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 the programs with the old versions of APIs to the ones with new APIs, as the cost is high while errors are likely to appear after manual upgrade, 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.

However, adapting programs between APIs is still not a easy task with existing program transformation systems, due to the complex mappings between the old and the new APIs. We designed a domain specific language PATL to solve the problem by 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6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534。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7）

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。

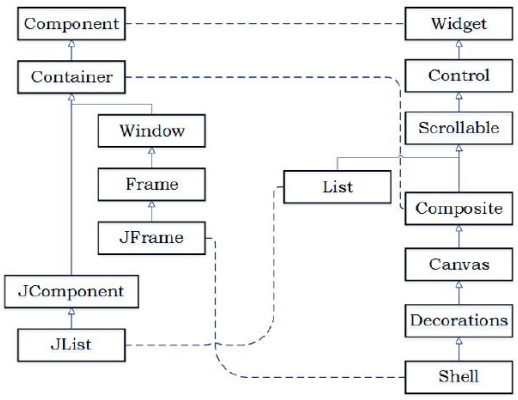

图 *1: Swing(图左)到SWT(图右)之间的类图对应关系，其中虚线箭头表示两个API之间的类映射，实线箭头表示同一API中类的继承关系（往下为父类）*

```
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:

```
column.setWidth(100);
TableCell cell1 = new TableCell();
column.add(cell1);
TableCell cell2 = new TableCell();
column.add(cell2);
```

(b) Target client code:

```
GridCell cell1 = new GridCell(100, column);
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。

```
/*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*

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。

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，[8]、[2,8]。

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

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

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

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

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

[8] 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，[8]、[2,8]。

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，[8]、[2,8]。

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