Eiffel 语言的语义*

李师贤 阮文江

(中山大学软件研究所,广州 510275) (中山大学计算中心,广州 510275)

摘要 本文采用类 VDM 的指称语义技术为 Eiffel(1988)语言建立了形式语义模型. 该模型首先为 Eiffel 语言定义了对象模型和两个语义环境(动态和静态环境),然后讨论 Eiffel 例程的语义. 为了描述 Eiffel 的意外处理,我们采用了 VDM 的"出口"机制.

关键词 Eiffel*,OOPL*,VDM,指语语义.

Eiffel 是面向对象语言(以下称"Eiffel")和环境的总称. Eiffel 强调软件的可重用性和可扩充性,支持纯对象性、断言、类属、多重和重复继承、多态性和动态连接.

为了描述 Eiffel 语言的语义,我们为它建立了形式模型,以便于我们讨论 Eiffel 语言的语义. 更进一步,它可指导我们实现 Eiffel 语言和环境.

1 Eiffel 语言的抽象文法

Eiffel 系统包括一组类定义,每个类定义说明了类的规范和实现:

System=Class_map

Class_map=map Class_name to Class_body

Class_body::Generics :seq of Id
Parents :seq of Parent
Attributes :map Id to Type
Routines :map Id to Routine_body
Class_inv :seq of Assertion
Rescue :seq of Instruction

Eiffel 允许多重继承,而类特征在继承过程中可以通过重命名和重定义加以调节:

Parent::Class :Class_name
Actuals :seq of Type
Renames :map Id to Id
Redefs :set of Id

* 本文 1992-01-13 收到,1992-12-14 定稿

作者**李师贤**,1944 年生,教授,主要研究领域为形式语义学与软件工程方法学. **阮文江**,1965 年生,硕士,主要研究领域为程序设计语言与编译程序.

本文通讯联系人:李师贤,广州 510275,中山大学软件研究所

Eiffel是强类型语言,其类型系统包括预定义类型(布尔型、整型、实型和字符型)、类属参量类型、类类型和相关声明类型:

Type={Boolean,Char,Integer,Real}∪Id∪Class_type∪Like_anchor

Class_type::Class :Class_name
Actuals :seq of Type

Like_anchor::Anchor:Id∪{Current}

每个例程体说明例程的标记、规范和实现:

Routine_body::Deferred :Boolean
Formals :seq of Dec
Type_mark :[Type]
Pre :Seq of Assertion
Externs :map Id to External
Locals :map Id to Type
Body :Compound
Post :Seq of Assertion

Compound=seq of Instruction

指令分为赋值、特征调用、条件、循环、检测、重试和调试等指令.

Instruction=Assignment∪Call∪Conditional∪Loop∪Check∪{retry}∪Debug

Assignment::LHS:Id∪{result}
RHS:Expression

特征调用有两种形式:对当前对象的非限定调用和对其它对象的限定调用:

Call=Qualified_call∪Unqualified_call

Qualified_call::Supplier:Expression
Feature_name:Id∪{Create,Void,Clone,Forget,Equal}
Args:seq of Expression

Unqualified_call::Feature_name:Id
Args:seq of Expression

条件指令:

Conditional::Then_list:seq of Then_part
Else_part:Compound

Then_part::Condition:Bool_exp
Then:Compound

循环指令:

Loop::From :Compound
Invariant :Bool_exp
Variant :Int_exp
Body :Compound
Exit :Bool_exp

表达式包括常量、局部变量、当前对象指针、特征调用、预定义实体 Result、不变表达式、原值表达式、一元表达式、二元表达式和括号表达式:

Expression=Constant ∪{Current,Result,Nochange}∪Old_value
∪Call∪Id∪Unary_exp∪Binary_exp∪(Expression)

良构性判定式精确地说明了各个语言成分的良构性:

WFProgram:Program→Boolean

WFClass:Class_body→SEnv→Boolean

WFParents:seq of Parent×SEnv→Boolean

WFParent:Parent→SEnv→Boolean

WFType:Type×SEnv→Boolean

exp_type:Expression×SEnv→Type

type_match:Type×Type×SEnv→Boolean

WFRoutine:Routine_body→SEnv→Boolean

WFInstruction:Instruction→SEnv→Boolean

WFCompound:seq of Instruction→SEnv→Boolean

WFAssertion_list:seq of Assertion→SEnv→Boolean

WFExpression:Expression→SEnv→Boolean

WFCall:Call→SEnv→Boolean

上述忽略了具体描述的函数精确地描述了 Eiffel 的静态语义,如继承的有效性、类型匹配和特征调用有效性等.

2 Eiffel 的对象模型

Eiffel 例程操作的对象存贮在单一的存贮空间中,它包含 Eiffel 系统运行时所创建的所有对象.

Object_memory=map Oop to Object

我们使用 σ 指称对象空间的值.

我们使用唯一的内部对象名或对象指针(称为 Oop)来标识对象.每个对象是类的实例,而类可能是含有形式类属参量的,因而实例也应当指称它的实际类属参量.实例的属性域对应于类的属性.

Object::Class　　:Class_name
　　Actuals　　:seq of Type
　　Fields　　:map Id to Value

属性域和局部变量指称整数、实数、布尔值、字符或对象指针:

Value=Char_value∪Bool_value∪Int_value∪Real_value∪Oop

3 环　境

在下面的语义方程中,我们将使用两种环境:静态环境和动态环境.静态环境描述例程、

指令和表达式等语言成分在文本中的语景.动态环境描述例程运行时的局部状态,它随着例程的运行而改变,而每次例程调用都有它自己的局部动态环境.另外,与它相关的对象空间是全局性的.

3.1 静态环境

静态环境(用 ρ 指称)包含所有例程的"文本"和当前例程、当前位置的指称:

SEnv::Class :Class_name
P :System
Routine:Id ⋃ {Create}
Where :{require,ensure,rescue,class_invariant}

3.2 动态环境

动态环境(用 δ 指称)记录当前调用例程的局部计算状态.为了实现后置断言中不变和原值表达式,δ 也存贮了当前对象的原始备份:

DEnv::Current :Oop
Args :map Id to Value
Result :[Value]
Locals :map Id to Value
Old :Oop

在一个例程调用中,当前对象和实际参量是确定的,并且在整个运行过程中不改变,而局部变量和预定义函数结果值实体 Result 则要以 Eiffel 初始化规则进行初始赋值.

4 例程的语义函数

在语义方程中,例程改变对象空间.

例程接收一个指向当前对象的指针和一组实际参量,而返回一个值(对于函数),并改变对象空间的状态.由于例程可能失败,例程的结果应当包含"出口标记"以指示该例程调用是否成功返回.

Routine=Oop×seq of Value×Object_memory→Object_memory
×[Exit_mark]×[Value]

由于"goto"指令破坏软件的可读性,Eiffel 放弃了该条指令.然而,"goto"指令处理意外事件的方便性是不容否认的.因此,为了处理意外,Eiffel 引入"重试"机制.重试指令产生一个意外并把控制流转向补救部.补救部把对象调整到稳定的状态,而补救部的重试指令将使例程重新运行.如果在补救部中没有执行重试指令,例程将以"失败"方式返回.下面指出的其它意外也将使控制流转向补救部.

4.1 意 外

下述几种意外可能在例程运行过程中发生:

(1)使用空指针进行限定性特征调用;(2)不满足断言;(3)由操作系统和硬件引起的意外.

4.2 出口机制

我们知道重试机制引起 Eiffel 例程的异常执行,VDM 的指称语义技术通过出口机制处

理这种异常情况.

这里,我们定义几个与出口机制有关的语义方程:

Exit_mark = {Failure, retry, exception}

a: Exit_mark

P: Exit_mark → Boolean

t1: Exit_mark → DEnv → Object_memory → DEnv × Object_memory × [Exit_mark]

t2: DEnv → Object_memory → DEnv × Object_memory × [Exit_mark]

exit(exception) △ λδ,σ · (δ,σ,exception)

(tixe [a → t1(a) | P(a)] in t2): DEnv → Object_memory
→ DEnv × Object_memory × [Exit_mark]

(tixe [a → t1(a) | P(a)] in t2) △ λδ,σ · let e = [a → t1(a) | P(a)] in
let r(δ,σ,a) = if a ∈ dom e then r · e(a)
else (δ,σ,a) in
r · t2(δ)(σ)

(trap (a) with t1(a); t2): DEnv → Object_memory
→ DEnv × Object_memory × [Exit_mark]

(trap (a) with t1(a); t2) △ λδ,σ · let h(δ,σ,a) = if a ≠ nil then t1(a)(δ)(σ)
else (δ,σ,nil) in
h · t2(δ)(σ)

4.3 与补救部相关的语义函数

补救部由例程意外激活,它接受当前对象的指针和对象空间,返回"重试"或"失败"的标记,并改变对象空间:

Rescue = Oop × Object_memory → Object_memory × [Exit_mark]

Rescue_body: Rescue_body → SEnv → Rescue

MRescue_body[mk − Rescue_body(rescue_clause)]ρ △ λcurrent,σ ·
let δ = mk − DEnv(current, [], nil, []) in
let rescue_body = if rescue_clause ≠ [] then rescue_clause else
Rescue(P(ρ)(Class(ρ))) in
if rescue_body = [] then (σ, failure)
else
let (δ′, σ′, exit_mark) =
(tixe [retry → λδ,σ · (δ,σ,retry), exception → λδ,σ · (δ,σ,failure)]
in MCompound[rescue_body]μ(ρ, Where → rescue)δσ)
if exit_mark = nil then (σ′, failure)
else (σ′, exit_mark)

4.4 例程体的语义函数

现在,我们为例程提供语义方程.它接受例程定义和静态环境,返回一个例程指称:

MRoutine_body: Routine_body → SEnv → Routine

MRoutine_body[mk－Routine_body(formals, result_type, pre, locals, body, post, rescue)]ρ △λcurrent, arglist, σ·

let (σ′, old_copy)=object_copy(current, σ) in
let δ= mk－DEnv(current,
[Var_id(formals(i))→arglist(i)|1≤i≤len formals],
initial_value(no_like_type(result_type, ρ)),
[id→initial_value(no_like_type(locals(id), ρ))|id
∈dom locals], old_copy) in
let class_inv=Class_inv(P(ρ)(Class(ρ))) in
let inv_pre=[mk－Invariant(class_inv)]↶[mk－Require(pre)] in
let inv_pre_body=inv_pre↶body in
let inv_pre_body_post=inv_pre_body↶[mk－Ensure(post)] in
let inv_pre_body_post_inv=inv_pre_body_post↶[mk－Invariant(class_inv)] in
let (δ′, σ″, exit_mark)=
(trap exception with
λexception·
(λδ, σ·
let (σ′, exit_mark)=
MRescue_body[rescue](ρ)(Current(δ), σ) in
if exit_mark=retry then
let (σ″, exit_mark1, v) =
MRoutine_body[mk－Routine_body(formals,
result_type, pre, post, locals, body, rescue)](ρ)(current, arglist, σ′)
in
(μ(δ, Result→v), σ″, exit_mark1)
else (δ, σ′, failure)
);
MCompound[pre_body_post]ρ
)δσ′ in
(σ″, exit_mark, Result(δ′))

必须注意的是，我们把前后置断言和类不变式作为一般指令来处理.

辅助函数 initial_value 返回对应类型的初始值.

指令组的语义函数是：

MCompound: Compound →SEnv→DEnv→Object_memory
→DEnv×Object_memory×[Exit_mark]

指令的语义函数定义为：

MInstruction: Instruction →SEnv→DEnv→Object_memory
→DEnv×Object_memory×[Exit_mark]

5 特征调用

特征调用是 Eiffel 对象间进行通讯的仅有方式. Eiffel 调用是同步的:调用方要等待被调用方返回一个值后才能继承执行. Eiffel 特征调用具有两种形式:一种是对当前对象的非限定调用,另一种是对其它对象的限定调用. 特征调用是动态的,要进行实际特征的搜索过程:从当前类到亲代类.

特征调用的语义方程可以定义为:

MCall:Call →SEnv→DEnv→Object_memory
→DEnv×Object_memory×[Exit_mark]×[Value]

这里,非限定调用的语义函数描述为:

MExpression[[mk−Unqualified_class(fn,arglist)]]ρδσ △
let (actuals,δ′,σ′)=MAll_Expression_list[arglist]ρδσ in
let externs=Externs(Routines(P(ρ)(Class(ρ)))(Routine(ρ)) in
if fn ∈dom extenrs
then
let (σ″,exit_mark,value)=
MExternal_routine[externs(fn)(actuals,current,σ′)]in
if exit_mark=failure then exit(exception)
else (δ′,σ″,exit_mark,value)
else
let newnm=new_name(Current(ρ),fn,Class(σ′(Current(δ))),P(ρ)) in
let find_result=find_call_body(Class(σ′(Current(δ))),newnm,P(ρ)) in
let (σ″,exit_mark,value)=find_result(Current(δ),client,actuals,σ′) in
if exit_mark=failure then exit(exception)
else (δ′,σ″,exit_mark,value)
MExternal_routine:External →seq of Value×Oop×Object_memory
→Object_memory×[Exit_mark]×[Value]

一组表达式的语义是:

MAll_Expression_list:seq of Expression→SEnv→DEnv→Object_Memory
→seq of Value×DEnv×Object_Memory×[Exit_mark]

辅助函数 new_name 找出某个亲代类的特征在后代类的新名:

new_name:Class_name×Id×Class_name×System → [Id]

辅助函数 find_call_body 找出调用的例程体.

find_call_body:Class_name×Id×System → [Routine]

6 对象创建

要生成一个新对象,首先必须为它分配内存,然后对象域要根据 Eiffel 初始化规则进行

初始赋值，并调用该对象的创建例程过程. 其语义描述如下：

MCall[mk—Qualified_call(id,Create,arglist)]ρδσ △

let (actuals,δ′,σ′)=MAll_Expression_list[arglist]ρδσ in

let object_type=no_like_type(exp_type(id,ρ),ρ) in

let (σ″,new_oop)=new_object(object_type,ρ,σ′) in

let (δ″,σ‴)=assign(id,new_oop,ρ,δ′,σ″) in

if Create∈dom Routines(P(Class(Object_type))) then

let (σ⁗,exit_mark,value)=Routines (P(Class(Object_type)))

(Create)(new_oop,actuals,σ‴) in

if exit_mark=failure then exit(exception)

else (σ⁗,nil)

else (σ‴,nil)

例程 new_object 创建一个新对象并初始化对象域：

new_object(type:Class_type):new_oop:Oop

ext wr σ:Object_memory

rd p:System

post new_oop∉σ∧$\tilde{\sigma}$=σ∪{new_oop→mk—Object(Class(type),

Actuals(type),{id→initial_value(fields(id))|id∈dom fields

=object_fields(Class(type),Actulas(type),p)}

辅助函数 object_fields 求出类的所有属性特征.

7 结束语

这里为 Eiffel 描述的形式语义有助于对面向对象语言的理解，并可作为我们实现 Eiffel 系统的基础.

参考文献

1 Cardelli, Wegner P. On understanding types, data abstraction and polymorphism. ACM Computing Surveys, 1985,17(4).

2 Meyer B. Object—oriented software construction. Prentice Hall, 1988.

3 屈延文. 形式语义学基础与形式说明. 北京：科学出版社，1991.

4 Wolczko M. Semantics of smalltalk—80. ECOOP'87, 1987. 108—120.

5 周巢尘. 形式语义学引论. 长沙：湖南科学技术出版社，1985.

SEMANTICS OF EIFFEL

Li Shixian

(*Institute of Computer Software, Zhongshan University, Guangzhou* 510275)

Ruan Wenjiang

(*Computing Center of Zhongshan University, Guangzhou* 510275)

Abstract　A formal model of the Eiffel(1988) programming language is described using the denotational style. The model defines Eiffel's object model and two environments (dynamic and static) at first, and then studies semantics of Eiffel's routines. The "exit" mechanism of VDM is used for processing Eiffel's exception.

Key words　Eiffel*, OOPL*, VDM, denotational semantics.