北京大学

本科生毕业论文

题目： **API迁移问题上的程序变换语言设计与实现**

姓　　名：汪成龙

学　　号：1100012785

院　　系：信息科学技术学院

专　　业：计算机专业

研究方向：程序语言

导　　师：熊英飞

2015.05.21

版权声明

摘要

现代软件的开发离不开对于API的依赖，API的版本升级问题，以及对于API需求变化带来的API迁移问题就成为了软件维护的一个重要问题。在大规模的软件开发过程中，手动维护会极大的增加软件的维护成本，同时较高的错误率也导致了其他维护问题的产生。因此，能够快速的开发一个能够保证程序变换正确性的自动化工具就成为了软件维护者的梦想。

然而，API之间复杂的映射，尤其是多对多的API调用映射关系，让自动转换工具的开发变得尤为困难。本文设计了一种领域特定语言(Domain Specific Language)来解决因API问题而进行程序改写的问题，通过语言机制来保证了API迁移过程中的程序变换的正确性，同时提供了用户友好的映射规则定义接口。本文定义了该程序变换语言的表里语义：表语义为用户书写转换规则过程中对于API规则的理解，而里语义则是转换规则在程序变换过程中的作用法则，并通过程序分析的手段来自动分析API转换规则在用户程序中的依赖关系，让程序的表里语义能够在实际转换过程中起到等同的作用，以此来保证变换过程在符合语言使用者理解的条件下自动推广到复杂的显示情况中。

关键词：程序语言，程序变换

A Safe Transformation Language for Code Adaptation between APIs

Chenglong Wang (Computer Science)
Directed by Prof. Yingfei Xiong

Abstract

Modern software development can not progress without the use of APIs, and this makes incompatible API update or API switching become an important but tough task for software maintainers. It is not feasible for programmers to manually update these programs between the APIs, as errors are likely to appear with manual effort, which even increases the cost of software maintenance. Thus it is desirable to have automatic program transformation tools to support the program transformation process that can maintaining the safety of the transformation process.

To make the transformation tool programming easier, we designed a domain specific language PATL, which providing safe checks in the language semantics with user-friendly interface. In the language design, we provide an external semantics, which enables users to understand the rules they write easily, as well as an internal semantics for the actual transformation of user programs that can handle complex program transformation cases. The magic between the two semantics is the design of an analysis system which support semantic-preserving program rewriting before adaptation, which actually achieves the goals to enable users to transform complex programs with their simple rules written with the understanding of the external semantics.

Keywords: Programming languages, Program transformation

目录

序言

现代的程序开发过程大量的依赖着对于API（Application Programming Interfaces，应用程序接口）的使用，因而在程序中进行API的替换也成为了程序开发过程以及软件维护过程中开发者的一项重要工作。其中，程序中的API改写主要可以分为以下两个类别。第一类是API的升级：当一个API推出的新版接口并不兼容旧版API接口时，使用旧版API的用户程序就需要进行改写来满足对于新API的使用。另一类则是API的切换，这一类情况常常出现在用户希望将原有的应用程序迁移到另一个平台上（例如从Android到IOS的程序切换），或是因为对于API性能以及维护功能上的其他需求而要求将原有的API切换到另一个API集合中去。在以上描述的这两种API更替场景中，我们需要将使用API_1的程序变换为使用API_2的程序，而API_1与API_2之间具有同构性，这样的替换也即相同功能之间的不同接口替换。然而，这样的程序变换并不容易。当用户手动进行这样的程序转换时，由于并非所有用户都是API的专家，他们并不能够全盘掌握两个API之间的映射关系，以及难以顾及到不同转换之间的交叉问题，手工的转换带来的结果就常常是错漏连篇的新程序，这带来结果就是更加高昂的软件维护成本和更多的人力资源消耗。

既然对于API升级的程序改写如此重要而又如此困难，让API发布者提供一个自动的程序转换工具就成为了帮助客户端程序员（即使用API的用户程序员）进行程序升级的一大利器：当API进行更新的时候，API发布者在发布API库的时候同时提供了一个用户升级工具来帮助客户进行程序升级，用户只需要让工具自动的将原有的程序进行改写就成为了使用新API的程序。让API提供者开发这样的转换工具具有相当的好处：一方面，作为API的专家，他具有更加丰富的API映射关系的知识，他能够处理还API之间的多方面映射关系，而另一方面，这样工具的开发是独立于用户程序的，一次开发就能够让广大用户都可以进行程序的升

级，可以广泛的解决用户的需求问题。这样的工具例子有1）微软为了推广自己的.Net平台，提供了Visual Basic到Visual Basic.Net的程序转换工具，2）RIM提供了一个从Android到Blackberry的平台迁移工具，让使用Android API的程序能够自动改写为使用Blackberry的工具，从而可以让转换后的程序在Blackberry上运行而不需更多的工作。

然后，开发这样的Source-to-source自动转换工具并不容易。从转换工具的市场上来看，只有极小的一部分API拥有了这样的自动转换工具来完成程序的改写工作，其中一个重要的原因就是这些API间的程序自动转换工具常常会引入一些新的错误，或是在具体的程序环境中并不能完全按照API开发者的意图进行程序之间的改写。其中一个典型的例子就是Python 2to3.script工具，这个工具是设计来将使用python 2.x版本的程序升级为使用python 3.x版本的程序。虽然这个升级脚本能够在一定程度上完成对于python API的升级工作，但在升级过程中也引入了较多的错误，在有Pilgrim与Willison的案例研究14中，2to3.script会导致包括类型错误在内的很多程序错误。从另一方面来看，程序变换开发的困难源自于专业开发环境的支持。由于并非所有的API开发者都能够熟悉程序语言的修改操作，让他们从底层开始实现这样的转换工具就显得困难重重。

事实上，为了解决程序变换的难题，程序语言社区设计出了很多的程序变换语言1256。这些语言通过高层次的抽象来帮助使用者避开了底层开发中容易产生的错误，例如转换语言中能够保证程序变换前后不会产生语法错误，同时能够帮助开发者更好的描述自己需要表达的转换对象。然而对于API迁移这样一个特定场景的问题，这些变换语言的设计就显出了明显的不足，主要体现在以下几个方面：

- API变换过程中的类型安全并没有得到保证：这些通用的程序变换语言由于广泛的解决了多类的程序变换问题，类型安全就显得难以控制，而在较为单一的API升级问题中，变换对象保持在了statement以及expression层次上，类型的正确性就变得可控同时具有高层次的操作性。
- API之间的映射没有得到高层次的抽象，因而这些转换仍然需要较多对于用户context的明确指定。由于API转换工具开发者需要能够在未知用户程序环境的情况下完成映射规则的指定，复杂的context就难以得到实现。
- API升级之中多对多的API映射没有得到良好的支持：API之间的多对多问题

涉及到了程序之间的环境分析问题以及转换语句之间的依赖关系问题，而大多数变换语言都是从语形上进行的转换，这样的转换就忽略了转换对象在运行时候的以来关系以及对应关系，从而导致了变换过程的实际语义遭到了违反。

我们接下来通过几个实际的例子来描述API之间程序改写中的常见问题。

类型安全问题 类型安全问题主要体现在程序迁移前是类型正确的而改写之后出现了类型错误，举例如下。（该案例来自15）

在Java图形界面编程API Swing到SWT的升级过程中，Swing在创建一个窗口时需要首先创建一个Container来存放窗口中的内容，SWT也同理需要创建一个对应的Composite类来进行存放。(类图见图.1)因此在升级过程就就有了这样的两条对应关系（此处的规则只是简单的描述了API之间的映射关系，其中的π_1可以读作“将`Container`类的构造函数映射到`Composite`的构造函数，其中`Composite`的构造函数参数为`new Shell()`以及0”，第二条规则则是将`JList`类映射到`List`类，并将其构造函数进行对应的替换。）：

$$\pi_1 = \texttt{new Container() -» new Composite(new Shell(),0)}$$

$$\pi_2 = \texttt{new JList() -» new List()}$$

在这样的转换规则下，用于转换如下用户程序时就会出现类型的错误。

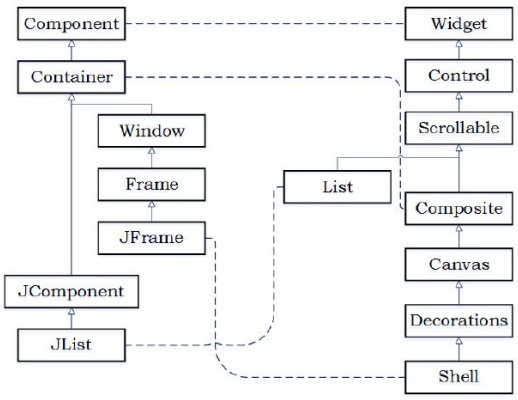

图 *1: Swing(图左)到SWT(图右)之间的类图对应关系，其中虚线箭头表示两个API之间的类映射，实线箭头表示同一API中类的继承关系（往下为父类）*

```
JList list = new Container(); ⟶ List list = new Composite(new Shell(), 0);
```

其原因在于在图.1中，Swing类型中的JList是Container的父类，因而左部满足子类关系而右部的List不再是Composite的子类，因而转换之前的类型是正确的，而转换之后就会导致类型错误的出现。

API之间多对多的映射问题 比起前面所提到的类型问题，程序变换中更复杂的多对多问题更为引人注目。API之间多对多映射带来的主要问题则是用户程序上下文之间的不一致性，这样的问题并不像类型错误那样能够通过静态检查就能够得到并修复，而复杂的runtime错误让这类问题更容易隐藏。

在这里我们先简单使用程序匹配语言SmPL1的语法来展示变换的规则。

```
(a:A->B) {
  - a.x();
  - a.y();
  + a.z();
}
```

这一条规则就代表了一组API到另一组API之间的映射，元变量定义部分的`a:A->A`就代表了元变量在原来的API集合中类型为A，在新的API集合中类型为B，而这条规则表示了“每当遇到一个类型为A的变量a连续调用了a.x()与a.y()两个方法，我们就删除这两条语句，并用a.z()这样的一条语句来进行替换。”

假设我们现在的转换空间具有如下两条规则：

```
(a:A->A) {
  - a.y();
  - a.x();
  + a.n();
}
```

Rule-1

```
(a:A->A,b:int->int) {
  - b = a.z();
  - a.x();
  + b = a.m();
}
```

Rule-2

让我们来考虑如下几种实际的转换类别来分析这两个例子的具体问题。

1. 情形1：在同一个context可以正确转换的例子。

```
                          if (true) {
                            a.y();
a.y();                      a.x();
a.x();                    } else {
                            b = a.z();
                            a.x(); }
```

在这两个例子中，用户规则可以直接的在一个基本块中匹配到需要进行转换的语序序列，在类型和语形都能够正确匹配的情况下就能够进行正确的转换。因此可以比较容易的完成操作。

2. 情形2：被匹配的代码并不直接出现在连续块中，而需要进行调整来进行转换。

```
                                  if (i<0)
b=a;              a.y();            a.y();
a.y();            i++;            else
b.x();            a.x();            b = a.z();
                                  a.x();
```

在上述的三个例子中，虽然并没有直接出现连续的转换指令，但是这些代码块却可以进行调整来使得代码变得可以转换。例如第一段代码中，由于b与a为alias关系，可以将b.x();重写为a.x();，从而让转换可以正常进行。第二段代码中i++;与a.x();并没有依赖关系，从而可以交换位置让a.y();a.x();连续出现，因而也可以进行转换。第三段代码则可以通过将a.x();移动到两个分支语句中，从而让两个分支分别进行转换。

3. 情形3：类似于第二种情形，但是却因为以来关系的出现而无法转换。

```
b=a.z();
if (b<0)
  a.x();
else
  a = new A();
```

在上面的这一段代码中，由于分支语句的条件b<0依赖于b=a.z();， b=a.z();就不能像上述2-3的语句一样进行移动，从而产生了一个转换异常需要报告给用户。

由于上述的种种困难，开发一个能够正确处理API之间程序改写的工具就显得尤为困难，因此在实际应用中，一个良好的程序转换工具应该能够保证独立于客户端程序的正确性：假设对于用户程序p，有一条在p上的性质ϕ，那就需要对于一个转换程序Π，使得对于所有的p，都能够做到$\phi(p)$成立可以得出$\phi(\Pi(p))$也成立。这个关系可以形式化的表述为：

$$\forall p.\phi(p) \Longrightarrow \phi(\Pi(p))$$

在我们的目标中，我们就把性质ϕ定义为类型正确性以及依赖关系的正确保持。此外，让程序能够完全自动的转换任意的客户端程序而不会产生错误显然并不是一个可行的手段。实际进行程序变换的过程中，应当能够有规则检查系统来对转换程序进行检查，可以将规则的错误以及规则在非规范程序上可能产生的不正常表现报告给用户，从而让用户能够理解转换工具将在程序中进行的操作，并保证能够在所有错误正确处理的情况下就能够保证性质π的保持。这个关系可以由以下关系表示：

Property 1 (Transformation Safety) *Given* p,Π,*if* $\mathsf{check}(p,\Pi) == \mathtt{true}$ *and* $\phi(p)$ *holds, then* $\phi(\Pi(p))$ *holds, or else* $\mathsf{check}(p,\Pi)$ *will report warnings for the transformation.*

除了对于转换目标正确性的支持，语言应当能够具有以下的特点以能够实际作为API之间程序变换问题的一个解决方案：

1. 能够高层次的描述API间的程序变换问题，表达能力要足够强，让用户能够表达绝大部分的API转换问题同时尽量减少对于非必要功能的依赖。
2. 转换规则的书写应当是独立于用户程序，需要能够保证作用在不同用户程序下都保持相同的转换特征。
3. 语言应当能够在用户程序中的API使用违反API使用规则时向用户提供错误信息，在转换前就能让用户知道需要进行修改的部分。

本文主要就从语言机制的角度探讨了对于API之间程序改写问题的解决方案。本文设计了程序语言Patl，一种用具解决API升级问题中的多对多问题程序变换语言。本文的主要启示就在于将匹配放在了控制流图之中进行，并且在实际的程序改写之前进行一系列的语义等价转换，最后才在等价变换的程序变换基础上进

行实际的API改写操作，同时在改写之中保持程序变化之间类型性质的转换正确性。

在接下来的章节中，我们首先会具体说明具体的研究问题难点，形式化的概述我们的解决方案（第一章），通过几个具体案例说明我们的解决方案（第二章），形式化的描述语言的语形语义（第三、四、五章），最后分析语言的应用以及在几个数据集下的分析结果（第六章）。

第一章　方法概述

在上一章中，我们提出了变换语言的设计目标。我们将在本章通过几个案例来阐释Patl语言的设计。

1.1　Patl的转换框架

1.1.1　表语义与里语义

为了缩短抽象的转换规则与具体的语言作用效果的鸿沟，我们为Patl赋予了表语义与里语义。其中表语义（也即抽象语义），是为用户提供理解的转换语义，用户在书写规则时可以根据表语义来理解其转换规则在程序中的作用方式。而在表语义之外，Patl还具有里语义（即算法语义），其作用是等效的将表语义所表示的变换最大化的推广到程序的控制流图中，同时将可能违背表语义转换法则的程序部分报给给用户进行手动处理。这样设计的重要原则就是让表语义与里语义能够具有等效性，否则转换程序作用在客户端程序上的时候就会导致违背用户规则意图的效果。当表里语义做到等效作用时，用户就可以在只需最小限度的理解程序的表象效果就能够完成复杂程序上下文中作用的工作。

接下来我们将形式化的描述这个转换框架。

Definition 1 (表语义转换) 给定一个转换程序Π，以及两个程序p_1，p_2，如果p_2满足：对于p_1中所有的被规则Π匹配的连续程序块，p_2中都由规则Π的映射结果替换，则我们称p_1与p_2具有表语义转换关系，我们使用符号$p_2 = \Pi \circ (p_1)$来表示。

简单来说，表语义就是在引言部分所使用的“块替换”的转换方法：给定一条转换规则$\Pi = (\,)\{-p_1; -p_2; ...; -p_n; +q_1; ...; +q_n\}$来转换一个程序语句序列$s_1; ...; s_n$，表语义所进行的转换即是找到$s_1; ...; s_n$中的一段顺序语句$s_{l_1}; ..; s_{l_u}$，

使得这段语句匹配$p_1;...;p_n;$，然后将语句序列替换为$q_1;...;q_n$，从而使得程序中的对应片段得以转换。

Definition 2 (里语义转换) 里语义转换定义为根据算法语义（见后文）进行的上下文有关的程序匹配替换，我们用符号$p_2 = \Pi \bullet (p_1)$来表示p_2是p_1在Π的里语义下的转换结果。（作为简化，我们也常常用$\Pi(p)$来表示里语义转换）

给定程序上的一个语义等价关系“$\doteq$”（即$p_1 \doteq p_2$意味着p_1与p_2语义等价），我们使用下图来描述这两个转换的关系。

$$\begin{array}{ccc} q & \xrightarrow{\Pi\circ} & q' \\ \updownarrow{\scriptstyle \doteq} & & \updownarrow{\scriptstyle \doteq} \\ p & \xrightarrow[\Pi\bullet]{} & \Pi(p) \end{array} \tag{1.1}$$

这个交换图表述了里语义与表语义之间的关系：里语义在一个程序p上的作用结果等价于表语义在q上的作用结果，而其中q与p是语义等价的两个程序。这个图的交换性就表述了Patl在里语义与表语义的设计目标：里语义与表语义的作用在等价关系$\doteq$的意义下等效（下文将在语言设计的基础上证明该图的交换性以及等价关系$\doteq$的等价性）。

此外这个等效关系也可以通过抽象解释的方式来进行阐释：首先我们将p到$\Pi(p)$的流程转化为$p \leftrightarrow q \rightarrow_{\Pi\circ} q' \leftrightarrow \Pi(p)$。那么，对于抽象语义函数$S$，我们持有以下关系：

$$\begin{array}{ccccccc} p & \xleftarrow{\doteq} & q & \xrightarrow{\Pi\circ} & q' & \xleftarrow{\doteq} & \Pi(p) \\ \downarrow{\scriptstyle S} & & \downarrow{\scriptstyle S} & & \downarrow{\scriptstyle S} & & \downarrow{\scriptstyle S} \\ S(p) & \xleftarrow{=} & S(q) & \xleftarrow{=} & S(q') & \xleftarrow{=} & S(\Pi(p)) \end{array}$$

这个关系的含义也即是上图中的交换图代表的含义，意味着每一步转换都保持了语义的等效性，（其中的API迁移过程也是在保持语义下的API）从而也表示了变换框架里语义与表语义所应当具有的对应关系。

1.2 Patl的转换流程

在表述了这两种语义的基本框架后，我们就开始着重介绍里语义的转换案例，通过这样的转换案例的介绍，我们将知道转换程序具体作用在客户端程序上的表现。

在这一个章节中，我们采用如下从SWT到GWT的一个转换案例来解释语言的作用法则。这一个样例展示的是从SWT到GWT的一个表格绘制功能的转换。

两个API在此处需要完成的工作都是创建一个表格，给表格添加一些列，然后在表格的第一列添加一个单元格。

(a) Source client code:

```
1 column.setWidth(100);
2 TableCell cell1 = new TableCell();
3 column.add(cell1);
4 TableCell cell2 = new TableCell();
5 column.add(cell2);
```

(b) Target client code:

```
1 GridCell cell1 = new GridCell(100,
      column);
2 GridCell cell2 = new GridCell(100,
      column);
```

图 *1.1: Client code*

上图中的a)部分表示的是SWT中的规则，SWT会首先创建一个列，为列设置好宽度属性之后创建一个单元格，再将单元格加入到列中。上图的b)部分则表示了GWT的修改方式：不需要将单元格加入到列中，而是把列`column`以及宽度`100`直接作为参数传入。

给定了这样的对应关系，我们可以定义如下的两条转换规则：

这两条转换规则就表述了上述的SWT到GWT程序之间的对应关系。

表语义转换规则

根据我们前文的表语义做法，我们就会进行语形以及类型上的匹配来确立需要转换的程序片段，从而进行直接的替换。例如对于图.1.1中的例子，我们就会通

```
/*tRow*/ ( col: TableColumn ->
    GridField,
       cell: TableCell -> GridCell,
       w : int -> int) {
  m col.setWidth(w);
  - cell = new TableCell();
  - col.add(cell);
  + cell = new GridCell(w, col);
}

/*tSetWidth*/ (col: TableColumn ->
    GridField,
           w: int -> int) {
  - col.setWidth(w);
}
```

图 *1.2: Transformation program*

过语形以及类型的匹配让这些规则与用户语句产生绑定，然后根据语句的绑定关系进行代码生成。

具体在图.1.1中，其中`tSetWidth`这一条规则就会使得其中的语句序列与设置宽度的语句产生绑定，而`tRow`这条语句能够让添加单元格的语句进行匹配。匹配完成后就会产生元变量与Java变量之间的绑定关系，然后在进行元变量的替换来产生需要生成的语句，从而替换已经被匹配的连续块。

里语义的转换规则

如上文所说（对应引言部分的转换方式），里语义还会在转换过程中引入一系列等价变换来确保表语义足以表达更复杂环境下的转换方式。我们使用下述的例子来解释我们的里语义作用方式。

```
TableColumn column = columnList.get(0);
column.setWidth(100);
if (toLeft) {
  column.add(new TableCell());
}
```

我们观察这个例子，与前文不同的是，这个例子中的一个分支语句`if(..){...}`把`column` 的 `setWidth` 以及 `add()` 方法出现在了不同的block中。因此我们在转换时需要考虑到不同block之间的影响。这个例子的转换可以分为以下步骤：

1. 预处理：在这个阶段，源代码之中的语句将会被转换为三地址风格的语句。这样的处理可以让源程序与其控制流图具有直接的对应关系。在这个例子中，第四行代码将会分裂为两端代码，包括创建单元格以及设置单元格，转换结果如下：

```
1 TableColumn column;
2 column = columnList.get(0);
3 column.setWidth(100);
4 if (toLeft) {
5   Cell cell;
6   cell = new TableCell();
7   column.add(cell);
8 }
```

2. 匹配：在匹配阶段，所有符合匹配模式的语句序列都会进行匹配。此处的语句序列需要满足的关系是这些语句在执行时满足序列关系的语句。意即：当一组语句将会在实际执行阶段满足前后关系，并且这一组语句能够与语句模式进行匹配，那这一组执行语句对应的程序语句就将会与模式产生绑定。在这个例子中，会产生两组绑定：tRow与3,6,7三行产生绑定，而tSetWidth与第3行的代码产生绑定。（这样的绑定是根据控制流图上执行序列关系进行的绑定，具体做法可以参见下文。）
3. 基于绑定的等价移动：在等价移动的阶段，我们会将可能呈现数据流序列中的匹配语句移动到同一个block中，同时保持其他的分支语句执行序列关系不变。在考虑依赖关系的条件下，这个操作的结果要做到匹配到的语句都出现在同一个基本块中，同时整个程序的语义完全等价于原有的程序。由于我们在上一步骤（匹配步骤）产生的语句出现在了不同的基本块中（tRow绑定的语句中 `column.setWidth(100);` 出现在了第一个block中，而6,7行的两个语句出现在了分支语句的第一个分支块中）。

   因此，根据之前的绑定关系，我们将上图中的 `column.setWidth(100);` 移动到分支语句的第一个分支中，但如果仅仅将这条语句放入分支中，该程序的语义就与原有的程序产生了不同，因而我们在移动的同时还会创建一个 `else` 分支，从而保持语义的等价性。

   在这个样例中最后的移动结果如图所示：（`column.setWidth(100);` 被复制到了分支语句的两个分支中，上一步匹配到的三条语句都同时出现在了if分支

中)。

```
TableColumn column;
column = columnList.get(0);
if (toLeft) {
  column.setWidth(100);
  Cell cell;
  cell = new TableCell();
  column.add(cell);
} else {
  column.setWidth(100);
}
```

4. 程序修改：在进行移动之后，那些被绑定的语句就可以通过表语义类似的修改方式来进行修改。在修改之中，我们将那些被“-”标注语句模式匹配的语句块替换为由“+”所标记模式生成的语句。由于被“-”标注语句匹配到的语句常常并不连续，我们将在这些删除语句中寻找一个合适的生成点来生成这个语句。(当这个生成点无法寻出时，我们就会产生一个warning来提示用户，让用户手动进行修改。)

   在这个例子中，我们会将第5行的语句删除（这一个语句与`tSetWidth`规则中的删除语句模式产生了绑定)，并将第4,6,7行的语句删除（同理，这三句话与`tRow`产生了绑定，因此对应的模式将会在对应的位置生成：`cell=new GridCell(100,column);`将会在第6行的位置生成，其他的删除语句将会进行删除。)

   进行程序修改之后的样例如下图所示：

```
TableColumn column;
column = columnList.get(0);
if (toLeft) {
  Cell cell;
  cell = new GridCell(100, column);
} else {
  column.setWidth(100);
}
```

   进行了这样的修改之后我们就成功的根据了匹配绑定的语句完成了程序中的语句删除与生成，至此，API的变换操作就已经完成。接下来的步骤则是用于还原最初正则化程序操作的后处理工作了。

5. 后处理：完成了程序的改写工作之后，我们就需要通过后处理工作来对之前normalization造成的基本修改进行一定程度的还原，将三地址风格的Java程序变换为普通风格的Java程序。这一步进行的主要操作就是将引入的可销毁变量通过inlining的方式进行销毁，这样就可以保证程序的风格最大限度的接近原有的程序风格。

   在这个例子中，我们在normalzition这一步中创建了一个`cell`变量，我们在这里就将`cell`变量的定义与其创建方法合并。

```
1 TableColumn column;
2 column = columnList.get(0);
3 if (toLeft) {
4   Cell cell = new GridCell(100, column);
5 } else {
6   column.setWidth(100);
7 }
```

   进行完后处理操作之后，我们的程序转换结果就呈现出了如上图的效果：原有的SWT的构造方法被GWT的构造方法替换，同时较大限度的保持了原有的程序风格。

这样从源到源的代码转换就产生了如图的所需转换结果，这也即是里语义的作用方式。

1.3 表里语义之间的关系

我们在这一章第一节讲述了表里语义之间应当具有的等价关系，在这一节，我们将具体阐述语言Patl的表里语义之间的等效关系。同样的，我们在这里简单的阐述两者的等效性，其证明将在后文介绍完语言形式化定义之后进行更加具体的证明。

事实上，一个转换规则Π的里语义可以分解成为三个部分，即$\Pi\bullet = \beta \cdot \Pi \circ \cdot \alpha$，其中$\beta$与$\alpha$为两个等价变化，使得$\beta(p) \doteq p$以及$\alpha(p) \doteq p$。其中的$\Pi\circ$表示的是其表语义。

在这里，我们将里语义的作用分解为两个等价变换与表语义的复合，此处的等价变换就起到了连接表里语义的关键步骤：让表语义作用在一个等效的程序上，

从而API迁移这一修改动作保持了用户所理解的操作：在这里，我们将等价变换操作定义为以下基本等价变换操作的复合：

- 正规化：将程序的语句转换为三地址代码形式。
- 重命名：当两个变量u,v在位置l，互为别名(Alias)关系时，在位置l处将变量u重命名为v。
- 移动：当一个语句s与分支语句条件不产生依赖关系时，将其从分支语句前复制进入两个分支语句的首部，或将分支语句后的一条语句复制到两个分支语句的尾部。
- 交换：当两条相邻的语句s_1, s_2不产生依赖关系时，将这两条语句的位置进行交换。

有了这样的一些基本等价操作，我们就保证了里语义的作用是等价变换与表语义作用的符合，从而保证里语义（实际的算法语义）作用在一段程序p上时能够保持作用行为的等效性。因此，我们就可以将里语义与表语以分别在不同的层次上展示：在用户书写转换规则时，只需按照表语义的方式来理解转换语言的作用原理而不需要考虑实际程序中复杂的依赖关系以及变量之间可能存在的别名关系，只考虑转换规则在顺序程序基本块上的作用；而当转换程序实际作用在客户端程序上时，实际作用就是前文所述通过等价变换的方式，让所有可能会被转换的语句移动到同一个基本块中，接下来再通过表语义的作用来进行转换。

在简单的介绍完语言的表里语义作用方式及其关系之后，我们就将在下文形式化的来介绍语言的设计。

第二章　语形(Syntax)

在这一个章节，我们将通过对Patl语形的定义来形式化的介绍这一门变换语言。

2.1　中量级Java (Middleweight Java)

出于形式化的清晰以及让读者可读性的考虑，我们将Patl的作用对象定义在了Middleweight Java4上。因此在介绍Patl之前，我们首先介绍Middleweight Java的特点及其语形。

Middleweight Java(简称作为MJ)，作为Featherweight9 的扩展形式（FJ是著名常用的Java的函数式抽象语言，其简洁的模型让很多论文都采用其来进行对于Java核心功能的模拟），MJ模型化了Java语言的很多命令式编程特征：例如赋值语句，域访问(Field Access)语句，空指针，构造函数，以及block结构与副作用系统。

由于我们的程序首先会在Middleweight Java上进行正规化操作，将MJ语言规范到三地址形式的MJ上，因此，我们在形式化的时候采用了MJ的一个三地址版本：我们将方法调用(Method Invocation)，或者对象构造(Object construction)，域访问语句的目标对象，分支语句的条件，循环语句的条件都降维到了单变量的形式（通过在语句执行前进行变量赋值来降维所需参数的形式）。因此我们在接下来的章节中将采用正规化的MJ形式而不再反复说明MJ的正规化过程，具体过程可以参见实现中的Java文档。

此外，在我们的形式化记号中，我们采用Pierce13的bar notation来模式化重复的元组：对于一个元素序列$a_1, a_2, ..., a_n$以及在单一值上定义的元素操作，我们采用如下的上划线记号来表示:$\bar{a}$。例如：$\bar{a} < \bar{c}$等价于$\forall i.a_i < c_i$，符号$\bar{C}\ \bar{f}$等价

于$C_1\ f_1, ..., C_n\ f_n$，其中n是这个序列的长度。

中量级Java的语形如下图所示：

p	::=	$\overline{cd}$	(Program)
cd	::=	`class C extends C` $\{\overline{fd}\ cnd\ \overline{md}\}$	(Class)
fd	::=	`C` f	(Field)
cnd	::=	`C(`$\bar{\texttt{C}}\ \bar{x}$`){`$vd$ `super(`$\bar{e}$`);` $\bar{s}$`}`	(Constructor)
md	::=	$\tau\ m$`(`$\bar{\texttt{C}}\ \bar{x}$`){`$vd\ \bar{s}$ `return` x`;}`	(Method)
vd	::=	$\bar{\texttt{C}}\ \bar{x}$`;`	(Variable Decl)
τ	::=	`void` | `C`	(Return Type)
s	::=	ps	(Statement)
	|	`if(`x`) {`$\bar{s}$`} else {`$\bar{s}$`}`	
	|	`while(`x`){`$\bar{s}$`}`	
ps	::=	$x = e;$ | $x.f = e;$ | $pe;$	(Primitive Stmt)
e	::=	`null` | x | $x.f$ | `(C)` x | pe	(Expression)
pe	::=	$x.m(\bar{x})$ | `new C(`$\bar{x}$`)`	(Promotable Expr)

图 *2.1: Syntax of normalized MJ, where metavariable x ranges over variables, f ranges over field names, and C ranges over type names.*

中量级Java的程序是由一组类的定义构成，而其中类的定义则是由类名，扩展对象名以及其体内的域定义，构造函数以及方式定义而组成。其中的域定义，方法定义以及构造者定义都与普通的Java类似。在语句层次，为了简明起见，我们没有model类似于断言语句这样实质是表达式语句的形式，而只考虑了最为典型的几种：包括基本语句（更细分为赋值语句，域赋值语句，以及可提升表达式语句）和分支、循环语句。我们在设计中将基本语句与赋值语句分离开，是为了让程序在进行变换时候的作用对象都是在基本语句的层次。最终的表达式层次则模式化了变量，创建语句，空指针，以及转类语句。

有了这样的一些基本语言定义，我们就可以通过定义Patl在这个正规化MJ的作用在定义Patl的语义了。

2.2 Patl语形

我们将Patl语形放置在如下图所示。Patl语言的特点在于其规则化程序，程序由一组规则构成，因而用户在书写程序时只需要指明旧API与新API之间的对应关

系就能作为转换程序的主题来描述整个转换过程。

$$
\begin{array}{lcll}
\Pi & ::= & \emptyset \mid \Pi, \pi & \text{(Rule Sequence)} \\
\pi & ::= & (\bar{x} : \overline{\mathsf{C} \hookrightarrow \mathsf{C}})\ \{I\} & \text{(Transformation Rule)} \\
I & ::= & I^m; I^-; I^+ & \text{(Rule Body)} \\
I^m & ::= & \emptyset \mid \mathbf{m}\ p; I^m & \text{(Context Pattern)} \\
I^- & ::= & -\,p \mid -\,p; I^- & \text{(Source Pattern)} \\
I^+ & ::= & \emptyset \mid +\,p; I^+ & \text{(Target Pattern)} \\
p & ::= & x = r \mid r & \text{(Statement Pattern)} \\
r & ::= & x.m(\bar{x}) \mid \mathtt{new}\ \mathsf{C}(\bar{x}) \mid x.f & \text{(Expression Pattern)}
\end{array}
$$

图 *2.2: PATL syntax, where metavariable x ranges over PATL metavariables and* C *ranges over MJ types.*

如上图，Patl语言是由一系列的转换规则构成的（图中的Π），每条规则(π)由两个部分组成，其中一部分是元变量的定义，另一部分是顺序语句模式的定义。其中的顺序模式定义又进一步分为了三个部分：1）上下文模式I^m，这部分是用来匹配一段代码上下文的，当这段模式被匹配上之后才能进行后面修改语句的匹配；2）第二部分是删除类语句I^-，删除类语句是在进行匹配之后将会进行删除的程序片段；3）第三部分是程序的添加模式，这个部分的模式是在进行替换阶段用来生成新的程序片段的模式：在进行程序替换时，我们将添加模式中的元变量替换为匹配绑定阶段得到的绑定的程序实际变量，从而让添加模式转化为实际程序中可以插入的代码部分，从而进行代码的生成。

在这些基本的语形要求之外，我们还对变换规则有一定的书写要求，从而保证规则在形式上不会产生不合法的行为，这些规则列举如下：

- 在上下文模式I^m，修改模式I^-以及添加模式I^+之中的语句模式都应该是SSA形式（静态单值模式）。这一条限制保证了一个元变量在匹配中绑定的语义对象都是固定的，而不会产生模式以及模式之间的变化，从而让用户书写的转换规则不会产生多义性。

  （相对的，我们也可以通过自动规则元变量的重命名来改变程序变换种非SSA形式的情况，但这样就会导致与用户意图之间的一个鸿沟，从而可能会在规则理解上产生一定程度的误解。）

  例如，当我们看以下的转换规则以及转换对象程序：

```
// Transformation program
(a:X->X, b:Y->Y) {
  - a.m();
  - a = b.n();
  + ...
}

// Target Program
x.m();
y = z.n();
```

在这一个例子中，我们左部的规则并非是SSA形式而右部的用户程序是正常的赋值语句。因此在绑定过程中，我们就会使得左部的元变量a绑定到不同的用户实际变量中（这里a会同时绑定到x，y，导致这两个变量并不成为别名关系），从而让用户理解的同一个变量规则产生了误解，让匹配时的绑定一致性收到了破坏。

- 在每一个变量中的删除语句 `-x=r`中，如果删除了一个元变量的使用，那我们就要求在生成的变量中必须有对应的添加语句来让删除的变量得到重生（要求在添加模式中存在语句`+x=r'`），从而避免变量被删除后的空引用问题。
  这条限制避免了在程序变化的时候我们因为匹配删除了一个变量的定义从而导致变量在程序转换之后的所有引用都得不到继续使用，引起上下文的错误依赖产生。
  给定以下的例子：

```
// Transformation Rule
(a:X->X, b:Y->Y) {
  - a = b.m();
}

// Target Program
X x;
x = y.m();
x.n();
```

如果不限制这样的规则，在上图的转换中就会导致`x=y.m();`这一条语句被删除，而由于没有对应的变量来填充上下文中被删除的x，从而导致下文的`x.n();`就失去了依赖，从而导致了程序变换过程中的依赖错乱问题。

- 映射规则中的一个类型只能映射到一个类型，以及子类之间的映射必须保持子类关系不变。这一条对于类型映射的要求就保证了我们前文提及的程序变换过程中类型正确性的问题。当我们在进行类型替换的时候，我们能够通过一一映射来保证类型映射的唯一性，并由子类关系保证变换过程中原有能赋值的部分仍然能够在新的环境中保持子类关系。

在拥有以上三条保证之后，通过检查的转换规则就已经满足了其静态的正确性，随后就将进入转换之中的动态执行与检查阶段。

第三章　表语义(Abstract Semantics)

在这一章中，我们将形式化的定义表语义，即前文所提的块内序列转转换语义，也即用户在书写转换规则过程中使用的转换语义。

我们对表语义的描述会在MJ程序p上进行，但是，由于这个转换会从程序p的语义上升格到对于p的程序体上的作用（也即p的内的语句序列），我们在接下来的描述就使用程序在语句序列上作用作为描述：转换过程是一个转换程序Π作用在一个方法M上，方法M中的变量都在变量环境Γ中。（在MJ中，由于变量定义都出现在程序的最上层，变量环境在程序的语句序列中并不会发生变化，所以我们都采用程序的变量上下文Γ来表示这个环境。）

在我们正式讲解表语义之前，我们先集中阐释一下我们所需要使用到的一些属于来避免在后文理解中的疑惑或误解。

- 位置：在程序中，我们为每一个语形单位都赋予了一个位置变量，来表示这个语形单位在程序中的位置，同时也用作识别程序的方式。在这里我们将所有的位置集合定义为L。我们使用$\mathsf{obj}(\iota)$来引用处在位置ι处的语形单位。
  例如，给定这样的一个语句（其中的位置用上标表示在每一个语形单位处）：$(b^{\iota_3} = (a^{\iota_1} : m())^{\iota_2})^{\iota_4}$，我们就可以使用$\mathsf{obj}(\iota_2)$来引用变量$a.m()$。
- 替换操作：我么使用符号 $p\backslash[s_1 \mapsto t_1, \ldots, s_n \mapsto t_n]$ 来表示对p中的变量替换。在这个替换中，如果s_i是一个位置，我们就用来表示把s_i位置处的语形单元替换为t_i。如果s_i代表的是一个语形单元，那这个替换就指定为在s_i所处的位置，将s_i替换为t_i。
- 路径：我们使用集合$\mathsf{Path}(p) \subseteq \mathsf{Loc}^*$ 来表示所有可能的流程p间可能的程序执行路径，其中这样的一条路径是一个序列的位置的集合。
- 类型：我们使用符号$\mathsf{type}(v)$来表示变量v在其方法内的类型，当v没有被定义时，$\mathsf{type}(v) = \bot$。正如我们前文所言，我们将此处的说明放在一个程序的方

法体内，因此类型不会产生变化，所以对于每一个变量的引用我们都可以正确的进行使用。

除了引用程序中的变量以外，我们同时还用这个符号来引用Patl中变量的源类型。例如：$x : \mathsf{C}_1 \hookrightarrow \mathsf{C}_2 \Rightarrow \mathsf{type}(x) = \mathsf{C}_1$。

在定义语句的匹配之前，我们首先定义语句与模板之间的匹配关系。

Definition 3 (语句匹配) *给定一条转换规则π, 以及一段程序语句序列$\bar{s}$, 一个程序与规则的匹配是一个三元组(p, σ, ι), 其中p是π中的一个语句模式, σ是一个映射, 将π中的元变量映射到$\bar{s}$中的程序变量, 并且ι是$\bar{s}$中的一个位置, 这些变量需要满足以下两个条件:*

1. *语形匹配. $p\backslash\sigma = s$.*
2. *类型匹配. 对于任何p中的变量x, $\mathsf{type}(\sigma(x))$应当是$\mathsf{type}(x)$的一个子类，如果x在p中作为右值出现, 并且当x在p中作为左值出现时，$\mathsf{type}(x)$应当是类型$\mathsf{type}(\sigma(x))$的子类。*

在有了语句的匹配之后，我们很自然的就可以把语句匹配推广到语句序列上的匹配操作:

Definition 4 (语句序列匹配) *一个三元组$(\overline{p}, \sigma, \bar{\iota})$在满足以下两个条件的情况下是一个语句序列的匹配:*

1. *$|\overline{p}| = |\bar{\iota}|$*
2. *(p_i, σ, ι_i)对于所有的下标$i < |\overline{p}|$都满足前文所说的语句匹配关系。*

以上的两条定义都是我们定义在程序的基本模式上的，接下来我们就可以更具这些模式上定义的匹配拓展到规则与用户程序之间的匹配关系，定义如下。

Definition 5 (规则匹配) *一个规则匹配是定义在一个语句序列$\bar{s}$以及一条转换规则π上的三元组$(\pi, \sigma, \bar{\iota})$, 其中$\pi$满足如下的形式$(\bar{x} : \overline{\mathsf{C}_1 \hookrightarrow \mathsf{C}_2})\{I^m; I^-; I^+;\}$。其中的$\sigma$是一个元变量的映射, 以及$\bar{\iota} = (\iota_i, \ldots, \iota_n)$是$\bar{s}$中语句的位置集合。以上这些要素需要满足以下这些关系:*

- *源匹配. $(I^-, \sigma, \bar{\iota})$形成一个语句序列的匹配关系。*
- *上下文匹配. 对于任意$\bar{\iota}_p \in \mathsf{Path}(p)$中一条可执行路径$\bar{\iota}$, 都存在一个子序列$\bar{\iota}_q$出现在$\bar{\iota}$之前, 使得$(I^m, \sigma, \bar{\iota})$在给定任意的元变量$x$ in I^m满足$\sigma(x)$在$\bar{\iota}$之前都没有被写操作覆盖的情况下形成一个语句序列匹配。*

- 独立. 对于任意两个元变量 x_1 与 x_2, 应当满足条件 $x_1 \neq x_2 \Rightarrow \sigma(x_1) \neq \sigma(x_2)$.
- 新变量. 对于任意只出现在 I^+ 并且不出现在 I^m *or* I^- 中的元变量 x, $\sigma(x)$ 是与程序中任意元变量都不同的变量。

上述定义的规则与程序之间的匹配关系可以根据如下的方式来进行理解：第一条规则定义了语句之间需要转换的程序部分。第二个条件要求了在需要匹配删除的程序语句之前需要有上下文能够与模板中的程序上下文相匹配，从而产生合理的上下文来确立转换位置，此外，由于我们要求被转换语句在任何执行情况下都满足这样的上下文匹配关系，我们就要求对于所有的执行路径都满足这个条件，从而让程序转换得以满足，因此我们在定义匹配时加上了对于所有路径的要求。第三个条件表示的则是不同的元变量都应该匹配到不同的转换对象，保证在转换中不会产生变量覆盖的情况。我们给出以下简单的例子来说明这个条件的实际效果：

```
1 //Transformation rule
2 (a:X->X, b:Y->Y) {
3   - a = A.m();
4   - b = A.n();
5   + b = A.n1();
6   + a = A.m1();
7 }
8 //Java program
9 i = A.m();
10 i = A.n();
```

在上面的这个例子中，用户希望的是把原有的 `A.n()` 用 `A.n1()` 替换，并将 `A.m();` 用 `A.m1();` 替换。但是如果没有第三个限制条件，产生匹配转换之后，由于Java程序之中的转换体都是对i的赋值，就会由于覆盖之后i的含义遭到破坏。因此我们增加这个限制就可以避免这个情况的发生：转换者意图转换的是不同的变量，而用户使用了相同的变量，这个情况当然就不应该进行匹配了。

此外，我们还应该保证在匹配进行之后，如果有一条语句匹配到了不同的语句上，我们就要保证这一条语句只匹配到一个删除模板，否则我们怎么理解一条语句被删除多次呢？因此尽管我们允许一条语句被多个环境模板匹配，我们只允许他最多匹配一个删除语句模板。

最后，有了这些单个规则的匹配，我们就可以定义匹配集合以及表语义的转换了。

Definition 6 (匹配集合) 一个建立在语句序列$\bar{s}$以及转换规则集合Π上的集合 S 被称为一个匹配集合如果他满足以下两个条件：

1. 对于任何S中的匹配$(\pi, \sigma, \bar{\iota}) \in S$，满足$\pi \in \Pi$
2. 对于任何 $\bar{s}$中的位置ι, 一个删除I^-中的模板p至多匹配到一个位置ι.

Definition 7 (表语义转换) 给定一段源程序语句序列 $\bar{s}$ 以及一个目标程序语句许略 $\bar{s}'$，我们定义目标序列为源序列在规则Π下的一个转换（我们标记为$\Pi^\circ(\bar{s}, \bar{s}')$），当且仅当存在转换程序$e$和转换规则集合$\Pi$之上的一个匹配集合$S$ 满足关系 $\bar{s}' = \bar{s} \backslash \tau$，并且其中 τ 是满足一下关系的最小集合：

$$\frac{(\pi, \sigma, \bar{\iota}) \in S \quad \pi = (\ldots)\{I^m; I^-; I^+;\}}{[\bar{\iota} \mapsto I^+ \backslash \sigma] \in \tau}$$

$$\frac{(x_1 : \mathsf{C}_1 \hookrightarrow \mathsf{C}'_1, \ldots, x_n : \mathsf{C}_n \hookrightarrow \mathsf{C}'_n)\{I\} \in \Pi}{[\mathsf{C}_1 \mapsto \mathsf{C}'_1] \in \tau, \ldots, [\mathsf{C}_n \mapsto \mathsf{C}'_n] \in \tau}$$

至此，我们就完成了表语义的转换定义：表语以转换就是定义在转换语言上的同义语形匹配与替换，其中的替换操作源于对添加模板之中的元变量替换产生。

此处的程序变换仅涉及匹配与替换，并不涉及程序结构的移动与修改，我们就在接下来的章节中进行算法语义的描述。

第四章　里语义(Algorithmic Semantics)

现在我们将正式的进行里语义的形式化定义，正如我们在第二章的描述，里语义是表语以在控制流结构下增加了对程序结构修改的语义。

同样的我们在介绍里语义之前先将里语义中所采用的符号列举如下，来避免后面阅读时的困难。

- MJ 类型环境 Γ.
- Patl 类型环境 Ψ, 其中的元素形如 $x : \mathsf{C}_1 \hookrightarrow \mathsf{C}_2$，表示了元变量$x$ 在源环境中是类型 C_1 而在新环境中类型为 C_2。
- 别名关系集合Ω, 其描述了程序之中的所有可能存在的别名关系。里面的元素包含以下两种类型： 1) *must aliases*关系 $[v_1^{\iota_1}, v_2^{\iota_2}] \in \Omega$ 以及 2) *may alias* $\langle v_1^{\iota_1}, v_2^{\iota_2} \rangle \in \Omega$.
- 依赖关系集合 Σ, 这个集合中同时包含了变量之间的依赖关系以及语句之间的依赖关系: $(v_1^{\iota_1}, v_2^{\iota_2})$ 表示了变量 $v_1^{\iota_1}$ 依赖于变量 $v_2^{\iota_2}$, 而 (l_1, l_2) 表示了具有标号 l_1 的语句依赖于具有标号 l_2的语句.
- 转换法则 Θ集合。每一个出现在 Θ 中的转换元素包含了一个匹配者 $\hat{\pi}$, 以及一列转换语句的标签 $l_1, ..., l_n$, 标明了标记为 $l_1, ..., l_n$ 的语句将会被移动转换, 由于他们被匹配者 $\hat{\pi}$所标记。(更多关于这个集合的东西将会在使用的位置进行阐述。)

接下来，我们就开始进行里语义算法流程的介绍。

4.1　里语义算法流程

在接下来的介绍中，我们采用类似于表语义中的介绍方式，介绍里语义作用在一个方法体内的规则（转换规则会将具体过程推进到方法体内，因此其等价于

对于Java程序的介绍）。

此外，我们的里语义对于上下文的依赖也定义在了一个方法之内，由于方法之间的转换存在不一致性，（而且根据API的使用规则，具有时序关系调用的API集合也会出现在一个方法内，否则在升级过程中不会产生合并替换这样的操作），因此任何时候我们检查到API之间的调用超越了方法边界时，我们都会产生一个警告给用户，来提示已经发现的错误。

正如前文所述，我们的转换过程可以分为以下几个步骤：

1. 分析：我们在第一阶段，会对即将转换的方法快进行程序分析，来获取其中的以来关系以及别名关系，从而在具体转换实现时进行辅助来保证依赖关系的正确保持。
2. 匹配：其中的第二步转换为匹配操作。在这一个步骤中Patl程序Π会与用户语句序列产生绑定，将其中的语句与模板进行绑定，同时产生元变量与变量之间的绑定关系。这一步匹配的结果就是会产生一个匹配者，用来指导后文的转换工作。
3. 移动：这一步又可以称作基于匹配者的移动操作，这个移动操作是用来帮助改变程序的块结构从而产生顺序修改条件程序的必要步骤。移动操作中我们需要使用匹配者（又称绑定者）来描述移动之间的关系，同时采用依赖集合来进行移动的指导。这一步操作是一个语义等价操作，可以用来保证变换前后的程序完全语义等价。
4. 修改：最后一步的修改操作则是类似于表语义之中的修改替换操作。这一步修改与替换操作进行的即是API的替换操作。这一步修改操作会读取匹配者之间的绑定信息，通过实例化操作将元变量实例化，从而产生新的API对应关系。

在进行了上述的这些转换步骤之后我们就能够成功的在语句层次上将程序变换为具有新API调用的程序了。如果在转换过程中出现了警告，我们就会将由于用户用户程序书写不当造成的错误报告给用户，让用户经过修改之后再次进行变换。

$$\boxed{\Pi(M) = M'}$$

$$\frac{\mathsf{depAnalysis}(\bar{s}) = \Sigma_d \quad \mathsf{aliasAnalysis}(\bar{s}) = \Omega \quad \{\bar{x} : \bar{\mathsf{C}}\} \vdash_{\Omega} \mathsf{Match}(\Pi, s) = \hat{\Pi} \quad \mathsf{MatchedSeqs}(\hat{\Pi}) = \Theta \quad \vdash^{\Theta}_{\Sigma_d \cup \mathsf{matchDep}(\Theta)} \bar{s} \rightsquigarrow_{shift} \bar{s}' \quad \mathsf{adapt}(\hat{\Pi}, \bar{s}') = \bar{s}''}{\Pi(\mathsf{C}_1\ m(\bar{\mathsf{C}}\ \bar{x})\{\bar{s}; \texttt{return}\ z;\}) = \Pi(\mathsf{C}_1)\ m(\overline{\Pi(\mathsf{C})}\ \bar{x})\{\bar{s}''; \texttt{return}\ z;\}}$$

上述的转换流程可以概述为如上图所示。

类型的转换 在语句层次的变化之外，我们还需要进行类型上的变化来保证转换的完整性。类型层次上的操作是从变换规则中提取类型映射条件，用来添加到类型的映射之中，其间Patl系统会分析类型之间的函数映射条件以及子类关系是否满足，用以解决用户程序之间的实际类型问题。

在接下来的几个章节中，我们将分步骤的介绍Patl的转换流程，通过形式化的方式来描述转换过程。

4.2 分析

我们首先进行的介绍是程序的分析，给定一个用户程序p用来作为转换源时，我们首先通过分析来给出一个对于程序之中依赖以及别名关系的分析结果用在后面的匹配操作中。

别名分析 我们首先会通过一个函数$\mathsf{aliasAnalysis}(\bar{s}) = \Omega$来进行别名的分析，在这个分析中我们要求分析记过是完整的：给定了两个变量以及他们的位置，我们要求 1) $[v_1^{\iota_1}, v_2^{\iota_2}] \in \Omega \Rightarrow v_1^{\iota_1}$ 可以推导出 $v_2^{\iota_2}$ 他们是必然别名关系, 2) $\langle v_1^{\iota_1}, v_2^{\iota_2} \rangle \in \Omega \Rightarrow v_1^{\iota_1}$ 与 $v_2^{\iota_2}$ 是可能别名关系, 以及 3) 他们之间没有别名关系。

在这样的分析下，我们才能够保证在程序匹配中每一次查询都能查询到合理的对应结果用来指导程序的匹配操作。

依赖关系分析 同样的，我们要求语句之间的依赖关系也具有完整性。保证了分析的结果是可确定的：1) 给定两个语句 s_1 标签为 l_1 与 s_2 标签为 l_2, 我们要求 $(l_1, l_2) \in \Sigma_d$, s_1 与 s_2 也许会有依赖关系，否则他们 *必然没有* 依赖关系。 2) 给定两个变量 $v_1^{\iota_1}$ 以及 $v_2^{\iota_2}$, $(v_1^{\iota_1}, v_2^{\iota_2}) \in \Sigma_d$ 意味着 $v_1^{\iota_1}$ 也许依赖于 $v_2^{\iota_2}$，否则$v_1^{\iota_1}$ *一定不* 依赖于 $v_2^{\iota_2}$。

同时应当注意的是，我们在程序中使用的依赖关系分析是更加保守的依赖关系分析，我们要求所有无法确定的程序分析部分都被标注为具有依赖关系，从而保证依赖关系的不完整不会损害转换与匹配的过程。

4.3 匹配

有了程序分析的结果之后，我们就可以开始使用数据流分析的匹配框架来进行程序中的控制流图上的匹配操作。匹配操作的目标是要将所有可能存在执行路径上的转换对象寻找出来，用于等价变换之后进行查找和替换。给定了一个Patl程序Π以及一列语句序列$\bar{s}$，我们希望找到$\bar{s}$中的一列子语句序列，是的这个子语句序列与Π中的一些规则匹配绑定，并且要求这些子语句序列必定存在与一个执行序列之中。

4.3.1 匹配者

在匹配过程中，我们采用了“匹配者”结构来满足语句之间匹配的需求（定义如下图所示图.4.1），匹配者结构是用来存储绑定匹配信息结构，在匹配个过程中随着匹配点的移动而更新，不断的增加绑定直至产生一组完整的匹配绑定序列。

$$
\begin{array}{lcll}
\hat{\Pi} & ::= & \{\hat{\pi}_1, ..., \hat{\pi}_n\} & \text{(Matcher set)} \\
\hat{\pi} & ::= & [\pi](\bar{x} : \bar{S})\{\bar{p} : \bar{l}\} & \text{(Matcher)} \\
S & ::= & \{v_1^{\iota_1}, ..., v_n^{\iota_n}\} & \text{(Bound Value Set)}
\end{array}
$$

图 *4.1: 匹配者的定义, 其中的 x 表示 PATL 元变量, p 表示语句模式, l 表示语句标签， v^{ι} 用来表示变量在位置ι位置的实例*

此处的匹配者在定义中表示了元变量与变量实例集合的绑定关系，匹配者体中包含了语句与模式之间的绑定关系。

4.3.2 数据流上的程序匹配操作

由于我们的程序匹配是与程序的实际执行有一定的对应关系，我们就通过数据流上的匹配来减少实际执行与源代码之间的鸿沟。这个数据流分析的框架如下

$$
\begin{aligned}
&\mathit{init} = \{[\pi_1]()\{\}, ..., [\pi_n]()\{\}\} \\
&\mathit{out}[s] = \bigcup_{\hat{\pi} \in \mathit{in}[s]} \mathsf{match}(\hat{\pi}, s) \\
&\mathit{in}[s] = \bigcup_{s' \in \mathit{pred}(s)} \mathit{out}[s'] \\
&\mathit{exit} = \mathsf{clear}(\mathit{in}[s])
\end{aligned}
$$

图 *4.2:* 匹配环节的数据流分析框架

图所示。

在这一个数据流的分析框架中，我们在初始点首先创建了一个空的匹配者集合（使用转换规则进行创建），接下来就在遍历每一个转换节点上进行更新操作，想所有可能的匹配者添加可能的绑定语句，从而进行匹配者的更新，用来满足匹配者完全匹配的要求。

转移函数 在数据流分析框架中，最重要的转移函数部分就进行了匹配添加的过程，一方面会对所有的匹配者进行尝试更新操作，我们会对所有产生绑定的匹配者进行更新，并将其加入到下一个匹配者集合中，用在下一个节点处进行更新。

值得注意的是我们为了避免回溯匹配的需要，我们采用的是类似于DFA的扩展匹配方式，每一步更新都会同时保留更新前后的状态。但是由于每一个匹配者的语句数量都是小的常数，这个变化就不会产生过高的复杂度。

我们就将转移函数的形式化定义放置如下。

$$\boxed{\Gamma \vdash_\Omega \mathsf{match}(\hat{\pi}, s) = \hat{\Pi}}$$

$$\frac{\mathsf{matchpoint}(\hat{\pi}) = p_1 \quad \exists \hat{\pi}' = [\pi](\bar{x}' : \bar{S}')(\bar{p}' : \bar{l}') \in \mathit{in}[s].(p_1 : \mathsf{label}(s) \in \bar{p}' : \bar{l}')}{\Gamma \vdash_\Omega \mathsf{match}(\hat{\pi} :: [\pi](\bar{x} : \bar{S})\{\bar{\beta}\}, s) = \left\{[\pi](\bar{x} : \bar{S})\{\bar{\beta}\}\right\}}$$

$$\frac{\begin{array}{c}\mathsf{matchpoint}(\hat{\pi}) = p_1 \quad \mathsf{decl}(\pi); \Gamma \vdash \mathsf{varmatch}(p_1, s) = \bar{x}' : \overline{v''} \\ \forall\, x_1 : v_1^{\iota_1} \in \bar{x}' : \overline{v''}.(\exists x_1 : S_1 \in \bar{x} : \bar{S}.(S_1 = \emptyset \vee \forall v_2^{\iota_2} \in S.([v_1^{\iota_1}, v_2^{\iota_2}] \in \Omega))) \\ \mathsf{insert}(\bar{x}' : \overline{v''}, \bar{x} : \bar{S}) = \bar{x} : \bar{S}'\end{array}}{\Gamma \vdash_\Omega \mathsf{match}(\hat{\pi} :: [\pi](\bar{x} : \bar{S})\{\bar{\beta}\}, s) = \left\{[\pi](\bar{x} : \bar{S}')\{\bar{\beta},\ p_1 : \mathsf{label}(s)\}, [\pi](\bar{x} : \bar{S})\{\bar{\beta}\}\right\}}$$

$$\frac{\begin{array}{c}\mathsf{matchpoint}(\hat{\pi}) = p_1 \quad \mathsf{decl}(\pi); \Gamma \vdash \mathsf{varmatch}(p_1, s) = \bar{x}' : \overline{v''} \\ \forall\, x_1 : v_1^{\iota_1} \in \bar{x}' : \overline{v''}.(\exists x_1 : S_1 \in \bar{x} : \bar{S}.(S_1 = \emptyset \vee \forall v_2^{\iota_2} \in S.([v_1^{\iota_1}, v_2^{\iota_2}] \vee \langle v_1^{\iota_1}, v_2^{\iota_2}\rangle \in \Omega))) \\ U = \{(x_1, v_1^{\iota_1}) \mid x_1 : v_1^{\iota_1} \in \bar{x}' : \overline{v''} \wedge \exists\, x_1 : S_1 \in \bar{x} : \bar{S}.(\exists v_2^{\iota_2} \in S_1.(\langle v_1^{\iota_1}, v_2^{\iota_2}\rangle \in \Omega))\} \neq \emptyset \\ \mathsf{insert}(\bar{x}' : \overline{v''}, \bar{x} : \bar{S}) = \bar{x} : \bar{S}'\end{array}}{\Gamma \vdash_\Omega \mathsf{match}(\hat{\pi} :: [\pi](\bar{x} : \bar{S})\{\bar{\beta}\}, s) = \left\{[\pi](\bar{x} : \bar{S}')\{\bar{\beta},\ *p_1 : \mathsf{label}(s)\ [U]\}, [\pi](\bar{x} : \bar{S})\{\bar{\beta}\}\right\}}$$

$$\frac{\text{otherwise}}{\Gamma \vdash_\Omega \mathsf{match}(\hat{\pi} :: [\pi](\bar{x} : \bar{S})\{\bar{\beta}\}, s) = \left\{[\pi](\bar{x} : \bar{S})\{\bar{\beta}\}\right\}}$$

$\boxed{\Psi; \Gamma \vdash \mathsf{varmatch}(p, s) = \{\bar{x} : \bar{v}\}}$

$$\frac{\Psi; \Gamma \vdash \mathsf{lhsvarmatch}(x; z) = \{x : z\} \quad \Psi; \Gamma \vdash \mathsf{varmatch}(r; e) = \{\bar{y} : \bar{v}\}}{\Psi; \Gamma \vdash \mathsf{varmatch}(x = r; z = e) = \{x : z, \bar{y} : \bar{v}\}}$$

$$\frac{\Psi \vdash x : \mathsf{C}_1 \hookrightarrow \mathsf{C}_1' \quad \Gamma \vdash x_1 : \mathsf{C}_2 \quad \mathsf{C}_2 <: \mathsf{C}_1}{\Psi; \Gamma \vdash \mathsf{varmatch}(x, x_1) = \{x : x_1\}}$$

$$\frac{\Psi \vdash \bar{x} : \overline{\mathsf{C}_1 \hookrightarrow \mathsf{C}_1'} \quad \Gamma \vdash \bar{x}_1 : \bar{\mathsf{C}}_2 \quad \bar{\mathsf{C}}_2 <: \bar{\mathsf{C}}_1}{\Psi; \Gamma \vdash \mathsf{varmatch}(\mathbf{new}\ \mathsf{C}(\bar{x}), \mathbf{new}\ \mathsf{C}(\bar{x}_1)) = \{\bar{x} : \bar{x}_1\}}$$

$$\frac{\begin{array}{c}\Psi \vdash x : \mathsf{C}_1 \hookrightarrow \mathsf{C}_1',\ \bar{y} : \overline{\mathsf{C}_2 \hookrightarrow \mathsf{C}_2'} \\ \Gamma \vdash x_1 : \mathsf{C}_3, \bar{y}_1 : \bar{\mathsf{C}}_4 \quad \mathsf{C}_3 <: \mathsf{C}_1, \bar{\mathsf{C}}_4 <: \bar{\mathsf{C}}_2\end{array}}{\Psi; \Gamma \vdash \mathsf{varmatch}(x.m(\bar{y}), x_1.m(\bar{y}_1)) = \{x : x_1, \bar{y} : \bar{y}_1\}}$$

$$\frac{\Psi \vdash x : \mathsf{C}_1 \hookrightarrow \mathsf{C}_1' \quad \Gamma \vdash x_1 : \mathsf{C}_2 \quad \mathsf{C}_1 <: \mathsf{C}_2}{\Psi; \Gamma \vdash \mathsf{lhsvarmatch}(x, x_1) = \{\bar{x} : \bar{x}_1\}}$$

$\boxed{\mathsf{matchpoint}(\hat{\pi})}$

$$\frac{\begin{array}{c}\pi = (\bar{x} : \overline{\mathsf{C} \hookrightarrow \mathsf{C}})\{\ldots \delta\ p_1; \delta\ p_2; \ldots\} \quad \delta = \mathbf{m} \mid -\\ \exists\, l_1.(p_1 : l_1 \in \{\bar{p} : \bar{l}\}) \wedge \neg \exists\, l_2.(p_2 : l_2 \in \{\bar{p} : \bar{l}\})\end{array}}{\mathsf{matchpoint}([\pi](\bar{x} : \bar{v})\{\bar{p} : \bar{l}\}) = p_2}$$

$$\frac{\pi = (\bar{x} : \overline{\mathsf{C} \hookrightarrow \mathsf{C}})\{\ldots \delta\ p, I^+\} \quad \delta = \mathbf{m} \mid - \quad \exists\, l_1.(p_1 : l_1 \in \{\bar{p} : \bar{l}\})}{\mathsf{matchpoint}([\pi](\bar{x} : \bar{v})\{\bar{p} : \bar{l}\}) = \epsilon}$$

上述的这些形式化定义包括了对于语形类型的匹配以及对于变量之间绑定对象的匹配，总体而言，这个匹配过程可以分为：

1. 语形以及类型匹配：这个阶段采用静态的语形与类型匹配，只要能满足语形一致，类型满足正确的子类关系即可。
2. 对象绑定匹配：在确定类型语形之后，还需要进行对象的绑定检查，这一步就需要前文涉及到的别名分析，将会检查是否同一变量匹配到的语形对象都互为别名关系，如是才能进行匹配。

随着数据流分析的进行，匹配者会得到逐步的更新，在出口节点得到一个匹配者集合。

出口操作 在程序的出口完成匹配操作之后，由于并非所有的匹配都是合法匹配，我们就需要通过出口处的一个分析清理操作来完成不合法匹配者的清除，规则如下。

$\boxed{\hat{\Pi} \rightsquigarrow_{clean} \hat{\Pi}'}$

$$\frac{\hat{\Pi}' = \hat{\Pi} - \{\hat{\pi} \ : \ \hat{\pi} \in \hat{\Pi} \wedge \mathsf{matchpoint}(\hat{\pi}) \neq \epsilon\} \quad \forall \hat{\pi} \in \hat{\Pi}'.(\mathsf{check}(\hat{\pi}) \rightsquigarrow ok)}{\hat{\Pi} \rightsquigarrow_{clean} \hat{\Pi}'}$$

$\boxed{\mathsf{check}(\hat{\pi}) \rightsquigarrow}$

$$\frac{V = \{ * p : l\ [U] \mid * p : l\ [U] \in \bar{\beta}\} \neq \emptyset}{\mathsf{check}([\pi](\ \bar{x} : \bar{S}\ )\{\bar{\beta}\}) \rightsquigarrow \textit{uncertain alias warning } V}$$

$$\frac{\exists\ p_1 : l_1, p_2 : l_2 \in \{\bar{p} : \bar{l}\}.(\mathsf{crossloop}(l_1, l_2))}{\mathsf{check}([\pi](\ \bar{x} : \bar{S}\ )\{\bar{\beta}\}) \rightsquigarrow \textit{cross loop boundary warning}}$$

$$\begin{aligned}\mathsf{crossloop}(l_1, l_2) = \quad & \exists\ \bar{s}_1\ \mathtt{while}(x)\{\bar{s}_2\}\ \bar{s}_3.(\\ & (\mathsf{stmt}(l_1) \in \bar{s}_1 \wedge \mathsf{stmt}(l_2) \in \bar{s}_2) \vee (\mathsf{stmt}(l_1) \in \bar{s}_2 \wedge \mathsf{stmt}(l_2) \in \bar{s}_3))\end{aligned}$$

$$\frac{\exists\ p_1 : l_1, p_2 : l_2 \in \{\bar{p} : \bar{l}\}.(\exists\ M_1, M_2.(\mathsf{stmt}(l_1) \in M_1 \wedge \mathsf{stmt}(l_2) \in M_2))}{\mathsf{check}([\pi](\ \bar{x} : \bar{S}\ )\{\bar{\beta}\}) \rightsquigarrow \textit{cross procedure warning}}$$

$$\frac{\neg\forall\ p_1 : l_1, p_2 : l_2 \in \{\bar{p} : \bar{l}\}.(p_1 \prec p_2 \Rightarrow l_1 \prec l_2)}{\mathsf{check}([\pi](\ \bar{x} : \bar{S}\ )\{\ \bar{p} : \bar{l}\}) \rightsquigarrow \textit{statement revisit warning}}$$

$$\frac{\text{otherwise}}{\mathsf{check}(\hat{\pi}) \rightsquigarrow ok}$$

这个清除操作需要去掉以下不合法情形：

1. 不合法的跨越了程序边界：在转换过程中，由于不同方法以及循环边界的跨越会导致转换的错误产生，我么就会通过边境检查筛除。
2. 未完成的匹配：匹配过程中即使到了终点也无法进行完全匹配的匹配者将会被标记为未完成。
3. 不确定别名关系：一个元变量绑定的变量存在无法确定的别名关系，这种情况会要求用户进行手动确定来完成修改。

进行了清除之后，剩余的匹配者就是可以用于指导程序等价变换的匹配者了，在下一阶段，就进入了匹配者指导下的程序变换。

4.4 匹配者指导下的程序移动

匹配者指导下的程序移动就表示为程序结构上的重构：通过将程序之间的程序等价移动来最大化程序变换中的匹配转换操作效益。由于前文已经对这个过程进行了较多的描述，在此我们就通过算法来阐述流程的构建。

首先我们介绍移动流程中的符号，如下图所示。

$$
\begin{array}{rcll}
\Theta & ::= & \emptyset \mid \Theta, \langle \hat{\pi}; l, ..., l \rangle & \text{(Shifting Rules)} \\
\mathsf{chunk}^{\hat{\pi}}[l] & ::= & \mathsf{stmt}^{\hat{\pi}}(l) & \text{(Statement Chunk Structure)} \\
& \mid & \texttt{if}(x)\{\bar{s}_1\ \mathsf{chunk}^{\hat{\pi}}[l]\ \bar{s}_2\}\texttt{else}\{\bar{s}_3\} & \\
& \mid & \texttt{if}(x)\{\bar{s}_3\}\texttt{else}\{\bar{s}_1\ \mathsf{chunk}^{\hat{\pi}}[l]\ \bar{s}_2\} &
\end{array}
$$

上图中的Θ用来表示移动的法则，这个法则标注了需要转换的语句对象及其转换的匹配者，其中的标记l是用来绑定语句的，这个标记会随着语句的移动而移动，不会改变。另外，chunk结构用来表示一个块状的语法结构，用来在移动过程中进行移动的统一整理。

Input: Statement sequence $\bar{s}$, Shifting set Θ, Dependency relation set Σ

Output: Shifted statement sequence $\bar{s}'$

begin

  $\Theta^{\#} \leftarrow \emptyset$;

  for *each shifting relation* $\theta = \langle \hat{\pi}; l_1, ..., l_n \rangle \in \Theta$ **do**

    for *each adjacent labels* l_k, l_{k+1} **do**

      /* Perform the shift rule with the given θ */

      $\bar{s} \leftarrow \mathsf{shift}(\Theta, \Sigma, \Theta^{\#}, \langle \hat{\pi}; l_k, l_{k+1} \rangle, \bar{s})$;

      $\Theta^{\#} \leftarrow \Theta^{\#} \cup \{\langle \hat{\pi}; l_1, l_2 \rangle\}$;

    end

  end

end

Algorithm 1: Shifting algorithm

在移动算法中，我们会根据语句来进行移动操作，移动直到不存在转换者继续能够作用为止。移动算法的主体shift函数将在下文通过转换规则进行表述。

$$\frac{\vdash_{\Sigma}^{\Theta} \mathsf{upshift}(\langle\hat{\pi}; l_1, l_2\rangle, \bar{s}) = \bar{s}'}{\mathsf{shift}(\Theta, \Sigma, \Theta^{\#}, \langle\hat{\pi}; l_1, l_2\rangle, \bar{s}) = \bar{s}'} \text{ (Shift)}$$

$$\frac{\begin{array}{c}\mathsf{chunk}^{\hat{\pi}}[l_2] = \texttt{if}(x)\{\bar{s}_{11}\}\texttt{else}\{\bar{s}_{21}\} \quad \exists\, \mathsf{chunk}^{\hat{\pi}}[l_1] \in \bar{s}_{11} \vee \exists\, \mathsf{chunk}^{\hat{\pi}}[l_1] \in \bar{s}_{21} \\ \vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{upshift}(\langle\hat{\pi}; l_1, l_2\rangle, \bar{s}_{11}) = \bar{s}'_{11} \quad \vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{upshift}(\langle\hat{\pi}; l_1, l_2\rangle, \bar{s}_{21}) = \bar{s}'_{21}\end{array}}{\vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{upshift}(\langle\hat{\pi}; l_1, l_2\rangle, \bar{s}_1\ \mathsf{chunk}^{\hat{\pi}}[l_2]\ \bar{s}_2) = \bar{s}_1\ \texttt{if}(x)\{\bar{s}'_{11}\}\texttt{else}\{\bar{s}'_{21}\}\ \bar{s}_2} \text{ (Upshift-0)}$$

$$\frac{\vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{downshift}(\langle\hat{\pi}; l_1, l_2\rangle, \mathsf{stmt}^{\hat{\pi}}(l_1)\ \bar{s}_2) = \bar{s}'}{\vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{upshift}(\langle\hat{\pi}; l_1, l_2\rangle, \bar{s}_1\ \mathsf{stmt}^{\hat{\pi}}(l_1)\ \bar{s}_2)\ =\ \bar{s}_1\ \bar{s}'} \text{ (Upshift-1)}$$

$$\frac{\begin{array}{c}\exists\, \mathsf{chunk}^{\hat{\pi}}[l_1] \in \bar{s}_1 \vee \mathsf{chunk}^{\hat{\pi}}[l_1] \in \bar{s}_2 \quad \vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{closure}^{\uparrow}(\mathsf{chunk}^{\hat{\pi}}[l_2], \bar{s}_3) = \bar{s}' \\ \mathsf{upshift}(\langle\hat{\pi}; l_1, l_2\rangle, \bar{s}_1\ \bar{s}'\ \mathsf{chunk}^{\hat{\pi}}[l_2]) = \bar{s}''_1 \quad \mathsf{upshift}(\langle\hat{\pi}; l_1, l_2\rangle, \bar{s}''_2\ \bar{s}'\ \mathsf{chunk}^{\hat{\pi}}[l_2]'') = \bar{s}''_2 \\ \exists\theta = \langle\hat{\pi}_x; l_u, l_v\rangle \in \Theta^{\#}.(\exists s_u \in \bar{s}' \wedge s_u = \mathsf{stmt}^{\hat{\pi}_x}(l_u)) \\ \mathsf{upshift}(\theta,\ \bar{s}_0\ \texttt{if}(x)\{\bar{s}''_1\}\texttt{else}\{\bar{s}''_2\}\ \bar{s}_3\backslash\bar{s}'\ \bar{s}_4) = \bar{s}'''\end{array}}{\begin{array}{c}\vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{upshift}(\langle\hat{\pi}; l_1, l_2\rangle, \bar{s}_0\ \texttt{if}(x)\{\bar{s}_1\}\texttt{else}\{\bar{s}_2\}\ \bar{s}_3\ \mathsf{chunk}^{\hat{\pi}}[l_2]\ \bar{s}_4) \\ = \bar{s}'''\end{array}} \text{ (Upshift-2)}$$

$$\frac{\neg\exists\, \mathsf{chunk}^{\hat{\pi}}[l_1] \in \bar{s}_1}{\vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{upshift}(\langle\hat{\pi}; l_1, l_2\rangle, \bar{s}_1\ \mathsf{chunk}^{\hat{\pi}}[l_2]\ \bar{s}_3)\ =\ \bar{s}_1\ \mathsf{chunk}^{\uparrow\hat{\pi}}[l_2]\ \bar{s}_3} \text{ (Upshift-3)}$$

$$\frac{}{\begin{array}{c}\vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{downshift}(\langle\hat{\pi}; l_1, l_2\rangle, \mathsf{stmt}(l_1)^{\hat{\pi}}\ \bar{s}_1\ \mathsf{stmt}^{\hat{\pi}}(l_2)\ \bar{s}_2) \\ =\ \mathsf{stmt}(l_1)^{\hat{\pi}}\ \bar{s}_1\ \mathsf{stmt}^{\hat{\pi}}(l_2)\ \bar{s}_2\end{array}} \text{ (Downshift-1)}$$

$$\frac{\begin{array}{c}\exists \mathsf{chunk}^{\hat{\pi}}[l_2] \in \bar{s}_2 \vee \mathsf{chunk}^{\hat{\pi}}[l_2] \in \bar{s}_3 \quad \vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{closure}^{\uparrow}(\mathsf{stmt}^{\hat{\pi}}(l_1), \bar{s}_1) = \bar{s}' \quad \mathsf{stmt}^{\hat{\pi}}(l_1) \not\leftarrow_{dep} x \\ \vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{downshift}(\langle\hat{\pi}; l_1, l_2\rangle, \mathsf{stmt}^{\hat{\pi}}(l_1)\ \bar{s}'\ \bar{s}_2) = \bar{s}''_2 \quad \vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{downshift}(\langle\hat{\pi}; l_1, l_2\rangle, \mathsf{stmt}^{\hat{\pi}}(l_1)\ \bar{s}'\ \bar{s}_3) = \bar{s}''_3 \\ \exists\theta = \langle\hat{\pi}_x; l_u, l_v\rangle \in \Theta^{\#}.(\exists s_v \in \bar{s}' \wedge s_v = \mathsf{stmt}^{\hat{\pi}_x}(l_v)) \quad \mathsf{downshift}(\theta,\ \bar{s}_0\ \bar{s}_1\backslash\bar{s}'\ \texttt{if}(x)\{\bar{s}''_2\}\texttt{else}\{\bar{s}''_3\}) = \bar{s}'''\end{array}}{\vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{downshift}(\langle\hat{\pi}; l_1, l_2\rangle, \bar{s}_0\ \mathsf{stmt}^{\hat{\pi}}(l_1)\ \bar{s}_1\ \texttt{if}(x)\{\bar{s}_2\}\texttt{else}\{\bar{s}_3\}\ \bar{s}_4)\ =\ \bar{s}'''\ \bar{s}_4} \text{ (Downshift-2)}$$

$$\frac{\text{otherwise}}{\begin{array}{c}\vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{downshift}(\langle\hat{\pi}; l_1, l_2\rangle, \bar{s}_0\ \mathsf{stmt}^{\hat{\pi}}(l_1)\ \bar{s}_1\ \texttt{if}(x)\{\bar{s}_2\}\texttt{else}\{\bar{s}_3\}\ \bar{s}_4) \\ \rightsquigarrow \textit{Cannot shift due to dependency of } x.\end{array}} \text{ (Downshift-2-Warning)}$$

$$\frac{\neg\exists\, \mathsf{chunk}^{\hat{\pi}}[l_2] \in \bar{s}_2}{\vdash_{\Sigma}^{\Theta;\Theta^{\#}} \mathsf{downshift}(\langle\hat{\pi}; l_1, l_2\rangle, \mathsf{stmt}^{\hat{\pi}}(l_1)\ \bar{s}_2)\ =\ \mathsf{chunk}^{\downarrow\hat{\pi}}[l_1]\ \bar{s}_2)} \text{ (Downshift-3)}$$

Post shifting check:

$$\frac{\begin{array}{c}\neg\exists s \in \bar{s}.(\exists \pi_1, \pi_2 \in \hat{\Pi}.(\exists p_1 : l \in \pi_1, p_2 : l \in \pi_2.(\\ \mathsf{label}(s) = l \wedge p_1 \textit{ and } p_2 \textit{ are deletion pattern})))\end{array}}{\textit{Interleaving source pattern matching warning}}$$

上述的转换规则就完成了一次调整的操作，而在算法框架中通过对多部分算法的调整我们就可以通过具体的调整来完成对于匹配者绑定对象的匹配。

这些移动规则的主要思路列举如下：在移动过程中，我们首先通过上移操作来使得要求上移的语句进入到分支之中（当需要上移的语句包含在一个块中时，我们会将这个块作为一个整体进行移动）。在上移完成后，我们就会出发下移操作将前文的匹配语句下移到语句块中。（在上下移动过程中，我们会采用连锁移动的方式来指导移动操作，从而保证前文引发的移动不会被现有移动破坏）。

进行移动完成后的代码就具有了所有绑定过的代码都出现在同一个语句块中的特性，这就让我们在接下来的修改阶段能够进行块内替换达到API迁移的目的。接下来我们就介绍修改操作。

4.5 代码修改

移动之后的代码转换就是在基本块中进行代码删除与代码生成操作。在代码删除阶段，我们会将源代码中匹配到匹配者中删除模板中的部分进行删除，同时在删除的块中寻找适当的位置将匹配者中的生成语句实例化，添加到源代码中。

在这一步操作中，我们主要考虑的是如下两个问题：1）代码应当生成在什么位置，2）添加模板中的元变量该由哪些程序变量替换得到。

对于第一个问题，我们采用了一个对于可交换位置的计算来生成代码的方式：程序中的可交换位置即是代码生成之中能够将所有匹配语句通过相邻交换操作进行归并的位置。而对于第二个问题，我们采用的方式即是通过找到元变量绑定变量的别名位置来寻找变量，另一种方式则是原有变量被删除，我们就会为该变量生成一个对应的名字来生成在该位置，从而满足程序的变量生成要求。

具体而言，代码生成算可以分为以下步骤：1）寻找可交换位置，2）使用元变量集合进行变量替换，3）通过回填的方式来进行变量实体化。

在主算法中，我们会穷举所有可以转换的代码生成点来判断哪些位置可以进行代码生成，在找到了一个合理的代码生成位置之后，我们并不立刻进行删除，而是将所有即将进行删除的代码保存起来，搜集完毕在进行整理操作。

在搜集完毕之后，会将所有标记为删除的代码进行删除，同时在删除的位置插入使用记号标记的变量位置，最后再通过回填的方式将代码中的变量进行插入。

值得注意的是，在代码生成过程中，一个变量在源代码中被删除以后，可能

Input: Statement sequence $\bar{s}$, corresponding matcher set $\hat{\Pi}$
Output: Transformed statement sequence $\bar{s}''$
begin
  */*Generate a map that maps each referenced variable v^{ι} to its definition $v_{s^s}^{\iota_s}$ */*;
  $\phi \leftarrow$ aliasToDefMap($\hat{\Pi}, \bar{s}$);
  $Frags \leftarrow \{\}$;
  for *each basic block* $\overline{ps}$ *in* $\bar{s}$ **do**
    for $\hat{\pi} \in \hat{\Pi}$ **do**
      if *no statement tagged with active labels of* $\hat{\pi}$ *exists in* $\overline{ps}$ **then**
        continue;
      end
      $\bar{s}^0 \leftarrow$ genPartialStmts($\hat{\pi}, \overline{ps}$);
      $Loc \leftarrow$ genPos($\hat{\pi}, \overline{ps}$);
      if $Loc! = \emptyset$ **then**
        Randomly choose $loc \in Loc$;
        $Frags \leftarrow Frags \cup \{(\bar{s}^0, loc, \hat{\pi})\}$;
      else
        Empty generating position warning
      end
    end
  end
  $\bar{s}' \leftarrow \bar{s}$;
  for *each* $(\bar{s}^0, loc, \hat{\pi}) \in Frags$ **do**
    */*Delete all primitive statements binded to a deletion pattern*/*
    $\bar{s}_1 \leftarrow$ deleteMatchedStmts($\bar{s}', \hat{\Pi}$);
    */*insert* $\bar{s}^0$ *in its corresponding location in* $\bar{s}_1$ **/*
    $\bar{s}'' \leftarrow$ insertPartialCode($\bar{s}^0, \bar{s}_1$);
    */*Backfill program variables in* $\bar{s}$ *to annotated alias locations*/*
    $\bar{s}'' \leftarrow$ backFill($\bar{s}'', \bar{s}_1, \phi$);
    $\phi \leftarrow$ updateLocationMap($\phi, \bar{s}'', \bar{s}', \hat{\pi}$);
  end
end

Algorithm 2: Code Adaptation Algorithm, the location map function ϕ is a function that maps a set of aliases to the first definition of the alias, i.e. $\phi(S) = v^{\iota}$, s.t. $\forall v_1^{\iota_1} \in S.(v^{\iota} \; and \; v_1^{\iota_1} \; are \; aliases)$.

Input: A statement sequence $\bar{s}$, partial filled statements $\bar{s}'$, location map ϕ
Output: Backfilled statement sequence $\bar{s}'$
begin
  for *each alias notation* χ = alias(S) **do**
    /* extract the location of χ */
    $\iota_\chi \leftarrow$ location(χ);
    $v^\iota \leftarrow \phi(S)$;
    /* If the name v remains an alias of v^ι */
    if v^ι *exists in the* $\bar{s}$ **then**
      if v^ι *and* v^{ι_χ} *are aliases* **then**
        /* Fill the position with v */
        $\chi \leftarrow v$;
      else
        $u \leftarrow$ fresh variable name;
        /*Generate a copy of v^ι using u right after the location ι;*/
        insert($u = v;, \iota + 1$);
        /* Fill the position with u */
        $\chi \leftarrow u$;
      end
    else
      /* The variable is defined in the deleted code, thus v is a free name now. */
      $\chi \leftarrow v$;
    end
  end
end

Algorithm 3: Backfill algorithm backfill($\bar{s}'$, $\bar{s}$)

Input: The original location map ϕ, new statements $\bar{s}''$, old statements $\bar{s}', \hat{\pi}$
Output: Updated alias definition mapping ϕ
begin
  $\gamma \leftarrow$ id;
  for *each* $x : S \in$ varBindings($\hat{\pi}$) **do**
    $v^\iota \leftarrow \phi(S)$;
    if $v^\iota \in \bar{s}' \wedge v^\iota \notin \bar{s}''$ **then**
      /* In this case, v^ι is defined in the deleted code. According to
        * the backfill algorithm, v will be used as the generation name of x in $\bar{s}''$ */
      v is defined in loctaion ι' in $\bar{s}''$;
      $\gamma \leftarrow \gamma \circ [v^\iota \mapsto v^{\iota'}]$
    end
  end
  for *all* $x_1 : S_1, x_2 : S_2 \in$ varBindings($\hat{\pi}$) **do**
    if $\phi(S_1)$ *depends on* $\phi(S_2)$ *in* $\bar{s}' \wedge (\gamma \circ \phi)(S_2)$ *depends on* $(\gamma \circ \phi)(S_1) \in \bar{s}''$ **then**
      Report contradictory dependency relation warning;
    end
  end
  $\phi \leftarrow \gamma \circ \phi$
end

Algorithm 4: The update algorithm for aliasToDefinition map. updateLocationMap

会在多个模板中使用，因此，变量的创建工作就应该由第一个生成语句完成，在创建完毕，就会更新待填充位置到变量的映射，从而让后面应用该变量的部分能够引用该生成的代码，因此在update_algorithm中，我们的工作就是不断的更新代码空位对于代码变量的引用。

在回填进行完成之后，代码的修改工作就已经完成，这就让程序的修改进入到尾声。在这个时候，用户代码的API替换工作就已经完成。最后需要进行的尾处理工作就是对于正规化阶段生成的多余变量进行inline（注意我们不会修改非自动生成的变量，并且只对inline无害的生成变量进行该操作），从而最大限度的保持源代码的用户风格，让用户能够良好的进行代码的继续使用。

4.6 表里语义一致性的证明

在这里我们将证明表里语义之间的一致性：即前文所描述的表里语义之间的交换图交换条件满足。

Theorem 1 给定任意的程序 p 以及 p'，满足 $\Pi(p) = p'$（即p'是p在Π的里语义下的转换结果），则一定存在另外两个程序q 与 q'，满足 $p \doteq q$, $p' \doteq q'$, 以及 $\Pi^0(q, q')$。也即，下图交换。

$$\begin{array}{ccc} q & \xrightarrow{\Pi^0} & q' \\ \updownarrow{\scriptstyle \doteq} & & \updownarrow{\scriptstyle \doteq} \\ p & \xrightarrow[\Pi]{} & p' \end{array}$$

证明概要. 首先, 在移动阶段, 我们可以将移动操作分解为原子移动操作，即相邻交换，朴素移动，以及别名变量对象重命名。这是因为我们在进行移动阶段保持了语句以及变量之间的以来关系 (在交换图中我们把这一步变换标记为α)。其次, 在修改算法中的代码的生成点 pt 是所有标记语句可以通过原始交换移动到的位置 (我们将这个可以产生的移动阶段记为 β)。其三，存在一个从绑定元变量到程序变量之间的映射 $x : S \in \mathsf{varBinding}(\Pi)$, 并且根据我们对绑定操作的要求，每一个元变量只会绑定在一个程序变量对象中，因此我们可以对每个绑定的变量对象重命名，从而让每个元变量绑定的变量对象都名字互不相同。在这里，我们将元变量到里语义中绑定的程序变量映射定义为β。因此这个符合的里语义转换就可以

看作是在表语义 Π^0 上的转换, 这个转换结果为 p'. 在完成这个转换之后，我们通过重命名逆变换 γ_1^{-1}，就能够得到 $\Pi^0(\gamma(\beta(\alpha(p))), \gamma^{-1}(p'))$, 因此我们可以将里语义转换的结果理解为表语以转换与等价变换的复合。□ 至此，我们就完成了表里语义的介绍及其之间等价关系的证明，在接下来的章节中，我们就将分析该变换程序在实验中的表现以及对于相关工作的介绍。

第五章　实验评估

我们将我们的转换工具实现为eclipse的一个插件，用来进行插件中的Project的转换操作。在实现中我们采用了以下部件：分析工具soot，AST操纵工具eclipse JDT。我们的项目页面在 https://github.com/Mestway/Patl4J。

在实验中，我们需要进行对于如下问题的评估：

1. 我们进行一个项目转换需要多少的规则来描述？
2. 在转换过程中我们需要解决多少的多对多问题？
3. 我们在转换过程中会进行多少错误提示？

接下来的部分我们就通过数据来回答以上几个评估问题。

在评估过程中，我们采用了Jdom到Dom4j以及Google calendar v2到Google calendar v3之间的转换。

在这两个项目中，我们进行了Jdom中读写API之间的转换，以及gCalender中全部API的转换。我们书写一下规则使用的代码行数如下图所示：

Transformation	Rules	Classes	Methods	M-to-m
Jdom/Dom4j	84	12	77	12(14.3%)
Calendar	42	14	45	21(50.0%)
Total	126	26	122	33(26.2%)

由此可见代码转换行数是可以达到人为编程水平的，以及我们在不同的转换中M-to-m（多对多）转换是具有一定数量需要解决的。

使用了上述的这些转换规则，我们在几个客户端程序之中进行了API之间转换，列表如下：

Client	KLOC	Classes	Methods	Case
husacct	195.6	1187	5977	Jdom/Dom4j
serenoa	12.2	52	523	Jdom/Dom4j
openfuxml	112.5	727	4098	Jdom/Dom4j
clinicaweb	3.9	74	213	Calendar
blasd	9.7	199	729	Calendar
goofs	8.6	78	643	Calendar
Total	342.5	2317	12183	–

这几个规则中，husacct是一个软件体系结构表现的检测工具，serenoa是一个用户交互界面的修改框架，openfuxml是一个XML的发布框架，他们都采用了Jdom作为内部的XML处理支持，我们在测试中将他们转换为使用Dom4j接口的XML处理平台。另外的cliniaweb是一个在线临床的诊断平台，blased是一个邮件系统，goofs是一个文件系统，他们都是用了google calendar v2作为其中的日历处理接口，我们在转换中将这些接口升级为v3版本。

这些项目的转换结果如下：

Client	CF	CL	U	W	MM
husacct	42	852	0	0(0%)	0(0%)
serenoa	8	273	0	3(1%)	9(3.3%)
openfuxml	72	983	15	54 (6.6%)	2(0.2%)
clinicaweb	5	81	8	0 (0%)	34(42%)
blasd	5	26	0	7 (21.2%)	13(50%)
goofs	13	100	27	13 (15.3%)	27(27%)
Total	145	2315	50	77(3.2%)	85(3.7%)

CF =修改的文件数, *CL*= 修改的代码行数, $U =$ 无法进行转换的代码行数, $W =$ 报转换错误的数目, 其中的比例表示 $W/(W+CL)$, $MM =$ 涉及*M-M*转换的代码行数, 变分比表示 $= MM/CL$

由此可见我们在转换过程中能够较好的处理M-M映射问题，同时能够比较完整的进行API转换工作。

关于项目实现以及使用，更多的消息以及更新请参见项目页面。

第六章　相关工作与未来展望

支持程序变换的程序语言在业界已经有了较多的研究，常见的程序变换语言比如Stratego5，TXL6，Twinning12，SWIN16等，而这些程序变换语言常常有自己的主要研究方向，接下来我们先介绍相关工作中对于程序变换语言的研究。

一对多映射　很多程序语言都是设计来用于解决程序变换之中的一对多映射问题12, 11, 3, 17, 7, 8，这些语言解决的问题都体现在：规则给定原有程序的一条语句，以及该语句对应的语句序列，从而在代码中将该语句替换为对应的语句序列。一对多程序映射是程序变换中比较常见的现象，但由于其扩展性不强，在复杂的程序变换中由于上下文之间的依赖关系，单一规则的复合难以进行对这样转换模式的描述。

其中的Twinning12是一种用于描述程序变换之中API变换的基本语言，其特色主要在于能够对API变换进行一对多的描述，并能够在异常抛出的问题上进行较好的条件生成，但由于缺乏安全性检查，Twinning语言在使用上需要用户来保证规则的正确性。此外SWIN11是基于Twinning的一种程序变换语言，该语言在Twinning的基础上增加了一个类型检查系统，保证了转换过程中的API迁移类型正确性，通过对于类型正确性的检查，确保了规则书写层面上就能避免很多的转换问题。

多对多映射　同时，也有较多的程序语言是设计在多对多上的，通过对上下文关系的表述来描述多个语句之间的联系，从而限定程序转换的范围。这一类语言通常工作范围比较广泛，能够比较好的解决程序变换中的不同情况。

其中Stratego5是一种通用的程序变换语言，能够解决程序变换中比较常见的变换情况，该语言通过对变换AST上的上下节点关系描述达到了比较高的描述能

力，从而能够在程序变换之中进行上较好的程序变换功能。此外，SmPL1是一种用来描述上下文之间匹配关系的程序语言，其特点在于能够在程序上下文中通过标识语句之间的依赖关系，通过Model Checking的方式来进行匹配，其工作能力等效于使用数据流分析上的程序匹配功能。

除此之外，程序语言中还有一类是通过程序变换的方式来优化编译之中的代码生成问题的。例如Lacey与De Moor10提出了一种在数据流框架中通过程序变换来优化代码生成，这种方法在解决代码生成中的优化问题中颇有成效，但由于其作用在数据流框架中的特点，难以拓展到源代码转换的层面。

结论

pkuthss 文档模版最常见问题：

在最终打印和提交论文之前，请将 pkuthss 文档类选项中的 `colorlinks` 改为 `nocolorlinks`，因为图书馆要求电子版论文的目录必须为黑色，且某些教务要求打印版论文的文字部分为纯黑色而非灰度打印。

`\cite`、`\parencite` 和 `\supercite` 三个命令分别产生未格式化的、带方括号的和上标且带方括号的引用标记：2，[18]、[2,18]。

若要避免章末空白页，请在调用 pkuthss 文档类时加入 `openany` 选项。

如果编译时不出参考文献，请参考 `texdoc pkuthss`“问题及其解决”一章“其它可能存在的问题”一节中关于 biber 的说明。

参考文献

[1] J. Andersen *et al.* "*Semantic patch inference*". In: *ASE*, **2012**: 382–385.

[2] Author. "*Title*" [J]. *Journal*, 2014-04-01.

[3] I. Balaban, F. Tip and R. Fuhrer. "*Refactoring Support for Class Library Migration*". In: *OOPSLA*, **2005**: 265–279.

[4] G. Bierman, M. Parkinson and A. Pitts. *MJ: An imperative core calculus for Java and Java with effects* [techreport], **2003**.

[5] M. Bravenboer *et al.* "*Stratego/XT 0.17. A Language and Toolset for Program Transformation*". *Sci. Comput. Program.* **2008**: 52–70.

[6] J. R. Cordy. "*The TXL Source Transformation Language*". *Sci. Comput. Program.* **2006**: 190–210.

[7] D. Dig *et al.* "*ReBA: Refactoring-aware Binary Adaptation of Evolving Libraries*". In: *ICSE*, **2008**: 441–450.

[8] J. Henkel and A. Diwan. "*CatchUp!: Capturing and Replaying Refactorings to Support API Evolution*". In: *ICSE*, **2005**: 274–283.

[9] A. Igarashi, B. C. Pierce and P. Wadler. "*Featherweight Java: a minimal core calculus for Java and GJ*". *ACM Transactions on Programming Languages and Systems*, **2001**: 396–450.

[10] D. Lacey and O. De Moor. "*Imperative program transformation by rewriting*". In: *CC*, **2001**: 52–68.

[11] J. Li *et al.* "*SWIN: Towards Type-Safe Java Program Adaptation Between APIs*". In: *PEPM*, **2015**: 91–102.

[12] M. Nita and D. Notkin. "*Using Twinning to Adapt Programs to Alternative APIs*". In: *ICSE*, **2010**: 205–214.

[13] B. C. Pierce. *Types and programming languages*. MIT press, **2002**.

[14] M. Pilgrim. *Dive Into Python 3*. APress, **2009**.

[15] T. Tonelli, Krzysztof and Ralf. "*Swing to SWT and Back: Patterns for API Migration by Wrapping*". In: *ICSM*, **2010**: 1–10.

[16] C. Wang *et al. Formal Definition of SWIN language* [techreport], **2014**.

[17] L. Wasserman. "*Scalable, Example-based Refactorings with Refaster*". In: *WRT*, **2013**: 25–28.

[18] 作者。“标题”[J]。期刊，2014-04-01。

附录 A　附件

pkuthss 文档模版最常见问题：

在最终打印和提交论文之前，请将 pkuthss 文档类选项中的 `colorlinks` 改为 `nocolorlinks`，因为图书馆要求电子版论文的目录必须为黑色，且某些教务要求打印版论文的文字部分为纯黑色而非灰度打印。

`\cite`、`\parencite` 和 `\supercite` 三个命令分别产生未格式化的、带方括号的和上标且带方括号的引用标记：2，[18]、[2,18]。

若要避免章末空白页，请在调用 pkuthss 文档类时加入 `openany` 选项。

如果编译时不出参考文献，请参考 `texdoc pkuthss`“问题及其解决”一章“其它可能存在的问题”一节中关于 biber 的说明。

致谢

pkuthss 文档模版最常见问题：

在最终打印和提交论文之前，请将 pkuthss 文档类选项中的 `colorlinks` 改为 `nocolorlinks`，因为图书馆要求电子版论文的目录必须为黑色，且某些教务要求打印版论文的文字部分为纯黑色而非灰度打印。

`\cite`、`\parencite` 和 `\supercite` 三个命令分别产生未格式化的、带方括号的和上标且带方括号的引用标记：2，[18]、[2,18]。

若要避免章末空白页，请在调用 pkuthss 文档类时加入 `openany` 选项。

如果编译时不出参考文献，请参考 `texdoc pkuthss` “问题及其解决”一章“其它可能存在的问题”一节中关于 biber 的说明。

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