A Graph Model for Object-Oriented Programming Languages

Chui, Patrick

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

August 2005

Your file Votre référence
ISBN: 0-494-10283-7
Our file Notre référence
ISBN: 0-494-10283-7

ABSTRACT

A Graph Model for Object-Oriented Programming Languages
Patrick Chui

Object-oriented programming lacks a simple theoretical foundation. This is manifested by the abundance of formal models for this programming paradigm. In its simplest form, an object can be viewed as an internal state plus a set of operations on objects. We use directed labelled graph, called state graph, to model the run-time behaviour of objects, with nodes as objects and edges as references to other objects. The graph model, based on conventional operational semantics, is natural and simple. We are then able to formulate an axiomatic semantics for reasoning about object-oriented programs. The axiomatic semantics is proved to be sound and complete with respect to the operational semantics. Our work suggests that graph is an good candidate for capturing the characteristic features of object-oriented programming languages.

ACKNOWLEDGEMENTS

I wish to express my sincere thank to Dr. Peter Grogono, my supervisor, for his direction, assistance and guidance. Dr. Grogono has also provided numerous suggestions and techniques in typesetting this document. Special thanks should be given to my wife Echo and my daughter Maya for their encouragement and support.

Contents

Chapter 1

Background

1.1 Introduction

Object-oriented programming lacks a simple theoretical foundation. Any answer to the question "what is the essence of object-oriented programming" reveals personal preference rather than objective truth. This gives rise to a wealth of formal models for this programming paradigm, which is reflected in the survey [Men94]. Since the popularization of object-oriented programming by `SmallTalk`[1], basic concepts of this paradigm have been interpreted differently. The situation has been aggravated by the continuing appearance of new object-oriented languages.

A simple theoretical foundation of object-oriented programming provides a basis for definition and discussion. We might then determine what are the essential features. For example, should class be a primitive concepts? It could also be used to give a formal semantics to obscure concepts with no generally accepted definition. A common framework would let us compare strength and weakness of object-oriented programming languages so as to develop new languages with more orthogonal design, and help us find more efficient compilation techniques.

Because functional programming, founded on λ-calculus, is so well understood, many existing formalisms of object-oriented programming are extensions of λ-calculus, often with emphasis on type-correctness. The problem with functional approach is that it is hard to talk about state. Even the σ-calculus of Abadi and Cardelli [AC96] does not include state as a primitive

[1]Though, the notion of object as a programming construct first appeared in `Simula`.

concept. Further, typing is a static property and is not able to describe run-time features of object-oriented programming such as dynamic binding.

In its simplest form, an object is some internal state, which is a collection of objects, plus a set of operations on objects. The internal state is defined by the content of the instance variables of the object. Each operation, called a method, has a host object. A method name may be associated with different operations. The actual method body (i.e. operation) to be executed depends on the run-time identify of the host object. This is dynamic binding.

State is memory configuration. Conventional approach for modelling state is based on address locations and values. This is adequate for procedural programming. The state of a procedural program is simply a list of variables and their content. An object-oriented system, however, consists of entities interacting with each other. The relationship between objects is lost.

We model an object-oriented system as a directed labelled graph, called state graph. Edges are labeled with the names of instance variables or local variables and have a direction. A vertex represents an object and an edge is a reference or pointer to an object. Such a model leads naturally to object identity, local state, and dynamic binding, side-effects, aliasing, the use of self to denote the current object.

We concentrate on systems in which each object is an instance of a particular class. Methods are defined by classes or via inheritance. We formulate semantic description of object-oriented programming languages by means of state graph. Semantics is a powerful tool in studying programming languages. It gives meaning to programs and allows us to reason about programs. Different semantics styles have different applications. Operational semantics guides language designers and implementers. Axiomatic semantics helps us prove programs meet specifications. Both semantic styles are therefore useful but they need to be compatible.

This thesis describes a graph model for object-oriented programming. We give operational semantics (Section 3.1) and axiomatic semantics (Section 5.1) of a sample language, and prove that the two semantics are equivalent (Section 5.2.2). Then we discuss related work (Chapter 6). Our work suggests that graph is a good candidate for modelling object-oriented languages.

1.2 Semantic Styles

Semantics is *meaning.* A computer can execute a program but does not necessarily know what the program *means.* People understand programs intuitively: "I use this program to pay my bills." Formal semantics is able to rigorously specify the behaviour of programs. The need for rigor arises because it resolves ambiguities and subtle complexities in apparently clear defining documents, e.g. language manuals. More importantly, it forms the basis for implementation, analysis and verification. Over the years, various approaches of giving meaning to programs have been widely studied. The major ones are operational semantics, denotational semantics and axiomatic semantics.

1.2.1 Operational Semantics

Operational semantics describes a programming language by defining a simple abstract machine for it. This machine is abstract because it uses the terms of the language as its machine code rather than some instruction set of low level hardware. A term is executed against an initial state[2] to give a next state. The meaning of the term is taken to be a sequence of machine states. This approach has the advantage that its rules can be used to generate interpreters for validating and testing the language definition.

Plotkin's seminal technical report [Plo81] is generally regarded as the beginning of structural operational semantics. The use of inductive systems to define programming constructs has then become a standard style of semantics, as exemplified in the textbooks [NN92], [Win93], [SK95] and [Pie02], to name but a few.

1.2.2 Denotational semantics

Denotational semantics takes a more abstract view of meaning. A program term is represented by some mathematical object, such as a number or a function. To define denotational semantics for a language, one has to search for a collection of semantics domains and then define an interpretation function mapping terms into elements of these domains.

Originally intended as a mechanism for the analysis of programming languages, denotational semantics was developed in the mid 1960s by Christopher Strachey and his Programming Research Group at Oxford University. Dana Scott supplied the mathematical foundations in

[2]State is also known as configuration.

1969, which grew into *domain theory.* A review of the subject, with extensive bibliography, can be found in [FJM$^+$96].

1.2.3 Axiomatic semantics

Axiomatic semantics, also known as *Hoare logic* or programming logic, aims at reasoning about programs. Instead of first describing behaviour of programs by giving some operational or denotational semantics and then deriving laws from this definition, axiomatic methods take the laws themselves as the definition of the language. The meaning of a program term is the relation between an initial condition and final condition, which is what can be proved about the term. This approach is better suited for verification of program properties.

The idea of verifying the correctness of a program by means of logic originated in the early work of Floyd [Flo67] and Hoare [Hoa69]. An early concrete application of this axiomatic technique to programming language specification was `Pascal` [Hoa74]. The main features of Hoare logic are summarized in [Apt81], which emphasizes soundness and completeness.

1.3 Deductive Systems

In this section, we give an informal review of deductive or inference system[3] that is fundamental to the study of semantics of programming languages. An *inference system* consists of a collection of *inference rules* defining one or more judgements. A `judgement` is an assertion stating that a property holds of certain object. The inference rules determine the conditions under which a judgement may be inferred or derived. Rules are written in the form

$$\frac{J_1 \; J_2 \; \cdots \; J_n}{J}$$

where the judgements $J_1, J_2, \cdots, J_n$ are called premises, and judgement J is the conclusion. The notation can be interpreted as follows. If $J_1, J_2, \cdots, J_n$ can be inferred, then J can also be inferred. Alternatively, in order to derive J, we have to first derive $J_1, J_2, \cdots, J_n$. Sometimes, rules may stand by themselves without any premise. Such rules are called *axioms.* They are written as

$$\frac{}{J}$$

or simply as J without the premise part, which is empty.

[3]Variants of the more technical term *inductive definition.*

For example, for a string of characters r, the judgement $r \in \mathsf{R}$ reads r is a regular expression. Then the axioms

$$\varepsilon \in \mathsf{R} \qquad \alpha \in \mathsf{R}$$

where α is any character, together with the rules

$$\frac{r \in \mathsf{R} \quad r' \in \mathsf{R}}{(r|r') \in \mathsf{R}} \qquad \frac{r \in \mathsf{R} \quad r' \in \mathsf{R}}{(rr') \in \mathsf{R}} \qquad \frac{r \in \mathsf{R}}{(r)^* \in \mathsf{R}}$$

define a deductive system of regular expressions.

When we use the axioms and rules to *derive* a judgement J, we obtain a *derivation tree.* The root of the derivation tree is J. The leaves are instances of axioms. The internal nodes are conclusions of rules, with premises of the rules as immediate children. In this case, judgement J is called *derivable.*

An important tool to reason about deductive systems is induction on derivation. To show that a property P holds for all derivable judgements, it suffices to show that for each rule

$$\frac{J_1 \; J_2 \; \cdots \; J_n}{J}$$

if P is true for all $J_1, J_2 \; \cdots \; J_n$, then P is also true for J. The assumption that P holds for every premise of the rule is known as induction hypothesis. In the case of axioms, we need to establish the conclusion without any assumption. This will be main proof technique used in this thesis. For detailed discussion of induction in the context of programming language semantics, see [Win93].

Chapter 2

A Graph Model Language

The *call graph* for a program P is a directed graph $G = (V, E)$ such that

- V is the set of functions in the program,
- an edge (u, v) belongs to $E \subseteq V \times V$ iff function u calls function v.

If P is written in FORTRAN, which does not allow recursion, then G has no path that starts and ends in the same node. On the other hand, if P is written in C or any language that allows recursion, then G may have *cycles*, including self-loops when a function calls itself. For a procedural program, the call graph can be constructed directly from the program text.

The situation happens to be much more complicated in object-oriented programming. A term $o.m(\ )$ or $o{\rightarrow}m(\ )$ represents a function call, but the function called depends on the identity of object o at run-time. Consequently, there is no static call graph for an object-oriented program.

The concept of state graph, first described in [GG94], models the call graph dynamically. It is based on the principle that, if an object o can evaluate a member function of another object o', there must be a link from o to o'. This is quite different from procedural languages, where any function can make a call to any other function.

2.1 Abstract Syntax of GM

We define a sample language GM. The language is small but includes all of the basic features of object-oriented programming. In the following BNF, the symbol ε stands for the empty string. Non-terminal terms are capitalized, except for class names, which are the only terminal terms in Roman uppercase.

Program A GM program is a list of class declarations.

$\mathsf{G} ::= \varepsilon \mid \mathsf{K}\ \mathsf{G}$

Class Declaration A class consists of a list of fields[1] and methods.

$\mathsf{K} ::= \texttt{class}\ C\ \texttt{extends}\ C\ \{\ \mathsf{FS}\ \mathsf{MS}\ \}$

Field Declaration List

$\mathsf{FS} ::= \varepsilon \mid C\ f\ ;\ \mathsf{FS}$

Method List

$\mathsf{MS} ::= \varepsilon \mid \mathsf{M}\ \mathsf{MS}$

Method Definition A method requires exactly one parameter.

$\mathsf{M} ::= \mathcal{T}\ \ m(C\ x)\ \{\ \mathcal{G}\ \}$

Method Body A method body consists of a list of variable declarations and a statement. A variable declaration is a variable name preceded by a class name.

$\mathcal{G} ::= C\ x\ ;\ \mathcal{G} \mid \mathcal{S}$

Statement A statement is a local variable assignment, field assignment, method invocation, return, conditional *if* statement, or sequence of statements.

$\mathcal{S} ::= x = \mathcal{E} \mid \mathcal{E}.f = \mathcal{E} \mid \mathcal{I} \mid \texttt{return}\ \mathcal{E} \mid \texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}\ \texttt{else}\ \mathcal{S}\ \texttt{fi} \mid \mathcal{S}\ ;\ \mathcal{S}$

Expression An expression is a local variable access, field access, method invocation, object creation, cast, the keyword *null*, or the keyword *this*.

$\mathcal{E} ::= x \mid \mathcal{E}.f \mid \mathcal{I} \mid \texttt{new}\ C \mid (C)\ \mathcal{E} \mid \texttt{null} \mid \texttt{this}$

Method Invocation A method invocation is a statement, for its side effects, or an expression, for its returned value.

$\mathcal{I} ::= \mathcal{E}.m(\mathcal{E})$

Type A type is a class name or the keyword *Void*.

$\mathcal{T} ::= C \mid \texttt{Void}$

[1]Also known as field variables, or instance variables.

Atoms These are user-defined identifiers.

x ::= `VARIABLE`
f ::= `FIELD`
C ::= `CLASS`
m ::= `METHOD`

We restrict our study to single inheritance. Fields defined in the subclass hide those fields that have the same name in the superclass. In method overriding, neither the parameter type nor the return type is allowed to change. Moreover, parameter name must remain the same in overriding. We assume a few syntactic regularities in order to make GM as simple and neat as possible.

- The *extends* clause is always required. This is achieved by means of a predefined class *Object*[2], which simply has no fields or methods. A class that inherits no user-defined class must therefore be declared with *extends Object*.
- Field access within the defining class has to be qualified by keyword *this*. Any unqualified identifier is understood to be a local variable.
- The keyword *this* is only allowed in a method body and is implicitly bound to the class in which the method is defined.
- Method invocation is an expression if and only if the method has type $\mathcal{T} \to C$. In this case, the last statement of the method body is a *return*. We shall see in Section 4 that this is enforced by the type system.
- Only fields and variables can appear on the left hand side of an assignment statement. They are the only valid L-values.
- Brackets, which are not part of the syntax, will be applied to clarify syntax whenever necessary.

We assume that there is always a class `Main`,

$$\texttt{class Main extends Object} \ \{ \ \texttt{Void main(Object } x\texttt{)} \ \{ \ \cdots\cdots \ \} \ \}$$

[2]As in Java.

which has only a single method `main` and has no field. Execution of program starts by creating a `Main` object and sending it the message `main`

```
(new Main).main(null)
```

This is equivalent to the *main* function in a Java or C++ program.

2.2 An Example GM Program

Figure 2.1 shows a GM class that may be used to build natural numbers. GM requires every method to take one parameter, even if the method does not need any. The successor method `succ` simply never refers to its parameter α in the method body, and the value `null` is passed to `succ` on invocation. This slight semantic redundancy is traded off by syntactic simplicity.[3]

2.3 Terminology of Casting

Upcast and *downcast* are well-established jargon in object-oriented programming. Inheritance introduces a class hierarchy, which may be thought as an inverted tree with parents (superclasses) on top of children (subclasses). Casting an expression from a child class to a parent class is called an upcast. Let $\texttt{class}\ C\ \texttt{extends}\ D\ \{\ \cdots\ \}$. If $e : C$, then $(D)\ e$ is an upcast. Visually we go up the tree. Casting in the other direction, i.e. from a parent class to a child class, is called a downcast. If $e : D$, then $(C)\ e$ is a downcast. In this case, we come down the tree. In Java, upcast and downcast are known as widening reference conversion and narrowing reference conversion, respectively. For details, see Chapter 5 of [GJS96].

2.4 Auxiliary Functions

The keyword *extends* introduces a *subclassing* relation $\preccurlyeq$. Class C is a subclass of D, written as $C \preccurlyeq' D$, if and only if

$$\texttt{class}\ C\ \texttt{extends}\ D\ \{\ \cdots\ \}$$

[3]In fact, parameter-less methods can be encoded as a syntactic sugar.

```
class NatNum extends Object
{
   Nat pred;

   NatNum succ(Object α)
   {
     NatNum x;
     x = new NatNum;
     x.pred = this;
     return x
   }

   NatNum eq(NatNum n)
   {
     NatNum x;
     if this.pred then
       if n.pred then
         x = this.pred.eq(n.pred);
       else
         x = n.pred;
       fi
     else
       if n.pred then
         x = this.pred;
       else
         x = this.succ(null);
       fi;
     fi;
     return x
   }
}
```

Figure 2.1: Natural Numbers in GM

i.e. C derives or inherits from D. The relation $\preccurlyeq$ is defined to be the reflexive, transitive closure of $\preccurlyeq'$.

Next, let

$$\begin{aligned} \mathcal{CLASS} &= \text{the set of class names} \\ \mathcal{METH} &= \text{the set of method names} \\ \mathcal{IDEN} &= \text{the set of variable names or identifiers} \\ \mathcal{TYPE} &= \text{the set of types} \end{aligned}$$

The set of types $\mathcal{TYPE}$is defined by the following grammar

$$\mathcal{T} ::= \ C \mid \texttt{Void} \mid \mathcal{T} \rightarrow \mathcal{T}$$

where C is a meta-variable ranging over class names.

A *class table* is a tuple $\Psi = (\Psi_{\mathcal{F}}, \Psi_{\mathcal{M}}, \Psi_{\preccurlyeq})$ made up of three partial maps to be interpreted as follows.

- $\Psi_{\mathcal{F}} : \mathcal{METH} \times \mathcal{IDEN} \rightarrow \mathcal{TYPE}$
  $\Psi_{\mathcal{F}}(C, f)$ returns the type of field f defined in class C or the nearest superclass of C. In other words, re-declaring an inherited field to a type hides the field of the superclass.
- $\Psi_{\mathcal{M}} : \mathcal{CLASS} \times \mathcal{METH} \rightarrow \mathcal{TYPE}$
  $\Psi_{\mathcal{M}}(C, m)$ returns the type of method m in class C. Specializing method type is not allowed.
- $\Psi_{\preccurlyeq} : \mathcal{CLASS} \rightarrow \mathcal{CLASS}$
  $\Psi_{\preccurlyeq}(C)$ returns the type of superclass of C. This is well-defined since we consider only single inheritance.

We need a function to access method body and the formal parameter

$$\texttt{method}(C, m) \triangleq \begin{cases} (x : B\ ,\ \mathcal{G}), & \text{if } m = \mathcal{T}\ \ m(C\ x)\ \{\ \mathsf{P}\ \} \text{ and } m \in \{m_1, m_2, \cdots\} \\ \texttt{method}(C', m) & \text{if } m \notin \{m_1, m_2, \cdots\} \end{cases}$$
$$\text{where } \ \texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \cdots\ m_1\ m_2\ \cdots\ \}$$

a function to extract the method body

$$\texttt{body}(C, m) \triangleq \mathcal{G} \quad \text{where} \quad (\_\ , \mathcal{G}) = \texttt{method}(C, m)$$

and finally a function to extract the fields of a class and its superclasses

$$\begin{aligned}\texttt{fields}(C) \;&\triangleq\; \{f_1, \cdots, f_n\} \cup \texttt{fields}(C') \\ &\quad \text{where } \texttt{class } C \texttt{ extends } C' \; \{ \; C_1 \; f_1 \cdots C_n \; f_n \; \mathsf{MS} \; \}\end{aligned}$$

2.5 State Graph

A state graph is a directed labelled graph with two distinguished vertices, namely a source vertex and a final vertex, plus a set of special edges, called the scoping environment. A vertex is a class name indexed by an integer or the symbol $\bot$. Each edge is labelled by a field name or variable name indexed by a number. Formally, let

$$\mathcal{VERTEX} = \mathcal{CLASS} \times \mathbb{N} \cup \{\bot\}$$

be the set of vertices and let

$$\mathcal{LABEL} = \mathcal{IDEN} \times \mathbb{N} \cup \mathcal{IDEN}$$

be the set of labels. Then a state graph is a tuple (V, E, c, r, Ω) where

- $V \subseteq \mathcal{VERTEX}$ and $\bot \in V$.
- $E : V \times \mathcal{LABEL} \rightarrow V$ is a partial function that defines the edges of the graph. Or equivalently, E can be viewed as a subset of $V \times \mathcal{LABEL} \times V$. We sometimes use the notation $u \overset{l}{\rightarrowtail} v$ to denote a directed edge.
- $c \in V$ is the source vertex or the current vertex.
- $r \in V$ is the final vertex or the result vertex.
- $\Omega \subseteq \mathcal{LABEL} \times \mathbb{N}$ is a list of non-field labels (x, i).

Intuitively, a vertex (C, i) represents an instance of class C. The index i is used to make every instance distinct and is selected by the function

$$\texttt{next}(C, V) = \min \{k \mid (C, k) \notin V\}$$

The min of an empty set is taken to be zero. A label (x, i), or simply a field name f, is a reference to an object. The index i is to make each local variable name unique, which together with the run-time environment Ω, will facilitate scoping. By a slight abuse of notation, we use the same function name to select the next unused labelling index.

$$\texttt{next}(x, E) = \min\{k \mid \exists u \exists v.(u, (x, k), v) \notin E\}$$

The source vertex represents the current object under consideration, or the keyword *this*. The final vertex is the result of a computation. The special vertex $\bot$ corresponds to *null*. It lets us differentiate between un-initialized versus un-declared variables. Further, it serves as logical comparison in conditional statements. It is also used in the initial state $(\phi, \phi, \bot, \bot, \phi)$ of an execution.

2.5.1 Operations on State Graph

We define three basic operations on state graphs. Let $u, w \in V$ and $\alpha \in \mathcal{LABEL}$. A new edge is added by the $\oplus$ operator if it is not there.

$$E \oplus (u, \alpha, w) = \begin{cases} E \cup \{(u, \alpha, w)\} & \text{if } (u, \alpha, w) \notin E \\ E & \text{otherwise} \end{cases}$$

An existing edge is modified by the $\odot$ operator.

$$E \odot (u, \alpha, w) = \begin{cases} (E \setminus \{(u, \alpha, v)\}) \cup \{(u, \alpha, w)\} & \text{if } (u, \alpha, v) \in E \text{ for some } v \\ E & \text{otherwise} \end{cases}$$

Finally, an edge is removed by the $\ominus$ operator.

$$E \ominus (u, \alpha) = \begin{cases} E \setminus \{(u, \alpha, v)\} & \text{if } (u, \alpha, v) \in E \text{ for some } v \\ E & \text{otherwise} \end{cases}$$

This will be used to de-allocate formal parameters and local variables.

2.5.2 Notational Convention

Unless otherwise stated, σ will always denote the tuple (V, E, c, r, Ω). A subscript or superscript will apply to every component of the tuple, e.g.

$$\sigma' = (V', E', c', r', \Omega')$$

$$\sigma_\kappa = (V_\kappa, E_\kappa, c_\kappa, r_\kappa, \Omega_\kappa)$$

An underscore '_' denotes an object of the proper kind which we do not care about. For instance, the underscore in a vertex $(C, \_)$ represents some integer of which the value is not important.

The symbol $\sigma[E'/E]$ stands for a new state graph created from σ by substituting a new edge function E' for the original one, i.e.

$$\sigma[E'/E] = (V, E', c, r, \Omega)$$

The substitution operation [/] applies similarly to other components of the state graph. Sometimes, we write

$$\sigma[\cdot, E', \cdot, \cdot, \cdot] = (V, E', c, r, \Omega)$$

$$\sigma'[\cdot, \cdot, c_2, r_3, \cdot] = (V', E', c_2, r_3, \Omega')$$

to emphasize the affected components. The symbol $\cdot$, a dot, is merely a place-holder to represent the original content.

Finally, in a well-behaved program, a variable x is accessed only if it is in scope. From time to time, we simply write $E(c, x)$ instead of $E(c, (x, j))$ for some $(x, j, \_) \in \Omega$. This is especially the case in dealing with axiomatic semantics.

Chapter 3

Operational Semantics

Operational semantics define a programming language by means of deductive system, in which the judgements are transitions of machine states. It has two main flavours that take opposite view of describing program execution. *Structural operational semantics* strives to capture the smallest possible changes in configuration, while *natural semantics* emphasizes how the overall result of computation is obtained. For this reason, structural operational semantics is sometimes called small-step semantics and natural semantics is called big-step semantics. The term *natural semantics* was coined by a research group at INRIA [Kah87]. They demonstrated the framework by actually implementing many examples in type checking, translation between programming languages and modelling execution.

We define GM by natural semantics and then make a brief detour into structural operational semantics. Denotational semantics resembles closely natural semantics and they can be viewed as notational variants of each other.[1] We do not pursue it here.

3.1 Natural Semantics

The natural semantics of GM is a deductive system that defines a transition relation of the form

$$\langle\!\langle\, \mathsf{t} \,,\, \sigma \,\rangle\!\rangle \rightsquigarrow \sigma'$$

Here t is a method body, a statement or an expression; σ and σ' are state graphs. Intuitively, the transition means that execution of t against initial σ will terminate and the resulting

[1]Chapter 8 of [SK95]

state is σ'. It describes run-time behaviour of GM. Class declaration in is merely a static entity used to build up the class table. Those syntactic elements for class declaration are therefore ignored in the semantics.

The semantics imposes reduction order in the following sense. Premises in the rules are read from left to right and from top to bottom. There is exactly one rule for each syntactic entity. At any point during execution, either one or no rule applies.

Recall that a state graph is a tuple $\sigma = (V, E, c, r, \Omega)$. We apply the notational convention laid out in Section 2.5.2.

3.1.1 Expression: Keywords

Keyword *this* means the object under consideration.[2] So evaluating *this* marks the current vertex as the result vertex.

$$\texttt{[E-THIS]} \qquad \langle\!\langle\ \texttt{this}\ ,\ (V, E, c, r, \Omega)\ \rangle\!\rangle \ \rightsquigarrow\ (V, E, c, c, \Omega)$$

Evaluating *null* sets $\perp$ as the result vertex.

$$\texttt{[E-NULL]} \qquad \langle\!\langle\ \texttt{null}\ ,\ (V, E, c, r, \Omega)\ \rangle\!\rangle \ \rightsquigarrow\ (V, E, c, \perp, \Omega)$$

3.1.2 Expression: Variable Access

Evaluating a variable gives the object pointed to by the current object via an edge labelled by the variable name. During program execution, there can be more than one edge originating from the current vertex labelled by the same variable name. We have to pick the one that is currently in scope, i.e. the one that is recorded in the run-time environment.

$$\texttt{[E-VACC]} \qquad \langle\!\langle\ x\ ,\ \sigma\ \rangle\!\rangle \ \rightsquigarrow\ (V, E, c, E(c, (x, i)), \Omega)$$
$$\text{where} \quad (x, i) \in \Omega$$

[2]The keyword *self* or *current* is used for this purpose in some other object-oriented languages.

3.1.3 Expression: Field Access

Accessing a field requires the owner of the field, which is the result of evaluating $\mathcal{E}$. No scoping is involved since the field name is unique within the object.

$$\texttt{[E-FACC]} \quad \frac{\langle\!\langle\, \mathcal{E} \,,\, \sigma \,\rangle\!\rangle \;\rightsquigarrow\; \sigma'}{\langle\!\langle\, \mathcal{E}.f \,,\, \sigma \,\rangle\!\rangle \;\rightsquigarrow\; (V', E', c', E'(r', f), \Omega')}$$

3.1.4 Expression: Instantiation

Object creation introduces a new vertex into the state graph. All fields come into existence and are initialized to null. The result vertex is, of course, the new object itself.

$$\begin{aligned} \texttt{[E-NEW]} \quad & \langle\!\langle\, \texttt{new}\ C \,,\, \sigma \,\rangle\!\rangle \;\rightsquigarrow\; (V \cup \{v\}, E', c, v, \Omega) \\ & \text{where}\ \ v = (C, \texttt{next}(C, V)) \\ & \qquad\quad E' \triangleq E \oplus (v, f_1, \bot) \oplus \cdots \oplus (v, f_n, \bot) \\ & \qquad\quad f_1, \cdots, f_n \in \texttt{fields(class}(v)\texttt{)} \end{aligned}$$

3.1.5 Expression: Casting

Casting has no run-time effect on the state. The expression to be cast is first reduced to an object. Then its type is checked against the target of the cast. If it is a parent class of the target, i.e. an upcast, the cast is thrown away and execution goes on. Otherwise, no rule applies and execution gets stuck.

$$\texttt{[E-CAST]} \quad \frac{\langle\!\langle\, \mathcal{E} \,,\, \sigma \,\rangle\!\rangle \;\rightsquigarrow\; \sigma' \qquad \texttt{class}(r') \preccurlyeq C}{\langle\!\langle\, (C)\ \mathcal{E} \,,\, \sigma \,\rangle\!\rangle \;\rightsquigarrow\; \sigma'}$$

Notice that casting is the only place in the semantics where a run-time type check is performed. Upcasts are always safe. In Java, they "never throw an exception at run time. They consist simply in regarding a reference as having some other type in a manner that can be proved correct at compile time."[3]

[3]Refer to Chapter 5 of [GJS96]

3.1.6 Method Invocation

Method invocation has the most complex rule in the semantics. It consists of finding the message receiver, calculating the parameter and then calling the proper method.

$$\texttt{[E-INVK]}\quad \frac{\langle\!\langle\, \mathcal{E}_0 \,,\, \sigma \,\rangle\!\rangle \rightsquigarrow \sigma_0 \qquad \langle\!\langle\, \mathcal{E}_1 \,,\, \sigma_0 \,\rangle\!\rangle \rightsquigarrow \sigma_1 \qquad \langle\!\langle\, \mathcal{G} \,,\, \sigma' \,\rangle\!\rangle \rightsquigarrow \sigma_2}{\langle\!\langle\, \mathcal{E}_0.m(\mathcal{E}_1) \,,\, \sigma \,\rangle\!\rangle \rightsquigarrow \sigma''}$$

$$\begin{aligned} \text{where}\quad (x : \_\,, \mathcal{G}) &= \texttt{method}(\texttt{class}(r_0), m) \\ \sigma' &= (V_1, E_1 \oplus r_0 \overset{(x,j)}{\rightarrowtail} r_1, r_0, r_1, \{(x,j)\}) \\ \sigma'' &= (V_2, \texttt{del}(E_2, r_0), c_0, r_2, \Omega_1) \end{aligned}$$

The message receiver is the result of evaluating the expression $\mathcal{E}_0$, represented by r_0. The parameter value is r_1, the result of evaluating the expression $\mathcal{E}_1$. The actual method body $\mathcal{G}$ is determined *dynamically* by the run-time type of $\mathcal{E}_0$, i.e. $\texttt{class}(r_0)$. The method is executed against an initial state σ', which can be interpreted as follows.

- The message receiver r_0 is the current vertex.
- The formal parameter x and local variables come into existence. Parameter x is bound to the value r_1.[4]
- A new scope Ω' contains the formal parameter and local variables.

Notice that it does not matter whether σ' has r_0 or r_1 as the result vertex. When method execution has finished, the final state σ'' can be understood as follows.

- Control is returned to c_0. In fact, we shall see later (Section 3.3) that c, c_0, c_1 are the same.
- Given a vertex v, define

  $$\texttt{del}(E, v) = E \ominus (v, (x, i)) \ominus \cdots$$

  where $(v, (x, i), \_\,) \in E$, to remove all variable-labelled edges from v. Then the expression $\texttt{del}(E_2, r_0)$ says that the formal parameter and local variables are destroyed.
- The previous scope Ω_1 is restored.

[4]This can be extended easily to methods accepting multiple parameters.

It is not crucial to de-allocate these variables. Any method call during execution of m will create a new scope. Once m has exited, the current scope will be removed. In both cases, these variables are not accessible. See also Chapter 6 for further discussion on this issue.

3.1.7 Statement: Variable Assignment

Variable assignment redirects the edge originating from the current object and labelled by the variable name, to the result object. This result object has been obtained by evaluating the R-value expression.

$$\texttt{[E-VASSN]} \quad \frac{\langle\!\langle\, \mathcal{E} \,,\, \sigma_0 \,\rangle\!\rangle \rightsquigarrow \sigma}{\langle\!\langle\, x = \mathcal{E} \,,\, \sigma_0 \,\rangle\!\rangle \rightsquigarrow (V, E \odot c \overset{(x,i)}{\rightarrowtail} r, c, r, \Omega)} \quad \text{where} \ (x,i) \in \Omega$$

3.1.8 Statement: Field Assignment

Field assignment works in a similar way.

$$\texttt{[E-FASSN]} \quad \frac{\langle\!\langle\, \mathcal{E} \,,\, \sigma_0 \,\rangle\!\rangle \rightsquigarrow \sigma' \qquad \langle\!\langle\, \mathcal{E}' \,,\, \sigma' \,\rangle\!\rangle \rightsquigarrow \sigma}{\langle\!\langle\, \mathcal{E}.f = \mathcal{E}' \,,\, \sigma_0 \,\rangle\!\rangle \rightsquigarrow (V, E \odot r' \overset{f}{\rightarrowtail} r, c, r, \Omega)}$$

First we find the owner of the field r' by evaluating the expression $\mathcal{E}$. Unlike variable assignment that makes use of the current vertex, we switch around the edge originating from r' to r, the result of evaluating the R-value.

3.1.9 Statement: Return

A *return* does nothing more than evaluating the expression to be returned. The *returned value* is then accessible in the result vertex.

$$\texttt{[E-RETN]} \quad \frac{\langle\!\langle\, \mathcal{E} \,,\, \sigma \,\rangle\!\rangle \rightsquigarrow \sigma'}{\langle\!\langle\, \texttt{return}\, \mathcal{E} \,,\, \sigma \,\rangle\!\rangle \rightsquigarrow \sigma'}$$

3.1.10 Statement: Conditional

There being no Boolean primitive type, a conditional statement simply tests for null, which signifies a false.

$$\texttt{[E-FALSE]} \quad \frac{\langle\!\langle\, \mathcal{E} \,,\, \sigma \,\rangle\!\rangle \leadsto \sigma' \qquad r' = \bot \qquad \langle\!\langle\, \mathcal{S}_2 \,,\, \sigma' \,\rangle\!\rangle \leadsto \sigma''}{\langle\!\langle\, \texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2 \texttt{ fi} \,,\, \sigma \,\rangle\!\rangle \leadsto \sigma''}$$

$$\texttt{[E-TRUE]} \quad \frac{\langle\!\langle\, \mathcal{E} \,,\, \sigma \,\rangle\!\rangle \leadsto \sigma' \qquad r' \neq \bot \qquad \langle\!\langle\, \mathcal{S}_1 \,,\, \sigma' \,\rangle\!\rangle \leadsto \sigma''}{\langle\!\langle\, \texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2 \texttt{ fi} \,,\, \sigma \,\rangle\!\rangle \leadsto \sigma''}$$

3.1.11 Statement: Sequencing

Statements are executed from left to right.

$$\texttt{[E-SEQ]} \quad \frac{\langle\!\langle\, \mathcal{S}_1 \,,\, \sigma \,\rangle\!\rangle \leadsto \sigma' \qquad \langle\!\langle\, \mathcal{S}_2 \,,\, \sigma' \,\rangle\!\rangle \leadsto \sigma''}{\langle\!\langle\, \mathcal{S}_1 \,;\, \mathcal{S}_2 \,,\, \sigma \,\rangle\!\rangle \leadsto \sigma''}$$

3.1.12 Method Body

Variable declaration introduces a new edge into the state graph, with the variable initialized to `null`.

$$\texttt{[E-MBODY]} \quad \frac{\langle\!\langle\, \mathcal{G} \,,\, \sigma'' \,\rangle\!\rangle \leadsto \sigma'}{\langle\!\langle\, C\ x \,;\, \mathcal{G} \,,\, \sigma \,\rangle\!\rangle \leadsto \sigma'}$$

$$\text{where } \sigma'' = (V, E \oplus c \overset{(x,j)}{\rightarrowtail} \bot, c, r, \Omega \cup \{(x, j)\})$$
$$j = \texttt{next}(x, E)$$

3.2 An Example

Figure 3.1 shows a class C with a single field n and two methods `get` and `foo`. Method `get`retrieves the internal state of its owner object. The sole purpose of `foo` is to show how the scoping environment in the state graph resolves names. We assume an already defined class D from which C is derived. For clarity, we use single letters for class names, field names and variable names.

Now we execute the program step by step according to the natural semantics. In the following series of figures, the current vertex is represented by a node with double border and the result vertex is indicated by an arrow. The un-named node is the $\bot$ vertex. A subscripted class

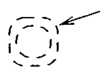
```
class C extends D
{
     D n;

     D get(Object x)
       { D y; y = this.n; this.foo(new D); return y }

     Void foo(D x)
       { D y; y = x }
}


class Main
{
   main(Object p)
   {
     C y; D x;
     y = new C;
     y.n = new D;
     x = y.get(null)
   }
}
```

Figure 3.1: A Demo Program

name C_i stands for a vertex (C, i) in the state graph. We simply write M for the `Main` node since there is one and only one such vertex. We ignore fields of D.

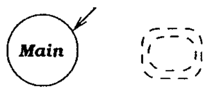
1. (`new Main`).`main(null)`

   The program starts by implicitly creating a `main` object and sends it the message `main`. The initial state is $(\{\perp\}, \{\}, \perp, \perp, \{\})$.



2. $\underbrace{(\texttt{new Main})}$`.main(null)`

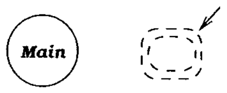
   Operator `new` introduces a new `Main` node and marks it as the result vertex. This new object will be used as the current vertex in executing the body of `main`.[5]



$$V = \{\perp, M\} \quad E = \{\}$$
$$c = \perp \qquad r = M \qquad \Omega = \{\}$$

3. (`new Main`)`.main(`$\underbrace{\texttt{null}}$`)`

   Evaluation of `null` marks $\perp$ as the result vertex.[6]

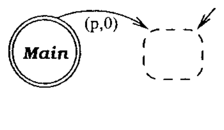


$$V = \{\perp, M\} \quad E = \{\}$$
$$c = \perp \qquad r = \perp \qquad \Omega = \{\}$$

4. (`new Main`)`.`$\underbrace{\texttt{main(null)}}$

   Invocation of `main` first sets up the parameter p. The current vertex is the M node.

[5]The resulting state corresponds to σ_0 in rule `[E-INVK]` for calling `main`.
[6]The resulting state corresponds to σ_1 in rule `[E-INVK]` for calling `main`.

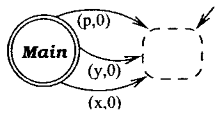


$$V = \{\perp, M\} \quad E = \{(M, p_0, \perp)\}$$

$$c = M \qquad r = \perp \qquad \Omega = \{p_0\}$$

5. $\underbrace{C\ y\ ;\ D\ x}$; $y = \texttt{new}\ C$; $y.n = \texttt{new}\ D$; $x = y.\texttt{get(null)}$

   Declaring variables y and x introduces two edges labelled by y and x, respectively, from the current vertex, the M node, to $\perp$.

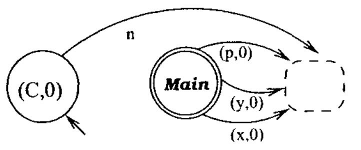


$$V = \{\perp, M\} \quad E = \{(M, p_0, \perp), (M, y_0, \perp), (M, x_0, \perp)\}$$

$$c = M \qquad r = \perp \qquad \Omega = \{p_0, y_0, x_0\}$$

6. $C\ y$; $D\ x$; $y = \underbrace{\texttt{new}\ C}$; $y.n = \texttt{new}\ D$; $x = y.\texttt{get(null)}$

   Operator `new` introduces a node of class C and makes it the result vertex. The field n of C comes into existence.

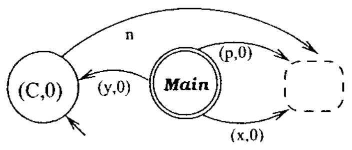


$$V = \{\perp, M, C_0\} \quad E = \{(M, p_0, \perp), (M, y_0, \perp), (M, x_0, \perp), (C_0, n, \perp)\}$$

$$c = M \qquad r = C_0 \qquad \Omega = \{p_0, y_0, x_0\}$$

7. $C\ y$; $D\ x$; $\underbrace{y = \texttt{new}\ C}$; $y.n = \texttt{new}\ D$; $x = y.\texttt{get(null)}$

   Assigning value to variable y switches the edge labelled y to the result vertex.

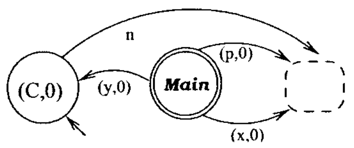


$$V = \{\perp, M, C_0\} \quad E = \{(M, p_0, \perp), (M, y_0, C_0), (M, x_0, \perp), (C_0, n, \perp)\}$$

$$c = M \qquad r = C_0 \qquad \Omega = \{p_0, y_0, x_0\}$$

8. $C\ y\ ;\ D\ x\ ;\ y = \texttt{new}\ C\ ;\ \underbrace{y}\ .n = \texttt{new}\ D\ ;\ x = y.\texttt{get(null)}$

   Evaluating y makes the destination of the edge labelled y the result vertex. In this case the state graph does not change.

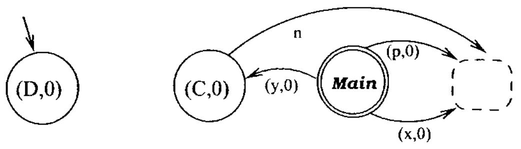


$$V = \{\perp, M, C_0\} \quad E = \{(M, p_0, \perp), (M, y_0, C_0), (M, x_0, \perp), (C_0, n, \perp)\}$$

$$c = M \qquad r = C_0 \qquad \Omega = \{p_0, y_0, x_0\}$$

9. $C\ y\ ;\ D\ x\ ;\ y = \texttt{new}\ C\ ;\ y.n = \underbrace{\texttt{new}\ D}\ ;\ x = y.\texttt{get(null)}$

   Operator new introduces a node of class D and makes it the result vertex.

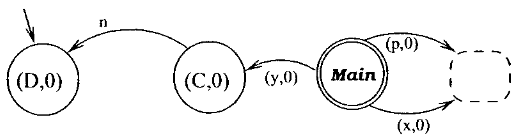


$$V = \{\perp, M, C_0, D_0\} \quad E = \{(M, p_0, \perp), (M, y_0, C_0), (M, x_0, \perp), (C_0, n, \perp)\}$$

$$c = M \qquad r = D_0 \qquad \Omega = \{p_0, y_0, x_0\}$$

10. $C\ y\ ;\ D\ x\ ;\ y = \texttt{new}\ C\ ;\ \underbrace{y.n = \texttt{new}\ D}\ ;\ x = y.\texttt{get(null)}$

    Assigning value to field n switches the edge labelled n to the result vertex.

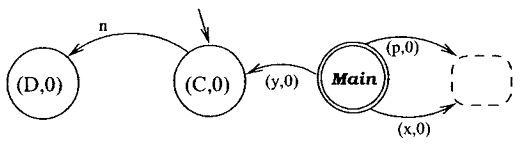


$$V = \{\bot, M, C_0, D_0\} \quad E = \{(M, p_0, \bot), (M, y_0, C_0), (M, x_0, \bot), (C_0, n, D_0)\}$$

$$c = M \qquad r = D_0 \qquad \Omega = \{p_0, y_0, x_0\}$$

11. $C\ y\ ;\ D\ x\ ;\ y = \texttt{new}\ C\ ;\ y.n = \texttt{new}\ D\ ;\ x = \underbrace{y}\texttt{.get(null)}$

    Evaluating y makes its content the result vertex, which will be used as the current vertex in executing the body of get.[7]

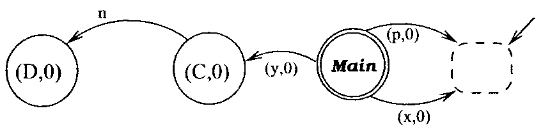


$$V = \{\bot, M, C_0, D_0\} \quad E = \{(M, p_0, \bot), (M, y_0, C_0), (M, x_0, \bot), (C_0, n, D_0)\}$$

$$c = M \qquad r = C_0 \qquad \Omega = \{p_0, y_0, x_0\}$$

12. $C\ y\ ;\ D\ x\ ;\ y = \texttt{new}\ C\ ;\ y.n = \texttt{new}\ D\ ;\ x = y\texttt{.get(}\underbrace{\texttt{null}}\texttt{)}$

    Evaluating null marks $\bot$ as the result vertex.[8]



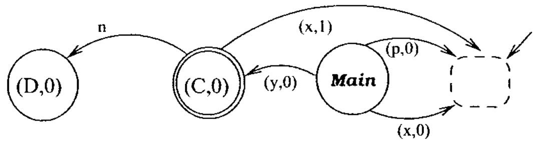
$$V = \{\bot, M, C_0, D_0\} \quad E = \{(M, p_0, \bot), (M, y_0, C_0), (M, x_0, \bot), (C_0, n, D_0)\}$$

$$c = M \qquad r = \bot \qquad \Omega = \{p_0, y_0, x_0\}$$

13. $C\ y\ ;\ D\ x\ ;\ y = \texttt{new}\ C\ ;\ y.n = \texttt{new}\ D\ ;\ x = \underbrace{y\texttt{.get(null)}}$

    Invocation of get first sets up the parameter x. The current vertex is the C node.

[7]The resulting state corresponds to σ_1 in rule [E-INVK] for calling get.
[8]The resulting state corresponds to σ_1 in rule [E-INVK] for calling get.

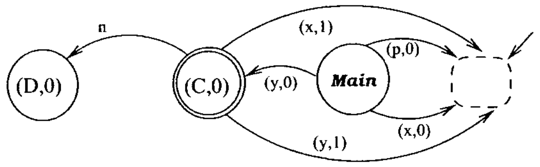


$$V = \{\bot, M, C_0, D_0\} \quad E = \{(M, p_0, \bot), (M, y_0, C_0), (M, x_0, \bot), (C_0, n, D_0), (C_0, x_1, \bot)\}$$

$$c = C_0 \qquad r = \bot \qquad \Omega = \{x_1\}$$

14. $\underbrace{D\ y}$; $y =$ this.n ; this.foo(new D) ; return y

Declaring y introduces an edge labelled y. Notice the edge is such indexed as to be different form the previously declared y.



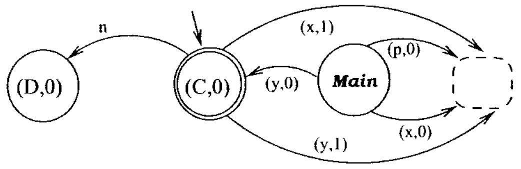
$$V = \{\bot, M, C_0, D_0\}$$

$$E = \{(M, p_0, \bot), (M, y_0, C_0), (M, x_0, \bot), (C_0, n, D_0), (C_0, x_1, \bot), (C_0, y_1, \bot)\}$$

$$c = C_0 \qquad r = \bot \qquad \Omega = \{x_1, y_1\}$$

15. $D\ y$; $y = \underbrace{\texttt{this}}.n$; this.foo(new D) ; return y

Evaluating this marks the current vertex as the result.



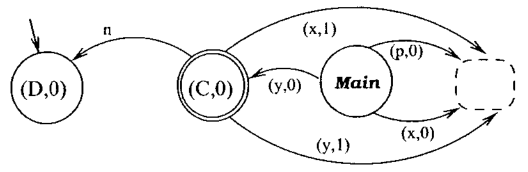
$$V = \{\bot, M, C_0, D_0\}$$

$$E = \{(M, p_0, \bot), (M, y_0, C_0), (M, x_0, \bot), (C_0, n, D_0), (C_0, x_1, \bot), (C_0, y_1, \bot)\}$$

$$c = C_0 \qquad r = C_0 \qquad \Omega = \{x_1, y_1\}$$

16. $D\ y\ ;\ y = \underbrace{\texttt{this}.n}\ ;\ \texttt{this.foo(new}\ D)\ ;\ \texttt{return}\ y$

Accessing the field n gives the destination of the edge labelled by n, which is the D node. It is marked as the result vertex.



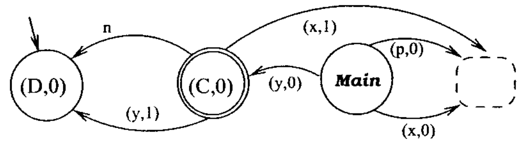
$$V = \{\bot, M, C_0, D_0\}$$

$$E = \{(M, p_0, \bot), (M, y_0, C_0), (M, x_0, \bot), (C_0, n, D_0), (C_0, x_1, \bot), (C_0, y_1, \bot)\}$$

$$c = C_0 \qquad r = D_0 \qquad \Omega = \{x_1, y_1\}$$

17. $D\ y\ ;\ \underbrace{y = \texttt{this}.n}\ ;\ \texttt{this.foo(new}\ D)\ ;\ \texttt{return}\ y$

Assigning value to y switches the edge labelled y from $\bot$ to the result vertex.



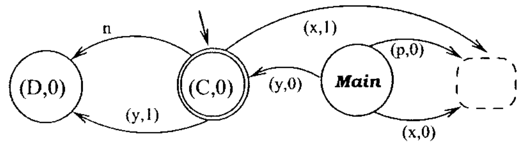
$$V = \{\bot, M, C_0, D_0\}$$

$$E = \{(M, p_0, \bot), (M, y_0, C_0), (M, x_0, \bot), (C_0, n, D_0), (C_0, x_1, \bot), (C_0, y_1, D_0)\}$$

$$c = C_0 \qquad r = D_0 \qquad \Omega = \{x_1, y_1\}$$

18. $D\ y\ ;\ y = \texttt{this}.n\ ;\ \underbrace{\texttt{this}}.\texttt{foo(new}\ D)\ ;\ \texttt{return}\ y$

Evaluating this marks the current vertex as the result vertex.



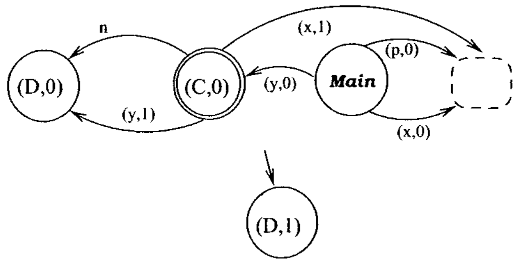
$$V = \{\perp, M, C_0, D_0\}$$

$$E = \{(M, p_0, \perp), (M, y_0, C_0), (M, x_0, \perp), (C_0, n, D_0), (C_0, x_1, \perp), (C_0, y_1, D_0)\}$$

$$c = C_0 \qquad r = C_0 \qquad \Omega = \{x_1, y_1\}$$

19. $D\ y$; $y = \texttt{this}.n$; $\texttt{this.foo}(\underbrace{\texttt{new}\ D})$; $\texttt{return}\ y$

Another D node is created. It is marked as the result vertex. Notice how it is made different from the existing D node by the index.



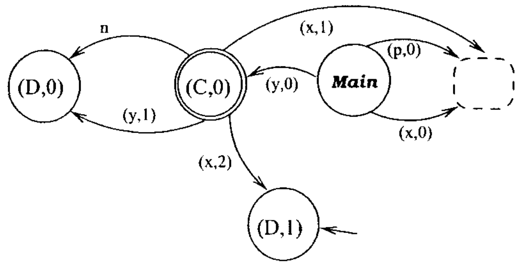
$$V = \{\perp, M, C_0, D_0, D_1\}$$

$$E = \{(M, p_0, \perp), (M, y_0, C_0), (M, x_0, \perp), (C_0, n, D_0), (C_0, x_1, \perp), (C_0, y_1, D_0)\}$$

$$c = C_0 \qquad r = D_1 \qquad \Omega = \{x_1, y_1\}$$

20. $D\ y$; $y = \texttt{this}.n$; $\underbrace{\texttt{this.foo(new}\ D)}$; $\texttt{return}\ y$

Invocation of `foo` first prepares the parameter x. The method body will be executed against a state in which the current vertex is the C node.

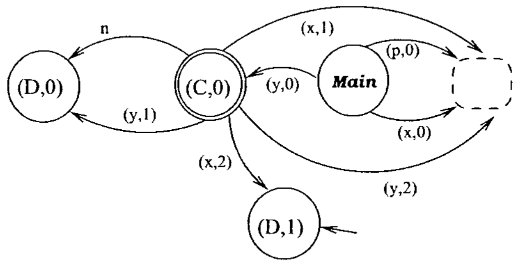


$$V = \{\perp, M, C_0, D_0, D_1\}$$

$$E = \{(M, p_0, \perp), (M, y_0, C_0), (M, x_0, \perp), (C_0, n, D_0), (C_0, x_1, \perp),$$
$$(C_0, y_1, D_0), (C_0, x_2, D_1)\}$$
$$c = C_0 \qquad r = D_1 \qquad \Omega = \{x_2\}$$

21. $\underbrace{D\ y}\ ;\ y = x$

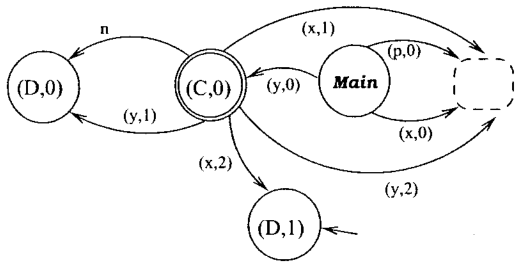
Declaring y creates a edge labelled by y.



$$V = \{\perp, M, C_0, D_0, D_1\}$$
$$E = \{(M, p_0, \perp), (M, y_0, C_0), (M, x_0, \perp), (C_0, n, D_0), (C_0, x_1, \perp),$$
$$(C_0, y_1, D_0), (C_0, x_2, D_1), (C_0, y_2, \perp)\}$$
$$c = C_0 \qquad r = D_1 \qquad \Omega = \{x_2, y_2\}$$

22. $D\ y\ ;\ y = \underbrace{\ x\ }$

Accessing variable x marks the destination of the edge labelled x as the result vertex.



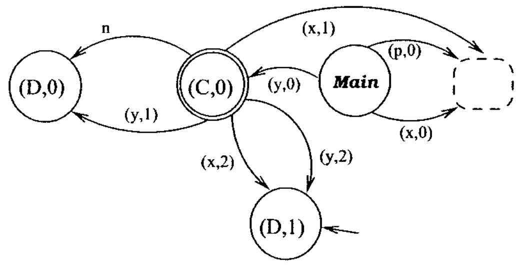
$$V = \{\perp, M, C_0, D_0, D_1\}$$
$$E = \{(M, p_0, \perp), (M, y_0, C_0), (M, x_0, \perp), (C_0, n, D_0), (C_0, x_1, \perp),$$

$$(C_0, y_1, D_0), (C_0, x_2, D_1), (C_0, y_2, \perp)\}$$

$$c = C_0 \qquad r = D_1 \qquad \Omega = \{x_2, y_2\}$$

23. $D\ y\ ;\ \underbrace{y = x}$

    Assigning value to y switches the edge labelled y to the result vertex. Now there are two such edges going out from the current vertex. The scoping environment decides which one should be used.



$$V = \{\perp, M, C_0, D_0, D_1\}$$

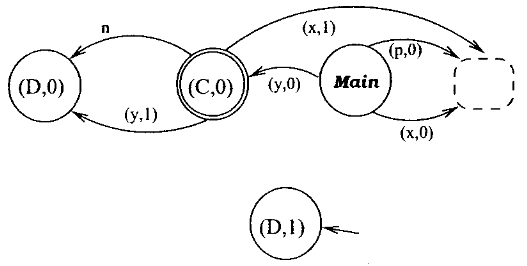
$$E = \{(M, p_0, \perp), (M, y_0, C_0), (M, x_0, \perp), (C_0, n, D_0), (C_0, x_1, \perp),$$

$$(C_0, y_1, D_0), (C_0, x_2, D_1), (C_0, y_2, D_1)\}$$

$$c = C_0 \qquad r = D_1 \qquad \Omega = \{x_2, y_2\}$$

24. $D\ y\ ;\ y = \texttt{this}.n\ ;\ \underbrace{\texttt{this.foo(new}\ D\texttt{)}}\ ;\ \texttt{return}\ y$

    Method `foo` exits and the original vertex and scoping environment are restored.



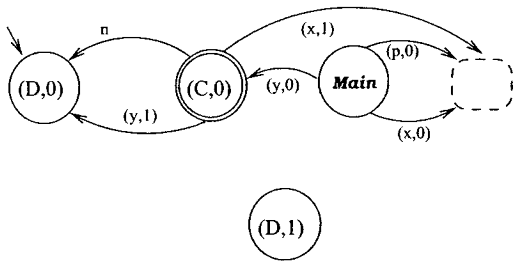
$$V = \{\perp, M, C_0, D_0, D_1\}$$

$$E = \{(M, p_0, \perp), (M, y_0, C_0), (M, x_0, \perp), (C_0, n, D_0), (C_0, x_1, \perp), (C_0, y_1, D_0)\}$$

$$c = C_0 \qquad r = D_1 \qquad \Omega = \{x_1, y_1\}$$

25. $D\ y$; $y =$ `this`.n ; `this.foo(new` D) ; $\underbrace{\texttt{return}\ y}$

Returning y is the same as evaluating y. The destination of the edge labelled y is marked as the result vertex.



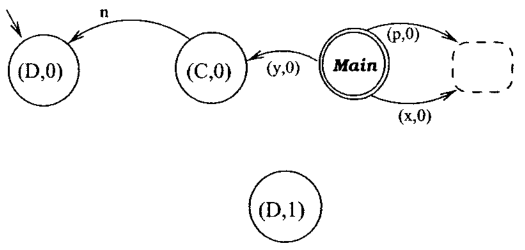
$$V = \{\perp, M, C_0, D_0, D_1\}$$

$$E = \{(M, p_0, \perp), (M, y_0, C_0), (M, x_0, \perp), (C_0, n, D_0), (C_0, x_1, \perp), (C_0, y_1, D_0)\}$$

$$c = C_0 \qquad r = D_0 \qquad \Omega = \{x_1, y_1\}$$

26. $C\ y$; $D\ x$; $y =$ `new` C ; $y.n =$ `new` D ; $x = \underbrace{y.\texttt{get(null)}}$

Method `get` has finished. Again the original vertex and scoping environment are restored.

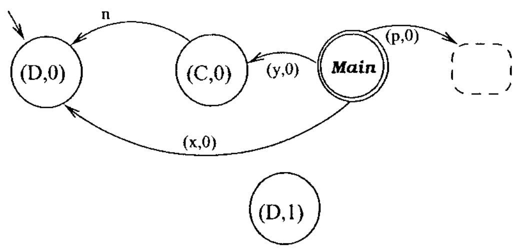


$$V = \{\perp, M, C_0, D_0, D_1\} \quad E = \{(M, p_0, \perp), (M, y_0, C_0), (M, x_0, \perp), (C_0, n, D_0)\}$$

$$c = C_0 \qquad r = D_0 \qquad \Omega = \{p_0, y_0, x_0\}$$

27. $C\ y$; $D\ x$; $y =$ `new` C ; $y.n =$ `new` D ; $\underbrace{x = y.\texttt{get(null)}}$

Assigning value to x switches the edge labelled x from $\perp$ to the result vertex.



$$V = \{\bot, M, C_0, D_0, D_1\} \quad E = \{(M, p_0, \bot), (M, y_0, C_0), (M, x_0, D_0), (C_0, n, D_0)\}$$

$$c = C_0 \qquad r = D_0 \qquad \Omega = \{p_0, y_0, x_0\}$$

3.3 Invariants

A method call in GM is in effect execution of a sequence of statements. Within a method call, the current object is always the method receiver. Local variables, once declared, are accessible throughout (and are accessible only within) the method body. Hence, among the five components in a state graph tuple, two of them remain unchanged during execution within the same scope.

Lemma 1 *Let t be a GM expression or statement. If* $\langle\!\langle\ \mathsf{t}\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma'$*, then* $c = c'$ *and* $\Omega = \Omega'$.

Proof

Straightforward induction on the derivation of $\langle\!\langle\ \mathsf{t}\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma'$.

□

3.4 Structural Operational Semantics

Now we investigate the small-step semantics of GM. To describe individual steps of executions, the transition relation must take two possible forms

$$\langle\!\langle\ \mathsf{t}\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \sigma' \tag{†}$$

$$\langle\!\langle\ \mathsf{t}\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \langle\!\langle\ \mathsf{t}'\ ,\ \sigma'\ \rangle\!\rangle \tag{††}$$

where t is a term, and σ, σ' are state graphs. While (†) indicates a completed execution as in the natural semantics, (††) says that the execution of t against σ has not yet finished and the remaining computation is expressed by $\langle\!\langle\ \mathsf{t}'\ ,\ \sigma'\ \rangle\!\rangle$.

For example, to access the field f via the expression $(x.f').f$, the owner $x.f'$ has to be evaluated first. In general, to access the content of a field $\mathcal{E}.f$, its owner $\mathcal{E}$ may need to be simplified a few times by a rule of the form (††)

$$\frac{\langle\!\langle\ \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \langle\!\langle\ \mathcal{E}'\ ,\ \sigma'\ \rangle\!\rangle}{\langle\!\langle\ \mathcal{E}.f\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \langle\!\langle\ \mathcal{E}'.f\ ,\ \sigma'\ \rangle\!\rangle}$$

and once $\mathcal{E}$ is completely evaluated, we can get the value of f by the rule

$$\frac{\langle\!\langle\ \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \sigma'}{\langle\!\langle\ \mathcal{E}.f\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow (V', E', c', E'(r', f), \Omega')}$$

These two rules together capture the essence of step-by-step execution. Similarly, small-step execution of sequence of statements can be formulated as

$$\frac{\langle\!\langle\ \mathcal{S}_1\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \langle\!\langle\ \mathcal{S}_1'\ ,\ \sigma'\ \rangle\!\rangle}{\langle\!\langle\ \mathcal{S}_1\ ;\mathcal{S}_2\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \langle\!\langle\ \mathcal{S}_1';\mathcal{S}_2\ ,\ \sigma'\ \rangle\!\rangle}$$

$$\frac{\langle\!\langle\ \mathcal{S}_1\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \sigma'}{\langle\!\langle\ \mathcal{S}_1\ ;\mathcal{S}_2\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \langle\!\langle\ \mathcal{S}_2\ ,\ \sigma'\ \rangle\!\rangle}$$

The treatment of method invocation in GM, unfortunately, is not quite suitable for a small-step description. We do have intermediate transitions of message owner and parameter,

$$\frac{\langle\!\langle\ \mathcal{E}_0\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \langle\!\langle\ \mathcal{E}_0'\ ,\ \sigma'\ \rangle\!\rangle}{\langle\!\langle\ \mathcal{E}_0.m(\mathcal{E}_1)\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \langle\!\langle\ \mathcal{E}_0'.m(\mathcal{E}_1)\ ,\ \sigma'\ \rangle\!\rangle}$$

$$\frac{\langle\!\langle\ \mathcal{E}_0\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \sigma' \qquad \langle\!\langle\ \mathcal{E}_1\ ,\ \sigma'\ \rangle\!\rangle \hookrightarrow \langle\!\langle\ \mathcal{E}_1'\ ,\ \sigma''\ \rangle\!\rangle}{\langle\!\langle\ \mathcal{E}_0.m(\mathcal{E}_1)\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \langle\!\langle\ \mathcal{E}_0.m(\mathcal{E}_1')\ ,\ \sigma''\ \rangle\!\rangle}$$

but then we are stuck with execution of the method body

$$\frac{\langle\!\langle\ \mathcal{E}_0\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \sigma_0 \qquad \langle\!\langle\ \mathcal{E}_1\ ,\ \sigma_0\ \rangle\!\rangle \hookrightarrow \sigma_1 \qquad \langle\!\langle\ \mathcal{G}\ ,\ \sigma'\ \rangle\!\rangle \hookrightarrow \langle\!\langle\ \mathcal{G}'\ ,\ \sigma''\ \rangle\!\rangle}{\langle\!\langle\ \mathcal{E}_0.m(\mathcal{E}_1)\ ,\ \sigma\ \rangle\!\rangle \hookrightarrow \langle\!\langle\ ?\ ,\ \sigma''\ \rangle\!\rangle}$$

While an expression evaluates to an expression and a statement reduces to a statement, there is no obvious in-between entity before the end of invocation.

Structural operational semantics has both advantages and disadvantages versus natural semantics. In structural operational semantics looping corresponds to an infinite sequence of transitions and abnormal termination ends in a stuck state. However they are indistinguishable in natural semantics. Natural semantics models better non-deterministic constructs but is not able to express interleaving of commands. For details, refer to [NN92].

Chapter 4

Type System

In object-oriented programming, the exact type of an expression cannot be determined until run-time. Yet type checking is able to predict a type for each expression based on the text of a program without running it. This type information is vital in reasoning about correctness of GM programs, as we shall see in Section 5.1.

4.1 Type Checking GM

GM has two categories of types. A simple type is made up of class names and a special type `Void`. A composite type consists of arrow types to describe methods. The type `Void` is simply a technicality that accounts for statements and methods that return nothing.

The type system of GM is built by four relations $\vdash_{\mathcal{P}}$, $\vdash_{\mathcal{C}}$, $\vdash_{\mathcal{M}}$ and $\vdash$. The first three relations specify that a program, a class declaration and a method declaration are well-formed, respectively. They depend on the class table described in Section 2.4. The last relation handles expressions and statements. It requires the class table and the typing environment as well. These relations are defined by a deductive system. Judgments in the system are sometimes called typing judgements.

A *typing environment* or a typing context Γ is an association of variables with types. It can be thought of as a finite unordered list $\{x_1 : T_1, x_2 : T_2, \cdots\}$ or a finite partial map $\Gamma : \mathcal{IDEN} \rightarrow \mathcal{TYPE}$. We use the set notation $\Gamma \cup x : C$ to denote the operation that the variable x of type C is added to environment Γ.[1]

[1]For conciseness, the brackets in $\Gamma \cup \{x : C\}$ are omitted.

4.2 Typing Rules

The type system is *syntax-directed*, in the sense that at most one rule is applicable at a time. An advantage is that the rules can be read "from bottom to top" and transformed into a type checking algorithm.

4.2.1 Method Body

A typing judgement of an expression or a statement has the form $\Psi, \Gamma \vdash e : C$, where Ψ is the class table and Γ is a typing environment. Expressions cannot have the type `Void`, while statements, except a `return`, have type `Void`.

The keyword `null` assumes any available class as its type.

$$\texttt{[T-NULL]} \qquad \Psi, \Gamma \vdash \texttt{null} : C$$

The type for keyword `this` is extracted from the environment. As a matter of fact, it may be regarded as a variable.

$$\texttt{[T-THIS]} \qquad \Psi, \Gamma \oplus \texttt{this} : C \vdash \texttt{this} : C$$

The type of a variable is extracted from the typing environment.

$$\texttt{[T-VACC]} \qquad \Psi, \Gamma \cup x : C \vdash x : C$$

Unlike variables, type of a field is read off from the class table, and is independent of the environment.

$$\texttt{[T-FACC]} \qquad \frac{\Psi, \Gamma \vdash \mathcal{E} : C \qquad \Psi_{\mathcal{F}}(C, f) = D}{\Psi, \Gamma \vdash \mathcal{E}.f : D}$$

The type of a newly created object is the class from which the object is instantiated.

$$\texttt{[T-NEW]} \qquad \Psi, \Gamma \vdash \texttt{new}\ C : C$$

A cast changes the type of an object. GM only allows upcasts, i.e. casting an expression to a superclass.

$$\texttt{[T-CAST]} \quad \frac{\Psi,\Gamma \vdash \mathcal{E} : C \qquad C \preccurlyeq D}{\Psi,\Gamma \vdash (D)\ \mathcal{E} : D}$$

The type of a method invocation is the result type of the method. When it is `Void`, the invocation acts as a statement. Otherwise, the invocation is an expression. The parameter type can be of a subclass of the domain type of the method.

$$\texttt{[T-INVK]} \quad \frac{\begin{array}{c}\Psi,\Gamma \vdash \mathcal{E} : C \qquad \Psi_{\mathcal{M}}(C,m) = B \to T \\ \Psi,\Gamma \vdash \mathcal{E}' : D \qquad D \preccurlyeq B\end{array}}{\Psi,\Gamma \vdash \mathcal{E}.m(\mathcal{E}') : T}$$

In assignment, the R-value object can be a subclass of the L-value object.

$$\texttt{[T-VASSN]} \quad \frac{\Psi,\Gamma \vdash x : C \qquad \Psi,\Gamma \vdash \mathcal{E} : D \qquad D \preccurlyeq C}{\Psi,\Gamma \vdash x = \mathcal{E} : \texttt{Void}}$$

Field assignment is typed in the same way as variable assignment.

$$\texttt{[T-FASSN]} \quad \frac{\Psi,\Gamma \vdash \mathcal{E}.f : C \qquad \Psi,\Gamma \vdash \mathcal{E}' : D \qquad D \preccurlyeq C}{\Psi,\Gamma \vdash \mathcal{E}.f = \mathcal{E}' : \texttt{Void}}$$

An if-then-else statement is well-typed if the test expression and both the if-branch and else-branch are well-typed. The final type of the statement is `Void`. Thus a *return* is prohibited in either branch $\mathcal{S}_1$ or $\mathcal{S}_2$.

$$\texttt{[T-IF]} \quad \frac{\Psi,\Gamma \vdash \mathcal{E} : C \qquad \Psi,\Gamma \vdash \mathcal{S}_1 : \texttt{Void} \qquad \Psi,\Gamma \vdash \mathcal{S}_2 : \texttt{Void}}{\Psi,\Gamma \vdash \texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2\ \texttt{fi} : \texttt{Void}}$$

The type of a sequence of two statements is the type of the last one. This enforces that a *return*, if there is one, can appear only at very end of a sequence of statements.

$$\texttt{[T-SEQ]} \quad \frac{\Psi,\Gamma \vdash \mathcal{S}_1 : \texttt{Void} \quad \Psi,\Gamma \vdash \mathcal{S}_2 : T}{\Psi,\Gamma \vdash \mathcal{S}_1\ ;\ \mathcal{S}_2 : T}$$

The type of a *return* is the same as that of the object to be returned.

$$\texttt{[T-RET]} \quad \frac{\Psi,\Gamma \vdash \mathcal{E} : C}{\Psi,\Gamma \vdash \texttt{return}\ \mathcal{E} : C}$$

Variable declaration introduces a new symbol into the environment and does not change the type of the statement in the body.

$$\texttt{[T-DECL]} \quad \frac{\Psi, \Gamma \cup x : C \vdash \mathcal{G} : T}{\Psi, \Gamma \vdash C\ x\ ;\ \mathcal{G} : T}$$

4.2.2 Class Declaration

A method is well-typed if the method body is well-typed and its type matches the returned type of the method. This is denoted by judgement of the form $\Psi \vdash_{\mathcal{M}} \mathsf{M}$.

$$\texttt{[T-METH]} \quad \frac{\Psi, \Gamma \cup \texttt{this} : C \cup x : D \vdash \mathcal{G} : T \qquad \Psi_{\mathcal{M}}(C, m) = C \rightarrow T}{\Psi \vdash_{\mathcal{M}} T\ \ m(D\ x)\ \{\ \mathcal{G}\ \}}$$

A class is well-typed if its methods are well-typed. This is denoted by judgement of the form $\Psi \vdash_{\mathcal{C}} L$.

$$\texttt{[T-CLASS-1]} \quad \frac{\Psi \vdash_{\mathcal{M}} \mathsf{M} \qquad \Psi \vdash_{\mathcal{M}} \mathsf{MS}}{\Psi \vdash_{\mathcal{M}} \mathsf{M\ MS}}$$

$$\texttt{[T-CLASS-2]} \quad \frac{\Psi \vdash_{\mathcal{M}} \mathsf{MS}}{\Psi \vdash_{\mathcal{C}} \texttt{class}\ C\ \texttt{extends}\ D\ \{\ \mathsf{FS\ MS}\ \}}$$

A program is well-typed if all the classes and the main statement are well-typed. This is denoted by judgement of the form $\Psi \vdash_{\mathcal{P}} P$.

$$\texttt{[T-PROG]} \quad \frac{\Psi \vdash_{\mathcal{C}} \mathsf{K} \quad \Psi \vdash_{\mathcal{P}} \mathsf{G}}{\Psi \vdash_{\mathcal{P}} \mathsf{K\ G}}$$

4.3 Type Safety

The fundamental purpose of a type system is to prevent the occurrence of execution errors during the running of a program. This property is known as *type soundness* or type safety. It is a relation between the type system and the operational semantics. Robin Milner used the slogan "Well-typed programs never go wrong" to describe type safety in the context of ML [Mil78]:

- Preservation
  "Well typedness" of programs remains invariant under the transition rules, i.e. evaluation rules or reduction rules of the language.

- Progress
  A well typed program never gets "stuck", i.e. never gets into an undefined state where no further transitions are possible.

Such formulation seems to be more appropriate for the paradigm of functional languages. The notion of type soundness varies between programming languages, and is in most cases far from trivial.

In object-oriented programming, there are two obvious kinds of run-time error. First, a field $\mathcal{E}.f$ is not recognized by its owner $\mathcal{E}$. In terms of state graph, it means the vertex representing $\mathcal{E}$ lacks an outgoing edge labelled by f. Second, a message $\mathcal{E}.m(\_)$ is not understood by its receiver $\mathcal{E}$. Again, in the setting of state graph, the vertex representing $\mathcal{E}$ is a class in which m is not declared. Ideally, if $\mathcal{E}$ is predicted by the type system to have type C, then $\mathcal{E}$ should evaluate to an object of type C' such that $C' \preccurlyeq C$. And indeed in a well-typed GM program, expressions are evaluated as expected. To see this, we need to extend typing to state graphs.

4.3.1 State Typing

We say that a state graph σ is *well-typed* with respect to a typing environment Γ, denoted by $\Gamma \Vdash \sigma$, if and only if for all $(x, j) \in \Omega$,

- $x : C \in \Gamma$ for some C;
- the value $w \triangleq E(c, (x, j))$ is defined;
- $w = \bot$ or $\texttt{class}(w) \preccurlyeq C$.

It essentially means that any variable currently in scope always contains an object of its proper kind. In this way the operational semantics of GM is related to its type system.

We have a similar notion for fields. Fields have infinite life span. While variables get their type information from the typing environment, field types are stored in the class table, independent of the typing environment. A state graph σ is *well-typed* with respect to the class table Ψ, denoted by $\Psi \Vdash \sigma$, if and only if for all $v \in V$ and for all fields f of $\texttt{class}(v)$,

- the value $w \triangleq E(v, f)$ is defined;
- $w = \bot$ or $\texttt{class}(w) \preccurlyeq \Psi_{\mathcal{F}}(C, f)$.

This is similar to store typing described in [BPP03] and [Pie02].

4.3.2 Type Soundness of GM

Well-typedness of state graph is preserved by well-typed GM programs. A expression is therefore evaluated to an object of the expected type.

Theorem 1 *Let P be a well-typed GM program. Assume that* $\langle\!\langle\ t\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma'$ *and* $\Gamma \vdash t : \mathcal{T}$, *where t is a method body ($\mathcal{G}$), a statement ($\mathcal{S}$) or an expression ($\mathcal{E}$).*

(i) If $t = \mathcal{G}$ *such that* $\Psi \Vdash \sigma$, *then* $\Psi \Vdash \sigma'$.

(ii) If $t = \mathcal{S}$ *such that* $\Psi \Vdash \sigma$ *and* $\Gamma \Vdash \sigma$, *then* $\Psi \Vdash \sigma'$ *and* $\Gamma \Vdash \sigma'$.

(iii) If $t = \mathcal{E}$ *such that* $\Psi \Vdash \sigma$ *and* $\Gamma \Vdash \sigma$, *then* $\Psi \Vdash \sigma'$ *and* $\Gamma \Vdash \sigma'$.

In all cases, if $\mathcal{T} = C$ *for some class C, then* $r' = \bot$ *or* $\texttt{class}(r') \preccurlyeq C$.

Proof

By induction on the derivation of $\langle\!\langle\ \_\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma'$. We demonstrate a few cases while others can be worked out similarly.

▷ Case $\langle\!\langle\ x\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma'$

Rule `[T-VAR]` ensures that $x : C \in \Gamma$. By `[E-VACC]`,

$$\langle\!\langle\ x\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma' \qquad \text{where } \sigma' = \sigma[\cdot, \cdot, \cdot, E(c, (x, i)), \cdot]$$

Since neither the edge function E nor the run-time environment Ω is changed, $\Gamma \Vdash \sigma$ implies $\Gamma \Vdash \sigma'$ while $\Psi \Vdash \sigma$ implies $\Psi \Vdash \sigma'$. Clearly, we have $r' = \bot$ or $\texttt{class}(r') \preccurlyeq C$.

▷ Case $\langle\!\langle\ \mathcal{E}.f\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma'$

Let $\Gamma \vdash \mathcal{E}.f : D$. Rule [T-FIELD] ensures that $\Gamma \vdash \mathcal{E} : C$ and $\Psi_{\mathcal{F}}(C, f) = D$ for some class C. By [E-FACC],

$$\frac{\langle\!\langle\ \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle \ \rightsquigarrow\ \sigma_0}{\langle\!\langle\ \mathcal{E}.f\ ,\ \sigma\ \rangle\!\rangle \ \rightsquigarrow\ \sigma'} \qquad \text{where } \sigma' = \sigma_0[\cdot, \cdot, \cdot, E_0(r_0, f), \cdot]$$

With induction hypothesis applied to the premise, we have $\Gamma \Vdash \sigma_0$ and $\Psi \Vdash \sigma_0$. Since σ_0 and σ' differ only in the result vertex, $\Gamma \Vdash \sigma'$ and $\Psi \Vdash \sigma'$. If $r_0 = \bot$, then $E_0(r_0, f)$ fails and $\mathcal{E}.f$ does not evaluate. So $\mathtt{class}(r_0) \preccurlyeq C$. Hence $r' = \bot$ or $\mathtt{class}(r') \preccurlyeq D$.

▷ Case $\langle\!\langle\ x = \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle \ \rightsquigarrow\ \sigma'$

First, we do not care about the type of r' since $\vdash x = \mathcal{E} : \mathtt{Void}$. According to rule [T-VAR-ASSGN], we get $\Gamma \vdash x : C$ and $\Gamma \vdash \mathcal{E} : D$, where $D \preccurlyeq C$. By [E-VASSN],

$$\frac{\langle\!\langle\ \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle \ \rightsquigarrow\ \sigma_0}{\langle\!\langle\ x = \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle \ \rightsquigarrow\ \sigma'} \qquad \text{where } \sigma' = \sigma_0[\cdot, E_0 \odot c_0 \overset{(x,i)}{\rightarrowtail} r_0, \cdot, \cdot, \cdot]$$

Applying induction hypothesis to the premise, we have $\Gamma \Vdash \sigma_0$, $\Psi \Vdash \sigma_0$ and $r_0 = \bot$ or $\mathtt{class}(r_0) \preccurlyeq C$. Now σ' differs from σ_0 by the edge functions. The edge functions E' is identical to E_0 except possibly on $(c_0, (x, i))$. But $E'(c_0, (x, i)) = r_0$. Hence $\Gamma \Vdash \sigma'$ and $\Psi \Vdash \sigma'$.

▷ Case $\langle\!\langle\ (D)\ \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle \ \rightsquigarrow\ \sigma'$

Let $\Gamma \vdash (D)\ \mathcal{E} : D$. Rule [T-CAST] ensures that $\Gamma \vdash \mathcal{E} : C$ for some class C such that $C \preccurlyeq D$. By [E-CAST],

$$\frac{\langle\!\langle\ \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle \ \rightsquigarrow\ \sigma'}{\langle\!\langle\ (D)\ \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle \ \rightsquigarrow\ \sigma'}$$

Applying induction hypothesis to the premise, we get $\Gamma \Vdash \sigma'$, $\Psi \Vdash \sigma'$, and $r' = \bot$ or $\mathtt{class}(r') \preccurlyeq C$. By transitivity of the subclassing relation, $\mathtt{class}(r') \preccurlyeq D$.

▷ Case $\langle\!\langle\ \mathcal{E}.m(\mathcal{E}')\ ,\ \sigma\ \rangle\!\rangle \ \rightsquigarrow\ \sigma''$

Suppose $\Gamma \vdash \mathcal{E}.m(\mathcal{E}') : T$. Rule [T-INVK] implies that $\Gamma \vdash \mathcal{E} : C$ and $\Gamma \vdash \mathcal{E}' : D$ for some classes C and D, where $\Psi_{\mathcal{M}}(C, m) = B \to T$ and $D \preccurlyeq B$. By [E-INVK],

$$\frac{\langle\!\langle\ \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle \ \rightsquigarrow\ \sigma_0 \qquad \langle\!\langle\ \mathcal{E}'\ ,\ \sigma_0\ \rangle\!\rangle \ \rightsquigarrow\ \sigma_1 \qquad \langle\!\langle\ \mathcal{G}\ ,\ \sigma'\ \rangle\!\rangle \ \rightsquigarrow\ \sigma_2}{\langle\!\langle\ \mathcal{E}.m(\mathcal{E}')\ ,\ \sigma\ \rangle\!\rangle \ \rightsquigarrow\ \sigma''}$$

where

$$\begin{array}{rcl} (x : \_\,, \mathcal{G}) & = & \mathtt{method}(\mathtt{class}(r_0), m) \\ \sigma' & = & (V_1, E_1 \oplus r_0 \overset{(x,j)}{\rightarrowtail} r_1, r_0, r_1, \{(x,j)\}) \\ \sigma'' & = & (V_2, \mathtt{del}(E_2, r_0), c_0, r_2, \Omega_1) \end{array}$$

Applying induction hypothesis to the first premise gives $\Gamma \Vdash \sigma_0$, $\Psi \Vdash \sigma_0$ and $r_0 = \bot$ or $\mathtt{class}(r_0) \preccurlyeq C$. Since $\Gamma \Vdash \sigma_0$ and $\Psi \Vdash \sigma_0$, we can apply induction hypothesis to the second premise to get $\Gamma \Vdash \sigma_1$, $\Psi \Vdash \sigma_1$ and $r_1 = \bot$ or $\mathtt{class}(r_1) \preccurlyeq D$.

Since all m is well-typed, we can let $\Gamma' \vdash \mathcal{G} : T$ for some Γ'. Again by induction hypothesis applied to $\mathcal{G}$, we have $\Psi \Vdash \sigma_2$ and if $T \neq \mathtt{Void}$, then $r_2 = \bot$ or $\mathtt{class}(r_2) \preccurlyeq T$.

From the construction of E'', $\Psi \Vdash \sigma_2$ implies $\Psi \Vdash \sigma''$. And because $\mathcal{G}$ has no access to the variables in Ω_1, $E''(c_0, (y,j)) = E_1(c_0, (y,j))$ for all $(y,j) \in \Omega_1$. Hence $\Gamma \Vdash \sigma''$.

□

Run-time errors can still occur in GM. For example, the invocation $x.m(\_)$ fails if the variable x does not contain any meaningful object, i.e. x is evaluated to $\bot$. Type checking merely ensures that evaluation of an expression yields an instance of its predicted type. However, $\bot$ is a value belonging to all types. The type system therefore will have no guarantee that x is $\bot$ as long as $x.m(\_)$ type checks. Other techniques, e.g. flow analysis, would be needed to detect occurrences of bottom, although such analysis does not seem to be obvious in the presence of dynamic binding.

Chapter 5

Axiomatic Semantics

The axiomatic semantics takes a somewhat unusual view of *meaning* of a program as the true properties that can be proved about it. The approach is to specify properties of programs as assertions. An assertion is a *triple* of the form $\{P\}$ t $\{Q\}$, where t is a term in the abstract syntax, and P, Q are themselves assertions of some kind. Intuitively, $\{P\}$ t $\{Q\}$ means that if P is true, then after execution of t provided that it terminates, Q is true. In this respect, P is called the *pre-condition* and Q the *post-condition*. The triple is also known as *partial correctness assertion*, an assertion that tells us nothing if t keeps on executing.

There are two approaches, *intensional* versus *extensional*, to specify the assertions P and Q in the triples. We may introduce an explicit language, known as the assertion language. Then P and Q will be formulae of that language. Ideally the assertion language has to be powerful, or expressive, enough to formulate all the assertions of interest. This is the intensional approach. Alternatively, the extensional approach takes a semantic view of assertions. An assertion is simply a predicate over states, i.e. a function of type

$$\textit{Set of state graphs} \longrightarrow \textit{Boolean}$$

This is kind of short-cut to avoid the expressiveness problem of the assertion language. Predicates can be combined to form new predicates. For example,

$$P \wedge Q \triangleq \lambda\sigma\ .\ P\sigma \wedge Q\sigma$$

Operations $P \vee Q$, $\neg P$, etc are defined similarly.

Partial correctness of object-oriented programs is more complex than that of procedural programs. A major difficulty is that object-oriented languages use dynamically bound method invocations, in contrast to procedural languages that use statically bound calls. For example, the invocation $x.m()$ where variable x is of class T with subclasses T' and T''. If x references an object of type T' at method invocation time, the implementation of m associated with T' is executed. On the other hand if x is an object of class T'', then different code is to be executed.

5.1 Proof Rules

First we exclude recursive methods. Similar to the operational semantics and the type system, the axiomatic semantics of GM is also described as a deductive system, in which judgements are triples. The first few rules are for general logic reasoning. The rest of the rules are closely geared at the operational semantics.

5.1.1 Consequence

If the result proved is stronger than required, it is then possible to weaken the post-condition.

$$\texttt{[A-POST]} \quad \frac{\{P\}\ \mathsf{t}\ \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\ \mathsf{t}\ \{Q\}}$$

On the other hand the given pre-condition may be stronger than necessary to complete the proof.

$$\texttt{[A-PRE]} \quad \frac{\{P'\}\ \mathsf{t}\ \{Q\} \qquad P \Rightarrow P'}{\{P\}\ \mathsf{t}\ \{Q\}}$$

These two rules are compactly represented by a single one.

$$\texttt{[A-CONSEQ]} \quad \frac{\{P'\}\ \mathsf{t}\ \{Q'\} \qquad P \Rightarrow P' \qquad Q' \Rightarrow Q}{\{P\}\ \mathsf{t}\ \{Q\}}$$

5.1.2 Sequencing

As usual, composition is proved by a common post-condition and pre-condition Q in the two premises. The link Q is then forgot in the conclusion.

$$\texttt{[A-SEQ]} \quad \frac{\{P\}\ \mathcal{S}\ \{Q\} \qquad \{Q\}\ \mathcal{S}'\ \{R\}}{\{P\}\ \mathcal{S}\ ;\ \mathcal{S}'\ \{R\}}$$

5.1.3 Conditional

An *if* statement consists of two alternatives. The entire statement is correct if we can prove each alternative given the appropriate value of the test expression $\mathcal{E}$. The value of $\mathcal{E}$ is stored as the result vertex.

$$\texttt{[A-COND]} \quad \frac{\begin{array}{c}\{P\}\ \mathcal{E}\ \{Q\} \\ \{\lambda\sigma\ .\ r \neq \bot \wedge Q\sigma\}\ \mathcal{S}\ \{R\} \\ \{\lambda\sigma\ .\ r = \bot \wedge\ Q\sigma\}\ \mathcal{S}'\ \{R\}\end{array}}{\{P\}\ \texttt{if}\ \mathcal{E}\ \texttt{then}\ \mathcal{S}\ \texttt{else}\ \mathcal{S}'\ \texttt{fi}\ \{R\}}$$

5.1.4 Return

A *return* statement is correct as long as the expression $\mathcal{E}$ to be returned has been correctly evaluated.

$$\texttt{[A-RETN]} \quad \frac{\{P\}\ \mathcal{E}\ \{Q\}}{\{P\}\ \texttt{return}\ \mathcal{E}\ \{Q\}}$$

5.1.5 Variable Access

In order to derive the post-condition P, we have to ensure the pre-condition on states with the result vertex equal to the content of x.

$$\texttt{[A-VACC]} \quad \{\lambda\sigma\ .\ P\ \sigma[E(c,x)/r]\}\ x\ \{P\}$$

5.1.6 Current Object

Expression `this` can be regarded as a special variable. The content of the special variable is always the current vertex.

$$\texttt{[A-THIS]} \quad \{\lambda\sigma\ .\ P\ \sigma[c/r]\}\ \texttt{this}\ \{P\}$$

5.1.7 Expression `null`

This can be regarded as a special case of accessing a variable. The content of `null` is always the vertex $\bot$.

$$\texttt{[A-NULL]} \qquad \{\lambda\sigma\ .\ P\ \sigma[\bot/r]\}\ \texttt{null}\ \{P\}$$

5.1.8 Variable Assignment

A premise is required to ensure the correctness of evaluating $\mathcal{E}$. The conclusion is then proved if Q in the premise holds for states with x equal to the value of expression $\mathcal{E}$, which is stored as the result vertex r.

$$\texttt{[A-VASSN]} \qquad \frac{\{P\}\ \mathcal{E}\ \{\lambda\sigma\ .\ Q\ \sigma[E \odot c \overset{(x,j)}{\rightarrowtail} r/E] \wedge (x, j, \_) \in \Omega\}}{\{P\}\ x = \mathcal{E}\ \{Q\}}$$

5.1.9 Field Access

Field access is similar to variable access, except that the owner $\mathcal{E}$ of the field has to be correctly evaluated in the first place.

$$\texttt{[A-FASS]} \qquad \frac{\{P\}\ \mathcal{E}\ \{\lambda\sigma\ .\ Q\ \sigma[E(r, f)/r]\}}{\{P\}\ \mathcal{E}.f\ \{Q\}}$$

5.1.10 Field Assignment

Again, a premise is required to prove the correctness of computing the field owner $\mathcal{E}$. In the second premise, this field owner, i.e. the result vertex r, is passed from the pre-condition to the post-condition by means of universal quantification

$$\texttt{[A-FASSN]} \qquad \frac{\begin{array}{c}\{P\}\ \mathcal{E}\ \{Q\} \\ \forall \hat{w}\ .\ \{\lambda\sigma\ .\ Q\sigma \wedge \hat{w} = r\}\ \mathcal{E}'\ \{\lambda\sigma\ .\ R\ \sigma[E \oplus \hat{w} \overset{f}{\rightarrowtail} r/E]\}\end{array}}{\{P\}\ \mathcal{E}.f = \mathcal{E}'\ \{R\}}$$

5.1.11 Casting

Evaluating a type cast causes a run-time error if the dynamic type of the object referred to is not a subclass of the target class C. So the post-condition Q in the premise need be shown only if the expression evaluates to *null* or to an object of a subclass of C.

$$\texttt{[A-CAST]} \qquad \frac{\{P\}\ \mathcal{E}\ \{\lambda\sigma\ .\ (r = \bot\ \vee\ \texttt{class}(r) \preccurlyeq C) \wedge Q\sigma\}}{\{P\}\ (C)\ \mathcal{E}\ \{Q\}}$$

5.1.12 Instantiation

If we can ensure P with the introduction of a new vertex of type C, then P should hold after creation of an object from class C.

$$\begin{array}{ll}\texttt{[A-NEW]} & \{\lambda\sigma\ .\ P\ \sigma[V \cup \{v\}, E', \cdot, v, \cdot]\}\ \texttt{new}\ C\ \{P\} \\ & \quad \text{where}\ \ v = (C, \texttt{next}(C, V)) \\ & \qquad\qquad E' \triangleq E \oplus (v, f_1, \bot) \oplus \cdots \oplus (v, f_n, \bot) \\ & \qquad\qquad f_1, \cdots, f_n \in \texttt{fields}(\texttt{class}(v))\end{array}$$

5.1.13 Method invocation

Method invocation is the most complex among the proof rules. The type of the message receiver will not be known until run-time. To ensure correctness of method call, it is therefore necessary to prove all method implementations compatible with the predicted type of the message receiver.

$$\texttt{[A-INVK]} \qquad \frac{\begin{array}{c}\{P\}\ \mathcal{E}\ \{Q\} \\ \forall\hat{w}\ .\ \{\lambda\sigma\ .\ Q\sigma \wedge \hat{w} = c\}\ \mathcal{E}'\ \{R\} \\ \forall\hat{C}\forall\hat{w}\forall\hat{\sigma}\ .\ \{R'\}\ \texttt{body}(\hat{C}, m)\ \{S'\} \wedge \hat{C} \preccurlyeq D\end{array}}{\{P\}\ \mathcal{E}.m(\mathcal{E}')\ \{S\}}$$

$$\begin{array}{ll}\text{where} & R' = \lambda\sigma\ .\ R\hat{\sigma} \wedge \texttt{class}(c) = \hat{C} \\ & \qquad\qquad \wedge\ \sigma = \hat{\sigma}[\cdot, \hat{E} \oplus \hat{w} \overset{(x,j)}{\rightarrowtail} \hat{r}, \hat{w}, \cdot, \{(x, j)\}] \\ & S' = \lambda\sigma\ .\ S\ \sigma[\cdot, \texttt{del}(E, \hat{w}), \hat{c}, \cdot, \hat{\Omega}] \\ & D = \text{ static type of } \mathcal{E} \\ & (x : \_, \_) = \texttt{method}(D, m)\end{array}$$

In essence, the third premise exhausts all possible implementations of method m in subclasses $\hat{C}$ of the statically inferred class D of the receiving object e. This is dynamic binding. The purpose of the universally quantified vertex $\hat{w}$ is to pass back control to the original object before the call, as described in the operational semantics.

5.1.14 Variable Declaration

A variable declaration is correct provided that the method body executes correctly against the state augmented with the variable.

$$\texttt{[A-DECL]} \qquad \frac{\{\lambda\sigma\ .\ P[\cdot, E \oplus c \overset{(x,j)}{\rightarrowtail} \bot, \cdot, \cdot, \Omega \cup (x,j)]\}\ \mathcal{G}\ \{Q\}}{\{P\}\ C\ x\ ;\mathcal{G}\ \{Q\}}$$

5.2 Soundness and Completeness

We say that a partial correctness assertion $\{P\}$ t $\{Q\}$ is *provable* if and only if it is the root of a derivation tree. In other words, an assertion is provable if and only if it is derivable within the axiomatic semantics. We write this as

$$\vdash \{P\}\ \mathsf{t}\ \{Q\}$$

The partial correctness assertion $\{P\}$ t $\{Q\}$ is said to be *valid* if and only if for all state graphs σ, if $\langle\!\langle\ t\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma'$ and $P\sigma$ is true, then $Q\sigma'$ is also true. We denote this property by

$$\models \{P\}\ \mathsf{t}\ \{Q\}$$

At the minimum, the axiomatic semantics need to be correct, in the sense that if a partial correctness result can be proved using the axiomatic semantics, then it does indeed hold according to the operational semantics. The axiomatic semantics is said to be *sound* with respect to the operational semantics if and only if provability implies validity, i.e. for every partial correctness assertion $\{P\}$ t $\{Q\}$

$$\vdash \{P\}\ \mathsf{t}\ \{Q\} \quad \Longrightarrow \quad \models \{P\}\ \mathsf{t}\ \{Q\}$$

Intuitively, this means that everything that can be proved is correct.

Moreover, for the semantics to be useful, at least theoretically, it has to provide all the truth. The axiomatic semantics is said to be *complete* with respect to the operational semantics if and only if validity implies provability, i.e. for every partial correctness assertion $\{P\}$ t $\{Q\}$

$$\models \{P\}\ \mathrm{t}\ \{Q\} \quad \Longrightarrow \quad \vdash \{P\}\ \mathrm{t}\ \{Q\}$$

This says that everything that is correct can be proved. We shall show that the axiomatic semantics of GM is sound and complete with respect to its operational semantics. Hence the axiomatic semantics is able to prove everything that is true, and no more.

Completeness sounds a bit appalling. Since GM has the power of arithmetic, completeness of its semantics would imply completeness of arithmetic. But the Austrian mathematician Kurt Goedel already showed that arithmetic is incomplete. Our *completeness* is with respect to the operational semantics. If we tried to encode Goedel's statement (which is true but not provable) in GM, the program simply would not terminate. "Partial correctness" comes to the rescue!

5.2.1 Proof of Soundness

Theorem 2 *The axiomatic semantics of GM is sound with respect to its operational semantics, i.e. for all partial correctness assertion $\{P\}$ t $\{Q\}$*

$$\vdash \{P\}\ t\ \{Q\} \quad \Longrightarrow \quad \models \{P\}\ t\ \{Q\}$$

Proof

The proof is by induction on the derivation tree used to derive $\vdash$ $\{P\}$ t $\{Q\}$.

$\triangleright$ Case t $::= x$

We show that the axiom [A-VACC] is valid. Suppose

$$\langle\!\langle\ x\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma'$$

and

$$P\ \sigma[u/r] = \mathtt{tt}$$

where $u = E(c,(x,j))$ for some integer j such that $(x,j) \in \Omega$. Then we have to prove that $P\sigma' = \mathtt{tt}$. This follows directly from [E-VACC] since

$$\sigma' = (V, E, c, E(c,(x,j)), \Omega) = \sigma[u/r]$$

▷ Case $\mathsf{t} ::= x = \mathcal{E}$

We show that, in rule [A-VASSN], validity of premise implies validity of conclusion. Suppose

$$\langle\!\langle\ x = \mathcal{E}\ ,\ \sigma_0\ \rangle\!\rangle \ \rightsquigarrow\ \sigma'$$

and

$$P\sigma = \mathtt{tt}$$

We need to prove that $Q\sigma' = \mathtt{tt}$. By induction hypothesis

$$\models \{P\}\ \mathcal{E}\ \{\lambda\sigma\ .\ Q\ \sigma[E \oplus c \overset{(x,j)}{\rightarrowtail} r/E]\}$$

By [E-VASSN], there is σ such that $\langle\!\langle\ \mathcal{E}\ ,\ \sigma_0\ \rangle\!\rangle \ \rightsquigarrow\ \sigma$ and

$$\sigma' = (V, E \oplus c \overset{(x,j)}{\rightarrowtail} r, c, r, \Omega) = \sigma[E \oplus c \overset{(x,j)}{\rightarrowtail} r/E]$$

The claim follows by applying the post-condition to σ.

▷ Case $\mathsf{t} ::= \mathcal{E}.f$

We show that, in [A-FASS], validity of premise implies validity of conclusion. Suppose

$$\langle\!\langle\ \mathcal{E}.f\ ,\ \sigma_0\ \rangle\!\rangle \ \rightsquigarrow\ \sigma'$$

and

$$P\sigma_0 = \mathtt{tt}$$

We need to deduce that $Q\sigma' = \mathtt{tt}$. By [E-FACC], there is a σ such that

$$\langle\!\langle\ \mathcal{E}\ ,\ \sigma_0\ \rangle\!\rangle \ \rightsquigarrow\ \sigma$$

and

$$\sigma' = (V, E, c, E(c,f), \Omega) = \sigma[E(c,f)/r]$$

The induction hypothesis implies that

$$\models \{P\}\ \mathcal{E}\ \{\lambda\sigma\ .\ Q\ \sigma[E(c,f)/r]\}$$

The claim follows by applying the post-condition to σ'.

▷ Case $\mathsf{t} ::= \mathcal{E} = \mathcal{E}'.f$

We show that, in [A-FASSN], validity of premises implies validity of conclusion. Suppose

$$\langle\!\langle\ \mathcal{E}.f = \mathcal{E}'\ ,\ \sigma_0\ \rangle\!\rangle \rightsquigarrow \sigma_1$$

and

$$P\sigma_0 = \mathtt{tt}$$

We need to deduce that $R\sigma_1 = \mathtt{tt}$. By [E-FASSN], there are σ_1 and σ such that

$$\langle\!\langle\ \mathcal{E}\ ,\ \sigma_0\ \rangle\!\rangle \rightsquigarrow \sigma'$$

$$\langle\!\langle\ \mathcal{E}'\ ,\ \sigma'\ \rangle\!\rangle \rightsquigarrow \sigma$$

and

$$\sigma_1 = (V, E \oplus r' \overset{f}{\rightarrowtail} r, c, r, \Omega) = \sigma[E \oplus r' \overset{f}{\rightarrowtail} r/E]$$

By induction hypothesis, we have

$$\models \{P\}\ \mathcal{E}\ \{Q\}$$

$$\forall \hat{w}\ .\ \models \{\lambda\sigma\ .\ Q\sigma \wedge \hat{w} = r\}\ \mathcal{E}'\ \{\lambda\sigma\ .\ R\ \sigma[E \oplus \hat{w} \overset{f}{\rightarrowtail} r/E]\}$$

The first validity implies that $Q\sigma' = \mathtt{tt}$. Specializing $\hat{w}$ to r' in the second validity, we get

$$\models \{\lambda\sigma\ .\ Q\sigma \wedge r' = r\}\ \mathcal{E}'\ \{\lambda\sigma\ .\ R\ \sigma[E \oplus r' \overset{f}{\rightarrowtail} r/E]\}$$

The pre-condition holds for $\sigma = \sigma'$, so that the post-condition is true when applied to σ. Hence,

$$R\sigma_1 = R\ \sigma[E \oplus r' \overset{f}{\rightarrowtail} r/E] = \mathtt{tt}$$

▷ Case $\mathsf{t} ::= (C)\ \mathcal{E}$

$$\texttt{[A-CAST]} \quad \frac{\{P\}\ \mathcal{E}\ \{\lambda\sigma\ .\ \mathtt{class}(r) \preccurlyeq C \wedge Q\sigma\}}{\{P\}\ (C)\ \mathcal{E}\ \{Q\}}$$

We show that, in [A-CAST], validity of premises implies validity of conclusion. Suppose

$$\langle\!\langle\ (C)\ \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle\ \leadsto\ \sigma'$$

and

$$P\sigma = \mathtt{tt}$$

We need to deduce that $Q\sigma' = \mathtt{tt}$. By [E-CAST],

$$\langle\!\langle\ \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle\ \leadsto\ \sigma'$$

where $\mathtt{class}(r') \preccurlyeq C$. By induction hypothesis,

$$\mathtt{class}(r') \preccurlyeq C \wedge Q\sigma' \Longrightarrow Q\sigma'$$

▷ Case $\mathtt{t} ::= \mathtt{new}\ C$

We show that axiom [A-NEW] is valid. Suppose

$$\langle\!\langle\ \mathtt{new}\ C\ ,\ \sigma\ \rangle\!\rangle\ \leadsto\ \sigma'$$

and

$$P\ \sigma[V \cup \{v\}, E', \cdot, v, \cdot] = \mathtt{tt}$$

where

$$\begin{aligned} &v = (C, \mathtt{next}(C, V)) \\ &E' = E \oplus (v, f_1, \bot) \oplus \cdots \oplus (v, f_n, \bot) \\ &f_1, \cdots, f_n \in \mathtt{fields}(\mathtt{class}(v)) \end{aligned}$$

Then we need to prove that $P\sigma' = \mathtt{tt}$. In fact, $\sigma' = \sigma[V \cup \{v\}, E', \cdot, v, \cdot]$ and so this follows readily from axiom [E-NEW].

▷ Case $\mathtt{t} ::= \mathcal{E}.m(\mathcal{E}')$

We show that, in [A-INVK], validity of premises implies validity of conclusion. Suppose

$$\langle\!\langle\ \mathcal{E}.m(\mathcal{E}')\ ,\ \sigma\ \rangle\!\rangle\ \leadsto\ \sigma''$$

and

$$P\sigma_0 = \mathtt{tt}$$

We need to deduce that $S\sigma'' = \mathtt{tt}$. By [E-INVK], there are σ_0, σ_1 and σ_2 such that

$$\langle\!\langle\, \mathcal{E} \,,\, \sigma \,\rangle\!\rangle \rightsquigarrow \sigma_0 \,,\ \langle\!\langle\, \mathcal{E}' \,,\, \sigma_0 \,\rangle\!\rangle \rightsquigarrow \sigma_1 \,,\ \langle\!\langle\, \mathcal{G} \,,\, \sigma' \,\rangle\!\rangle \rightsquigarrow \sigma_2$$

and

$$\begin{array}{rcl}
C & = & \mathtt{class}(r_0) \quad \text{and} \quad (x : C', \mathcal{G}) = \mathtt{method}(C, m) \\
\sigma' & = & (V_1, E_1 \oplus r_0 \overset{(x,j)}{\rightarrowtail} r_1, r_0, r_1, \{(x, j, C')\}) \\
 & = & \sigma_1[E_1 \oplus r_0 \overset{(x,j)}{\rightarrowtail} r_1/E_1][r_0/c_1][\{(x, j, C')\}/\Omega_1] \\
\sigma'' & = & (V_2, \mathtt{del}(E_2, r_0), c_0, r_2, \Omega_1) \\
 & = & \sigma_2[\mathtt{del}(E_2, r_0)/\Omega_2][c_0/c_2][\Omega_1/\Omega_2]
\end{array}$$

By induction hypothesis, we have

$$\models \{P\}\ \mathcal{E}\ \{Q\}$$

$$\models \{Q\}\ \mathcal{E}'\ \{R\}$$

$$\forall \hat{C} \forall \hat{w} \forall \hat{\sigma}\ .\ \models \{R'\}\ \mathtt{method}(\hat{C}, m)\ \{S'\}$$

The first two premises lead to $R\sigma_1 = \mathtt{tt}$. In the last premise, specializing

- $\hat{w}$ to r_0, the receiving object of message m
- $\hat{C}$ to C, the class of the receiving object
- $\hat{\sigma}$ to σ_1, the *pre-state* for executing the message

we have

$$\begin{array}{rcl}
R' & = & \lambda\sigma\ .\ \exists\sigma_k\ .\ R\sigma_k \wedge \sigma_1 = \sigma_k \wedge \mathtt{class}(c) = C \\
 & & \wedge\ \sigma = \sigma_k[E_k \oplus r_0 \overset{(x,j)}{\rightarrowtail} r_k/E_k][r_0/c][\{(x, j, C')\}/\Omega_k]
\end{array}$$

and

$$S' = \lambda\sigma\ .\ S\ \sigma[\mathtt{del}(E, r_0)/E][r_1/c][\Omega_1/\Omega]$$

Now such a σ_k does exist, e.g. σ_1, so that $R'\sigma' = \mathtt{tt}$. By validity,

$$S\sigma'' = S'\sigma_2 = \mathtt{tt}$$

as claimed.

□

5.2.2 Proof of completeness

To prove completeness, direct application of induction does not work. We consider a special predicate $\mathsf{sp}(P,\mathsf{t})$ for a program term t and an assertion P, defined as

$$\mathsf{sp}(P,\mathsf{t})\sigma' = \mathtt{tt}$$

if and only if there is a state graph σ,

$$\langle\!\langle\, \mathsf{t}\ ,\ \sigma \,\rangle\!\rangle \leadsto \sigma' \wedge P\sigma$$

The assertion $\mathsf{sp}(P,\mathsf{t})$ is called the *strongest postcondition* for P and t.

Lemma 2 *Let t be a program term and P a predicate.*

(i) $\models \{P\}\ t\ \{sp(P,t)\}$

(ii) If $\models \{P\}\ t\ \{Q\}$, *then* $sp(P,t) \Rightarrow Q$

Proof

Property (i) is no more than a re-formulation of the definition of strongest post-condition. Property (ii) is also obvious because for all σ',

$$\mathsf{sp}(P,\mathsf{t})\sigma' \implies \exists\sigma\ .\ \langle\!\langle\, t\ ,\ \sigma \,\rangle\!\rangle \leadsto \sigma' \wedge P\sigma \implies Q\sigma'$$

□

Theorem 3 *The axiomatic semantics of GM is complete with respect to its operational semantics, i.e. for every partial correctness assertion $\{P\}\ t\ \{Q\}$,*

$$\models \{P\}\ t\ \{Q\} \implies \vdash \{P\}\ t\ \{Q\}$$

Proof

It suffices to show that, for all program term t and assertion P,

$$\models \{P\}\ \mathsf{t}\ \{Q\} \implies \vdash \{P\}\ \mathsf{t}\ \{\mathsf{sp}(P,\mathsf{t})\} \qquad (\sharp)$$

because by lemma 2,

$$\mathsf{sp}(P, \mathsf{t}) \implies Q$$

so that [A-CONSEQ] together with ($\sharp$) give

$$\models \{P\}\ \mathsf{t}\ \{Q\} \implies \vdash \{P\}\ \mathsf{t}\ \{\mathsf{sp}(P, \mathsf{t})\} \implies \vdash \{P\}\ \mathsf{t}\ \{Q\}$$

Now we prove ($\sharp$) by structural induction on t.

$\triangleright$ Case $\mathsf{t} ::= x$

Applying rule [A-VACC] to $Q \triangleq \lambda\sigma\ .\ \mathsf{sp}(P, x)\ \sigma[E(c, x)/r]$, we have

$$\vdash \{Q\}\ x\ \{\mathsf{sp}(P, x)\}$$

Now for all σ,

$$\begin{aligned}
Q\sigma &\iff \mathsf{sp}(P, x)\ \sigma[E(c, x)/r] \\
&\iff \exists\sigma'\ .\ \langle\!\langle\ x\ ,\ \sigma'\ \rangle\!\rangle \rightsquigarrow \sigma[E(c, x)/r] \wedge P\sigma' \\
&\iff \langle\!\langle\ x\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma[E(c, x)/r] \wedge P\sigma \\
&\iff \mathtt{tt} \wedge P\sigma \\
&\iff P\sigma
\end{aligned}$$

where [E-VACC] ensures $\sigma = \sigma'$. Hence $\vdash \{P\}\ x\ \{\mathsf{sp}(P, x)\}$ by [A-CONSEQ].

$\triangleright$ Case $\mathsf{t} ::= x = \mathcal{E}$

The induction hypothesis applied to e and P gives

$$\vdash \{P\}\ \mathcal{E}\ \{\mathsf{sp}(P, \mathcal{E})\}$$

If we can show that $\mathsf{sp}(P, \mathcal{E}) \Rightarrow R$, where

$$R \triangleq \lambda\sigma\ .\ \mathsf{sp}(P, x = \mathcal{E})\ \sigma[E \oplus c \overset{(x,j)}{\mapsto} r/E] \wedge (x, j, \_) \in \Omega$$

then $\vdash \{P\}\ x = \mathcal{E}\ \{\mathsf{sp}(P, x = \mathcal{E})\}$ will follow from rule [A-CONSEQ]. For all σ,

$$\begin{aligned}
\mathsf{sp}(P, \mathcal{E})\sigma &\iff \exists\sigma\ .\ \langle\!\langle\ \mathcal{E}\ ,\ \sigma'\ \rangle\!\rangle \rightsquigarrow \sigma \wedge P\sigma' \\
&\iff \exists\sigma'\ .\ \langle\!\langle\ \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma' \wedge \sigma' = \sigma[E(c, x)/r] \wedge P\sigma \\
&\iff \langle\!\langle\ x\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma[E(c, x)/r] \wedge P\sigma \\
&\iff \exists\sigma'\ .\ \langle\!\langle\ \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma' \wedge \sigma' = \sigma[E(c, x)/r] \wedge P\sigma \\
&\iff R\sigma
\end{aligned}$$

▷ Case $\mathsf{t} ::= \mathcal{E}.f = \mathcal{E}'$

The induction hypothesis applied to e and P gives

$$\vdash \{P\}\ \mathcal{E}\ \{\mathsf{sp}(P, \mathcal{E})\}$$

Let $\hat{w}$ be a vertex. Applying induction hypothesis to e' and

$$Q_{\hat{w}} \triangleq \lambda\sigma\ .\ \mathsf{sp}(P, \mathcal{E})\sigma \wedge \hat{w} = r$$

we have

$$\vdash \{Q_{\hat{w}}\}\ \mathcal{E}'\ \{\mathsf{sp}(Q_{\hat{w}}, \mathcal{E})\}$$

It suffices to show that $\mathsf{sp}(Q_{\hat{w}}, \mathcal{E}') \Rightarrow R_{\hat{w}}$, where

$$R_{\hat{w}} \triangleq \lambda\sigma\ .\ \mathsf{sp}(P, \mathcal{E}.f = \mathcal{E}')\ \sigma[E \oplus \hat{w} \overset{f}{\rightarrowtail} r/E]$$

For then $\vdash \{P\}\ \mathcal{E}.f = \mathcal{E}'\ \{\mathsf{sp}(P, \mathcal{E}.f = \mathcal{E}')\}$ will follow from [A-CONSEQ] and [A-FASSN]. By definition of strongest postcondition, and [E-FASSN]

$$\begin{array}{rl}
\mathsf{sp}(Q_{\hat{w}}, \mathcal{E}')\sigma & \Longrightarrow\ \exists\sigma'\ .\ \langle\!\langle\ \mathcal{E}'\ ,\ \sigma'\ \rangle\!\rangle \rightsquigarrow \sigma \wedge Q_{\hat{w}}\sigma' \\
& \Longrightarrow\ \exists\sigma'\ .\ \langle\!\langle\ \mathcal{E}'\ ,\ \sigma'\ \rangle\!\rangle \rightsquigarrow \sigma \wedge \mathsf{sp}(P, \mathcal{E}')\sigma' \wedge \hat{w} = r' \\
& \Longrightarrow\ \exists\sigma_0\exists\sigma'\ .\ \langle\!\langle\ \mathcal{E}'\ ,\ \sigma'\ \rangle\!\rangle \rightsquigarrow \sigma \wedge \langle\!\langle\ \mathcal{E}\ ,\ \sigma_0\ \rangle\!\rangle \rightsquigarrow \sigma' \wedge P\sigma_0 \wedge \hat{w} = r' \\
& \Longrightarrow\ \exists\sigma_0\exists\sigma'\ .\ \langle\!\langle\ \mathcal{E}.f = \mathcal{E}'\ ,\ \sigma_0\ \rangle\!\rangle \rightsquigarrow \sigma[E \oplus \hat{w} \overset{f}{\rightarrowtail} r/E] \wedge P\sigma_0 \wedge \hat{w} = r' \\
& \Longrightarrow\ \mathsf{sp}(P, \mathcal{E}.f = \mathcal{E}')\ \sigma[E \oplus \hat{w} \overset{f}{\rightarrowtail} r/E] \\
& \Longrightarrow\ R_{\hat{w}}\sigma
\end{array}$$

▷ Case $\mathsf{t} ::= (C)\ \mathcal{E}$

The induction hypothesis applied to $\mathcal{E}$ and P gives

$$\vdash \{P\}\ \mathcal{E}\ \{\mathsf{sp}(P, \mathcal{E})\}$$

It suffices to show that $\mathsf{sp}(P, \mathcal{E}) \Rightarrow Q$, where

$$Q \triangleq \lambda\sigma\ .\ r = \bot\ \vee\ \mathtt{class}(r) \preccurlyeq C \wedge \mathsf{sp}(P, (C)\ \mathcal{E})\sigma$$

Then $\vdash \{P\}\ (C)\ \mathcal{E}\ \{\mathsf{sp}(P, (C)\ \mathcal{E})\}$ will follow from [A-CONSEQ] and [A-CAST]. Consider a state graph σ. If $r = \bot$, then $Q\sigma = \mathtt{tt}$ and therefore $\mathsf{sp}(P, \mathcal{E})\sigma \Rightarrow Q\sigma$. Now assume that

$r \neq \bot$ and $\texttt{class}(r) \preccurlyeq C$.[1]

$$\begin{array}{rcl} \texttt{sp}(P,\mathcal{E})\sigma & \Longrightarrow & \exists\sigma' \,.\, \langle\!\langle\, \mathcal{E}\,,\, \sigma' \,\rangle\!\rangle \leadsto \sigma \wedge P\sigma' \\ & \Longrightarrow & \exists\sigma' \,.\, \langle\!\langle\, (C)\,\mathcal{E}\,,\, \sigma' \,\rangle\!\rangle \leadsto \sigma \wedge P\sigma' \\ & \Longleftrightarrow & \texttt{sp}(P,(C)\,\mathcal{E})\sigma \\ & \Longrightarrow & Q\sigma \end{array}$$

▷ Case t ::= new C

Applying rule [A-NEW] to

$$Q \triangleq \lambda\sigma \,.\, \texttt{sp}(P,\texttt{new}\ C)\ \sigma[V \cup \{v\}, E', \cdot, v, \cdot]$$

where

$$\begin{array}{rcl} v & = & (C, \texttt{next}(C,V)) \\ E' & = & E \oplus (v, f_1, \bot) \oplus \cdots \oplus (v, f_n, \bot) \\ f_1, \cdots, f_n & \in & \texttt{fields}(\texttt{class}(v)) \end{array}$$

we have

$$\vdash \{Q\}\ \texttt{new}\ C\ \{\texttt{sp}(P,\texttt{new}\ C)\}$$

Now for all σ,

$$\begin{array}{rcl} P\sigma & \Longleftrightarrow & \langle\!\langle\, \texttt{new}\ C\,,\, \sigma \,\rangle\!\rangle \leadsto \sigma[V \cup \{v\}, E', \cdot, v, \cdot] \wedge P\sigma \\ & \Longleftrightarrow & \exists\sigma' \,.\, \langle\!\langle\, \texttt{new}\ C\,,\, \sigma' \,\rangle\!\rangle \leadsto \sigma[V \cup \{v\}, E', \cdot, v, \cdot] \wedge P\sigma' \\ & \Longleftrightarrow & \texttt{sp}(P,\texttt{new}\ C)\ \sigma[V \cup \{v\}, E', \cdot, v, \cdot] \\ & \Longleftrightarrow & Q\sigma \end{array}$$

It follows from [A-CONSEQ] that $\vdash \{P\}\ \texttt{new}\ C\ \{\texttt{sp}(P,\texttt{new}\ C)\}$.

▷ Case t is method invocation $\mathcal{E}.m(\mathcal{E}')$

The induction hypothesis applied to $\mathcal{E}$ and P gives

$$\vdash \{P\}\ \mathcal{E}\ \{\texttt{sp}(P,\mathcal{E})\}$$

Take a class $\hat{C} \preccurlyeq D$, a vertex $\hat{w}$ and a state graph $\hat{\sigma}$. Let

$$Q \triangleq \lambda\sigma \,.\, \texttt{sp}(P,\mathcal{E})\sigma \wedge \hat{w} = c$$

[1]Otherwise the cast gets stuck.

Applying induction hypothesis to Q and e', we have

$$\vdash \{Q\}\ \mathcal{E}'\ \{\mathsf{sp}(Q,\mathcal{E}')\}$$

Let

$$R \triangleq \lambda\sigma\ .\ \mathsf{sp}(Q,\mathcal{E}')\hat{\sigma} \wedge \mathtt{class}(c) = \hat{C} \wedge \sigma = \hat{\sigma}[\cdot, \hat{E} \oplus \hat{w} \overset{(x,j)}{\rightarrowtail} \hat{r}, \hat{w}, \cdot, \{(x,j,B)\}]$$

Again, by induction hypothesis applied to R and $\mathsf{body}(\hat{C},m)$, we get

$$\vdash \{R\}\ \mathsf{body}(\hat{C},m)\ \{\mathsf{sp}(R,\mathsf{body}(\hat{C},m))\}$$

Then it suffices to show that $\mathsf{sp}(R,\mathsf{body}(\hat{C},m)) \Rightarrow S$, where

$$S \triangleq \lambda\sigma\ .\ \mathsf{sp}(P,\mathcal{E}.m(\mathcal{E}'))\ \sigma[\cdot, \mathtt{delv}(\hat{w},E,\Omega), \hat{c}, \cdot, \hat{\Omega}]$$

Consider an arbitrary state σ_2.

- $\mathsf{sp}(R,\mathsf{body}(\hat{C},m))\sigma_2 \implies \exists\sigma'\ .\ \langle\!\langle\ \mathsf{body}(\hat{C},m)\ ,\ \sigma'\ \rangle\!\rangle \rightsquigarrow \sigma_2 \wedge \underbrace{R\sigma'}$
- $R\sigma' = \underbrace{\mathsf{sp}(Q,\mathcal{E}')\hat{\sigma}} \wedge \mathtt{class}(c') = \hat{C} \wedge \sigma' = \hat{\sigma}[\cdot, \hat{E} \oplus \hat{w} \overset{(x,j)}{\rightarrowtail} \hat{r}, \hat{w}, \cdot, \{(x,j,\hat{C})\}]$
- $\mathsf{sp}(Q,\mathcal{E}')\hat{\sigma} \Leftrightarrow \exists\sigma_0\ .\ \langle\!\langle\ \mathcal{E}'\ ,\ \sigma_0\ \rangle\!\rangle \rightsquigarrow \hat{\sigma} \wedge \underbrace{\mathsf{sp}(P,\mathcal{E})\sigma_0} \wedge \hat{w} = c_0$
- $\mathsf{sp}(P,\mathcal{E})\sigma_0 \Leftrightarrow \exists\sigma\ .\ \langle\!\langle\ \mathcal{E}\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma_0 \wedge P\sigma$

The operational semantics of method invocation [E-INV] ensures

$$\langle\!\langle\ \mathcal{E}_0.m(\mathcal{E}_1)\ ,\ \sigma\ \rangle\!\rangle \rightsquigarrow \sigma_2[\cdot, \mathtt{delv}(r_0,E_2,\Omega_2), c_0, \cdot, \Omega_1]$$

which, together with $P\sigma$, implies that $\mathsf{sp}(S,\mathcal{E}.m(\mathcal{E}'))\sigma_2$.

□

5.3 Recursive Methods

Consider the method eq in the example GM program in Figure 2.1.

```
NatNum eq(NatNum n) {...  x = this.pred.eq(n.pred); ...}
```

Application of the proof rule

$$\texttt{[A-INVK]} \qquad \frac{\begin{array}{c}\dots \\ \forall\hat{C}\forall\hat{w}\forall\hat{\sigma}\ .\ \{R'\}\ \mathsf{body}(\hat{C}, m)\ \{S'\} \wedge \hat{C} \preccurlyeq D\end{array}}{\{P\}\ \mathcal{E}.m(\mathcal{E}')\ \{S\}}$$

will result in an endless application of the rule itself because eq appears in both the premise and the conclusion. This problem can be solved by a trick similar to mathematical induction. To establish a proposition by induction, we first prove a base case and then we prove that the proposition is true for n assuming that it is true for $n-1$. For recursion, we prove that the current method call is correct if we assume that the result from any previous call is correct. The base case corresponds to the situation in which the method is called, but it does not call itself again.

Now the assertions (judgments) of the proof system are of the form

$$\mathcal{A} \rhd \{P\}\ \mathsf{t}\ \{Q\}$$

where the *antecedent* $\mathcal{A}$ is a set of assertions. It can be interpreted as follows. Assuming that every assertion in $\mathcal{A}$ is true, if P is true, then Q is true when execution of t terminates. This way we can rewrite the axiomatic semantics to accommodate recursive methods.

Chapter 6

Related Work

6.1 Graph Models

Our graph model is a direct extension of the work of Grogono and Gargul [GG94]. They describe object-oriented programming by graphs, of which vertices represent objects, and edges represent links between the objects. They show how to model simple values, records, and recursive data structures such as lists, trees, and graphs. A record with n fields is represented by a vertex with n out-edges. Their state graph is a tuple (V, E, c, a, r) with an argument vertex a and does not handle local variables. We add local variables to their model, which enables us to remove the argument vertex, and use the extended model to construct semantics for object-oriented programming languages.

Graphs arise naturally in systems with pointer structures. The approach closest to GM is the *OO-machine* of Schmidt and Zimmermann [SZ94]. OO-machine is an abstract machine that uses graphs to represent the states of a state space and graph transformations to model creation of objects and their evolution. Each vertex is an object. Edges are references to other objects, and labels reference fields. The graph transformations, which constitutes the instruction set of the OO-machine, resemble the operational semantics of GM. For example, object creation introduces a new vertex and assignment corresponds to switching a edge from one vertex to another. Cost metrics are then defined in terms of weights for vertices and weights for edges. The authors develop a framework for analyzing space and time complexity of object-oriented programs, and demonstrate the formal model by a subset of the language `Sather` [Omo94]. We use the same tool for different purposes. The goal of Schmidt and Zimmermann is software metrics. We aim at semantics.

In their *trace model* [HH99], Hoare and He describe heap storage by a directed labelled graph. Nodes are objects and edges are fields referencing other objects. Object instantiation is represented by node creation. Updating a value of the heap is equivalent to swinging a directed edge from one node to another. Their goal is to reason about inaccessible memory instead of semantics of object-oriented languages. Along the same line is Jackson's *object models* [Jac03]. The heap of an object-oriented program is represented by a graph. Nodes are objects and edges are field references. Hoare and He use graphs to reason about memory management, not semantics.

GROOVE (GRaphs for Object-Oriented VErification) is a project centered around graphs and graph transformations in a much wider context. Again it uses directed edge-labelled graphs to models state snapshots of systems that involve dynamic allocation and de-allocation of storage space and dynamic method invocation. The ultimate goal is, as the project title suggests, system verification. A first-order graph logic, called *Local Shape Logic (LSL)* is developed to perform model checking on states [Ren04]. A major issue addressed by this modelling technique is the state space explosion. There is an upper bound for the number of state graphs up to isomorphism. We gain insight from GROOVE, as well as from Hoare and He, to tackle the state complexity problem. See Chapter 6.

6.2 Axiomatic Semantics of Object-oriented Languages

Our axiomatic semantics is inspired from the work of Nipkow and Oheimb [vO00, vO01, ON02]. Using the extensional approach, they put forward a Hoare logic for a subset of Java, including dynamic binding of recursive methods and conventional sequential constructs. They formalize the entire system and show it to be sound and (relatively) complete in the theorem prover `Isabelle/HOL`[Isa]. We adopt their technique of universal quantification to formulate proof rules of GM that require passing values between pre-condition and post-condition, and between premise and conclusion. We apply the same technique to handle dynamic binding of method calls. Our treatment of side-effect is simpler since the evaluation result is built into the state graph.

Poetzsch-Heffter and Müller [PHM98, PHM99] present a similar Hoare logic via the extensional approach. They also study a sequential subset of Java and prove that the logic is sound. They develop the concept of *virtual methods*. Briefly, a virtual method is a specification that reflects the properties of all implementations that might be executed, i.e. the implementations in the class under consideration and those in all its subclasses. They use a separate subtype rule to handle dynamic binding. Nipkow and Oheimb also use virtual methods in their logic, using universal quantification instead of a subtype rule. We built our proof rules by comparing virtues and shortcomings of thier work. In particular, we are still investigating to what extent could virtual methods would simplify the rule `[T-INVK]` in Section 5.2.2 for method invocation.

Axiomatic semantics of object-oriented languages using intensional approach include the work of Pierik and de Boer [PdB03]. Their logic is proved to be sound and complete. We did consider an axiomatic semantics for GM using intensional approach but finally abandoned it. Our view is that an assertion language for expressing assertions would only add unnecessary clutter, obscuring the original language features.

6.3 Type Soundness

Our formulation and proof of type soundness is more or less standard. The history of type soundness proof can be traced as far back as the subject reduction theorem for typed λ-calculus. [WF94] has a review of various approaches to soundness proof of functional languages. In [IPW01], Igarashi, Pierce and Wadler prove that a functional fragment of Java is type safe.[1] Their type soundness is stated in terms of preservation and progress, as described in Section 4.3.

Bierman, Parkinson and Pitts [BPP03] formalize an imperative subset of Java which includes many interesting language features such as block structure. They extend the type system to configurations[2] used in the operational semantics and show that a terminal configuration is of a subtype of the original configuration. We borrow their idea of configuration typing to define state graph typing (Section 4.3.1).

[1]A slightly different proof is presented in Chapter 19 of [Pie02].

[2]Bierman et. al. use the term configuration in their work. Configuration is in fact the same as state, as we have already pointed out.

Chapter 7

Conclusion and Future Work

Graph is natural representation of systems with pointer structures, in particular object-oriented programming. Graph is a well-established mathematical object. Its rich theory may be carried over to the study of programming languages. Our graph model is able to describe formally local state, object identity, dynamic binding (by associating methods with vertices), side-effects, aliasing and the use of *self* or *this* for the current object.

Semantics can be formulated in a straightforward way by means of state graph. Our semantic approach uses a state that is more complicated than a simple environment, but which does not need to be extended for additional data structures. It is self-contained. All the run-time information is recorded in a single entity. We believe that the axiomatic semantics of GM is also complete with recursive methods (Section 5.3).

GM is a model for *pure* object-oriented programming languages, e.g. Java, as opposed to *hybrid* languages such as C++. A hybrid language allows *free* function call, i.e. $f(x)$ without a host object. It would be straightforward to extend GM to include free functions. Basically we just need one *great big object* $\mathfrak{b}$ with all of the free functions inside it, so that $f(x)$ is merely an abbreviation for $\mathfrak{b}.f(x)$. Technically, we can use the main object (see Section 2.1) for this great big object.

A more interesting application is garbage collection or memory management. A local variable is no longer accessible when it goes out of scope, i.e. when the method call returns. It can then be deleted from the memory. This is de-allocation of variables in Rule `[E-INVK]` (Section 3.1) and is equivalent to removal of edges from the state graph. As a result, the state graph is broken down into connected components during program execution. The idea is to view state graph as the machine memory and recycle the components. There are well-known algorithms for finding these components. Notice that, since the current vertex is replaced and then restored in method call, disconnected components may be united again into a single piece. We would need to keep a list of accessible vertices and delete those components that do not contain these vertices.

GM models sequential object-oriented programming languages. An obvious future undertaking is the semantics for concurrency and aspect-oriented programming. The need of a formal model in this area is expressed clearly in the following quotation:

> "The development of concurrent object-based programming languages has suffered from the lack of any generally accepted formal foundations for defining their semantics. Furthermore, the delicate relationship between object-oriented features supporting reuse and operational features concerning interaction and state change is poorly understood in a concurrent setting." ([Nie92])

We believe this will be a fruitful direction of the graph model.

Bibliography

[AC96] Martn Abadi and Luca Cardelli. *A theory of objects.* Monographs in computer science. New York : Springer-Verlag, 1996.

[Apt81] Krzysztof R. Apt. Ten years of hoare's logic: A survey - part i. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[BPP03] G.M. Bierman, M.J. Parkinson, and A.M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report UCAM-CL-TR-563, University of Cambridge, Computer Laboratory, april 2003.

[Bru02] Kim B. Bruce. *Foundations of Object-Oriented Programming Languages: Types and Semantics.* Cambridge, Mass. : MIT Press, 2002.

[Car97] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook.* CRC Press, Boca Raton, FL, 1997.

[FJM+96] M. P. Fiore, A. Jung, E. Moggi, P. O'Hearn, J. Riecke, G. Rosolini, and I. Stark. Domains and denotational semantics: History, accomplishments and open problems. *Bulletin of the EATCS*, 59:227–256, June 1996. Also published as Technical Report CSR-96-2, University of Birmingham School of Computer Science.

[Flo67] R. W. Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Proceedings of the International Sympoisum on Theoretical Programming*, volume 19 of *Proceedings of symposia in applied mathematics*, pages 19–32. Providence : American Mathematical Society, 1967.

[GG94] Peter Grogono and Mark Gargul. A graph model for object oriented programming. *ACM SIGPLAN Notice*, 29(7):21–18, 1994.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Reading, Mass. : Addison-Wesley, 1996.

[HH99] C. A. R. Hoare and Jifeng He. A trace model for pointers and objects. In *ECCOP99 - Object-Oriented Programming, 13th European Conference, Lisbon, Portugal, June 14-18, 1999, Proceedings*, volume 1628 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1999.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa74] C. A. R. Hoare. An axiomatic definition of the programming language PASCAL. In *Proceedings of the International Sympoisum on Theoretical Programming*, pages 1–16, London, UK, 1974. Springer-Verlag.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[Isa] `http://isabelle.in.tum.de`. Official website for Isabelle.

[Jac03] Daniel Jackson. Object models as heap invariants. In *Programming Methodology*, pages 247–268, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[Kah87] Giles Kahn. Natural semantics. In F. Bandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Fourth Annual Symposium on Theoretical Aspects of Computer Science*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.

[Men94] Tom Mens. A survey of formal models of OO. Technical Report VUB-TINF-TR-94-03, Department of Computer Science, Vrije Universiteit Brussel, 1994.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Nie92] Oscar Nierstrasz. Towards an object calculus. In Mario Tokoro, Oscar Nierstrasz, and Peter Wegner, editors, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, volume 612, pages 1–20. Springer-Verlag, 1992.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction.* John Wiley & Sons, 1992.

[Omo94] Stephen M. Omohundro. The Sather 1.0 specification. Technical Report TR-in preparation, International Computer Science Institute, Berkeley, 1994.

[ON02] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *Formal Methods – Getting IT Right (FME'02)*, volume 2391 of *LNCS*, pages 89–105. Springer, 2002.

[PdB03] Cees Pierik and Frank S. de Boer. A syntax-directed hoare logic for object-oriented programming concepts. In *Formal Methods for Open Object-Based Distributed Systems*, volume 2884 of *Lecture Notes in Computer Science*, pages 64–78. Springer-Verlag, 2003. Full paper published as Technical Report UU-CS-2003-010, Utrecht University, Department of Information and Computing Sciences.

[PHM98] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET 98)*. Chapman & Hall, 1998.

[PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symosium un Programming (ESOP '99)*, volume 1576, pages 162–176. Springer-Verlag, 1999.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. Cambridge, Mass. : MIT Press, 2002.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[Ren04] Arend Rensink. Canonical graph shapes. In D. A. Schmidt, editor, *Programming Languages and Systems: European Symposium on Programming (ESOP)*, volume 2986 of *Lecture Notes in Computer Science*, pages 401–415. Springer-Verlag, 2004.

[SK95] Kenneth Slonneger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages : A Laboratory Based Approach*. Reading, Mass. : Addison-Wesley Publishing, 1995.

[SZ94] Heinz W. Schmidt and Wolf Zimmermann. Reasoning about complexity of object-oriented programs. In *Proceedings of International Conference Programming Concepts, Methods and Calculi (PROCOMET 94), San Miniato, Italy, 1994*, pages 553–572, 1994.

[vO00] David von Oheimb. Axiomatic semantics for Java$^{\ell ight}$ in Isabelle/HOL. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and

A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs.* Technical Report 269, 5/2000, Fernuniversität Hagen, Fernuniversität Hagen, 2000.

[vO01] David von Oheimb. Hoare logic for Java in Isabelle/HOL. *Concurrency and Computation: Practice and Experience*, 13(13):1173–1214, 2001.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[Win93] Glynn Winskel. *The Formal Semantics of Programming Languages : an Introduction.* Cambridge, Mass. : MIT Press, 1993.

Index