定的流属性即为当前字面量的标号。
- **Let** 为了实现多态，Let 的规则比较复杂。首先在环境 A 中推导出 e 表达式的流属性类型 κ 以及相对应的类型约束集合 D 和子类型约束集合 C。注意我们不能直接把推导 e 时得出的类型约束集合并入最终答案中，因为 e 的类型都是多态类型，即其中的类型变量会在以后实例化成新的类型变量，所以我们需要在这里直接求解类型约束集合 D 并将其解 S 应用到当前环境 A 和后续的类型推导中。但是由于子类型约束集合不能部分求解，所以需要在泛化后的流属性类型模式中绑定上子类型约束的集合，即形如 $\forall\vec{\alpha}.C \Rightarrow \forall\vec{X}.\kappa$ 的流属性类型模板。而 e 的类型多态只体现在 κ 中自由类型变量和自由流属性变量中，所以只需为其添加约束并在后续使用的位置进行实例化即可。

3.6 求解类型约束集合

在这里我们使用联合算法求解类型约束集合，其中 $unify$ 函数接受一个类型约束集合 D 并返回其一个解 S。该算法的伪代码描述如下：

```
1    unify(D) = if C is empty then id
2             else let {κ1 = κ2} ∪ C' = C in
3                 if κ1 = κ2
4                     then unify(C')
5                 else if κ1 is X and X ∉ FV(κ2)
6                     then unify(C'[X/κ2]) ∘ [X ↦ κ2]
7                 else if κ2 is X and X ∉ FV(κ1)
```

```
8                       then unify(C'[X/κ1]) ∘ [X ↦ κ1]
9                   else fail
```

注意基于流属性的类型系统不支持递归类型，所以如果检测到变量 X 应该替换为 κ 且 X 在 κ 中自由出现则会直接报错。

3.7　求解子类型约束集合

对于一个形如 $\{\kappa_1 \preceq \kappa_2, \kappa_3 \preceq \kappa_4 \cdots\}$ 子类型约束集合 C, 我们首先需要将其转化为形如 $\{\ell_1 \subseteq \ell_2, \ell_3 \subseteq \ell_4 \cdots\}$ 的流属性约束集合，转换的算法伪代码描述如下：

```
1     trans(D) = if D is empty then ∅
2             else let {d} ∪ D' = D in
3             if d = Bool^ℓ1 ⪯ Bool^ℓ2
4                 then trans(D') ∪ subset(ℓ1, ℓ2)
5             if d = Int^ℓ1 ⪯ Int^ℓ2
6                 then trans(D') ∪ subset(ℓ1, ℓ2)
7             if d = κ1 -ℓ1-> κ2 ⪯ κ3 -ℓ2-> κ4
8                 then trans(D'∪{κ3 ⪯ κ1}∪{κ2 ⪯ κ4}) ∪ subset(ℓ1, ℓ2)
9             if d = (κ1, κ2)^ℓ1 ⪯ (κ3, κ4)^ℓ2
10                then trans(D'∪{κ1 ⪯ κ3}∪{κ2 ⪯ κ4}) ∪ subset(ℓ1, ℓ2)
```

subset 函数用于生成一个流属性约束集合，定义如下, 其中 α, β 代指流属性变量，ℓ 指流属性：

$$
\begin{aligned}
&subset(\alpha, \beta) = \alpha \subseteq \beta \\
&subset(\ell, \beta) = \ell \subseteq \beta \\
&subset(\ell, \ell') = \varnothing && \text{if} \quad \ell \subseteq \ell' \\
&subset(\ell, \ell') = fail && \text{if} \quad \ell \nsubseteq \ell' \\
&subset(\alpha, \ell) = fail
\end{aligned}
$$

3.8 求解流属性约束集合

一个形如 $\mathcal{D} = \{\alpha \subseteq \beta, \ell \subseteq \gamma, \cdots\}$ 的集合被称为流属性约束集合，其中每一条约束 $\alpha \subseteq \beta$ 的含义为流属性 α 是 β 的真子集。我们希望找到一个从流属性变量映射到流属性的函数 $\mathcal{F}$，使得对于所有的约束 $\alpha \subseteq \beta$ 都有 $\mathcal{F}(\alpha) \subseteq \mathcal{F}(\beta)$.

为了求解流属性约束集合，我们可以将其抽象为一个图论问题。构建有向图 $G = (V, E)$，其中 V 为流属性约束集合 $\mathcal{D}$ 中所有出现的流属性变量，$E = \{(\beta, \alpha) \mid \{\alpha \subseteq \beta\} \subseteq \mathcal{D}\}$. 有向图 G 中一条边 (α, β) 的含义为，流属性变量 β 所代表的流属性是 α 所代表的流属性的子集。同时定义从 V 映射到流属性的函数 $\mathcal{H}$:

$$\mathrm{H}(\alpha) = \bigcup \ell \qquad \forall\{\ell \subseteq \alpha\} \subseteq \mathcal{D}$$

$\mathcal{H}$ 函数的含义为 $\mathcal{H}(\alpha)$ 是流属性变量 α 所代表的流属性的子集. 于是可以发现，$\mathcal{F}(\alpha)$ 实际上是 α 节点所有可以到达的节点的 $\mathcal{H}$ 函数值的并集。于是一个朴素的算法为对于每一个节点 α，从它为起点进行一遍图的深度优先遍历，记录所有到达的节点，然后计算出这些节点的 $\mathcal{H}$ 函数值并将其取并就可以计算出 $\mathcal{F}$。但是这个算法需要对于图中每一个节点都进行深度优先遍历，每次深度优先遍历的时间复杂度为线性，所以总的时间复杂度是平方级别。这个算法在项目规模不大的时候运行时间尚能接受，但是对于规模较大的项目，我们就必须寻找更优的算法。

我们从一个特殊情况开始考虑：如果 G 中的边没有形成环，即 G 是一个有向

无环图，是否有更优的算法。实际上，对于这种特殊情况，我们可以使用动态规划的算法解决。我们按照出度对图 G 进行拓扑排序，令 $v_1, v_2, \cdots, v_n$ 是 G 的拓扑序列，即对于任意 v_i，如果 $(v_i, v_j) \in E, \quad$ 则必有 $j < i$. 我们可以从前向后处理整个序列，对于当前的 v_i, 找到其出边对应的所有 v_j。由于是拓扑序列，$\mathcal{F}(v_j)$ 一定已经被计算出了，于是直接令 $\mathcal{F}(v_i) = \bigcup \mathcal{F}(v_j), \quad \forall (v_i, v_j) \in E$ 即可。

对于图 G 是一般图的情况，我们可以使用极大强连通分分量的算法进行解决 [3]。一个有向图 $G = (V, E)$ 的一个强连通分量定义为 V 的一个子集 V' 使得对于任意 V' 中两个节点 v_i, v_j，它们均可以互相到达。而一个极大强连通分量即为一个强连通分量 V'，且对于任意 $v' \in V - V', V' \cup \{v'\}$ 都不是强连通分量。由于强连通分量的定义，我们可以发现对于属于同一强连通分量的中的两个节点 v_i 和 v_j，必有 $\mathcal{F}(v_i) = \mathcal{F}(v_j)$, 于是我们可以对于一个强连通分量，只计算一个 $\mathcal{F}$ 即可. 对于有向图 $G = (V, E)$, 我们可以使用 Tarjan 算法在线性时间内将 V 划分为若干个极大强连通分量的并集，即 $V = V_1 \cup V_2 \cup \cdots \cup V_k$ 且 $\forall i, j \in [1, k], i \neq j, V_i \cup V_j = \varnothing$. 求得极大强连通分量之后，我们构建新图 $G' = (V', E')$, 其中 $V' = \{V_1, V_2, \cdots, V_k\}$, $E' = \{(V_i, V_j) \mid \exists v_i \in V_i \ \exists v_j \in V_j \ s.t. \ (v_i, v_j) \in E\}$, 容易发现 G' 是有向无环图（如果不然，可以将其环上的节点求并从而得到一个更大的强连通分量），于是可以套用上文中有向无环图的算法求得新图中每个节点的 $\mathcal{F}$ 函数，然后原图中每个节点的 $\mathcal{F}$ 值即为对应的极大强连通分量的 $\mathcal{F}$ 值。Tarjan 算法运行时间为线性，在有向无环图中求 $\mathcal{F}$ 函数的时间复杂度也是线性，所以总的时间复杂度是线性。

3.9　流属性约束集合的优化

再次审视 let 规则，我们会发现如果 let 嵌套层数过多，流属性类型模板中的约束集合的中约束个数就呈指数级增长：

```
1    let x0 = e in
2        let x1 = λ.y if y then x0 else x0  in
3            let x2 = λ.z if z then x1 else x1 in
4                ...
```

这里考虑所有最外层的 let 绑定，不妨类型推导 e 时返回了子类型约束集合 C_0，因为 x0 的多态性，我们可以发现 x1 的约束集合 C_1 中会包含 C_0 的两个拷贝，同理由于 x1 的多态性，x2 的约束集合 C_2 中含有 C_1 的两个拷贝，即 C_0 的四个拷贝。如此下去，我们可以发现约束集合的中约束的个数是随着 let 的嵌套层数指数增长的。

为了限制其约束集合的增长，我们必须去除约束集合中的一些无用约束, 这里无用的约束由以下两种组成：

- 如果约束集合中存在 $\{\ell_1 \subseteq \alpha, \ell_2 \subseteq \alpha\}$ 的约束，可以将其合并为 $\{\ell_1 \cup \ell_2 \subseteq \alpha\}$.
- 如果约束集合中存在 $\{\ell_1 \subseteq \alpha, \ell_2 \subseteq \alpha, \alpha \subseteq \beta\}$，且 α 在表达式无自由出现，那么可以将其合并为 $\{\ell_1 \cup \ell_2 \subseteq \beta\}$

经过如此合并之后，可以发现约束集合中不同的流属性变量个数最多为待分析的表达式中出现的自由变量的个数，故最多约束个数的级别为 $O(n^2)$

3.10 let 绑定中流属性变量的实例化

由于流属性类型模板的实例化，导致最终加入到子类型约束集合中的流属性变量是实例化之后的新的流属性变量，而原有的 let 绑定中的流属性变量并没有被添加进约束集合。考虑程序:

```
1        let f = `$\lambda$`x. if x then False else True
2        in (f True, f False)
```

这里如果在 f 的定义中，x 分配的流属性变量是 α，那么它在后续调用 f 的时候就会被实例化为 β 和 γ（实例化的流属性变量总是被替换为全新的流属性变量）。这样如果我们需要询问 if 语句中 x 的流属性，就会得到一个流属性变量 α 而不是一个真实的流属性。

事实上，为了解决这个问题，我们需要在类型分析之后，再次扫描整个程序追踪所有 f 的调用，并将传入的参数的流属性求并集，作为 x 的流属性。但是这样不仅需要多次扫描程序，降低分析效率，而且算法较为繁琐，不易实现。

这里我们可以使用一个技巧，即如果将 α 流属性变量实例化为 β 流属性变量的时候，可以在子类型约束集合中添加一条约束：$\beta \preceq \alpha$，然后我们就正常求解所

有的约束集合并执行替换操作。由于此类型的存在，α 就会被自动替换为 β 和其余所有调用的参数的流属性的并集，从而得到正确的流属性结果。

3.11　算法总结

给定待分析的表达式 e，首先使用 4.5 章的类型推导规则推导出 e 的流属性类型 κ，与此同时得到其类型约束集合 D 和子类型约束集合 C。然后使用 4.6 中的算法求解类型约束集合，得到解 S。将类型约束集合的解 S 应用到子类型约束集合 C 得到 C'。接下来使用 4.7 中的算法将子类型约束集合转换为流属性约束集合 $\mathcal{D}$, 再利用 4.8 中的算法求解流属性约束集合 $\mathcal{D}$ 得到其解函数 $\mathcal{F}$. 于是对于 e 中任意一子表达式 e'，且 e' 有流属性类型 κ'。那么 $\ell = \mathcal{L}(\mathcal{F}(S(\kappa')))$ 就是 e' 的流属性，即 e' 最终可能求值结果的标号是 ℓ 的一个子集。注意在子类型约束集合构造和求解的构造过程中，使用 4.9 和 4.10 中提到的优化，优化实现并提高分析的效率。

第四章　实现细节

4.1　类型定义

本框架使用 Haskell 实现。其核心数据结构使用如下定义：

```
1 type Infer a = (RWST
2                     Env
3                     [TypedProgram]
4                     InferState
5                     (Except TypeError)
6                     a)
```

其中 RWST 是一个 monad，实现了 MonadReader, MonadWriter, MonadState 接口。这里 Env 是一个 reader monad，用于储存类型推导过程中的环境。TypedProgram 是一个 writer monad，用来存储各个子表达式的流属性类型。InferState 是 state monad，使用用来存储类型推导过程中所需要的信息，比如已使用的类型变量和流属性变量等。Except TypeError 是一个 except monad，用来表示类型推导过程中遇到的各种错误。

注意到控制流分析过程中，我们频繁地进行了替换操作，于是可以定义一个 Substitutable 类来抽象表示这一系列操作，以后遇到需要替换的类型，直接对其实现 Substitutable 接口，就可以调用 apply 来应用一个替换、调用 fv 获取其自由变量：

```
1 class Substitutable var value contr | var -> value where
```

```
2         apply :: Subst var value -> contr -> contr
3         fv    :: contr -> Set.Set var
```

这里 Subst 的定义为一个 Data.Map.Map 类型，即 Subst a b 为从类型 a 到类型 b 的替换。在 Substitutable 的声明中，var、value 和 contr 分别为需要替换的变量类型、替换的结果类型以及需要应用替换的类型，注意到由替换的变量类型可以直接确定其替换的结果类型（即流属性变量只可能替换为流属性，类型变量只可能替换为类型），所以我们可以使用 GHC 的 FunctionalDependencies 扩展 [6] 来提供更精确的的类型信息，简化后续实例的实现。

而整个推导算法的主体由一个函数 infer 实现，其中 infer 的类型签名为:

```
1 infer :: Expr -> Infer (Type, [TConstraint], [FConstraint])
```

infer 函数接受一个 expr 类型的输入，返回一个 Infer monad，其中 monad 的结果为该表达式类型, 类型约束集合与子类型约束集合的三元组。由上文 Infer monad 的定义我们知道，这其中也包含了输入表达式中所有子表达式的类型、类型推导错误信息等其他信息。

4.2　类型推导实现

由于 Haskell 提供的简洁又强大的 Monad 编程范式，类型推导的实现异常简单，我们只以相对复杂的 *Let* 和 λ 规则作为说明。

```
1 Let loc name e1 e2 -> do
2     (t, d, c) <- infer e1
3     case runSolve d of
4         Left error -> throwError error
5         Right sub -> do
6             t' <- local (apply sub) generalize t c
7             (tv, d', c') <- inEnv (name, t') $
8                         local (apply sub) (infer e2)
```

```
9             tell [(loc, tv)]
10            return (tv, d' ++ d, c' ++ c)
```

首先我们对于 let 语句中，所绑定给 name 的表达式 e_1 进行类型推导，结果为 (t, d, c)。然后根据类型推导规则，我们对类型约束集合 d 进行求解，如果求解失败，则直接返回失败的异常即可，否则得到一个解 sub。求解完毕之后，我们在应用过 sub 的环境之中对类型 t 进行泛化。这里 local 是定义在 reader monad 上的一个函数，其类型签名为

```
1 local :: (e -> e) -> m a -> ma
```

接受一个修改 reader monad 的环境的函数以及一个 reader monad，返回在新的环境中求值的这个 reader monad. 我们在这里用它来暂时修改类型环境，并在修改之后的环境中对 t 类型进行泛化。对 t' 泛化之后，我们将 $name : t'$ 加入到环境中，对 e_1 继续进行类型推导。这里 $inEnv$ 函数的定义为：

```
1 inEnv :: (Name, TypeScm) -> Infer a -> Infer a
2 inEnv (v, t) m = do
3     let scope e = (remove e v) `extend` (v, t)
4     local scope m
```

定义一个函数 scope 为接受一个环境 e，将其内原有的 name 键值删去，并新添加当前绑定的类型。然后利用 local 函数，对当前环境应用 scope 函数并在新的环境中推导出 e_2 的类型。推导出 e_2 的类型后，利用 writer monad 的 tell 的函数记录下当前表达式的类型，并返回其结果即可。

```
1 Lambda loc name e -> do
2         a <- freshTyvar
3         (t, d, c) <- inEnv (name, TyForall [] (FpForall [] [] a))
4                   (infer e)
5         let tv = TyArr a t (fpSingleton loc)
```

```
6         tell [(loc, tv)]
7         return (tv, d, c)
```

对于 λ 表达式，我们首先新建一个类型变量 a, 为了统一环境中类型，我们强制环境中的类型都是流属性类型模式的形式，即 $\forall\vec{\alpha}.C \Rightarrow \forall\vec{X}.\kappa$. 在将参数和新建的流属性类型模式添加进环境中之后，在新的环境中对 e 进行类型推导，在记录当前子表达式的类型后直接返回结果类型即可。这里值得一提的是 freshTyvar 函数的实现, 它内部使用了一个函数叫做 lettersTy：

```
1 lettersTy :: [String]
2 lettersTy = [1..] >>= flip replicateM ['a' .. 'z']
```

这里 lettersTy 使用了 list monad，并利用 replicateM 对其进行无限次重复，由于 haskell 的惰性求值特性，最终编译器只会把我们需要的类型变量求值出放入列表中。lettersTy 的结果是形如 $a, b, \cdots, z, aa, ab, \cdots, az, ba, bb, \cdots, zz, aaa, aab \cdots$ 无限长的列表，而这个函数如此简洁的实现更加体现了 haskell 这门语言高度的抽象能力和强大的表现力。

4.3　错误处理

众所周知，在实现程序静态分析的过程中，最为繁琐的缓解就是错误的处理。因为在静态分析过程中，错误的来源五花八门，可能来自分析算法自身的局限性, 甚至来自于用户不合法的输入。程序分析框架必须要识别出这些错误，并为用户提供友好的错误信息，甚至给出错误处理的建议。这给我们的实现带来了巨大的挑战。

我项目中定义了一个类型用于表示各类错误：

```
1 data TypeError
2     = InfinitType String String
3     | UnificationFail Type Type
4     | UnboundVariables Name
```

```
5       | GenerateFPConstraintFail Type Type
6       | IncompatibleFPConstraints FP FP
```

其中各个错误类型的代表的意义如下：

- **InfinitType** 如果将变量 α 替换为 κ，但是同时 α 也出现在了 κ 的自由变量中。此时这个替换可以无限次递归进行下去，形成一个递归的类型。由于我们的算法目前不支持递归类型，所以一旦检测出该情况，就会直接抛出一个 InfinitType 错误，并记录下参与替换的变量和类型作为错误信息打印给用户，方便用户修正代码。
- **UnificationFail** 在 unify 算法运行过程出现类型不匹配的错误，例如调用 unify 算法的两个参数一个是 Int 类型另外是 Bool 类型，这种情况下此类型约束集合就不存在解。我们一旦检测到这种情况就直接抛出 UnificationFail 异常并记录 unify 算法中不匹配的两个变量。
- **UnboundVariables** 在类型推导过程中如遇到一个变量未在环境中出现，则抛出 UnboundVariables 异常，并记录该变量名称方便用户排查。
- **GenerateFPConstraintFail** 在对子类型约束集合进行求解的过程中，如果遇到类型不匹配的情况，则直接抛出 GenerateFPConstraintFail 异常，并记录下不匹配的两个流属性类型。例如 $Bool^{\alpha} \preceq Int^{\beta}$，这里无论 α 和 β 取什么值都不可能使得 $Bool^{\alpha}$ 是 Int^{β} 的子类型，故抛出 GenerateFPConstraintFail 异常。
- **IncompatibleFPConstraints** 在求解流属性约束集合的过程中，如果遇到流属性不匹配的情况，则直接抛出 IncompatibleFPConstraints 异常并记录下不匹配的流属性。例如 $\ell_1 \subseteq \ell_2$，这里 $\ell_1 = \{l_1, l_2\}, \ell_2 = \{l_1, l_3\}$，此时直接抛出 $IncompatibleFPConstraints$ 即可。

第五章　试验评估

待填坑

总结与未来展望

5.1 总结

本项目通过扩展 Hindley-Milner 类型系统，定义了一种基于标号和流属性的类型系统，实现了对于静态类型系统的在程序编译时期进行控制流分析的工作。本算法相比于传统的控制流分析算法，具有求值顺序无关、时间效率高以及可模块化等优点，适用于大型项目的控制流分析。本算法可以通过形式化证明证明其正确性，也可以用对于正确性要求较高的项目中。除此之外，该算法的可扩展性强，可以用于所有基于 Hindley-Milner 类型系统的编程语言，如 Haskell, Ocaml 等。

在 Haskell 上对于本算法的实现为 Haskell 社区提供了一个简单易用的控制流分析框架。本项目由良好的 API 接口，用户可以直接使用其对自己的 Haskell 项目进行控制流分析，或者作为一个库包含进自己的项目，利用控制流分析的中间结果进行后续的静态分析。

5.2 相关工作

像 Ocaml, Haskell 等函数编程语言由于强大的抽象能力和表达力所带来的陡峭的学习曲线、较高的入门门槛，目前在互联网公司中仍未大规模使用，相关的编程工具仍处于一个相对匮乏的状态。例如 Haskell 语言上的语义层面的控制流分析项目目前仍是一片空白。虽然在学术界，函数式语言控制流分析已经有很多优秀的算法，如 kCFA 等，但是由于没有商业利益的推动，所以业界目前没有人将其实现为可用的库。

不过函数式语言由于其可靠性和易扩展性，在金融领域受到了青睐，目前有

很多金融公司都将其作为主力开发语言开发交易系统，量化交易模型等。使用函数式语言进行开发的金融公司内部有一大批专业人员进行函数式语言的理论研究和内部工具的开发，不过这些理论和工具往往不对外界公开。例如美国的金融公司 Jane Street，其交易系统主要使用 Ocaml 语言开发。Jane Street 公司内部有专业的研究人员研究基于 Ocaml 的程序静态分析理论，并自行开发了一套 Ocaml 工具链，其中就包括功能强大的 Ocaml 静态分析程序，不过这些工具并不公开，所以外界的开发人员并不能使用这些工具帮助自己项目的开发。

5.3 局限与未来

本项目可以在一定程度上对 Haskell 语言进行控制流分析，但是由于 Haskell 的类型系统不是完全的 Hindley-Milner 类型系统，而包括诸如 kind 高阶类型、GADT 等更复杂的类型特性，故目前可以成功分析的只有 Haskell 的一个子集。同时本算法目前不支持递归类型，这也一定程度上阻碍了它向更复杂的类型系统的扩展。

因此在未来的工作中，我们可以考虑从递归类型和高阶类型角度扩展该算法，能够处理更加复杂的类型系统，从而将项目推广到整个 Haskell 语言。此外，对于有静态类型系统但是非函数式语言的控制流分析也是一个比较有潜力的方向，比如目前在 C++, Scala 上的控制流分析都没有从类型系统角度来进行算法设计，我们可以将我们的算法推广到这些语言上，从而得到更快更精确的结果。另外本算法中流属性概念不仅可以用于进行控制流分析，也可以辅助类型推导，比如在有子类型的编程语言中，利用流属性概念就可以得到更为精确的子类型推导结果。总之，未来的研究会朝着表达能力更加强大的流属性类型系统的方向发展、更加通用的分析框架。

参考文献

[1] C. Mossin *"Flow Analysis of Typed Higher-Order Programs"* . In:cs.ucla.edu. 1997, pp. 19-58. 85-97.

[2] Shivers, Olin. *"Control-flow analysis of higher-order languages"*. Diss. Carnegie Mellon University Pittsburgh, PA, 1991.

[3] Tarjan, R. E. *"Depth-first search and linear graph algorithms"*. In:SIAM Journal on Computing 1 (2). 1972, pp. 146–160

[4] Hindley, J. Roger *"The Principal Type-Scheme of an Object in Combinatory Logic"*. In: Transactions of the American Mathematical Society 146, pp. 29–60

[5] Milner, Robin *"A Theory of Type Polymorphism in Programming"*. In:Journal of Computer and System Science (JCSS) 17, pp. 348–374

[6] Hallgren, T. *"Fun with Functional Dependencies or Types as Values in Static Computations in Haskell"*. In:Proceedings of the Joint CS/CE Winter Meeting (Varberg, Sweden) Jan. 2011

致谢

在研究过程中，非常感谢熊英飞老师的悉心指导，我们在研究过程中进行了很多的讨论和设计，熊老师给我提了很多具有建设性的意见。

此外我还要感谢辛苦的答辩评委们进行阅读并提出宝贵的意见，没有你们这篇论文也失去了写作的意义，感谢各位能够让我们的研究得到一个良好的总结。

最后感谢父母，同学，学院，学校对我们科学研究开展的支持以及对于良好环境的提供。

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：　　日期：　　年　　月　　日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校在 □ 一年 / □ 两年 / □ 三年以后在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：　　导师签名：　　日期：　　年　　月　　日