软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2013,24(7):1495−1511 [doi: 10.3724/SP.J.1001.2013.04269]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种利用适合性测试支持方法重定向的演算*

赵银亮, 朱常鹏, 韩 博, 曾庆花

(西安交通大学 电子与信息工程学院,陕西 西安 710049)
通讯作者: 朱常鹏, E-mail: is99zcp@hotmail.com

摘 要: 一些面向上下文的编程语言使用结构化的块结构(block-structured construct)将方法调用重定向到层中方法.但该结构无法支持层的动态添加与激活,这增加了程序可执行文件的大小.为了解决该问题,提出一种新方法:使用适合性测试支持方法的重定向,并定义一个运行时的适合性测试演算(runtime fitness testing calculus on top of featherweight Java calculus)形式化描述该方法.该演算以FJ演算(featherweight Java calculus)为核心,通过融入新的语言结构——层,基于上下文的方法查找与对象转化描述基于适合性测试的方法重定向,分析它对程序类型安全的影响,制定相应约束,并证明在满足该约束的条件下能够保持程序的类型安全,从而证明所提方法的有效性.以该演算为指导,描述如何通过扩展Java的编译器与虚拟机,实现将层、基于上下文的方法查找与对象转化融入到Java语言,并通过实验测试实现,证明所提方法的可行性.该演算及其实现可用于指导如何扩展类似Java(Java-like)的语言以支持程序基于上下文动态调整其行为,并同时保证程序的类型安全.
关键词: 面向上下文的编程;层;适合性测试;featherweight Java 演算;类型系统
中图法分类号: TP311 **文献标识码**: A

中文引用格式: 赵银亮,朱常鹏,韩博,曾庆花.一种利用适合性测试支持方法重定向的演算.软件学报,2013,24(7):1495−1511. http://www.jos.org.cn/1000-9825/4269.htm
英文引用格式: Zhao YL, Zhu CP, Han B, Zeng QH. Calculus using fitness testing for method redirection. Ruan Jian Xue Bao/Journal of Software, 2013,24(7):1495−1511 (in Chinese). http://www.jos.org.cn/1000-9825/4269.htm

Calculus Using Fitness Testing for Method Redirection

ZHAO Yin-Liang, ZHU Chang-Peng, HAN Bo, ZENG Qing-Hua

(School of Electronic and Information Engineering, Xi’an Jiaotong University, Xi’an 710049, China)
Corresponding author: ZHU Chang-Peng, E-mail: is99zcp@hotmail.com

Abstract: Some context-oriented programming languages use block-structured constructs to redirect method invocation. However, these constructs do not support dynamic layer addition and activation, which increases the binary size of the program. To address the problem, this paper proposes a new approach that uses fitness testing to support method redirection, and presents a runtime fitness testing calculus, an extension of featherweight Java calculus with context-based method resolution and object transformation, to describe the approach. Based on the calculus, the paper analyzes the influence approach as on the type safety of the program, imposes constraints on it, and formally proves that it preserves the type safety of the program when these constraints are satisfied, which validates the approach. The paper also shows the implementation of the approach and assesses the implementation, which indicates that the approach is feasible. The calculus and the implementation illustrate how to extend Java-like languages to support runtime behavior adjustment that is context-based and type-safe.
Key words: context-oriented programming; layer; fitness testing; featherweight Java calculus; type system

大量的应用领域,尤其是在移动和普适计算领域,要求程序能够根据上下文动态地调整其行为,这导致程序

* 基金项目: 国家自然科学基金(61173040); 国家高技术研究发展计划(863)(2008AA01Z136)
收稿时间: 2011-12-18; 修改时间: 2012-02-28; 定稿时间: 2012-05-18

© 中国科学院软件研究所 http://www.jos.org.cn

包含行为变体(behavioral variations)[1,2].由于目前主流的编程语言缺少相应的机制封装该类变体,因此它们横切(crosscut)程序的多个模块,导致代码交织和分散(code tangling and scattering),影响了程序的维护与扩展[3].为了解决该问题,面向上下文编程[1](context-oriented programming,简称 COP)提出了新的封装机制——层,行为变种根据上下文封装到不同层(layer)中.层能根据上下文激活(activate)促使方法调用重定向(redirect)到层中相应方法.因此,层和层激活(layer activation)可实现程序根据上下文动态调整其行为,同时保持程序代码简洁,提高了程序的可维护性.

一些 COP 语言已实现,如文献[4−6].在大多数 COP 语言中,层被表示为新的语言结构,行为变种视为依赖上下文的行为(context-dependent behavior),被表示为部分方法(partial methods),并被封装于层中.层可使用结构化的块结构(block-structured construct)激活.块结构在编译阶段被转化为基于上下文的消息派送,实现将方法调用重定向到层中的方法.通过在不同的上下文中使用块结构,可实现程序根据上下文动态地调整其行为.

块结构的语义简单且容易实现,但缺点是:1) 阻碍层的动态添加.在程序开发阶段,影响程序的上下文不完全可知[7],因此需要程序能够动态地增加新层,使之能够根据新的上下文动态地调整其行为,但块结构在编译阶段已被转化为基于上下文的消息派送,因此无法将方法调用重定向到新添加的层中的方法,阻碍了层的动态添加;2) 无法分离上下文的改变与依赖上下文的方法的执行[2,8],限制了这些 COP 语言的应用范围[2].

针对块结构存在的不足,本文提出一种新方法:使用运行时的适合性测试(fitting testing)支持方法重定向.该方法在运行时先查找可用于当前上下文的层,若查找到,则在该层中查找被调方法,实现将方法调用重定向到层中方法;否则,先在指定路径或远程服务器端获取可用于当前上下文的层,并将层中的层化类(layered class)与相应类组合,产生组合类(composed class),转化方法调用的接受者为组合类的对象,然后在新获取的层中查找被调方法,实现将方法调用重定向到新获取的层中的方法.由于适合性测试并不改变上下文,仅根据当前上下文在相应层中查找方法,实现方法重定向,因此,该方法能够实现上下文的改变与依赖上下文方法执行的分离.

实现适合性测试的关键在于如何实现将方法调用重定向到新添加的层中的方法,并确保重定向保持(preserve)程序的类型安全.鉴于此,我们定义一个运行时的适合性测试演算(runtime fitness testing on top of featherweight Java calculus,简称 RFT-FJ 演算),它以 Featherweight Java(简称 FJ 演算)[9]为核心,通过加入层、基于上下文的方法查找和对象转化到 FJ 演算中,描述适合性测试.基于该演算,我们能够有效地分析适合性测试对程序类型安全的影响,并根据分析、定义对层化类和对象转化的约束,证明在该约束下,由适合性测试触发的方法重定向能够保证程序的类型安全.我们也以该演算为指导,主要扩展 Java 虚拟机的类加载、解析和执行等模块实现基于上下文的方法查找和对象转化,证明适合性测试的可行性.本文提出的演算和实现可用于指导如何扩展类似 Java(Java-like)的语言以支持程序基于上下文动态调整其行为,并同时保证程序的类型安全.

本文第 1 节通过一个简单的实例阐述适合性测试.第 2 节主要描述 RFT-FJ 演算,包括语法、语义、类型系统和性质.第 3 节首先描述如何以 RFT-FJ 演算为指导,扩展 Java 编译器和虚拟机实现 RFT-FJ 演算描述的基于上下文的方法查找和对象转化,证明适合性测试是可行的;然后,通过一个应用程序验证实现.第 4 节讨论相关工作.第 5 节是结论.

1 适合性测试

本节首先通过一个应用程序描述当前 COP 语言存在的不足,然后提出使用适合性测试解决此类问题,最后描述实现适合性测试需解决的问题.该程序是一个基于移动设备的上下文感知应用程序,用于以不同形式提供用户位置信息.该程序根据电池电量为用户创建一个相应的界面,比如当电量充足时,创建一个用于以图形方式显示用户位置信息的界面;当电量不足时,创建一个用于以语音或文本形式显示用户位置信息的界面.并且程序能够在运行时动态地加载层,从而实现执行文件大小的减少,以便能运行于更多环境,比如移动设备.

在该程序中,界面显示是一个依赖上下文的行为,图 1 展示了根据 COP 编程范式编写的界面显示方法.类 *Display* 中的 *CreateUI* 方法创建一个用户界面;类 *DisplayActionListener* 监听界面中按钮的点击事件,*actionPerformed* 方法则处理对应的事件.*CreateUI* 方法代码被分割,其中,为电量充足时建用户界面的代码置

© 中国科学院软件研究所 http://www.jos.org.cn

于层化类 *Display* 的 *CreateUI* 方法中,而层化类 *Display* 定义在层 *G_Suff* 中.层 *G_Low* 与层 *G_Suff* 类似. *actionPerformed* 方法代码的分割方式与类 *Display* 中的 *CreateUI* 方法类似.类 *Battery* 中的 *power* 方法返回当前移动设备的电池电量状态.

```
class Battery{
   public static String power(){...}
}
class Display extends{
   public Display(){super();}
   public void createUI(){...}
}
class DisplayActionListener implements ActionListener{
   private JPanel panel;
   private ServerInfo server;
   private Display display;
   public DisplayActionListener(JPanel p,ServerInfo s, Display
c)
    {super();...}
   public void actionPerformed(ActionEvent e){...}
}
layer G_Suff{
   boolean fitness(){return (Battery.power()).equals("Suff");}
   layered class Display{
      ...
      public Display(){super();}
      public void CreateUI(){...}
   }
   ...}
```

```
layer G_ DisplayActionListener_Suff{
   boolean fitness(){return (Battery.power()).equals("Suff");}
   layered class DisplayActionListener{
      ...
      public DisplayActionListener(){super();}
      public void actionPerformed(ActionEvent e){...}
   }
   ...
}
layer G_Low{
   boolean fitness(){
      return (Battery.power()).equals("Low");
   }
   layered class Display{...}
      ...
}
layer G_DisplayActionListener_Suff{
   boolean fitness(){
   return (Battery.power()).equals("Low");}
layered class DisplayActionListener{...}
   ...
}
with(G_Suff){CreateUI()}
```

Fig.1 The code snippet of a context-aware application

图 1 一个上下文感知程序的代码片段

为了根据电池电量显示不同的界面,必须使用块结构语句,比如图 1 中深色阴影部分中的代码.该语句激活层 *G_Suff*,然后将 *CreateUI* 方法调用重定向到层中的同特征(signature)方法,实现电量充足时的界面显示.块结构的语义简单且使用方便,但它不支持对层的动态添加.比如在运行时,添加层 *G_Low* 到程序中,由于块结构无法将 *CreateUI* 方法调用重定向到该层中的同特征方法,导致程序无法通过动态添加新层来适应新的上下文,这也导致所有出现在块结构语句中的层都必须封装到程序的可执行文件中,增加了程序的可执行文件的大小.

为了支持层的动态添加,我们提出运行时的适合性测试,其实质是将层激活从程序文本中分离出来,由运行系统在运行时根据当前上下文选择加载所需层,并在该层中查找被调方法,将方法调用重定向到该层中的方法,支持层的动态添加.比如,假设图 1 中块结构语句不再是程序的一部分,当电量为“Suff”,并且 *O* 是类 *Display* 的一个对象时.当 *O* 的 *CreateUI* 方法被调用时,运行系统根据电池电量状态,加载层 *G_Suff*,然后在该层中查找被调方法.为了支持层的动态添加,需要解决的第 1 个关键问题是:如何使运行系统能够根据电池电量状态决定需被添加的层.由于被调方法 *CreateUI* 的接受者为 *O*,而被查找到的方法是层中的 *CreateUI* 方法而不是 *O* 的方法,因此,当运行系统执行层中的 *CreateUI* 方法时可能导致运行错误(runtime error).所以,第 2 个需解决的关键问题是:如何使新添加的层成为程序的一部分,比如使层 *G_Suff* 中的 *CreateUI* 方法成为对象 *O* 的方法.适合性测试分别使用基于上下文的方法查找和对象转化解决这两个问题.

- 基于上下文的方法查找

我们首先扩展层的定义:每个层都包含一种谓词方法 *fitness*.它描述一个特定上下文,并用于判断当前的上下文是否匹配这一特定上下文[10].如果 *fitness* 返回 true,则匹配,层中的方法可用于当前的上下文;否则,不可以.如图 1 中浅色阴影部分是 *fitness* 的定义,被添加到程序中,其中,层 *G_Suff* 的 *fitness* 描述如果当前上下文匹配电池电量充足这一特定上下文,则该层中的方法可用于当前上下文.

- 对象转化

© 中国科学院软件研究所 http://www.jos.org.cn

我们首先组合相关对象的类与新添加的层中的同名类,产生新类,其中,新类的成员由类和层化类的成员组成,然后转化该对象为新类的对象.如当层 *G_Suff* 被添加时,类 *Display* 与该层中的类 *Display* 组合成新类,然后转化 *O* 为新类的对象 *O′*.因此,发生在 *O* 上的 *CreateUI* 方法调用转变为 *O′*上的同特征方法调用.基于上下文的方法查找,层化类 *Display* 的 *CreateUI* 方法被执行.由于 *O′*是新类的对象,因此执行不会导致运行错误.

适合性测试可能导致新层被添加和已创建的对象被转化.由于对象转化会改变包含被转化对象的表达式的语义,因此,程序在编译阶段被确定的(verified)类型安全性质可能在运行时被破坏.由此,需对适合性测试制定详细约束,以保证其能够保持程序的类型安全.在下一节,我们提出运行时的适合性测试演算,基于该演算能够有效地分析适合性测试对程序类型安全的影响,制定相应约束,并证明在满足该约束的条件下,能够保持程序的类型安全.

2 运行时的适合性测试演算

本节提出一个运行时的适合性测试演算——RFT-FJ 演算,包含其语法、辅助方法、操作语义和类型规则这 4 部分.本节重点说明每部分的核心,它们构成 RFT-FJ 演算的核心.本节最后研究该演算的性质并加以证明.

2.1 语 法

FJ 演算是一个基于 Java 的最小核心演算.它虽然只建模 Java 中的 5 种表达方式:对象创建、方法调用、域访问、类型转换和变量的读写,但却能对 Java 的核心功能进行模拟,并且其本身构成一个完备的系统.RFT-FJ 演算通过引入新的语言结构——层、基于上下文的方法查找和对象转化对 FJ 演算进行扩展,形式化地描述适合性测试.

图 2 中列出了 RFT-FJ 演算的抽象语法,其语法的核心是层和层化类声明语法.元变量 A,B,C,D,E 表示类名,f 和 g 表示域名,m 表示方法名,x 表示变量名,e 表示表达式;L 表示类声明,K 表示构造函数声明,M 表示方法声明,$\overline{f}$ 表示可能为空的序列 $f_1,\dots,f_n$($\overline{M},\overline{T},\overline{x}$ 等与此类似),$\overline{Tf}$ 表示可能为空序列对 $T_1f_1,\dots,T_nf_n,\overline{Tf}$;表示可能为空的域声明序列 $T_1f_1;\dots;T_nf_n$,GL 表示层声明,F 表示谓词方法 *fitness* 的声明,LC 表示层化类声明,G 表示层名,C_G 和 D_G 表示层化类名,u,v,w 表示值,T 表示类型.我们假设序列中不包含同名元素,类 *Object* 是所有类的超类,并且它既没有域和方法,也不包含于类表 CT 中.

一个层声明 layer $G\{F;\overline{LC}\}$定义一个层 G.声明中的 F 和 $\overline{LC}$ 分别定义一个无参且返回 Boolean 型的方法 *fitness* 和一组层化类.方法 *fitness* 也只包含一条 return e 语句,用于描述特定上下文(specific context).

一个层化类声明 layered class $C\{\overline{Cf};K_l;\overline{M}\}$定义一个层化类 C,记为 C_G.层化类没有超类,但有一个基类——由 L 定义的同名类.C_G 由一组类型为 $\overline{C}$ 的域 $\overline{f}$、构造函数 K_l 和一组方法 $\overline{M}$ 组成.K_l 使用初始化 $\overline{f}.\overline{M}$ 依赖由 *fitness* 描述的特定上下文,其使用范围如前文所述.层化类仅用于类的组合,产生组合类,生成组合类对象,因此,程序员禁止实例化层化类.为了保证类型安全,组合类对象中定义在层化类中的域被初始化为给定类型的中性值[11].

本文使用三元组(CT,LT,e)表示一个程序.表达式 e 可以是变量、对象创建、域访问、方法调用和 if 条件语句,e 的值可以是 true、false、类对象或组合类对象,类表 CT 是记录类及其声明对应关系的表,该表映射类 C 到其声明 L,即 $CT(C)$=class C extends $D\{\dots\}$,层表 LT 与 CT 类似,它们可形式化定义如下:

$$CT=\begin{cases}\varnothing, & \text{如果没有类声明}\\ CT\cup\{(C,\text{class } C \text{ extends } D\{\dots\})\}, & \text{否则}\end{cases},$$

$$LT=\begin{cases}\varnothing, & \text{如果没有类声明}\\ LT\cup\{(G,(fitness(),\text{layered class } C\{\dots\},\dots))\}, & \text{否则}\end{cases}.$$

- 类型与子类型关系

RFT-FJ 演算包含 4 种类型:布尔型、类类型、层化类类型和组合类类型,其中,布尔型用于 *fitness* 方法,类类型用于域和方法的声明.与 FJ 演算一样,在 RFT-FJ 演算中,类名本身表示一种类型,因此,类名、层化类类名和组

© 中国科学院软件研究所 http://www.jos.org.cn

合类类名分别表示相应类型.组合类类型仅在运行时由运行系统根据需要产生,并由类型系统通过类名组合(composition of class names)[12]记录.与文献[12]一样,我们使用符号::表示组合类类型,比如,$\overline{C}_G::C$ 表示由一组层化类 $\overline{C}_G$ 和类 C 所组成的组合类类型,而它的对象表示为 new $\overline{C}_G::C(\overline{v})$.由于组合类仅产生于运行时,因此,RFT-FJ 演算的语法不包含类的组合操作,组合类类型也不能用于域和方法的声明,该限制与文献[12]相同.

RFT-FJ 演算的子类型关系如图 3 所示,其核心是组合类类型的子类型关系,图中符号<:表示子类型关系.RFT-FJ 演算的子类型关系是自反和传递闭包.由于组合类的域和方法由其所有构成者的域和方法组成,因此,组合类类型是其所有构成者类型的子类型.此外,当一组可能为空的层化类序列同时与具有子类型关系的两个类组合或仅与子类组合时,该子类型关系保持不变;但不同组合类之间不存在子类型关系,比如 $C<:D$,即 C 是 D 的子类,那么 $C_G::C<:C_G::D$,$C_G::C<:D$.若 C 非 D 的子类,则 $C_G::C$ 和 $C_G::D$ 不存在子类型关系.由于 $\overline{C}_G$ 可以为空,记作ε,我们约定$\varepsilon::C=C$,$C_G::\varepsilon::C=C_G::C$,因此,图 3 中 $\overline{C}_G::C$ 可以统一地表示组合类和类.

注意:因为 Java 中的方法无法作为参数被传递,所以 RFT-FJ 演算不包含方法的子类型关系,这与 FJ 演算相一致.

$$
\begin{aligned}
L &::= \text{class } C \text{ extends } D\{\overline{C f};K\overline{M}\}\\
M &::= Cm(\overline{C x})\{\text{return } e;\}\\
K &::= C(\overline{C' g},\overline{C f})\{super(\overline{g});this.\overline{f}=\overline{f};\}\\
GL &::= \text{layer } G\{F;\overline{LC}\}\\
F &::= \text{boolean } fitness()\{\text{return } e;\}\\
LC &::= \text{layered class } C\{\overline{C f};K_l\overline{M}\}\\
K_l &::= C(\overline{C f})\{this.\overline{f}=\overline{f};\}\\
e &::= x \mid \text{if } (e_1)e_2 \text{ else } e_3 \mid \text{new } C(\overline{e}) \mid e.f \mid e.m(\overline{e})\\
v &::= \text{true} \mid \text{false} \mid \text{new } C(\overline{v}) \mid \text{new } \overline{C}_G::C(\overline{v})\\
T &::= \text{boolean} \mid C \mid C_G \mid \overline{C}_G::C
\end{aligned}
$$

Fig.2 RFT-FJ calculus: Syntax

图 2 RFT-FJ 演算:语法

$$
T<:T \qquad \frac{T_1<:T_2 \quad T_2<:T_3}{T_1<:T_3}
$$

$$
\frac{C \text{ extends } D\{\ldots\}}{C<:D}
$$

$$
\frac{\overline{C}_G::C<:\overline{C}_G::D}{\overline{C}'_G::\overline{C}_G::C<:\overline{C}'_G::\overline{C}_G::D}
$$

$$
\frac{\overline{C}_G::C<:\overline{C}_G::D}{\overline{C}'_G::\overline{C}_G::C<:\overline{C}_G::D}
$$

$$
\begin{aligned}
C_{G_n}::\ldots C_{G_1}::C &<: C_{G_{n-1}}::\ldots C_{G_1}::C\\
&<: C_{G_n}::C_{G_{n-2}}::\ldots C_{G_1}::C\\
&<: \ldots\\
&<: C_{G_n}::\ldots::C_{G_2}::C
\end{aligned}
$$

Fig.3 RFT-FJ calculus: Subtyping rules

图 3 RFT-FJ 演算:子类型关系

2.2 辅助方法

在描述 RFT-FJ 演算的归约和类型规则之前,我们首先定义若干辅助函数,即域查找函数(*fields*)、方法重写函数(*override*)、方法类型查找函数(*mtype*)、方法体查找函数(*mbody*)和对象转化函数(*trans*).图 4 展示它们的定义,其中最主要的函数是 *mbody* 和 *trans*,它们分别形式化地描述了基于上下文的方法查找和对象转化.*fields*, *override* 和 *mtype* 与 FJ 演算中同名函数类似,因此本节仅对它们作简要说明,主要说明 *mbody* 和 *trans*.

fields,*override* 和 *mtype* 是对 FJ 演算中同名函数的扩展,主要添加用于层化类和组合类的操作.*fields* 返回指定类(包括层化类和组合类)的域;*mtype* 返回指定类(包括层化类和组合类)中指定方法的类型;*override* 不仅用于判断定义在子类中的重写方法的有效性,也用于判断定义在层化类中的重写方法的有效性:如果定义在类的子类或层化类中的方法 m 重写该类中的方法 m,并且它们的类型相同,那么重写方法 m 有效.

mbody 主要返回方法的定义.它或者返回指定类中指定方法的定义,或者基于上下文返回层化类中指定方法的定义.如方法调用 $e.m()$,如果 e 的类型为类 C,那么 *mbody* 首先在类 C 中查找 m.如果类 C 定义 m,则返回 m 的定义;否则,递归地在类 C 的父类中查找 m.如果 e 的类型为组合类 $\overline{C}_G::C$,那么,*mbody* 先自左向右地遍历层化类序列 $\overline{C}_G$.如果当前的层化类定义了 m,并且定义该类的层适用于当前上下文,即 C_G IN G 且 *fitness*() true IN G,则 *mbody* 返回该类中 m 的定义;否则,递归地遍历下一个层化类.如果 $\overline{C}_G$ 为空,则 *mbody* 直接返回空.

trans 转化对象 new $\overline{C}_G::C(\overline{v})$ 转化为组合类 $C_G::\overline{C}_G::C$ 的对象.具体而言,*trans* 首先创建一个新对象 new $C_G::\overline{C}_G::C(\overline{v})$,其中,层化类 C_G 的构造函数被调用初始化它的域为给定类型的中性值,以保证类型安全;然

© 中国科学院软件研究所 http://www.jos.org.cn

后,复制 new $\overline{C}_G::C(\overline{v})$ 中域的值到 new $C_G::\overline{C}_G::C(\overline{v})$ 的同名同类型域中,保证对象的数据不会因转化而丢失. *trans* 表明,在转化对象之前,层化类 C_G 需与组合类 $\overline{C}_G::C$ 组合成新的组合类,在实现中,这由运行系统来实现.

方法体查找

$$\frac{\text{class } C \text{ extends } D\{\overline{Cf};K\overline{M}\}\ B\ m(\overline{Bx})\{\text{return } e;\}\in\overline{M}}{mbody(m,C)=(\overline{x},e)}$$

$$\frac{\text{class } C \text{ extends } D\{\overline{Cf};K\overline{M}\}\ m\notin\overline{M}}{mbody(m,C)=mbody(m,D)}$$

$$\frac{\text{layered class } C\{\overline{Cf};K_l\overline{M}\}\ B\ m(\overline{Bx})\{\text{return } e;\}\in\overline{M}\quad C_G\text{ IN } G\ fitness()\text{ true IN } G}{mbody(m,C_G)=(\overline{x},e)}$$

$$\frac{\text{layered class } C\{\overline{Cf};K_l\overline{M}\}\ m\in\overline{M}\quad C_G\text{ IN } G\ fitness()\text{ false IN } G}{mbody(m,C_G)=nil}$$

$$\frac{\text{layered class } C\{\overline{Cf};K_l\overline{M}\}\ m\notin\overline{M}}{mbody(m,C_G)=nil}$$

$$\frac{mbody(m,C_{G_n})=(\overline{x},e)}{mbody(m,C_{G_n}::\ldots::C_{G_1}::C)=(\overline{x},e)}$$

$$\frac{mbody(m,C_{G_n})=nil}{mbody(m,C_{G_n}::\ldots::C_{G_1}::C)=mbody(m,C_{G_{n-1}}::\ldots::C_{G_1})}$$

$$\frac{\overline{C}_G=\varepsilon}{mbody(m,\overline{C}_G)=nil}$$

对象转化

$$\frac{fields(\overline{C}_G::C)=\overline{Cf}}{\text{new }\overline{C}_G::C(\overline{v})^{trans(C_G)}=\text{new } C_G::\overline{C}_G::C(\overline{v});\ \{C_i\text{ new } C_G::\overline{C}_G::C(\overline{v}).f_i\leftarrow C_i\text{ new }\overline{C}_G::C(\overline{v}).f_i\}^{i\in 1,\ldots,n;}}$$

方法类型查找

$$\frac{\text{class } C\text{ extends } D\{\overline{Cf};K\overline{M}\}\quad B\ m(\overline{Bx})\{\text{return } e;\}\in\overline{M}}{mtype(m,C)=\overline{B}\to B}$$

$$\frac{\text{class } C\text{ extends } D\{\overline{Cf};K\overline{M}\}\ m\notin\overline{M}}{mtype(m,C)=mtype(m,D)}$$

$$\frac{\text{layered class } C\{\overline{Cf};K_l\overline{M}\}\ B\ m(\overline{Bx})\{\text{return } e;\}\in\overline{M}}{mtype(m,C_G)=\overline{B}\to B}$$

$$\frac{mtype(m,C_{G_n})=\overline{B}\to B}{mtype(m,C_{G_n}::\ldots::C_{G_1}::C)=\overline{B}\to B}$$

$$\frac{mtype(m,C_{G_n})=nil}{mtype(m,C_{G_n}::\ldots::C_{G_1}::C)=mtype(m,C_{G_{n-1}}::\ldots::C_{G_1}::C)}$$

域查找

$$fields(Object)=\bullet$$

$$\frac{\text{class } C\text{ extends } D\{\overline{Cf};K\overline{M}\}\ fields(D)=\overline{C'g}}{fields(C)=\overline{C'g}\,\overline{Cf}}$$

$$\frac{\text{layered class } C\{\overline{Cf};K_l\overline{M}\}}{fields(C_G)=\overline{Cf}}$$

$$\frac{fields(C_{G_n})=\overline{C'g}\ fields(C_{G_{n-1}}::\ldots::C)=\overline{Cf}}{fields(\overline{C}_G::C)=\overline{Cf}\,\overline{C'g}}$$

有效的方法重写

$$\frac{mtype(m,C)=\overline{B'}\to\overline{B}\ impiles\ \overline{B}=\overline{B'}\ B=B'}{override(m,C,\overline{B}\to\overline{B})}$$

Fig.4 RFT-FJ calculus: Auxiliary functions

图 4 RFT-FJ 演算:辅助方法

2.3 语 义

归约关系形如 $e\to e'$,表示表达式 e 可以通过一步归约为 e'.$\to^*$表示$\to$的自反传递闭包.RFT-FJ 演算的归约规则可分为计算规则(computation rule)与等价规则(congruence rule),它们的定义分别展示在图 5 的左边与右边,其中,方法调用表达式规则是计算规则的核心.等价规则的含义直观,所以本节不详细说明.本节主要说明计算规则中的方法调用表达式规则,它体现了程序的执行特点.

条件表达式规则(E-IF-T 和 E-IF-F). 如果 if 条件语句中的条件表达式是值 true,那么规则 E-IF-T 归约该表达式为它的第 1 分支语句 e_2;否则,规则 E-IF-F 归约该表达式为它的第 2 分支语句 e_3.

域访问表达式规则(E-PROJ). 如果类 $\overline{C}_G::C$ 有对应的域,则该规则归约域访问表达式为该域的值.该规则同时隐含地要求首先将表达式 new $\overline{C}_G::C(\overline{e})$ 归约为值 new $\overline{C}_G::C(\overline{v})$.

方法调用表达式规则(E-INVK,E-LINVK,E-LINVK$_2$和 E-LINVK$_3$). 这组计算规则包含了在第 1 节中讲述的适合性测试的形式化描述,即包含了基于上下文的动态层加载和方法调用重定向的形式化描述,同时也阐明该测试是基于对象(on a per-object basis).如方法调用 new $\overline{C}_G::C(\overline{v}).m(\overline{u})$, 计算规则在归约它时,首先使用基于上下文的方法查找策略在层化类列表 $\overline{C}_G$ 中查找可用于当前上下文的 m,如果查找成功,则归约方法调用为 m 的定义,同时,实参 $\overline{u}$ 替换形参 $\overline{x}$,new $\overline{C}_G::C(\overline{v})$ 替换 *this*,实现将方法调用重定向到层中的方法(E-LINVK);如果查找失败,则试图获取并加载所需层,然后,将层中相应的层化类与 $\overline{C}_G::C$ 组合成新的组合类,同时转化对象 new $\overline{C}_G::C(\overline{v})$ 为新组合类的对象,使其获取可用于当前上下文的 m,最后归约方法调用为新获取的 m 的定义,

© 中国科学院软件研究所 http://www.jos.org.cn

同时,替换形参 $\overline{x}$ 和 *this*(E-LINVK$_2$).如果无法获取到所需层,则归约方法调用为类 C 中的 m 的定义,同时替换形参 $\overline{x}$ 和 *this*(E-LINVK$_3$),这样可避免方法调用归约失败.如果 $\overline{C}_G$ 为空,因为类中的方法与上下文无关,所以直接归约方法调用为类 C 中的 m 的定义,同时替换形参 $\overline{x}$ 和 *this*(E-INVK).

与 FJ 演算一样,RFT-FJ 演算的归约规则并不包含对变量的归约,这是因为替换操作(substitution)用于归约变量.替换操作源于λ演算(lambda-calculus),主要用于替换项(term)中的变量,它的形式化定义可参见文献[13].例如,E-INVK 规则将方法体 e 中的变量 $\overline{x}$ 替换为值 $\overline{u}$.第 2.4 节中的定理 3 保证,经过有限次替换,变量最终可被替换为值.

计算规则(computation rules)

$$\text{if (true) } e_2 \text{ else } e_3 \rightarrow e_2 \quad \text{E-IF-T}$$

$$\text{if (false) } e_2 \text{ else } e_3 \rightarrow e_3 \quad \text{E-IF-F}$$

$$\frac{fields(\overline{C}_G :: C) = \overline{T}\overline{f}}{\text{new } \overline{C}_G :: C(\overline{v}).f_i \rightarrow v_i} \quad \text{E-PROJ}$$

$$\frac{mbody(m, \overline{C}_G) = (\overline{x}, e)}{\text{new } \overline{C}_G :: C(\overline{v}).m(\overline{u}) \rightarrow [\overline{x} \mapsto \overline{u}, this \mapsto \text{new } \overline{C}_G :: C(\overline{v})]e} \quad \text{E-LINVK}$$

$$\frac{mbody(m, \overline{C}_G) = nil \;\; mbody(m, C_G) = (\overline{x}, e) \quad \exists G: fitness() \text{ true IN } G \;\; m \in C_G \;\; C_G \text{ IN } G}{\text{new } \overline{C}_G :: C(\overline{v}).m(\overline{u}) \rightarrow [\overline{x} \mapsto \overline{u}, this \mapsto \text{new } \overline{C}_G :: C(\overline{v})^{trans(C_G)}]e} \quad \text{E-LINVK}_2$$

$$\frac{mbody(m, \overline{C}_G) = nil \;\; mbody(m, C) = (\overline{x}, e) \quad \neg(\exists G: fitness() \text{ true IN } G \;\; m \in C_G \;\; C_G \text{ IN } G)}{\text{new } \overline{C}_G :: C(\overline{v}).m(\overline{u}) \rightarrow [\overline{x} \mapsto \overline{u}, this \mapsto \text{new} C(\overline{v})]e} \quad \text{E-LINVK}_3$$

$$\frac{mbody(m, C) = (\overline{x}, e)}{\text{new } C(\overline{v}).m(\overline{u}) \rightarrow [\overline{x} \mapsto \overline{u}, this \mapsto \text{new} C(\overline{v})]e} \quad \text{E-INVK}$$

等价规则(congruence rules)

$$\frac{e_1 \rightarrow e_1'}{\text{if } (e_1)\ e_2 \text{ else } e_3 \rightarrow \text{if } (e_1')\ e_2 \text{ else } e_3} \quad \text{EC-IF}$$

$$\frac{e \rightarrow e'}{e.f_i \rightarrow e'.f_i} \quad \text{EC-PROJ}$$

$$\frac{e_i \rightarrow e_i'}{\text{new } C(\overline{v}, e_i, \overline{e}) \rightarrow \text{new } C(\overline{v}, e_i', \overline{e})} \quad \text{EC-NEWARG}$$

$$\frac{e \rightarrow e'}{e.m(\overline{e}) \rightarrow e'.m(\overline{e})} \quad \text{EC-INVKREV}$$

$$\frac{e_i \rightarrow e_i'}{\text{new } \overline{C}_G :: C(\overline{v}).m(\overline{u}, e_i, \overline{e}) \rightarrow \text{new } \overline{C}_G :: C(\overline{v}).m(\overline{u}, e_i', \overline{e})} \quad \text{EC-INVKARG}$$

Fig.5 RFT-FJ calculus: Reduction rules

图 5 RFT-FJ 演算:规约规则

2.4 类型化

为了保证适合性测试保持程序的类型安全,我们对表达式、层和层化类制定了详细的约束,它们包含于 RFT-FJ 演算的类型规则(typing rule)中.其类型规则由表达式、方法声明、类声明、层化类声明和层声明的类型规则组成,它们展示在图 6 和图 7 中,其中,方法调用表达式、对象创建表达式、层化类声明和层声明的类型规则是该演算类型规则的核心,形式化地描述我们制定的约束,本节主要对其加以说明.

图 6 展示了表达式的类型规则.每个语法表达式都有对应的类型规则,它们形如$\Gamma \vdash e{:}T$,表示在类型环境(type environment)Γ下可推导出表达式 e 的类型为 T,其中,Γ是一个从变量到类型的映射,可形式化地描述为一个由变量和它们的类型构成的序列[13].$x{:}T$ 表示变量 x 的类型为 T.表达式类型规则是语法制导(syntax- directed),具有直观的含义.如,类型规则 T-TRUE 和 T-FALSE 推导值 true 和 false 的类型为布尔型 boolean,类型规则 T-IF 推导条件表达式的类型为它的两个分支类型的合类型[13],类型规则 T-VAR 推导变量 x 类型为$\Gamma(x)$.下面我们主要说明方法调用表达式类型规则 T-INVK 和对象创建表达式类型规则 T-NEW,它们不仅可用于编译阶段,也可用于运行时的表达式类型检测与推导.

T-INVK:该类型规则检测方法调用表达式是否是良类型(well-typed),同时推导表达式的类型.对于方法调用表达式,该类型规则要求方法接受者的类型定义了该方法,方法声明中的参数类型与返回类型只能为类类型,并且实参类型必须是形参类型的子类型.如果该条件满足,那么该类型规则确定方法调用表达式为良类型,并可推导出该表达式的类型为方法声明中的返回类型.

T-NEW:该类型规则检测对象创建表达式是否是良类型,同时推导表达式的类型.对于创建对象表达式,该类型规则要求表达式中的实参类型是类中域的声明类型的子类型,并且域的声明类型只能是类类型.如果该条

© 中国科学院软件研究所 http://www.jos.org.cn

件得到满足,那么该类型规则确定对象创建表达式为良类型,并可推导表达式的类型为 C.

对于域访问表达式,类型规则 T-FIELD 要求 e 的类型必须定义了该域,并且域的声明类型只能为类类型,但 e 的类型可以是类、层化类或组合类(仅在运行时产生).如果条件满足,那么该类型规则可确定域访问表达式是良类型,并可推导表达式的类型为域的声明类型.类型规则 T-COBJ 推导组合类的对象的类型,它仅用于运行时的对象类型推导,因为组合类及其对象都在运行时产生.此外,由于ε不可类型化[12],因此 T-COBJ 也暗示 $\overline{C}_G \neq \varepsilon$.

表达式类型化

$$\Gamma \vdash \text{true} : \text{boolean} \quad \text{T-TRUE}$$

$$\Gamma \vdash \text{false} : \text{boolean} \quad \text{T-FALSE}$$

$$\frac{\Gamma \vdash e : T \quad fields(T) = \overline{Cf}}{\Gamma \vdash e.f_i : C_i} \quad \text{T-FIELD}$$

$$\Gamma \vdash x : \Gamma(x) \quad \text{T-VAR}$$

$$\frac{\Gamma \vdash e : T \quad mtype(m,T) = \overline{C} \to C \quad \Gamma \vdash \overline{e} : \overline{T'} \quad \overline{T'} <: \overline{C}}{\Gamma \vdash e.m(\overline{e}) : C} \quad \text{T-INVK}$$

$$\frac{\Gamma \vdash e_1 : \text{boolean} \quad \Gamma \vdash e_2 : T_1 \quad \Gamma \vdash e_3 : T_2 \quad T_1 \vee T_2 = T}{\Gamma \vdash \text{if } (e_1)\ e_2 \text{ else } e_3 : T} \quad \text{T-IF}$$

$$\frac{fields(C) = \overline{Cf} \quad \Gamma \vdash \overline{e} : \overline{T'} \quad \overline{T'} <: \overline{C}}{\Gamma \vdash \text{new } C(\overline{e}) : C} \quad \text{T-NEW}$$

组合对象类型化

$$\Gamma \vdash \text{new } \overline{C}_G :: C(\overline{v}) : \overline{C}_G :: C \quad \text{T-COBJ}$$

Fig.6 RFT-FJ calculus: Typing rules

图 6 RFT-FJ 演算:类型规则

图 7 展示类方法声明类型规则(T-CMETHOD)、类声明类型规则(T-CLASS)、层化方法声明类型规则(T-GMETHOD)、层化类声明类型规则(T-LCLASS)和层声明类型规则(T-LAYER),其中,T-GMETHOD,T-LCLASS 和 TLAYER 为新添加的规则,用于检测层化方法声明、层化类声明和层声明是否符合要求.它们包含了为保证程序类型安全所制定的约束,下面对此详细加以说明.

方法类型化(method typing)

$$\frac{\overline{x} : \overline{C}, this : C \vdash e : E \quad E <: C_0 \quad C <: D \quad override(m, D, \overline{C} \to C_0)}{C_0\ m(\overline{C}\,\overline{x})\{\text{return } e;\}\ \text{OK IN } C} \quad \text{T-CMETHOD}$$

层化方法类型化(layered method typing)

$$\frac{LT(G) = \{\ldots, \text{layered class } C\{\ldots\}, \ldots\} \quad override(m, C, \overline{C} \to C_0) \quad \overline{x} : \overline{C}, this : C_G \vdash e : E \quad E <: C_0}{C_0\ m(\overline{C}\,\overline{x})\{\text{return } e;\}\ \text{OK IN } C \text{ IN } G} \quad \text{T-GMETHOD}$$

层类型化(layer typing)

$$\frac{F = \text{boolean } fitness()\{\text{return } e;\} \quad \Gamma \vdash e : \text{boolean} \quad \overline{LC}\ \text{OK IN } G}{G\{F; \overline{LC}\}\ \text{OK}} \quad \text{T-LAYER}$$

类类型化(class typing)

$$\frac{K = C(\overline{C'}\,\overline{g}, \overline{Cf})\{super(\overline{g}); this.\overline{f} = \overline{f};\} \quad fields(D) = \overline{C'}\,\overline{g} \quad \overline{M}\ \text{OK IN } C}{\text{class } C \text{ extends } D\ \{\overline{Cf}; K\overline{M}\}\ \text{OK}} \quad \text{T-CLASS}$$

层化类类型化(layered class typing)

$$\frac{\begin{array}{c} LT(G) = \{\ldots, \text{layered class } C\{\overline{Cf}; K_l\overline{M}\}, \ldots\} \\ CT(C) = \text{class } C \text{ extends } D\{\overline{C'}\,\overline{f'}; K\overline{M'}\} \\ \overline{M} \subseteq \overline{M'} \quad fields(C) \cap fields(C_G) = \varnothing \\ K_l = C(\overline{Cf})\{this.\overline{f} = \overline{f};\}\ \overline{M}\ \text{OK IN } C_G \end{array}}{\text{layered class } C\{\overline{Cf}; K_l\overline{M}\}\ \text{OK IN } G} \quad \text{T-LCLASS}$$

Fig.7 RFT-FJ calculus: Typing rules (continue)

图 7 RFT-FJ 演算:类型规则(续)

T-GMETHOD:该类型规则用于检测层化类中的方法声明是否是良形式(well-formness),规则中的 M OK IN C IN G 表示层 G 中的层化类 C 的方法 m 的声明是良形式.该规则形式化表示层化类 C 中的 m 的声明为良形式需满足:1) 如果 m 的参数 $\overline{u}$ 类型为 $\overline{C}$、返回类型为 C_0,那么根据用于表达式的类型规则可推导出 m 的方法体 e 的类型为 E 且为 C_0 的子类型;2) 如果 m 为重写方法,那么 m 必须与类 C 中的 m 的类型相同.

T-LCLASS:该类型规则用于检测层化类的声明是否是良形式,规则中的 LC OK IN G 表示层 G 中的层化类 C 声明是良形式.该规则形式化表示层 G 中的层化类 C 的声明为良形式需满足:1) 层化类中的域被正确地初始化,并且不允许和类 C 中的域同名;2) 层化类 C 中的方法声明都是良形式,并且必须是类 C 的重写方法,这样可以保证程序运行时的语义安全[14].

T-LAYER:该类型规则用于检测层的声明是否是良形式,规则中的 GL OK 表示层的声明是良形式的.该规

© 中国科学院软件研究所 http://www.jos.org.cn

则形式化表示层 G 的声明为良形式需同时满足如下条件:1) 根据表达式的类型化规则可推导出层 G 中的方法 *fitness* 的方法体 e 的类型为 boolean;2) 层 G 中的所有层化类声明是良形式的.

规则 T-CMETHOD 和 T-CLASS 源于 FJ 演算,在此仅作简要说明.T-CMETHOD 要求良形式的方法声明需满足:1) 方法体的返回类型必须是方法返回类型的子类型:2) 如果该方法是重写方法,那么它必须与被重写方法的类型相同.T-CLASS 要求良形式的类声明需满足:1) 构造函数必须初始化类 C 及其超类中的域;2) 类 C 的方法都是良形式.

上述 5 个类型规则同时保证:层化类和组合类禁止成为域和方法声明的一部分,这是因为:

1) 由于层化类仅由运行系统用于类的组合产生新对象,为了保持 RFT-FJ 演算的简单化,我们禁止层化类成为域和方法声明的一部分;
2) 由于组合类仅在运行时产生,其类型由类型系统记录,因此,我们也禁止组合类成为域和方法声明的一部分.

2.5 性 质

RFT-FJ 演算具有类型安全(type soundness)这一性质,即一个良类型表达式可最终归约为值.根据标准的保持与进展理论(preservation and progress theorem)[15],该性质可描述为如下两部分:1) 保持性质,即一个良类型表达式的类型在归约过程中被保持;2) 进展性质,即一个良类型表达式在一步归约后或为值或可以进一步归约.这两个性质分别被形式化地描述为定理 1 和定理 2,而 RFT-FJ 演算的类型安全性质被形式化地描述为定理 3.尽管存在差别,但总体而言,定理 1~定理 3 的证明步骤类似于 FJ 演算,附录提供了主要定理的证明,下面我们先引入两个定义[13].

定义 1(闭包表达式(closed expression)). 如果表达式 e 中不包含自由变量,那么 e 称为闭包表达式,其中,自由变量是指出现在 e 中但未被绑定的变量.

定义 2(范式(normal formal)). 如果表达式 e 不能进一步归约,那么 e 称为范式.

定理 1(保持). 如果 $\Gamma \vdash e{:}T$ 并且 $e \rightarrow e'$,那么存在类型 T' 使得 $\Gamma \vdash e'{:}T'$,其中,$T' <: T$.

定理 2(进展). 假设 e 为闭包表达式,如果存在类型 T 使得 $\vdash e{:}T$,并且存在 e' 使得 $e \rightarrow e'$,那么 e' 或者是值或者可进一步归约.

定理 3(RFT-FJ 演算的类型安全). 如果 $\varnothing \vdash e{:}T$ 并且 $e \rightarrow^{*} e'$,其中,e' 是范式,那么 e' 是值 v 并且有 $\varnothing \vdash v{:}T'$ 和 $T' <: T$.

证明:根据定理 1 和定理 2,可直接推出定理 3 成立. □

3 实现和实验

本节展示如何以 RFT-FJ 演算为指导,通过扩展 Java 编译器与虚拟机,融入层、基于上下文的方法查找和对象转化到 Java,实现运行时的适合性测试,证明其可行性,并通过测试和实例评估验证我们的实现.

3.1 实 现

- 编译器

JastAddJ[16]是一个开源 Java 编译器,我们主要扩展它的扫描器、词法分析器、抽象语法树和语义分析模块支持层和层化类的编译,其中,层及其所包含的层化类被编译成标准 Java 字节码(*.class*)文件,层中的方法 *fitness* 被编译成静态方法.由于层化类与其基类同名,所以层化类在编译时被重命名以避免与类名冲突.

- 虚拟机

为了融入基于上下文的方法查找和对象转化到 Java 中,我们扩展 JamVM 的类加载(class loading)模块、内存分配(allocation)模块、解析(resolution)模块和解释器(interpreter)模块,同时整合数据转移模块到 JamVM.我们选择扩展 JamVM 虚拟机是因为此虚拟机不仅符合 Java 虚拟机规范(第 2 版),而且小巧、紧凑,易于扩展.

JamVM 使用 *ClassBlock* 结构体表示类.为了能够统一地表示类和组合类,我们首先扩展该结构体:一个新域

© 中国科学院软件研究所 http://www.jos.org.cn

layered_classes 和一个整型域 *real_size* 添加到该结构中.*layered_classes* 是一个指向指针的指针,用于记录类的所有层化类,当虚拟机加载类时,它被初始化为空;*real_size* 记录当前类的域所需空间,虚拟机根据它的值给对象分配空间,并且每当层化类被加载时,它的值将被重新计算.类的域由它及其所有层化类定义的域、其祖先及祖先的所有层化类定义的域组成.在对象的空间中,类的域的位置位于层化类的域之前;在层化类列表前端的类的域的位置位于后端的类的域之前;父类的层化类的域的位置位于子类的层化类的域之前.

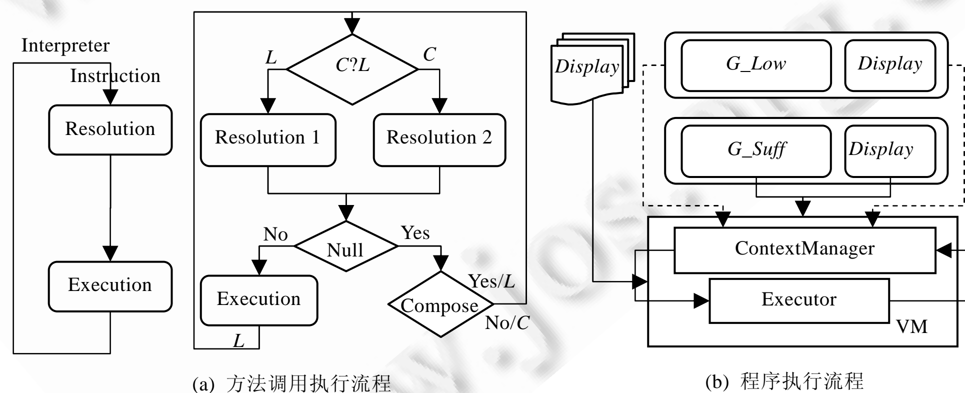
- 基于上下文的方法查找与执行

解释模块负责指令的执行,其中,Java 方法调用指令的执行流程如图 8(a)左边所示.当执行方法调用指令时,如 *invokevirtual* 指令,解析模块首先解析被调方法,获取一个指针,它指向被调方法的方法体(一组指令),然后执行模块执行方法体.

为了实现基于上下文的方法查找与执行,我们扩展解析模块和解释模块,结果如图 8(a)右边所示.解析模块扩展如下:在解析被调用方法之前,先确定是在类还是在层化类列表中解析该方法,其中,类中方法解析流程与标准的 Java 方法解析流程一致.若在列表中解析被调方法,则先从其末端自左向右开始查找被调用方法.如果查找成功,则进行适合性测试,即执行层化类所在层中的 *fitness* 方法:若它返回 true,则解析模块返回一个指针,该指针指向查找到的方法的方法体;否则,在下一个层化类中递归地查找.若无法在列表中找到被调用方法,则返回空,这与 *mbody* 的定义一致.为了减少查找方法的时间开销,我们保持适合当前上下文的层化类位于列表末端.此外,为了减少适合性测试的时间开销,我们将 *fitness* 的结果保存在一个 hash 表中,将 *fitness* 的执行转化为对 hash 中相应值的读取.

解释模块中,方法调用指令语义扩展如下:如果解析模块返回一个非空指针,则执行它所指向的方法体,这与 E-LINVK 和 E-INVK 的语义一致;否则,在指定的目录下查找如下条件的层:1) 可用于当前上下文;2) 包含一个层化类,它定义了与被调用方法同名同类型的方法.如果查找成功,则加载此层,组合层中的层化类与方法接受者的类,产生新的组合类,转化接受者为新组合类的对象,并再次解析执行被调用方法,这与 E-LINVK$_2$ 的语义一致;如果无法查找合适的层,则在类中重新查找被调方法(良类型的程序总能保证查找成功),然后执行返回的方法体,这与 E-LINVK$_3$ 的语义一致.

JamVM 是一个 Java 解释器,并不包含即时编译模块(just-in-time compiler),因此,只需扩展解析模块和解释器模块就可实现当方法下次被调用时,新加载的层化类的同类型方法能被正确地执行,无需重新编译方法的机器码(machine code)[17].此外,正在运行的方法(active method)并不受新加载的层化类的影响,这与文献[18]相似.



(a) 方法调用执行流程 (b) 程序执行流程

Fig.8 Execution process

图 8 执行流程图

- 对象转化

对象转化可分为如下 3 步:

1) 创建一个 *ClassBlock* 变量,由于接受者的类被加载后已被表示为一个 *ClassBlock* 变量,因此复制该变

量的内容到新建变量中,同时添加新加载的层化类到新变量的 *layered_classes* 域,用于表示新生成的组合类.这可通过扩展类加载模块实现;

2) 创建新组合类的对象,并将接受者中域的值赋值给该对象中同名同类型域,保证数据在对象转化过程中不丢失,然后赋初始值给该对象中其他域,比如整型域和指针类型域分别赋值为 0 和空.因为每当层化类被加载时,*real_size* 被重新计算,所以新建对象有空间存储新组合类的所有域的值;

3) 扫描堆栈,将所有指向接受者的指针转指向新建对象,由于没有指针再指向接受者,所以当垃圾回收器再次执行时,它被回收,这样可以避免程序执行中发生内存溢出错误.

步骤 2)和步骤 3)均通过扩展内存分配模块实现.

为了使新建对象中层化类的域能被正确地访问,还需扩展域的解析和读写指令的语义.首先扩展解析模块:先从类的层化类列表的末端自左向右开始查找.如果查找成功,则返回;否则,继续在类中查找(一个良类型的程序可保证这次查找成功).JamVM 使用一个整型变量 *offset* 记录域在对象中的偏移量,其值在编译时确定.在成功查找到域之后,如果被访问的域由类定义,则直接访问;否则,增加 *offset* 的值,增量等于位于此层化类之前的所有层化类的域、类定义的域和类的祖先及其层化类的域所需空间之和.这可通过扩展域读写指令的语义实现.

3.2 实 例

本节通过第 1 节描述的实例程序来验证我们的实现.程序的运行流程如图 8(b)所示,其中,圆角框表示层被编译后产生的字节码文件,执行环境是扩展后的 JamVM 虚拟机.执行模块 Executor 是扩展后的解释模块,主要执行 Java 指令,而上下文管理模块 ContextManager 主要监视位置的改变,并根据当前位置获取相应的层和层化类.

假设虚拟机正在执行程序,其中,程序中的类 *Display* 和 *G_Suff* 中的层化类 *Display* 已组合为组合类,并且类 *Display* 的对象也已转化为组合类的对象.当 Executor 执行类 *Display* 的 *CreateUI* 方法时,基于上下文的方法查找策略,首先,层化类 *Display* 中的 *CreateUI* 方法被返回,实现方法的重定向,然后执行返回方法的体.因此,一个专门用于电量充足时的操作界面被显示.如果在运行中电量的状态发生改变,比如由 *Suff* 变为 *Low*,当 Executor 再次执行类 *Display* 的 *CreateUI* 方法时,基于上下文的方法查找策略,没有方法被返回,因此,Executor 触发 ContextManager 用于获取层 *G_Low* 和它的层化类 *Display*.Executor 将新获取的层化类与被调方法接受者的类型组合,产生新的组合类,并转化接受者为新组合类的对象,然后执行新层化类中的 *CreateUI* 方法.

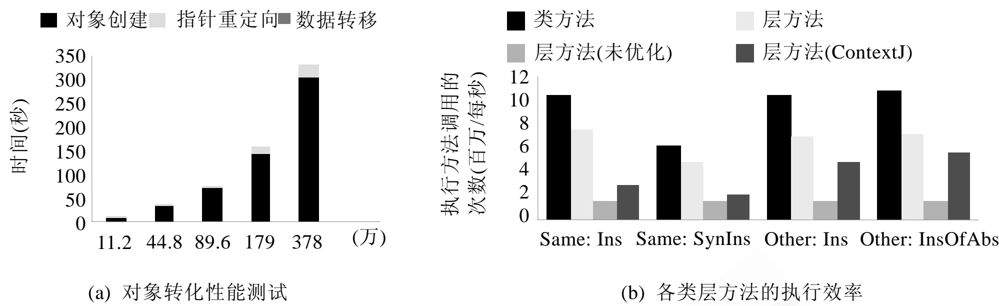
3.3 测 试

本节测试分析我们的实现.测试工具基于 Java Grande Forum Benchmark Suite(JGFBS)[19],并以文献[4,17]为例,开发一组微基准测试程序(micro-benchmark),用于测试层方法调用和对象组合的性能.测试所用计算机的 CPU 为 Intel Pentium Dual 2.2GHz,内存为 3GB,操作系统为 Ubuntu10.04 LTS,虚拟机中堆最大值设置为 1280MB.

3.3.1 对象转化

我们以文献[17]为例,开发一个微基准测试程序以测试对象转化的执行效率.该程序包含一个类和它的一个组合类,其中,类包含 3 个整型域和 2 个引用类型域,而组合类则包含额外的 2 个整型域和 2 个引用类型域.我们分别将 1.12×10^5,4.48×10^5,8.96×10^5,1.792×10^6 和 3.784×10^6 个类的对象转化为组合类的对象.运行测试程序 21 次,获取的测试数据的平均值展示在图 9(a)中,其中,黑色部分表示创建对象的时间开销,浅灰色部分表示重定向指针的开销,而深灰色部分表示复制数据的开销.由于复制数据的时间开销占整个开销的比值过小,因此无法在图中有效地显示.测试数据展示转化 1.12×10^5 个对象需耗时大约为 9.64s,转化 1.792×10^6,则耗时大约为 157s.该数据也展示出创建对象、重定向指针和复制数据分别与转化对象的数量呈线性关系,比如分别创建 1.12×10^5 和 1.792×10^6 个对象需耗时大约 8.93s 和 144.2s.

© 中国科学院软件研究所 http://www.jos.org.cn



(a) 对象转化性能测试 (b) 各类层方法的执行效率

Fig.9 Micro-Benchmark for object transformation and layered method invocation

图 9 对象转化和层方法执行的测试

3.3.2 层化方法的执行

我们以文献[4]为例,开发一种微基准测试程序测试执行类方法和层化方法的时间开销.该程序主要由不同类型的方法调用组成:

- 定义在同一类中的实例方法、同步实例方法;
- 定义在其他类中的实例方法、抽象实例方法.

其中,所有的方法都只包含一条语句:增加一个变量的值.*fitness* 方法返回 false 会中断层化方法的执行,并触发新层的加载和对象转化.这两个操作主要用于实现层的动态加载,所以在测试层化类方法的执行效率时,不应计算这两个操作的时间开销.因此,我们仅测试应用归约规则 E-LINK 执行层化方法的时间开销,即在微基准测试程序中,*fitness* 方法只包含一条返回 true 的语句.

运行该程序 17 次,获取的数据平均值展示在图 9(b)中.测试结果表明,层化方法的执行效率要比原 Java 方法的执行效率低 30%左右.比如,第 1 组数据展示每秒执行定义在同一类中的简单实例方法的次数为 1.04×10^7,而执行对应的层化方法次数则为 7.50×10^6.效率低是因为每执行层化方法都必须做额外的操作:执行 *fitness* 方法.

上一节描述了一种优化方法,可减少遍历层化类列表和执行 *fitness* 方法的时间开销,我们屏蔽掉虚拟机中该优化功能,重复同样的测试,测试结果展示在图 9(b)中,其中,每秒执行 4 种类型层化方法的次数分别为 1.60×10^6,1.46×10^6,1.48×10^6 和 1.48×10^6.这表明,执行层化方法的主要时间开销是遍历层化类列表和执行 *fitness* 方法.

下面我们简要分析比较执行类方法和层化方法的时间空间复杂度.假设执行类方法的时间复杂度函数 $T(n)=f(n)$,其中,$f(n)$为执行 n 次方法调用的时间,那么应用归约规则 E-LINK 执行层化方法的时间复杂度函数可以表示为 $T'(n)=(k+1)\times f(n)$,其中 k 为层化类列表的长度,$k>=1$.因此,执行两种方法的时间复杂度相同,但是减少 k 的值可以减少执行层方法的时间开销,上面的测试证明了这一点.由于在执行层方法过程中无需创建额外的对象,因此执行两种方法的空间复杂度也是相同的.如果在执行层方法时需加载新层,那么应用规则 E-LINVK$_2$ 执行层化方法的时间复杂度函数可表示为 $T'(n)=(k+1)\times f(n)+g(n)+h(p)$,其中,$g(n)$为转化 n 个对象的时间,而 $h(p)$为获取新层的时间,若新层在本地可以忽略 $h(p)$.此外,在这种情况下,每执行一次层化方法都会产生一个额外的空间开销,因此假设执行类方法的空间复杂度函数为 $S(n)=n\times s(k)$,其中,n 为执行方法的次数,$s(k)$表示每执行一次方法调用所需的空间开销,那么执行层方法的空间复杂函数可表示为 $S'(n)=n\times s(k+1)$.应用规则 E-LINVK$_3$ 执行层化方法的时间空间复杂函数与 E-LINVK$_2$ 类似.

3.3.3 比 较

ContextJ[4]提出了一种实现方法重定向的方法:层感知消息传递(layer-aware message dispatch).基于该方法,每次执行层化方法之前需要执行 3 次额外的方法调用和 1 次层列表遍历.基于相同的基准程序,ContextJ 中的层化方法执行效率的测试结果如图 9(b)所示.测试结果表明,基于层感知消息传递,同一类中层方法的执行效率大约只有相应类方法执行效率的 28%,而其他类中层方法的执行效率大约只有相应类方法的执行效率的 50%,该

© 中国科学院软件研究所 http://www.jos.org.cn

测试结果与文献[4]中的测试结果类似.我们的方法执行各类层方法的效率大约只有执行对应类方法的效率的70%.

导致 ContextJ 中的层化类执行方法效率更低的原因是层感知消息传递通过扩展编译器实现,3 次额外的方法调用和 1 次层列表遍历被转化为相应的 Java 语句,虚拟机在执行完这些语句对应的指令后才查找到相应层中的方法.我们的方法是通过扩展虚拟机来实现,它能使虚拟机以更直接的方式找到相应层中的方法.此外,层感知消息传递的实现会导致虚拟机无法优化层方法的执行,这是导致 ContextJ 中同一类中层方法的执行效率要比对应类方法的执行效率低近 72%的原因,而我们方法的实现不会导致这一问题的出现.

4 相关工作

本文工作主要涉及面向上下文的编程和基于 FJ 演算的扩展,而且本文的实现与动态软件更新相关.因此,本节主要讨论这 3 个领域的相关工作.

4.1 面向上下文的编程

ContextL[5]与 ContextJ 分别是对 Common Lisp Object System 和 Java 的面向上下文的扩展,它们都通过层的激活支持程序动态地调整其行为,其中,层激活通过语法块结构实现.但块结构无法实现上下文的改变与依赖上下文的方法的执行的分离,因此限制了它们的应用范围.RFT-FJ 演算通过适合性测试支持程序基于上下文动态地调整其行为,可实现这种分离.此外,ContextJ 与 RFT-FJ 演算的不同还包括:前者是基于 Java 编译器的,层中方法的选择均在编译阶段确定,因此它不支持层的动态加载,而 RFT-FJ 演算中的适合性测试是基于虚拟机的,层中方法的选择在运行时才被确定,因此 RFT-FJ 演算支持层的动态加载;ContextL 支持在运行时引入新的层,但是新引入的层的定义在程序编写时已经确定,而 RFT-FJ 演算并没有此要求,这是两者之间的最大差别.

JCop[8]和 EventCJ[2]是对 Java 的面向上下文的扩展.它们通过指定的事件而不是语言构造触发层的激活与去活,并且使用类似于 AspectJ[20]中的切入点(pointcut)与处理(advice)避免代码混乱.RFT-FJ 演算与其相同之处在于都使用谓词决定代码的可用性.不同之处是:

1) RFT-FJ 演算通过方法的适合性测试触发层的动态加载,转化对象,实现动态的行为调整,而 EventCJ 与 JCop 使用指定的事件触发层的切换实现动态的行为调整;
2) RFT-FJ 演算支持动态地添加新层到程序中,使得程序能适应新的上下文环境,但是 EventCJ 与 JCop 不支持.

RFT-FJ 演算与 AspectJ 存在一些相似之处.面向方面的编程(aspect-oriented programming)[3]已被广泛应用[21,22],AspectJ 是第 1 种对 Java 的面向方面的扩展,它主要使用方面(aspect)封装横切关注(crosscutting concerns),减少了程序代码的分散与纠缠.在 RFT-FJ 演算中,横切关注是特定上下文方法,RFT-FJ 演算使用层封装该类方法,实现相同目的,这是两者相同之处.其不同点是:AspectJ 在编译阶段通过方面编织器(aspect weaver)将方面编织成程序的一部分,所以在运行时所有的横切方法都被载入;而在 RFT-FJ 演算中,因为层可以在程序运行时添加,所以在运行时只需载入符合当前执行上下文的横切方法即可,这使得运行时的代码更加精简.

4.2 基于Featherweight Java演算的扩展

通过扩展 FJ 演算研究新的语言结构与机制的特性,是当前的一个热点.文献[23]提出多版本类的动态更新演算 MCUFJ,在该演算中,类可以动态地更新,比如添加、删除与修改类的域和方法,并且新旧对象共存.文献[11]提出 Feature Featherweight Java 演算,在该演算中,类通过与特征合成实现自身的扩展,比如重写已存在的方法或添加新的方法.文献[24]提出 FeatherTrait Java 演算,在该演算中,特征(trait)是一个方法集,类通过导入(import)特征实现扩展.RFT-FJ 演算与这些演算的共同之处是:引入新的语言结构和机制到 FJ 演算,研究它们的本质,分析它们对 FJ 演算的影响,制定相应约束以确保 FJ 演算的类型安全,从而保证它们能够融入 Java 语言.

4.3 动态软件更新

本文对象转化的实现与动态软件(dynamic software updating)的更新有相似之处.动态软件更新是一种解决

© 中国科学院软件研究所 http://www.jos.org.cn

软件缺陷和提高软件适用性的通用机制,当前已有较多相关的理论研究[18,23,25]和实现[17,18].文献[25]形式化描述了如何动态地更新程序的代码,但是它并不是针对面向对象程序的动态更新演算.文献[18]形式化描述了如何动态地更新Java类,但是形式化并不是基于FJ演算的扩展.文献[23]描述一个基于FJ演算的多版本的动态软件更新演算,该演算通过允许不同版本的类共存,实现软件的动态更新.但该文献并未给出具体的实现.

文献[17,18]分别描述了一个针对 Java 软件的动态软件更新系统 JVOLVE 和 DVM,它们都通过扩展 Java 虚拟机来实现.这与我们的实现存在共同之处:利用Java虚拟机提供的功能实现程序代码的动态改变,使得对象可以在运行时更新已有的方法或获取新的方法.不同之处是:

1) JVOLVE 和 DVM 是基于类的(class-based)动态更新,而我们的实现是基于对象的(object-based)动态更新;
2) JVOLVE 和 DVM 支持灵活的代码更新,比如它们不仅支持方法体而且支持方法特征的更新,而我们的实现仅支持前者.

导致这些不同的主要原因是:RFT-FJ 演算关注的重点是如何支持程序根据当前的上下文动态地调整其行为,而不是支持更灵活的代码更新.

5 结 论

本文提出一种新的方法:使用适合性测试重定向方法调用.该方法不仅能支持层的动态加载,减少COP程序执行文件的大小,同时也解决了目前 COP 语言中块结构无法分离上下文的改变与依赖上下文的执行的缺点.为了研究该方法的本质以及它对程序的影响,本文定义一个运行时的适合性测试演算——RFT-FJ 演算.作为类型化系统,该演算准确地描述了适合性测试的实质,清楚地反映了它对程序的影响.同时,该演算也定义了层和层化类的限制,以保证适合性测试保持程序的类型安全.本文对相关结论进行了形式化证明,说明 RFT-FJ 演算中定义的限制能够保证程序的类型安全.最后,本文以 RFT-FJ 演算为指导,扩展 Java 编译器和虚拟机实现适合性测试. RFT-FJ 演算和实现可用于指导如何扩展类似 Java 的语言,支持程序基于上下文动态地调整其行为.

References:

[1] Hirschfeld R, Costanza P, Nierstrasz O. Context-Oriented programming. Journal of Object Technology, 2008,7(3):125−151. [doi: 10.1145/940923.940926]

[2] Kamina T, Aotani T, Masuhara H. EventCJ: A context-oriented programming language with declarative event-based context transition. In: Borba P, Chiba S, eds. Proc. of the 10th Int'l Conf. on Aspect-Oriented Software Development. New York: ACM Press, 2011. 21−25. [doi: 10.1145/1960275.1960305]

[3] Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes CV, Loingtier JM, Irwin J. Aspect-Oriented programming. In: Aksit M, Matsuoka S, eds. Proc. of the ECOOP'97. LNCS 1241, Jyvaskyla: Springer-Verlag, 1997. 220−242. [doi: 10.1145/242224.242420]

[4] Appeltauer M, Hirschfeld R, Haupt M, Masuhara H. ContextJ: Context-Oriented programming with Java. Journal of the Japan Society for Software Science and Technology on Computer Software, 2011,28(1):272−292.

[5] Costanza P, Hirschfeld R. Language constructs for context-oriented programming: An overview of ContextL. In: Wuyts R, ed. Proc. of the Dynamic Languages Symp. 2005. New York: ACM Press, 2005. 1−10. [doi: 10.1145/1146841.1146842]

[6] Lincke J, Appeltauer M, Steinert B, Hirschfeld R. An open implementation for context-oriented layer composition in ContextJS. Science of Computer Programming, 2011,76(12):1194−1209. [doi: 10.1016/j.scico.2010.11.013]

[7] Ding B, Wang HM, Shi DX, Li X. Component model supporting trustworthiness-oriented software evolution. Ruan Jian Xue Bao/Journal of Software, 2011,22(1):17−27 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3813.htm [doi: 10.3724/SP.J.1001.2011.03813]

[8] Appeltauer M, Hirschfeld R, Masuhara H, Haupt M, Kawauchi K. Event-Specific software composition in context-oriented programming. In: Baudry B, Wohlstadter E, eds. Proc. of the Software Composition. LNCS 6144, Heidelberg: Springer-Verlag, 2010. 50−65. [doi: 10.1007/978-3-642-14046-4_4]

© 中国科学院软件研究所 http://www.jos.org.cn

[9] Igarashi A, Pierce BC, Wadler P. Featherweight Java: A minimal core calculus for Java and GJ. ACM Trans. on Programming Languages and Systems, 2001,23(3):396−450. [doi: 10.1145/503502.503505]

[10] Zhao YL. Granule-Oriented programming. ACM SIGPLAN Notices, 2004,39(12):107−118. [doi: 10.1145/1052883.1052896]

[11] Apel S, Kästner C, Lengauer C. Feature featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In: Smaragdakis Y, Siek JG, eds. Proc. of the 7th Generative Programming and Component Engineering. New York: ACM Press, 2008. 101−112. [doi: 10.1145/1449913.1449931]

[12] Bettini L, Bono V, Venneri B. Delegation by object composition. Science of Computer Programming, 2011,76(11):992−1014. [doi: 10.1016/j.scico.2010.04.006]

[13] Pierce BC. Types and Programming Languages. Cambridge: MIT Press, 2002.

[14] Kniesel G. Type-Safe delegation for run-time component adaptation. In: Guerraoui R, ed. Proc. of the ECOOP'99. LNCS 1628, Heidelberg: Springer-Verlag, 1999. 351−366. [doi: 10.1007/3-540-48743-3_16]

[15] Wright AK, Felleisen M. A syntactic approach to type soundness. Information and Computation, 1994,115(1):38−94. [doi: 10.1006/inco.1994.1093]

[16] Ekman T, Hedin G. The Jastadd extensible Java compiler. In: Gabriel RP, Bacon DF, Lopes CV, Jr GLS, eds. Proc. of the 22nd Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications. New York: ACM Press, 2007. 1−18. [doi: 10.1145/1297105.1297029]

[17] Subramanian S, Hicks MW, McKinley KS. Dynamic software updates: A VM-centric approach. In: Hind M, Diwan A, eds. Proc. of the 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation. New York: ACM Press, 2009. 1−12. [doi: 10.1145/1542476.1542478]

[18] Malabarba S, Pandey R, Gragg J, Barr ET, Barnes JF. Runtime support for type-safe dynamic Java classes. In: Bertino E, ed. Proc. of the ECOOP 2000. LNCS 1850, Heidelberg: Springer-Verlag, 2000. 337−361. [doi: 10.1007/3-540-45102-1_17]

[19] Bull JM, Smith LA, Westhead MD, Henty DS, Davey RA. A methodology for benchmarking Java grande applications. In: Proc. of the ACM '99 Conf. on Java Grande (Java'99). New York: ACM Press, 1999. 81−88. [doi: 10.1145/304065.304103]

[20] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An overview of AspectJ. In: Knudsen JL, ed. Proc. of the ECOOP 2001. LNCS 2072, Heidelberg: Springer-Verlag, 2000. 327−354. [doi: 10.1007/3-540-45337-7_18]

[21] Chen XQ, Yang FQ. Research on aspect oriented operating systems. Ruan Jian Xue Bao/Journal of Software, 2006,17(3):620−627 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/20060330.htm [doi: 10.1360/jos170620]

[22] Cui ZQ, Wang LZ, Liu HG, Li XD. Computational error handling as aspects: A case study and evaluation. Ruan Jian Xue Bao/Journal of Software, 2011,22(11):2639−2651 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3892.htm [doi: 10.3724/SP.J.1001.2011.03892]

[23] Zhang S, Huang LP. FJ extended calculus for multi-version class dynamic update. Ruan Jian Xue Bao/Journal of Software, 2008,19(10):2562−2572 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/20081008.htm [doi: 10.3724/SP.J.1001.2008.02562]

[24] Liquori L, Spiwack A. FeatherTrait: A modest extension of featherweight Java. ACM Trans. on Programming Languages and Systems, 2008,30(2):1−32. [doi: 10.1145/1330017.1330022]

[25] Hicks M, Nettle S. Dynamic software updating. ACM Trans. on Programming Languages and Systems, 2005,27(6):1049−1096. [doi: 10.1145/1108970.1108971]

附中文参考文献:

[7] 丁博,王怀民,史殿习,李骁.一种支持软件可信演化的构件模型.软件学报,2011,22(1):17−27. http://www.jos.org.cn/1000-9825/3813.htm [doi: 10.3724/SP.J.1001.2011.03813]

[21] 陈向群,杨芙清.面向 Aspect 的操作系统研究.软件学报,2006,17(3):620−627. http://www.jos.org.cn/1000-9825/20060330.htm [doi: 10.1360/jos170620]

[22] 崔展齐,王林章,刘慧根,李宣东.面向方面的计算误差处理技术:实例研究与评估.软件学报,2011,22(11):2639−2651. http://www.jos.org.cn/1000-9825/3892.htm [doi: 10.3724/SP.J.1001.2011.03892]

© 中国科学院软件研究所 http://www.jos.org.cn

[23] 张仕,黄林鹏.基于 FJ 的多版本类动态更新演算.软件学报,2008,19(10):2562−2572. http://www.jos.org.cn/1000-9825/20081008.htm [doi: 10.3724/SP.J.1001.2008.02562]

附录

引理 1. 如果 $mtype(m,T)=\bar{B}\to B$, 并且 $mbody(m,T)=(\bar{x},e)$, 那么存在 B'<:B 使得:

1) 如果 T=C,那么存在 D<:C,使得 $\bar{x}:\bar{B}, this:D\vdash e:B'$;

2) 如果 T=C_G,那么 $\bar{x}:\bar{B}, this:C_G\vdash e:B'$.

证明:直接对 $mtype(m,T)=\bar{B}\to B$ 作归纳即可.注意,如果 T=C_G,那么 T 没有超类. □

引理 2. 如果 $mtype(m,T)=\bar{B}\to B$, 那么对所有 T'<:T,都有 $mtype(m,T')=\bar{B}\to B$.

证明:如果 T=C,那么证明与FJ中的引理A.1.1[9]的证明类似.如果 T=C_G,因为 C_G 没有超类,所以我们有 T'=T,结论成立;如果 $T'=\bar{C}_G::C$, 由于 RFT-FJ 演算仅支持方法的重写,因此同名方法的类型一定相同,由 *mtype* 定义可知结论成立. □

引理 3(替换). 如果 $\Gamma,\bar{x}:\bar{B}\vdash e:T$ 并且 $\Gamma\vdash\bar{s}:\bar{T}'$,其中,$\bar{T}'<:\bar{B}$,那么存在 T'<:T 使得 $\Gamma\vdash[\bar{x}\mapsto\bar{s}]e:T'$.

证明:直接对 $\Gamma,\bar{x}:\bar{B}\vdash e:T$ 的推导作归纳,我们只考虑如下两种关键情况:

1) T-VAR $e=x$ $\Gamma\vdash x:B$

如果 $x\notin\bar{x}$,则有 $[\bar{x}\mapsto\bar{s}]x=x$,结论成立;如果 $x\in\bar{x}$,则有 $[\bar{x}\mapsto\bar{s}]x=s_i$,根据已知,结论成立.

2) T-INVK $e=e_0m(\bar{e})$ $\Gamma,\bar{x}:\bar{B}\vdash e_0:T_0$ $mtype(m,T_0)=\bar{B}'\to B$ $\Gamma,\bar{x}:\bar{B}\vdash\bar{e}:\bar{T}'$ $\bar{T}'<:\bar{B}'$

我们分别使用 e_0^* 和 $\bar{e}^*$ 表示 $[\bar{x}\mapsto\bar{s}]e_0$ 和 $[\bar{x}\mapsto\bar{s}]\bar{e}$.根据归纳假设有 $\Gamma\vdash e_0^*:T_0'<:T_0$ 和 $\Gamma\vdash\bar{e}^*:\bar{T}''<:\bar{T}'$. 由引理 2,有 $mtype(m,T_0')=mtype(m,T_0)$. 根据类型化规则 T-INVK 有 $\Gamma\vdash e_0^*.m(\bar{e}^*):B$, 设 T'=B,结论成立. □

引理 4(弱化). 如果$\Gamma\vdash e$:T,那么Γ,x:$T'\vdash e$:T.

定理 1 的证明.

证明:对 $e\to e'$ 作归纳,我们只考虑如下两种关键情况:

1) E-LINVK $e=\text{new }\bar{C}_G::C(\bar{v}).m(\bar{u})$ $mbody(m,\bar{C}_G)=(\bar{x},e_0)$ $e'=[\bar{x}\mapsto u, this\mapsto\text{new }\bar{C}_G::C(\bar{v})]e_0$

推导出$\Gamma\vdash e$:T 的最后一条类型规则一定是 T-INVK,并有如下前提:

$$\Gamma\vdash\text{new }\bar{C}_G::C(\bar{v}):\bar{C}_G::C\ \ mtype(m,\bar{C}_G::C)=\bar{B}\to B\ \ \Gamma\vdash\bar{u}:\bar{T}<:\bar{B}$$

及结论 T=B.

根据 *mbody* 定义,可假设 m 定义在 C_{G_i} 中且 $C_{G_i}\in\bar{C}_G$.根据引理 1 和引理 4 有, $\Gamma,\bar{x}:\bar{B}, this:C_G\vdash e_0:B'<:B$, 根据引理 3 有, $\Gamma\vdash[\bar{x}\mapsto u, this\mapsto\text{new }\bar{C}_G::C(\bar{v})]e_0:T'<:B'$, 根据<:的传递性,结论成立.

2) E-LINVK$_2$

$e=\text{new }\bar{C}_G::C(\bar{v}).m(\bar{u})$ $mbody(m,C_G)=(\bar{x},e_0)$ $e'=[\bar{x}\mapsto u, this\mapsto\text{new }C_G::\bar{C}_G::C(\bar{v})]e_0$ $mbody(m,\bar{C}_G)=\bullet$

推导出$\Gamma\vdash e$:T 的最后一条类型规则一定是 T-INVK,并有如下前提:

$$\Gamma\vdash\text{new }\bar{C}_G::C(\bar{v}):\bar{C}_G::C\ \ mtype(m,\bar{C}_G::C)=\bar{B}\to B\ \ \Gamma\vdash\bar{u}:\bar{T}<:\bar{B}$$

及结论 T=B.

在归约中,新对象 new $C_G::\bar{C}_G::C(\bar{v})$ 被创建,因此 $\Gamma=\Gamma\cup\{\text{new }C_G::\bar{C}_G::C(\bar{v}):C_G::\bar{C}_G::C\}$, 其中,根据子类型关系有 $C_G::\bar{C}_G::C<:\bar{C}_G::C$.

由于层化类只能定义其基类中的重写方法,因此根据 $m\in C_G$ 有 $mtype(m,C_G)=\bar{B}\to B$, 根据引理 1 和引理 4 有, $\Gamma,\bar{x}:\bar{B}, this:C_G\vdash e_0:B'<:B$. 根据引理 3 有, $\Gamma\vdash[\bar{x}\mapsto u, this\mapsto\text{new }C_G::\bar{C}_G::C(\bar{v})]e_0:T'<:B'$.

根据<:的传递性,结论成立. □

定理 2 的证明.

证明:对 $e\to e'$ 作归纳,我们只考虑下面一种关键情况:

© 中国科学院软件研究所 http://www.jos.org.cn

E-LINVK e=new $\bar{C}_G :: C(\bar{v}).m(\bar{u})$ $mbody(m,\bar{C}_G)=(\bar{x},e_0)$ $e'=[\bar{x}\mapsto\bar{u}, this\mapsto$ new $\bar{C}_G :: C(\bar{v})]e_0$

根据归纳假设有:$\bar{v}$ 和 $\bar{u}$ 都是值,我们根据 e_0 的不同情况分别加以证明.

1) $e_0=x$

由于 e_0 是闭包表达式,因此 x 只能是 $this$ 或者 x_i,其中, $x_i\in\bar{x}$,因此, $e'=$ new $\bar{C}_G :: C(\bar{v})$ 或 u_i,结论成立.

2) $e_0=e_0'.f_i$

- 如果 $e_0'=this$,则 $e'=$ new $\bar{C}_G :: C(\bar{v}).f_i$, 因此可应用 E-PROJ 于 e';
- 如果 $e_0'\neq this$,则 $e_0=[\bar{x}\mapsto\bar{u}, this\mapsto$ new $\bar{C}_G :: C(\bar{v})]e_0'.f_i$.

据归纳假设, $[\bar{x}\mapsto\bar{u}, this\mapsto$ new $\bar{C}_G :: C(\bar{v})]e_0'$ 是值或可进一步规则.如果是值,则可应用 E-PROJ 于 e';否则可进一步归约 e',结论成立.

3) $e_0=e_0'.m'(\bar{w})$

如果 $e_0'=this$,则可应用规则 E-LINVK,E-LINVK$_2$,E-LINVK$_3$ 和 E-INVK 可应用于 e';

如果 $e_0'\neq this$, $e'=[\bar{x}\mapsto\bar{u}, this\mapsto$ new $\bar{C}_G :: C(\bar{v})]e_0'.m'(\bar{w})$.

据归纳假设, $[\bar{x}\mapsto\bar{u}, this\mapsto$ new $\bar{C}_G :: C(\bar{v})]e_0$ 是值或可进一步归约.若是值,则 E-LINVK,E-LINVK$_2$,E-LINVK$_3$ 和 E-INVK 可应用于 e';否则,e'可进一步归约,结论成立. □

赵银亮(1960－),男,陕西岐山人,博士,教授,博士生导师,CCF 高级会员,主要研究领域为程序语言与编译系统,并行计算,机器学习.
E-mail: zhaoy@mail.xjtu.edu.cn

韩博(1975－),男,博士生,高级工程师,主要研究领域为软件编程方法学,信息管理学.
E-mail: bohan@xjtu.edu.cn

朱常鹏(1981－),男,博士生,主要研究领域为 Java 虚拟机技术,类型系统.
E-mail: is99zcp@hotmail.com

曾庆花(1981－),女,博士生,主要研究领域为 Java 编译技术.
E-mail: qhzeng08@gmail.com

© 中国科学院软件研究所 http://www.jos.org.cn