一种用于 Java 程序验证编译的标签类型*

陈 晖, 陈意云+, 茹祥民

(中国科学技术大学 计算机科学系,安徽 合肥 230027)

A Tag Type for Certifying Compilation of Java Program

CHEN Hui, CHEN Yi-Yun+, RU Xiang-Min

(Department of Computer Science, University of Science and Technology of China, Hefei 230027, China)

+ Corresponding author: Phn: +86-551-3607043, E-mail: yiyun@ustc.edu.cn

Received 2004-09-01; Accepted 2004-11-03

Chen H, Chen YY, Ru XM. A tag type for certifying compilation of Java program. *Journal of Software*, 2005,16(3):346−354. DOI: 10.1360/jos160346

Abstract: In the area of language-based security, programs written in typed high-level languages need to be translated into those written in typed low-level languages. This work is not trivial when type dispatch constructs are involved and implemented with tag mechanism. This paper proposes a new type that can deal with a special type dispatch construct occurring in the interface invokation of Java. A low-level language with this type is able to efficiently implement the interface invokation. This implementation approach is adopted in a Just-In-Time compiler with a typed low-level language as a certifying language.

Key words: type-preserving compilation; type dispatch; tag mechanism

摘 要: 在基于语言考虑代码安全性的工作中,往往需要将高级语言程序翻译成类型化低级语言的程序进行类型检查.许多高级语言具有类型调度结构,在向低级语言的编译过程中需要用标签机制来实现.针对具有多继承接口的 Java 程序包含的一种特殊的类型调度结构,提出了一种新的标签类型.包含这种标签类型的低级语言能够有效地实现 Java 程序中的接口调用.这种对接口调用的编译方法被用在一个以类型化低级语言为验证语言的 Java 字节码即时编译器中.

关键词: 类型保持编译;类型调度;标签机制

中图法分类号: TP301 **文献标识码**: A

经典的计算机安全原则和受信任计算基础(trusted computer base,简称 TCB)尽可能小的原则,要求软件系统的 TCB 尽可能地小而简单[1].TCB 是系统中这样的代码或机制:我们未能证明它们是安全的,而又不得不基于它们是安全的假设来证明系统是安全的.显然,系统的 TCB 越大,则存在安全漏洞的可能性就越大.在过去的 10

* Supported by the National Natural Science Foundation of China under Grant No.60173049 (国家自然科学基金); the Project of the Intel China Research Center (英特尔中国研究中心资助)

作者简介: **陈晖**(1979−),女,福建永春人,博士生,主要研究领域为程序语言设计与实现,类型论,基于语言的代码安全;**陈意云**(1946−),男,教授,博士生导师,主要研究领域为形式描述,程序设计语言的理论、设计与实现,编译理论与技术,软件安全;**茹祥民**(1978−),男,硕士生,主要研究领域为程序语言设计与实现.

年中,出现了许多基于语言的代码安全性研究,这些研究从构造软件系统的基础——语言和编译器着手,力求使软件系统具有尽可能小的 TCB.其中具有代表性的研究项目有:携带证明代码(proof-carrying code,简称 PCC)[2]、类型化汇编语言(typed assembly language,简称 TAL)[3,4]和高阶类型化语言 FLINT[5].这些研究的共同特点是:通过验证编译器将从源程序获得的有用信息以不同的表示形式(逻辑谓词、类型注释)保持到目标代码中.这些信息将被用于目标代码安全属性的验证,从而将编译器排除出系统的 TCB.以类型系统为基础的研究是这些研究中的一个重要方向,TAL,FLINT 都是这个方向上的成果.类型安全的语言确保通过类型检查的程序能够不犯由其类型系统排除出的那类错误.而验证编译在这个方向上的应用表现为类型保持编译(type-preserving compilation):从类型安全(type safe,或称类型可靠,type sound)的高级语言到类型安全的低级语言的翻译.

借鉴这些研究工作,我们设计了一种类型化低级语言(typed low-level language,简称 TLL),并以此为验证语言构造 Java 虚拟机,以探讨构造具有微小 TCB 的运行系统的方法.在这项工作中,Java 字节码将被类型保持翻译为低级的 TLL 程序,在其通过类型检查之后翻译成二进制代码.此举降低了代码的验证层次,从而将 Java 字节码翻译到 TLL 代码的那部分排除出 TCB.TLL 需要用更一般、更低级的类型和操作原语来实现 Java 语言中高级的对象系统.有些高级对象抽象的实现较为直观,例如对象实例可以用高阶的记录类型来实现.而有些对象抽象的实现则需要在低级语言中引入一些特殊的类型.

在我们的研究工作中发现,在 Java 对象的接口方法调用中包含了一种特殊类型调度(type dispatch)过程.类型调度就是根据值的不同类型来执行不同的处理,它的底层往往包含了标识类型的标签.接口调用中的这种类型调度使用的标签与普通类型标签不同,它标识了一类具有相同结构的类型.本文扩充了 Glew 提出的标签机制[6,7],引入一种新的标签类型,用于解决在翻译具有多继承的 Java 程序时出现的问题.

本文第 1 节描述翻译具有多继承接口的 Java 程序时出现的问题.第 2 节简要地介绍类型调度结构的一般形式以及如何利用标签机制来实现它.第 3 节提出一种新的标签类型,并介绍如何将这种标签类型用在翻译 Java 程序的接口方法调用中.第 4 节介绍相关研究.最后给出结论.

1 Java 程序接口调用中的类型翻译问题

Java 语言的接口定义一组方法,实现该接口的类必须实现这些方法.实现某个接口的类的对象可以被当作这个接口的实例来调用.图 1 描述了一个多继承接口的例子(本文对 Java 语法做了简化,认为类的所有成员都是公共的).类 Rectangle 实现了 Scalable 和 Turnable 两个接口,另外还间接地实现了它的父类 Shape 实现的接口 Movable;类 Round 则实现了 Movable 和 Scalable 两个接口.在 main 函数中这两个类的对象都可以作为实参传递给 scale_shape 函数.

```
interface Movable {void move (int dx, int dy);}
interface Turnable {void turn(int degree);}
interface Scalable {void zoom(int s);}
class Shape implements Movable{
   int x; int y;
   void move(int dx, int dy){… …}
}
class Rectangle
   extends Shape implements Turnable, Scalable
{int width; int high;
   void zoom(int s){… …}
   void turn(int degree){… …}}
class Round
      extends Shape implements Scalable
{int radium;
      void zoom(int s){… …}}
class Test{
 Rectangle r=new Rectangle;
 Round s=new Round;
 static void scale_shape (Scalable s){…s.zoom(2);}
 static main(){
      …scale_shape (r);
      scale_shape (s);…
      }}}
```

Fig.1 Java program example

图 1 Java 程序示例

图 2 展示了一个 Rectangle 对象的运行时表示,也就是对象布局(object layout),它是实现对象抽象的低层数据结构.不同的编译器在实现对象系统时采用的对象布局不尽相同,文献[8]中提到了一些面向对象语言编译器

的对象布局方案.本文采用的是一种较常见的对象布局,Intel 的 Java 虚拟机实现 ORP[9]就是采用类似的一种布局.

图 2(a)中是一个 Rectangle 类的对象,除了实例数据以外,它还包含了虚方法表(vtable),用来存放 Rectangle 类实现的所有虚方法地址(如图 2(b)所示).vtable 中还包含了接口方法表(itable)数组指针.数组的每个元素对应了该类实现的一个接口,包含了这个接口定义的各种方法的地址.图 2(c)是 Rectangle 类的 itable 指针数组.由于接口的多继承性,一个接口的 itable 在实现其不同的类的 itable 数组中并不具有固定的下标.例如,图 2 中的 Rectangle 对象,Scalable 接口对应的 itable 指针是数组的第 3 个元素,而对于一个 Round 类对象,按照同样的对象布局方案,Scalable 接口对应的 itable 指针则是数组的第 2 个元素.因此,scale_shape 函数中调用接口方法时(*s*.zoom(2)),并不知道对象 *s* 中 Scalable 接口的 itable 的确切位置,需要执行一个动态的方法查找过程.在图 2 中的对象布局方案中,每个 itable 被附以相应接口的标签.当调用接口方法时,已知接口的标签与各个 itable 的标签比较以确定所需的 itable.

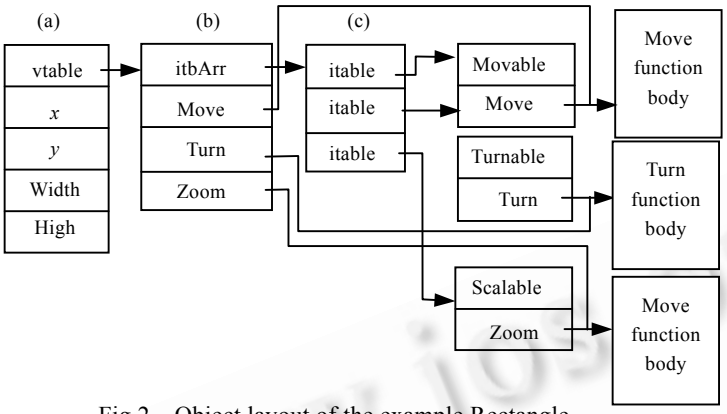

Fig.2 Object layout of the example Rectangle
图 2 程序示例的对象布局(Rectangle)

一般编译器中的目标语言并没有严格的类型系统.而对于目标语言是类型安全的低级语言的验证编译器,不仅要将高级操作原语翻译成低级语言的一系列操作,还要为低级的数据结构赋予低级语言的类型,并且使得这一系列操作符合类型安全的低级语言的定型规则.按照以上的对象布局方案,接口方法的调用被翻译为这样的底层处理过程:取得方法表→取接口方法表数组→获得正确接口方法表(或无法得到方法表而进行异常处理)→取得方法地址→调用方法.除了第 3 步,其他几个步骤的翻译都是平凡的,容易被验证为可靠的过程.第 3 步的翻译则需要在低级语言中引入特殊的类型,以保证利用标签查找的过程不会出错.本文扩展了对类型调度结构进行类型保持翻译的工作,提出了翻译接口调用的方案.

2 类型调度结构

第 1 节提出的接口方法表查找过程很容易与类型调度过程联系在一起.许多高级语言都包含类型调度结构,这种结构通过对值的动态测试确定值的更精确类型,并依此执行不同的处理.例如 ML 风格的异常匹配是一种典型的类型调度:当异常被抛出时,异常处理机制将比较异常的名字以确定异常的类型,从而决定异常处理的例程.除此之外,Java 的类造型也是一种类型调度结构.在 Java 程序中对象变量的静态类决定了它的哪些域和方法能够被引用.对象变量能够被赋予不同的动态类型,屏蔽其中的一部分域和方法.Java 提供类的下溯造型操作((*c*)*e*):表达式 *e* 被计值为一个对象,向下造型操作测试它的静态类是否为 *c* 的子类,如果是,那么对象将被赋予更精细的动态类型 *c*,否则抛出一个异常.这些类型调度结构的核心就是标签机制:对象的静态类和异常的名字即标签,它们总是依附于值或是置于值内;与这些标签相联系的是被标注值的类型;标签匹配成功时,被标注值的类型得到细化.

在具有类型调度结构的高级语言中,标签机制不出现在语法中而包含于语言的语义,由编译器实现.无论是在形式化地分析高级语言特征,或是用类型化低级语言翻译高级语言的过程中,都需要用不存在动态调度结构的低级语言来实现类型调度结构.在这些低级语言中,标签的相关操作语句显式地出现,并且定型规则严格地定义了标签比较时的类型约束.如果这些带有标签操作的低级语言的类型安全性质被证明,就能够确保通过标签比较完成的类型细化是可靠的.在 Glew[6]的文章中,用一种标签演算概括了几种常见的类型调度结构,并且提出

了从这种标签演算到一种不含类型调度原语而包含标签类型的低级语言的翻译算法(本文称 Glew 文章中的源语言为标签演算,而目标语言为标签语言,以示区别).

接口方法表查找也具有类似的类型调度特征:标签、由标签比较结果决定类型并决定不同的处理过程.但是这种标签与上文中各种类型调度结构中的标签存在差异.直观上看,附在 itable 上的标签标识了一个方法地址元组,通过对 itable 标签的比较,可以知道这个 itable 包含了哪些类型的方法.从 Java 语言的层次来看,这个元组中的方法类型是确定的,接口定义声明了这些方法的类型.但是从更低级的语言层次看,这些方法包含的隐式参数(this)因实现接口的类的不同而不同.因此,itable 上附着的标签事实上标识了一类具有相同结构的类型,类型中的某个部分是可变的.这种标签并不包含在 Glew 提到的几种类型调度结构中.

3 具有特殊标签类型的低级语言

限于篇幅,本文并不给出能够完整翻译面向对象特征的 TLL 语言,而给出一种低级标签语言 $\mathbf{tag}^{\forall}$.它具有能够标识一类具有相同结构的标签类型,包含了翻译接口方法调用的核心机制.第 3.1 节给出这种语言.第 3.2 节将使用这种标签语言实现第 1 节描述的接口方法表的查找过程,说明这样实现的查找过程是可靠的.

3.1 $\mathbf{tag}^{\forall}$语言

$\mathbf{tag}^{\forall}$语言的语法如下:

类型: $\tau,\sigma ::= \alpha \mid \mathrm{int} \mid \mathrm{uint} \mid \mathbf{tag}^{\varphi}(\tau_1,\tau_2) \mid \mathbf{rec}\alpha.\tau \mid \exists\alpha.\tau \mid \tau? \mid \{\vec{\tau}\} \mid \mathbf{array}(\tau) \mid \forall\alpha \le \tau.\tau_1 \to \tau_2$

标签标记: $\sigma ::= \uparrow\alpha \mid \downarrow\sigma$

项: $e,b ::= x \mid \mathbf{if}\ e_1 = e_2[\sigma]\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2\ \mathbf{fi} \mid \mathbf{roll}^{\tau}(e) \mid \mathbf{unroll}(e) \mid \Lambda\alpha \le \tau.\mathbf{fixf}(\overrightarrow{x:\tau}):\tau'.b \mid e_1 e_2 \mid e[\tau]$
$\mathbf{pack}[\tau_1,e]\ \mathbf{as}\ \tau_2 \mid \mathbf{unpack}[\alpha,x] = e_1\ \mathbf{in}\ e_2 \mid \mathbf{none} \mid \mathbf{if}\ ?\ e_1\ \mathbf{then}\ x.b_1\ \mathbf{else}\ b_2\mathbf{fi} \mid \{\vec{e}\} \mid e.i \mid$
$\mathbf{let}\ \overrightarrow{x = e}\ \mathbf{in}\ e \mid \mathbf{tag}([\alpha,\tau],\{\vec{e}\}) \mid \mathbf{tagapp}[\sigma,e] \mid \langle\vec{e}\rangle \mid e.\mathbf{size} \mid e.i \Rightarrow x.b_1\ \mathbf{else}\ b_2 \mid$

语言的定型规则有如下描述,其中Δ和Γ是类型检查的上下文:Δ包含了自由类型变量及其约束条件;Γ包含了自由程序变量的类型指派.附录 A 中展示了语言的操作语义,也就是表达式的计值规则.

$$\frac{\Delta;\Gamma \rhd e:\sigma[\tau/\alpha]}{\Delta;\Gamma \rhd \mathbf{roll}^{\tau}(e):\tau}(\tau = \mathbf{rec}\alpha.\tau)\ (\mathrm{R1})$$

$$\frac{\Delta;\Gamma \rhd e:\tau}{\Delta;\Gamma \rhd \mathbf{unroll}(e):\tau[\tau/\alpha]}(\tau = \mathbf{rec}\alpha.\tau)\ (\mathrm{R2})$$

$$\frac{\Delta;\Gamma \rhd \overrightarrow{x:\tau}\ \Delta;\Gamma,\overrightarrow{x:\tau} \rhd e':\tau'}{\Delta;\Gamma \rhd \mathbf{let}\ \overrightarrow{x = e}\ \mathbf{in}\ e':\tau}\ (\mathrm{R3})$$

$$\frac{\Delta \rhd \tau_1\ \Delta;\Gamma \rhd e:\tau_2[\tau_1/\alpha]}{\Delta;\Gamma \rhd \mathbf{pack}[\tau_1,e]\mathbf{as}\ \exists\alpha.\tau_2:\exists\alpha.\tau_2}\ (\mathrm{R4})$$

$$\frac{\Delta;\Gamma \rhd e_1:\exists\alpha.\tau_1\ \Delta,\alpha;\Gamma,x:\tau_1 \rhd e_2:\tau_2\ \Delta \rhd \tau_2}{\Delta;\Gamma \rhd \mathbf{unpack}[\alpha,x] = e_1\ \mathbf{in}\ e_2}(\alpha \notin \Delta)\ (\mathrm{R5})$$

$$\frac{\Delta,\alpha \le \tau;\Gamma \rhd \vec{\tau} \to \tau'\ \Delta,\alpha;\Gamma,\overrightarrow{x:\tau} \rhd b:\tau'}{\Delta;\Gamma \rhd \Lambda\alpha \le \tau.\mathbf{fixf}\ (\overrightarrow{x:\tau}):\tau'.b}\ (\mathrm{R6})$$

$$\frac{\Delta;\Gamma \rhd e:\alpha \le \tau.\tau_1 \to \tau_2\ \Delta \rhd \sigma \le \tau}{\Delta;\Gamma \rhd e[\sigma]:\tau_1[\sigma/\alpha] \to \tau_2[\sigma/\alpha]}\ (\mathrm{R7})$$

$$\frac{\Delta;\Gamma \rhd e:\mathbf{tag}^{\varphi}(\tau_1,\tau_2)}{\Delta;\Gamma \rhd e:\tau_2}\ (\mathrm{R8})$$

$$\frac{\Delta,\alpha \rhd \tau(\alpha \notin \Delta)\ \Delta;\Gamma \rhd e:\tau'}{\Delta;\Gamma \rhd \mathbf{tag}([\alpha,\tau],e):\mathbf{tag}^{\downarrow\alpha}(\tau,\tau')}\ (\mathrm{R9})$$

$$\frac{\begin{array}{c}\varepsilon \rhd \sigma\Delta;\Gamma \rhd e_1:\mathbf{tag}^{\downarrow\sigma}(\beta,\tau_1)\ \Delta;\Gamma \rhd e_2:\mathbf{tag}^{\uparrow\alpha}(\tau',\tau_2)\\ \Delta_1,\beta = \tau'[\sigma/\alpha],\Delta_2;\Gamma \rhd b_1:\tau\ \Delta;\Gamma \rhd b_2:\tau\end{array}}{\Delta;\Gamma \rhd \mathbf{if}\ e_1 = e_2[\sigma]\ \mathbf{then}\ b_1\ \mathbf{else}\ e_2\ \mathbf{fi}:\tau}(\Delta = \Delta_1,\beta,\Delta_2)\ (\mathrm{R10})$$

$$\frac{\Delta;\Gamma \rhd e:\mathbf{tag}^{\uparrow\alpha}(\tau_1,\tau_2)\ \Delta \rhd \sigma}{\Delta;\Gamma \rhd \mathbf{tagapp}(\sigma,e):\mathbf{tag}^{\downarrow\sigma}(\tau_1[\sigma/\alpha],\tau_2}\ (\mathrm{R11})$$

$$\frac{\begin{array}{c}\varepsilon \rhd \sigma,\sigma'\ \Delta;\Gamma \rhd e_1:\mathbf{tag}^{\downarrow\sigma}(\sigma',\tau_1)\ \Delta;\Gamma \rhd e_2:\mathbf{tag}^{\uparrow\alpha}(\tau',\tau_2)\\ \varepsilon \rhd \sigma' = \tau'[\sigma/\alpha] \Rightarrow \Delta;\Gamma \rhd b_1:\tau\ \Delta;\Gamma \rhd b_2:\tau\end{array}}{\Delta;\Gamma \rhd \mathbf{if}\ e_1 = e_2[\sigma]\ \mathbf{then}\ b_1\ \mathbf{else}\ e_2\ \mathbf{fi}:\tau}\ (\mathrm{R12})$$

$$\frac{\Delta;\Gamma \rhd e_i:\tau}{\Delta;\Gamma \rhd < \vec{e_i} > \mathbf{array}(\tau)}\ (\mathrm{R13})$$

$$\frac{}{\rhd i:\mathrm{int}}\ (\mathrm{R14})$$

$$\frac{\Delta;\Gamma \rhd e:\mathbf{array}(\tau)}{\Delta;\Gamma \rhd e.\mathbf{size}:\mathrm{int}}\ (\mathrm{R15})$$

$$\frac{\begin{array}{c}\Delta;\Gamma \rhd e:\mathbf{array}(\tau)\\ \Delta;\Gamma,x:\tau \rhd b_1:\tau'\ \Delta;\Gamma \rhd b_2:\tau'\end{array}}{\Delta;\Gamma \rhd e.i \Rightarrow x.b_1\ \mathbf{else}\ b_2:\tau'}\ (\mathrm{R16})$$

语言具有标签类型 $\mathbf{tag}^{\varphi}(\tau,\{\vec{\tau}\})$,其中的标识$\varphi$有两种:$\uparrow\alpha$和$\downarrow\sigma$.在这种语言中,两种表达式可以在堆中产生一个新的元组:$\{\vec{e}\}$和 $\mathbf{tag}([\alpha,\tau],\{\vec{e}\})$.$\{\vec{e}\}$是$\{e_1,...,e_n\}$的简写,它产生一个新的元组,元组中第 i 个域的值为 e_i.$\mathbf{tag}([\alpha,\tau],\{\vec{e}\})$的操作语义和$\{\vec{e}\}$相同,区别在于,它产生的元组同时还可以作为类型为$\tau$的值的标签.类型$\tau$中含有可变的类型变量$\alpha$,$\alpha$可以被任意类型替换.表达式 $\mathbf{tag}([\alpha,\tau],\{\vec{e}\})$产生类型为 $\mathbf{tag}^{\uparrow\alpha}(\tau,\tau')$的值(规则 R9).类型为 $\mathbf{tag}^{\downarrow\sigma}(\tau_1,\tau_2)$的值是类型为 $\mathbf{tag}^{\uparrow\alpha}(\tau_1,\tau_2)$标签的通过类型强制 **tagapp** 得到的一个替换结果,它用σ替换了τ_1中特定的类型变量α(规则 R11).类似于多态类型中的称呼,本文把替换结果叫作原标签的一个实例,$\mathbf{tagapp}[\sigma,x]$表达式的计值结果也可写作 $x[\sigma]$.

在进行标签检查时(规则 R10),假如两个标签相等,且具有一对互补标识(↑和↓),则在标签相等的程序分支(then 子句)的类型检查上下文中增加未知类型变量β的约束条件:β应该是已知标签可标注类型的一个σ替换实例.假如类型为 $\mathbf{tag}^{\uparrow\alpha}(\tau',\tau_1)$的标签和某个已知类型为 $\mathbf{tag}^{\downarrow\sigma}(\sigma',\tau_2)$的标签相等(规则 R12),则它们的类型必须存在这样的关系:$\sigma'=\tau'[\sigma/\alpha]$,并且在此类型关系下能够推断两个分支的类型.

除此之外,$\mathbf{tag}^{\forall}$包含了高阶类型$\exists\alpha.\tau$和$\forall\alpha\le\tau.\tau_1\to\tau_2$.存在类型$\exists\alpha.\tau$通过表达式 $\mathbf{pack}[\tau_1,e]\mathbf{as}\exists\alpha.\tau_2$ 生成,这个表达式隐蔽了类型为τ_2的表达式 e 中某一部分类型τ_1(规则 R4).程序的其他部分在使用这个值时并不知道τ_1是什么,而用一个类型变量α来代替它(规则 R5).类型为$\forall\alpha\le\tau.\tau_1\to\tau_2$函数包含了多态参数,即它的参数类型具有可变性,可以被任意满足约束的类型所替换.规则 R7 描述了多态函数实例化的定型规则.

为了实现图 2 中的对象布局,$\mathbf{tag}^{\forall}$中还引入了数组类型 $\mathbf{array}(\tau)$,以及相关的数组构造表达式$\langle\vec{e}\rangle$,取数组长度表达式 e.size 以及数组元素表达式 $e.i\Rightarrow x.b_1$ **else** b_2.前两个表达式的操作语义显而易见.表达式 $e.i\Rightarrow x.b_1$ **else** b_2 包含了这样的操作语义:如果下标 i 超出数组的长度,则表达式计值结果为 b_2 的计值结果;否则为 b_1 的计值结果,b_1中的名字变量 x 的值绑定为数组 e 的第 i 个元素.

为了避免使类型检查算法和类型安全属性证明变得复杂,在标签比较语句中相等的分支 $\mathbf{tag}^{\downarrow\sigma}(\tau_1,\tau_2)$与 $\mathbf{tag}^{\uparrow\alpha}(\tau_1,\tau_2)$的$\sigma$实例仅具有相等关系,而并不像 Glew 的层次结构标签那样还包括了子定型关系.其他子定型关系的规则具有一般性,由于篇幅的关系本文并不一一列出.

3.2 使用扩展的标签语言翻译接口方法表

本节仍以图 1 中的程序为例来介绍对接口方法表的翻译.在用类型化低级语言翻译对象系统时,需考虑对象的自应用语义[10],还需要引入一些复杂的类型来保证翻译过程中自应用语义的保持.为了简化问题,本文中的翻译不考虑自应用语义的问题,用 ObjTP_C 来表示类 C 对象的类型.图 2 中 3 个接口方法表的标签分别翻译为 x_{Movable},x_{Scalable},x_{Turnable}.它们分别是一个无内容且可作标签的元组.

$\mathbf{let}\ x_{\mathrm{Movable}}=\mathbf{tag}([\alpha,\{\alpha\to(\mathrm{int}\to(\mathrm{int}\to\mathrm{unit}))\}],\mathbf{none})\ \mathbf{in}\ldots$

$\mathbf{let}\ x_{\mathrm{Scalable}}=\mathbf{tag}([\alpha,\{\alpha\to(\mathrm{int}\to\mathrm{unit})\}],\mathbf{none})\ \mathbf{in}\ldots$

$\mathbf{let}\ x_{\mathrm{Turnable}}=\mathbf{tag}([\alpha,\{\alpha\to(\mathrm{int}\to\mathrm{unit})\}],\mathbf{none})\ \mathbf{in}\ldots$

构造图 2(c)中的 Rectangle 类的接口方法表数组的表达式则为

$$\begin{aligned}\mathbf{let}\ \mathrm{iTbArr}=\langle&\mathbf{pack}[\tau_1,\{\mathbf{tagapp}[\sigma,x_{\mathrm{Movable}}],\{\mathrm{move}[\sigma]\}\}]\mathbf{as}\ \exists\alpha.\{\mathbf{tag}^{\downarrow\sigma}(\alpha,\{\mathrm{unit}\});\alpha\},\\&\mathbf{pack}[\tau_2,\{\mathbf{tagapp}[\sigma,x_{\mathrm{Scalable}}],\{\mathrm{zoom}[\sigma]\}\}]\mathbf{as}\ \exists\alpha.\{\mathbf{tag}^{\downarrow\sigma}(\alpha,\{\mathrm{unit}\});\alpha\},\\&\mathbf{pack}[\tau_2,\{\mathbf{tagapp}[\sigma,x_{\mathrm{Turnable}}],\{\mathrm{turn}[\sigma]\}\}]\mathbf{as}\ \exists\alpha.\{\mathbf{tag}^{\downarrow\sigma}(\alpha,\{\mathrm{unit}\});\alpha\}\rangle\\\mathbf{in}\ldots&\end{aligned}$$

其中$\tau_1=\{\sigma\to(\mathrm{int}\to(\mathrm{int}\to\mathrm{unit}))\}$,$\tau_2=\{\sigma\to(\mathrm{int}\to\mathrm{unit})\}$.而$\sigma$为 Rectangle 对象的类型 $\mathrm{ObjTP}_{\mathrm{Rectangle}}$.move,zoom,turn 则分别是 Rectangle 类实现的 3 种方法的翻译结果.它们的第 1 个参数都具有多态性,这是因为类的方法可以被它的子类的对象调用,例如从 Shape 类继承得到的 move 方法将被翻译成类型为$\forall\alpha\le\mathrm{ObjTP}_{\mathrm{Shape}}.\alpha\to(\mathrm{int}\to(\mathrm{int}\to\mathrm{unit}))$的多态函数.这些函数在 Rectangle 类的接口方法表中都被 $\mathrm{ObjTP}_{\mathrm{Rectangle}}$ 实例化.由此可以得到 iTbArr 的类型为数组类型 $\mathbf{array}(\exists\alpha.\{\mathbf{tag}^{\downarrow\sigma}(\alpha,\{\mathrm{unit}\});\alpha\})$.它的每个元素都是由标注了某个未知类型的以及这种类型的值构成的元组.这个标签是一个多态标签的 $\mathrm{ObjTP}_{\mathrm{Rectangle}}$ 实例,通过标签比较可以确定这个未知类型.

在翻译查找接口方法表时,一个比较标签并得到相应接口方法表的函数将被调用.例如,在翻译调用 Scale

接口时,下面的函数将被用来得到相应的接口方法表:

0 **let** get_scale_itabl=$\Lambda\alpha'\leq \mathrm{ObjTP}_{\mathrm{Object}}$.**fix** ($x_1$:**array**($\exists\beta$.{**tag**$^{\downarrow\alpha'}$($\beta$,{unit});$\beta$})):{$\alpha'\rightarrow$(int→unit)}?.

1 **let** loop=**fix**(x_2:int):{$\alpha'\rightarrow$(int→unit)}?.

2 $x_1[x_2]\Rightarrow x_3$.(**unpack**[γ,x_3]=x_4 **in**

3 **if** x_4.**0**=$x_{\mathrm{Scalable}}[\alpha']$ **then** x_4.**1** **else** loop x_2+1)

4 **else none**

5 **in** loop 0

6 **in**…

这个函数接受某个对象的接口方法表数组(x_1)作为参数,这个对象的类型是α',它可以是基本对象(object)的子类型.函数的返回值类型为{$\alpha'\rightarrow$(int→unit)}?.这是一个可选类型,这种类型的值可以是一个{$\alpha'\rightarrow$(int→unit)}类型的元组,或是一个空元组.语句序列依次取出数组的每个元素,将这个元组的第 1 个数据也就是一个未知类型的接口方法表标签与x_{Scalable}作比较.在存在类型的包执行了解开操作之后(第 2 行),这个未知类型的接口方法表标签(x_4的第 1 个数据)具有类型:**tag**$^{\downarrow\alpha'}$(γ,{unit}).其中γ为新引入的类型变量,同时也是跟随在标签之后数据的类型.第 4 行执行的就是比较操作,由上文中x_{Scalable}的声明可知:

x_{Scalable}:**tag**$^{\uparrow\alpha}$({$\alpha\rightarrow$(int→unit)},{unit}).

由标签比较语句的定型规则(R10)可知,在比较成功的分支中,类型变量γ得到更细化的类型:

γ={$\alpha'\rightarrow$(int→unit)}.

由此可知,标签引导的数据(x_4的第 2 个数据)为存放类型为$\alpha'\rightarrow$(int→unit)的函数元组.在多态函数的参数被实例化之后,这个函数元组的类型中的类型变量 α 也被替换为具体的对象类型.如果数组中每个标签都匹配不成功,则返回空元组(第 4 行).

在 **tag**$^{\forall}$这样一种可以被证明是一个类型安全的语言的低级语言中,以上描述的函数经检查为良类型(well-typed)的,从而保证这个查找过程是可靠的.这种能标注一类具有相同结构类型的标签类型被扩充到我们的类型化低级语言 TLL 中,使之能够实现 Java 接口方法调用原语的类型保持翻译.

4 相关工作

上文中提到的 Glew 的标签语言能够很好地实现各种具有层次结构的类型调度结构.用于类造型中的标签就是典型的具有层次结构的标签:与标签(类)联系的是该标签相应类型及其子类型(类及它的子类);进行标签比较时,则不仅仅比较值的标签与已知标签是否相等,而且还比较值的标签是否是已知标签的子标签.这些标签类型能够有效地实现对象的类标签.Glew 的解决方案还被用于 TAL 对面向对象语言的类型保持翻译中[7].我们的低级语言 TLL 也引入了这种标签来实现 Java 的向下造型表达式.

除了 Glew 的工作之外,Crary 和 Weirch 也为动态类型分析作出了重要的贡献.他们设计的语言λ_{R}[11]及在后继工作中设计的 LX[12]能够将运行时类型信息表示成一般的项.在类型制导的编译(type-directed compilation)中,这些语言可以用于分析上下文中静态无法确定的类型信息(包括实现本文中提到的基于标签机制的类型分析),并把这些信息用于代码的优化和转换.他们的语言具有一般性,但是也相对复杂,每一种类型和每一种类型构造符都有唯一的项对应.与他们的工作相比,Glew 的工作更具有针对性.标签语言和标签类型针对具有明显特征并且广泛用于各种高级语言的类型调度结构提出.而本文在其工作基础上对标签的表现能力作了补充,使标签能够标注具有一类相同结构的类型的值.

League 用 FLINT 实现的 Java 编译器[13]是一个具有相当影响的类型保持编译器.在其工作中,对接口方法调用的处理和本文中提到的并不相同.当一个类实例被当作接口实例时,例如第 1 节程序中,Rectangle 类对象被传给形参为 Scalable 的函数 scale_shape,这个实例被重新包装为具有唯一接口方法表的接口实例.这种处理方法事实上是生成了一个新实例,它的实例数据和原实例相同,不同的是提取了相应方法形成接口方法表.因此在调用接口方法时,同一接口实例的接口方法表的位置是固定的.这种处理方法将会产生大量的临时对象耗费空间.

另一种处理方法就是把类的所有接口方法表都放到一个无顺序元组中.无顺序元组的域选择和顺序元组

中的相应操作类似,但是需要编译器来实现运行时查找.这仍然是一个较高级的语言抽象,与验证编译器希望在较低级层次上翻译高级语言程序的目的不符合.

5 结 论

由于对 Java 程序验证编译的需要,本文提出一种新的标签类型,它能够标注一类具有相同结构的类型的值.包含这种标签类型的低级语言能够有效地实现 Java 程序中的接口调用.本文的工作扩展了对类型调度结构的研究.文中提出的对具有多继承接口的 Java 对象编码方案为分析 Java 语言特征提供了一种新的视角.这种标签类型被引入我们设计的类型化低级语言 TLL 中.采用文中所述的编码方案,TLL 实现了对具有接口的 Java 语言的翻译.基于 TLL 构造的 Java 虚拟机能够在低层代码上对程序进行类型检查,从而达到减小 TCB 的目的.在许多具有丰富表现力的语言中存在着类型调度结构,例如具有和类型(sum type)ML 语言以及具有可重载特性的面向对象语言(Java 和 C++).本文的工作对将这些高级语言翻译到类似于类型化低级语言的验证编译具有一定的参考价值.

References:

[1] Schneider FB, Morrisett G, Harper R. A language-based approach to security. LNCS: Informatics-10 Years Back. 10 Years Ahead. Springer-Verlag, 2001. 86−101.

[2] Necula G. Proof-Carrying code. In: ACM Symp. on Principles of Programming Language. New York: ACM Press, 1997. 106−119.

[3] Morrisett G, Walker D, Crary K, Glew N. From system F to typed assembly language. ACM Trans. on Programming Languages and Systems, 1999,21(3):528−569.

[4] Morrisett G, Crary K, Glew N. TALx86: A realistic typed assembly language. In: ACM SIGPLAN Workshop on Compiler Support for System Software. Atlanta, 1999. 25−35.

[5] Shao Z. Typed common intermediate format. In: USENIX Conf. on Domain-Specific Languages. Santa Barbara, 1997. 82−102. http://flint.cs.yale.edu/flint/publications/tcif.html

[6] Glew N. Type dispatch for named hierarchical types. In: Proc. of the 1999 ACM SIGPLAN Int'l Conf. on Functional Programming. Paris: ACM Press, 1999. 172−182.

[7] Glew N. An efficient class and object encoding. Technical Report STAR-TR-00. 07-02, STAR Laboratory, 2000.

[8] Chen YY, Zhang Y. Theory of Compiler. Beijing: Higher Education Press, 2003. 344−351 (in Chinese).

[9] Intel Microprocessor System Lab. ORP Online Resource. http://intel.com/research/mrl/ORP

[10] Kamin S. Inheritance in Smalltalk-80: A denotational definition. In: Proc. of the 15th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages. San Diego: ACM Press, 1988. 80−87.

[11] Crary K, Weirich S. Flexible type analysis. In: Int'l Conf. on Functional Programming. Paris: ACM Press, 1999. 233−248.

[12] Crary K, Weirich S, Morrisette. Intensional polymorphism in type-erasure semantics. Journal of Functional Programming, 2002,12(6):567−600.

[13] League C, Shao Z, Trifonov V. Type-Preserving compilation of featherweight Java. ACM Trans. on Programming Languages and Systems, 2002,24(2):112−152.

附中文参考文献:

[8] 陈意云,张昱.编译原理.北京:高等教育出版社,2003.344−351.

附录

A tag$^{\forall}$语言的操作语义(归约规则)

附录 A 展示了 **tag**$^{\forall}$语言的操作语义.了解操作语义有助于理解各表达式的运算功能及理解语言的类型安全属性证明.**tag**$^{\forall}$语言的计算模型是在一个存储区上执行表达式.本文用二元组$\langle H,e\rangle$来描述程序状态:H 是存储区,它是堆值标识(或可以看成是堆值的指针,x)到其对应堆值(h)的有限映射集合;e 是当前正在计算的表达式.堆

值 h 可以是值(v)的元组$\{\vec{v}\}$,数组$\langle\vec{v}\rangle$或标签值 $\mathbf{tag}([\alpha,\tau],\{\vec{v}\})$.

$P ::= \langle H,e\rangle$

$v ::= x \mid i \mid \mathbf{roll}^{\tau}(e) \mid \mathbf{unroll}(\mathrm{e}) \mid \Lambda\alpha\le\tau.\mathbf{fixf}(\overrightarrow{x:\tau}):\tau'.b \mid v[\tau] \mid \mathbf{pack}[\tau_1,e]\mathbf{as}\ \tau_2 \mid \mathbf{none}$

$h ::= \{\vec{v}\} \mid \langle\vec{v}\rangle \mid \mathbf{tag}([\tau_1,\tau_2],\{\vec{v}\})$

下面给出的语言的操作语义描述了表达式的一步归约对程序状态的影响.当前的表达式总是这样的形式:$E[\iota].E[.]$为表达式ι的计算环境,经过一步归约后表达式变化为 $E[e]$,而存储区在表达式作用下变为 H'.

$$\langle H,E[\iota]\rangle\rightarrow\langle H',E[e]\rangle$$

$\mathbf{tagapp}(\sigma,x))$	$x[\sigma]$	H	$H(x)=\mathbf{tag}([\alpha,\tau],\{\vec{v}\})$
$\mathbf{if}\ x[\sigma]=x[\sigma]\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2$	b_1	H	
$\mathbf{if}\ x=y[\sigma]\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2$	b_2	H	$x\neq y$
$\mathbf{unroll}(\mathbf{roll}^{\tau}(v))$	v	H	
$\mathbf{unpack}[\alpha,x]=(\mathbf{pack}[\tau_1,v]\mathbf{as}\ \tau_2)\mathbf{in}\ e$	$e[\tau_1/\alpha,v/x]$	H	
$v[\tau]$	$v[\tau/\alpha]$	H	$v=\Lambda\alpha\le\tau.\mathbf{fix}\ f(\overrightarrow{x_i:\tau_i}):\tau'.b$
$v_1\ \vec{v_i}$	$b[v_1/f,x_i/v_i]$	H	$v_1=(\Lambda\alpha\le\tau.\mathbf{fix}\ f(\overrightarrow{x_i:\tau_i}):\tau'.b)[\tau]$
$x.i$	v_i	H	$H(x)=v$ or $\mathbf{tag}([\alpha,\tau],v)$
			$1\le x\le n, v=\{v_1,\ldots,v_n\}x$
$x.\mathbf{size}$	n	H	$H(x)=\langle v_1,\ldots,v_n\rangle$
$x[i]\Rightarrow x'.b_1\ \mathbf{else}\ b_2$	$b_1[v_i/x']$	H	$H(x)=\langle v_1,\ldots,v_n\rangle, 1\le i\le n$
$x[i]\Rightarrow x'.b_1\ \mathbf{else}\ b_2$	b_2	H	$H(x)=\langle v_1,\ldots,v_n\rangle,\neg(1\le i\le n)$
$\mathbf{let}\ \overrightarrow{x_i=v_i}\ \mathbf{in}\ e'$	$e'[\overrightarrow{v_i/x_i}]$	H	

B tag$^{\forall}$语言的类型安全性证明

语言的类型安全属性确保了良类型的程序不会运行时阻塞,也就是说,不会计值到一个非正常结果.类型安全属性是保持性(preservation)和前进性(progress)两条定理的推论.这两条定理表明了良类型语言的静态语义和动态语义的内在关系.保持性说明了良类型程序经过一步归约后仍然是良类型的;前进性则说明了良类型程序除非处于计算终结状态,否则总能进行归约.在证明这两个定理之前必须先定义良类型程序.上下文Γ具有堆的形式$\{\overrightarrow{x:\tau}\}$,它指派了每个变量的类型.我们把它看做是堆 H 的类型:$\rhd H:\Gamma$.一个程序 P 是良类型的,它的状态二元组需满足这样的条件:$\dfrac{\rhd H:\Gamma\ \varepsilon;\Gamma\rhd e}{\rhd P=\langle H,e\rangle}$.

引理 1(上下文弱化引理). 如果 Δ_1 扩展了 Δ_2,Γ_1 扩展了 Γ_2,那么由上下文($\Delta_2;\Gamma_2$)推断得到的结果,上下文($\Delta_1;\Gamma_1$)也能够推断得到.

引理 2(上下文定型引理). 如果 $\varepsilon;\Gamma\rhd E(i):\tau$,那么存在着某个类型$\sigma$使得 $\varepsilon;\Gamma\rhd i:\sigma$,且所有满足条件且 $\varepsilon;\Gamma,\Gamma'\rhd e:\sigma$ 的表达式 e 断言 $\varepsilon;\Gamma,\Gamma'\rhd E(e):\tau$ 成立.

证明:对 $\varepsilon;\Gamma,\rhd E(i):\tau$ 的推断分情况讨论(可以是某个值,或是结构化表达式).根据引理 1,将 $\varepsilon;\Gamma,\Gamma'\rhd e:\sigma$ 代入 $\varepsilon;\Gamma\rhd E(i):\tau$ 推断的最后一步规则可以得到要证明的结果. □

定理 1(保持性). 如果 $\rhd P$ 且 $P\rightarrow P'$,那么 $\rhd P'$.

证明:若 $P=\langle H_1,E[\iota]\rangle$,而 P'为$\langle H_2,E[e]\rangle$,那么ι,e,和 H_2 由附录 A 中给出的操作语义描述某个操作语义规则给出.推断 $\rhd P$ 需要条件 $\rhd H_1:\Gamma_1$ 及 $\varepsilon;\Gamma_1\rhd E(i):\tau$.根据引理 2,存在某个类型$\sigma'$使得 $\varepsilon;\Gamma_1\rhd i:\sigma'$.证明的目标是找到 Γ_2 使得 $\rhd H_2:\Gamma_1,\Gamma_2$ 及 $\varepsilon;\Gamma_1,\Gamma_2\rhd e:\sigma'$ 成立.然后运用引理 2 得到 $\varepsilon;\Gamma_1,\Gamma_2\rhd E[e]:\tau$ 成立,则 $\rhd P'$ 的结论显而易见.证明通过对 $\rhd P$ 推断最后一步的各种情况进行分析,这里我们只列出在标签相关的情况下如何选择Γ_2.

ι=**tag**($[\alpha,\tau],\{\vec{v}\}$):这种情况下,根据操作语义规则 $e=x$,其中 x 为新标识符,也就是在堆中分配了以此为标识的新的数据块,$H_2=H_1,x=\iota$.很容易推断出 H_2 的类型 $\triangleright H_2:\Gamma_1,x:\sigma$.选择$\Gamma_2$ 为 $x:\sigma$,根据引理 1 可以得到 $\Gamma_1,\Gamma_2\triangleright x:\sigma$.

ι=**if** $x[\sigma]=x[\sigma]$ **then** b_1 **else** b_2:这种情况下,$e=b_1$ 且 $H_1=H_2$.根据定型规则 R12(ι检查的类型上下文为 ε), $\varepsilon;\Gamma_1\triangleright i:\sigma'$ 推断具有以下形式: $\dfrac{\varepsilon\triangleright\sigma,\sigma'\ \varepsilon;\Gamma_1\triangleright x[\sigma]:\mathbf{tag}^{\downarrow\sigma}(\sigma_1,\tau_1)\ \varepsilon;\Gamma_1\triangleright x:\mathbf{tag}^{\uparrow\alpha}(\tau',\tau_2)\quad \varepsilon\triangleright\sigma_1=\tau'[\sigma/\alpha]\Rightarrow\varepsilon;\Gamma_1\triangleright b_1:\tau\ \Delta;\Gamma_1\triangleright b_2:\tau}{\varepsilon;\Gamma_1\triangleright\mathbf{if}\ x[\sigma]=x[\sigma]\ \mathbf{then}\ b_1\ \mathbf{else}\ e_2\ \mathbf{fi}:\tau}$.由于值的相等,$x$ 的σ实例即为 $\mathbf{tag}^{\downarrow\sigma}(\tau'[\sigma/\alpha],\tau_2)$(定型规则 R11),又因为对于这种标签类型没有子定型关系,在 x 和它的实例类型推断之间没有子定型包含(subsumption)规则的作用,所以可以直接得到$\sigma_1=\tau'[\sigma/\alpha]$.那么根据推断的条件, $\varepsilon;\Gamma_1\triangleright b_1:\tau$ 成立.因此这种情况下,Γ_2 选择ε即可.

ι= **if** $x=y[\sigma]$ **then** b_1 **else** b_2:这种情况下 $e=b_2$ 且 $H_1=H_2$.同上一种情况,Γ_2 选择ε. □

定理 2(前进性). 如果 $\triangleright P$ 且 $P\rightarrow P'$,那么或者 $P=\langle H,v\rangle$,或者存在着 P',使得 $P\rightarrow P'$.

证明:程序 $P=\langle H_1,e\rangle$.如果 e 为一个值 v,那么 P 处于计算终结状态.这种情况显而易见,是符合定理的.若 e 为某种结构化的表达式,那么可以找到某个ι,使得 $e=E[\iota]$,其中ι总是属于以下某种情况:

$$i ::= h \mid \mathbf{if}\ v_1=v_2[\sigma]\ \mathbf{then}\ b_1\ \mathbf{else}\ b_2\ \mathbf{fi} \mid v_1 v_2 \mid v[\tau] \mid \mathbf{unroll}(v) \mid \mathbf{unpack}[\alpha,x]=v\ \mathbf{in}\ e \mid \mathbf{tagapp}[\sigma,v] \mid$$
$$\mathbf{if}\ ?v\ \mathbf{then}\ x.b_1\ \mathbf{else}\ b_2\ \mathrm{fi} \mid \{\vec{e}\} \mid v.i \mid \mathbf{let}\ \overrightarrow{x=e}\ \mathbf{in}\ e \mid v.\mathbf{size} \mid v.i \Rightarrow x.b_1\ \mathbf{else}\ b_2$$

也就是说,可以进行计算归约的情况(通过对语法观察和对 e 的结构分析可证得).P 的良类型推断$\triangleright P$ 需要条件 $\triangleright H_1:\Gamma_1$ 及 $\varepsilon;\Gamma_1\triangleright E(i):\tau$.根据引理 2 后一个条件可以推断,存在某个类型$\sigma'$使得 $\varepsilon;\Gamma_1\triangleright i:\sigma'$.前进性证明的目的是要证明$\iota$总能够在操作语义规则中找到对应的情况,且符合附录 A 中给出的操作语义描述附注的附加条件.证明通过对ι的分情况讨论,并根据各种类型值的规范形式(canonical form)得到.这里同样也对标签相关情况进行证明.

ι= **tag**($[\tau_1,\tau_2],\{\vec{v}\}$):这种情况符合操作语义的第 1 条规则,并不需要满足其他附加条件.

ι=**if** $v_1=v_2[\sigma]$ **then** b_1 **else** b_2:这种情况下,因为有推断 $\varepsilon;\Gamma_1\triangleright i:\sigma'$ 那么,根据定型规则这个推断,必然具有形式 $\dfrac{\ldots\varepsilon;\Gamma_1\triangleright v_1:\mathrm{tag}^{\downarrow\sigma}(\sigma_1,\tau_1)\ \varepsilon;\Gamma_1\triangleright v_2:\mathrm{tag}^{\uparrow\alpha}(\tau'),\tau_2\ldots}{\varepsilon;\Gamma_1\triangleright i:\sigma'}$.从上面定型规则中可以看出,能够推断出类型为 $\mathbf{tag}^{\uparrow\alpha}(\tau',\tau_2)$的值只有某个变量(标签堆数据的某个标识)$x$,而类型为 $\mathbf{tag}^{\downarrow\sigma}(\sigma_1,\tau_2)$也只能是某个标签的$\sigma$实例.这两个推断容易得到,也是因为这两类标签类型没有牵涉到子定型关系的结果.所以ι必然符合附录 A 中给出的操作语义描述中第 3 或第 4 种情况中的某一种,能够继续归约. □

推论 1(类型安全性). 如果 $\triangleright P$,那么 $P\rightarrow^*\langle H',v\rangle$,也就是 P 总能计算到某个终结状态.

证明:根据保持性和前进性定理,对 $\triangleright P$ 推断的步数做归纳即可证得. □

从以上的证明可以看出,这两个标签类型的引入并没有给语言类型安全性的证明增加太多的负担,我们能够将它方便地引入到 TLL 语言当中.