OBJECT-ORIENTED PROGRAMMING: INHERITANCE TO ADOPTION

by

T. B. Dinesh

An Abstract

Of a thesis submitted in partial fulfillment of the requirements for the Doctor of Philosophy degree in Computer Science in the Graduate College of The University of Iowa

May 1992

Thesis supervisors: Professor Arthur C. Fleck
Professor Paul Klint

OBJECT-ORIENTED PROGRAMMING: INHERITANCE TO ADOPTION

by

T. B. Dinesh

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science in the
Graduate College of The
University of Iowa

May 1992

Thesis supervisors: Professor Arthur C. Fleck
Professor Paul Klint

ABSTRACT

Object oriented programming is recognized for its ability to inherit and reuse code and this is reflected by its widespread adoption by the industry in the recent years. A high degree of reusability makes it especially suited for developing and maintaining large software. But as programs get larger, static type-checking becomes imperative, and this has been a concern of the OOP community from the beginning. However, as illustrated by the language Eiffel, the flexibility of inheritance appears to conflict with a language being statically type-checkable.

In this thesis, we generalize the notion of inheritance and delegation, and use this generalization to propose an applicative object oriented programming language which accomodates updating of *instance variables* and a flexible message passing mechanism. The proposed message passing mechanism uses the type declaration information to appropriately alter the message passing semantics in a type sensitive manner. A particular implication of such a system is the notion of *adoption*, where an object could get adopted by a context thus giving an orphan object some features of the context. Inheritance with adoption results in a desirable solution to static type-checking in several contexts when *inherited types* are not subtypes. A flexible and statically type-checkable applicative OOPL is developed to demonstrate the utility of adoption with inheritance. This language is described using ASF+SDF, an algebraic specification formalism.

Abstract approved: ______________________________

Thesis supervisor

____

Title and department

____

Date

____

Thesis supervisor

____

Title and department

____

Date

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

PH.D. THESIS

This is to certify that the Ph.D. thesis of

T. B. Dinesh

has been approved by the Examining Committee for the thesis requirement for the Doctor of Philosophy degree in Computer Science at the May 1992 graduation.

Thesis committee: ______________________________
Thesis supervisor

____
Thesis supervisor

____
Member

____
Member

____
Member

____
Member

To

Amba

ACKNOWLEDGEMENTS

I am deeply indebted to my advisor Art Fleck for his consistent support and many illuminating discussions and meetings over the years, Mahesh Dodani for introducing me to object–oriented programming through many formal and informal discussions, Jon Kuhl for giving me the opportunity to experience the problems of software engineering, G. Steve Hirst and Chung-Shin Tsai for enthusiastically collaborating in many projects we undertook.

Paul Klint and the GIPE group at CWI, Amsterdam for giving me an oppurtunity to work with the best. Their ASF+SDF system has helped demonstrate many of my ideas in a straight forward manner.

I am grateful to my committee for comments on my thesis, to the department of Computer Science for financial support and to CWI for supporting me as a visitor.

I have appreciated the help of many people over the years. Arie van Deursen, Stu Hansen, Wilco Koorn, Gary Leavens, Richard Marciano, Emma van der Meulen, Steve Miller, Halit Oguztuzun, Ramani Ramchandran, Jan Rekers, Teo Rus, Srinivas Sataluri, Ramachandra Topannavar, Lizette Velazquez, Hantao Zhang deserve special mention.

I am unable to thank Susan for the multitude of wonderful moments, Michael for his jovial company and our families for constant encouragement.

TABLE OF CONTENTS

LIST OF EXAMPLES

CHAPTER 1

INTRODUCTION AND OVERVIEW

Object oriented programming is recognized for its ability to inherit and reuse code and this is reflected by its widespread adoption by the industry in recent years. A high degree of reusability makes it especially suited for developing and maintaining large software. But as programs get larger, static type-checking becomes imperative, and this has been a concern of the OOP community from the beginning. However, as illustrated by the language Eiffel, the flexibility of inheritance appears to conflict [16, 42] with a language being statically type-checkable. In this thesis, we study the concepts involved in object oriented programming and discuss alternative perspectives of these concepts that suit development of an applicative object oriented language and also provide for new techniques that facilitate "code reuse" in a statically typed system.

We characterize *objects* by their ability to receive messages and execute methods. A *method* is defined either simply as an object[1] or by sending messages to other objects (including *self*). The methods that are simply defined as other objects are sometimes referred to as *instance variables* in this thesis since these methods are amenable to updating of object by specifying an alternate object. This kind of updating corresponds to the notion of assigning a new value to an instance variable (in an alias free language) in order to obtain an *updated* object.

[1]The set of values may include primitive data as in Section 2.2 or [43], and may not as in Section 5.5 or [26].

A *class* is the basic programming unit, the instances of which are objects. Class definitions are descriptions of objects. They are static and can be accessed during compilation. Objects have a dynamic nature and are found at runtime. A behavioral way of thinking about the objects of a class is that they all accept the same messages and for an identical message history they give the same response.

Classes are named and can be used as a basis for a subsequent class definition and moreover a new class can be defined as a modification of an earlier one[2], using an incremental description. The incremental description is a class schema that defines a new class using the *inheritance* operation. The accepted use of inheritance has changed over the years from a concatenation of description blocks as found in SIMULA [20] to effectively redefining the self reference of the inherited class to that of the new class [26].

An analogous operation that uses objects rather than their descriptions is known as *delegation* [41]. Delegation defines a new object using another object as a prototype, with some incremental information.

There are languages based on delegation only [59, 41] and those based on inheritance only [43, 58, 20]. Some languages effectively provide for the capabilities of both delegation and inheritance [26, 55].

Among the several aspects of combining delegation and inheritance we discuss in this thesis, we identify *adoption* as a runtime augmentation of inheritance that allows an object not modified by inheritance to specialize to a given context.

---

[2] When more than one old class is used, it is called *multiple inheritance*. We will only be concerned with *single inheritance* in this thesis.

1.1 Case for OOP

Simula [20], the first object oriented programming language was developed 25 years ago by Oel-Johan Dahl and Kristen Nygaard. The main emphasis of this language was to provide for the ability to simulate real word objects using classes (later popularized as data abstraction), to introduce the notion of message passing, and to provide code reuse (later popularized as inheritance). Although many programming concepts, such as abstract data types, developed as a result of the introduction of the language Simula, it was not until the development of the language SmallTalk [26] that object oriented programming itself was popularized. SmallTalk is the first language that is developed in a purely object oriented manner, although the notion of classes and inheritance slightly differed from that of Simula. Even though the kernel of SmallTalk is small, the language is widely used due to the availability of many library classes which has inturn promoted the idea of code reuse (inheritance). Although both Simula and SmallTalk advocate code reuse through inheritance, the inheritance mechanism provided in these languages are different. The one provided by SmallTalk is the most popular notion these days.

Simula is a statically type-checkable language, but SmallTalk is not a statically type-checkable language. Mainly due to the nature of large software development environments that SmallTalk has advocated and demonstrated, the later work on OOPL design has converged on developing statically type-checkable languages that are also flexible.

Object-oriented design is the method which leads to software architectures based on the objects every system or subsystem manipulates (rather than the function it is meant to provide).

```
Procedure Print(item: IntCharStringType);
        – – IntCharStringType = Int ∪ Char ∪ String
  Begin
    Case item.tag of
      inttag: Printint(item);
      chartag: Printchar(item);
      stringtag: Printstring(item);
      otherwise: error("Don't know how to print this type");
    end case
end proc;

– – Array A has been arbitrarily assigned Int, Char and String values
for i:=1 to MAX do Print(A[i]) ;
```

Example 1: Procedure to print items of a union type

The program in example 1.1, in a Pascal like language, prints the contents of an array of items. Items in the array belongs to the type IntCharStringType, which can be defined as Int ∪ Char ∪ String, where ∪ denotes a discriminated type union. Printing an item of the array therefore requires the more specific type information, and thus the need for a case statement in the procedure that checks the type tag.

If at a later date, another type element is to be added into the array A, it is necessary to redefine the 'IntCharStringType', the 'Print' procedure and all other procedures that perform operations on items in the array. However, if the design was based on objects in the system, the new object that is to be added to the array A would provide information on how it could be printed. The above 'print' procedure would then be distributed to individual objects. For example, if A[i] is an object of type 'Int', it would have the information on how an 'Int' object can be printed. Thus a statement of the form: for i:=1 to MAX do (A[i]).print would be interpreted as sending the print message to the specific object at the ith location of the array. This would invoke the print method specific to the object found at runtime and therefore message passing semantics is said to give the *delayed* or *late binding* effect.

Objects encapsulate a *state* which is usually represented by instance variables. This state can change over time while the identity of the object remains the same in most object oriented languages; thus *mutating* an object in a prescribed manner. Writing methods that change the state of the object is very common in OOP languages. This state-change should not be regarded as a side effect of an operation, in the sense that good programming practice should avoid it, since this effect is intended in the paradigm. Mutation, however, should be controlled and we feel that the proliferation of mutatability as found in many OO languages is more due to implementation baises than due to need. The languages developed in this thesis are functional, with no aliasing or mutation. In the language SOOL (Section 3.2) delegation is used to update an object, while in language ASOOL (Section 5.7) updating self mutates the current environment for the scope of the invocation of a method.

1.2 Specialization of classes using inheritance

Inheritance, as found in many languages is not very restrictive and has lead to many discussions of controlled uses of it. Cardelli [9] used it to mean subtypes (substitutability in all contexts), while Bruce [7] uses it to identify a little more relaxed relation that need not be just subtypes, but *inherited types* [17].

We can think of inheritance as a mechanism to help specialize an earlier defined class to a more specific context, by making it easy to add new methods or modify the behavior of some of the methods.

When there is a need for changing the behavior of a class to a more specific use, say when a black and white terminal is replaced with a colored terminal, one would want to change the behavior of "points" to "colored points". Editing the point class to add color attributes could uniformly replace the old notion of points to the new one. However, such editing would either be impossible or disastrous. Impossible, since the class might be part of a library of classes that cannot be changed or might be part of a monolithic software that cannot be directly altered, but can only be extended, due to the unavailability of source code specification. Disastrous, since the class might be used for specifying other classes or for purposes very different than where the need for specialization exists. *Specialization* defines a new class without changing the old, with the hope of substituting the resulting objects where ever the specialization is desired. However, circumstances may prevent all objects encountered within the specialization from being within the scope of the (static) description of the specialization. We call those objects which cannot be captured by a static description of specialization in this manner, *orphan* objects.

In this thesis, we defer to delegation to help specialize certain orphan objects that appear in a specialized context. Since, in the class of problems that we are interested, the context in which the *not yet specialized* objects get detected will have the information for the required specialization, we recognize this as the case where a context can *adopt* the orphan object.

1.3 Statically typed object oriented languages

The problem that needs to be resolved by strongly-typed OOP languages, like Eiffel, is the precise definition of when an object of type A can be substituted by an object of type B. In order to do static type checking, this problem translates to a problem of defining when a class behaves like another. A naive notion of "behaves" is the notion of subtype that can be exploited by the static type checker. A natural definition of *subtype* is: A is subtype of ($\leq$) B iff for every f, a method, in B there exists an f in A such that the *contravariance* rule (f:C$\rightarrow$ D $\leq$ f:C$'$$\rightarrow$D$'$ iff D$\leq$D$'$ and C$'$$\leq$C) holds.

Consider the trivialized POINT and ColoredPOINT example in an Eiffel [43] like language:

class POINT **exports**

x, y, print

method print **is**

return = "POINT"

class ColoredPOINT **inherits** POINT

redefines print **as** prt

exports x, y, c, print

method print **is**

return = "Colored"#prt() – – # is string concatenation

Even though ColoredPOINT behaves like POINT, we cannot conclude that ColoredPOINT $\leq$ POINT, with this definition because

print: ColoredPOINT $\rightarrow$ STRING $\not\leq$ print: POINT $\rightarrow$ STRING

This is not a surprise, as the *polymorphism* (use of a single function for different types of data) that can be observed is more apparent than real. The print message sent to an instance of ColoredPOINT invokes the method described in ColoredPOINT whose definition is different from that of the print method in

POINT. Where as, if it were real polymorphism, the same print method would be invoked for arguments of different types.

This apparent polymorphism suggests that we should consider the signature of print instead to be print: $\rightarrow$ STRING. However for operations with two or more arguments, we still have to follow the contravariance rule for the arguments of the methods since the apparent polymorphism only allowed us to hide the type of the first argument (which was the type of the receiving object). The apparent polymorphism also illustrates that inheritance does preserve algorithmic abstraction, as is the case with parametric polymorphism and therefore a careful statement of contract compliance [43, 1] is demanded over the general subtyping rules. In this thesis, we consider only subtype conformance that can be realized using type signatures.

In general, inheritance (subclass) does not imply a subtype relation, as demonstrated by [54, 17], since inheritance does not require a contravariance rule.

The message passing paradigm facilitates a simple algorithm for selecting the target method dynamically. Selection of the method to be invoked is not affected by all the arguments of the method, but by only one (the target object). This selection rule together with dynamic binding, we claim, has led to incomplete operational semantics in languages like Eiffel. Thus a proper (in the sense of completeness) interpretation of message passing is necessary for proper static type checking. Several alternatives have been considered [58, 36, 25]. The adoption mechanism suggested in this thesis provides for an orthogonal approach that accommodates for some desirable cases that cannot be handled by other techniques.

Type denotes a collection of objects that share the same externally observable behavior. Type, as used in this thesis, is more general than a class as it considers type conformance using signatures and it is possible that objects of several different classes conform to the same type.

Definition 1.3.1 *The* conformance *or* subtype *relation is defined as [9, 1]: The type τ is a subtype of σ if, for each method μ_σ of σ there is a corresponding method μ_τ of τ, such that*

- *μ_σ and μ_τ have the same name.*
- *μ_σ and μ_τ have the same number of arguments.*
- *The ith parameter type of μ_σ is a subtype of the ith parameter type of μ_τ (the contravariant parameter type rule).*
- *The result type of μ_τ is a subtype of the result type of μ_σ (the covariant result type rule).*

Let us consider the following type declarations that reflect the signatures of the methods, of the objects that belong to this type (including its subtypes). The comments provide an intuition for the objects of this type.

```
Type Line
    p: → Point                  – – return p point
    q: → Point                  – – return q point
    samelen: Line → Boolean– – check lengths

Type LineLen subtype³of Line
    length: → Real              – – distance from p to q
```

implying the following type:

Type LineLen

p: → Point

q: → Point

length: → Real

samelen: Line → Boolean

The subtype relation is important in two situations: assignments and parameter passing. The messages defined in LineLen are at least the messages defined in Line and we can desire an instance of LineLen to occur where ever an instance of Line might be permitted. In particular, note that samelen accepts Line type arguments. Let L1, L2 be variables declared to be of type Line and LL be of type LineLen, then statements of the form:

if (L1.samelen(L2)) then ...

if (LL.samelen(L2)) then ...

would be valid even though either of L1 or L2 could have at runtime, a more specific object of type LineLen.

A declaration of the form of LineLenS given below, is however, invalid:

Type LineLenS subtype of Line

length: → Real

samelen: LineLenS → Boolean

Since it would imply the type:

[3]For this example we assume that "A **subtype of** B" means extend the method signatures of B, using the additional information to add/override signatures as long as the resulting type (A) conforms to B

Type LineLenS

p: $\rightarrow$ Point

q: $\rightarrow$ Point

length: $\rightarrow$ Real

samelen: LineLenS $\rightarrow$ Boolean

samelen in LineLenS accepts only arguments of type LineLenS and is therefore not at least as defined as samelen in type Line. The reason to disallow this as a subtype is explained by the following pathological case:

L1, L2 : Line ; ...

if (L1.samelen(L2)) then ...

looks like an obviously type correct set of statements. However,

L1, L2 : Line ; LL1 : LineLenS ;

L1 := LL1 ;

if (L1.samelen(L2)) then ...

may fail at runtime, since the samelen invoked will be LL1.samelen, which might require L2.length, which need not be defined (no length attribute in type Line).

Some languages do not require the contravariance of parameters, even though they emphasize static type-checkability (Eiffel [43] and BETA [38]). The redefinition rule, for redefining a method of the superclass in Eiffel [43, page 263], is paraphrased here as:

> An *attribute*, a function result or a *formal routine argument* declared in a class may be redeclared with a new type in a descendant class, provided the *new type conforms to* the original one.

Note that the emphasized words imply a covariance of arguments, unlike the contravariant type rule in the subtype definition. The result being that even though the programs are statically type checked as correct, say by Eiffel [43], they can fail at runtime when subtype means substitutability in all contexts.

Cook remarks, in his proposal to make Eiffel type-safe [16], that the "contravariance rule has the unfortunate effect of making argument type redefinition almost useless, since it is usually not very useful to allow a redefined method to accept a larger class of arguments". Since in practice, it is very common and useful to have types evolve in the manner of LineLenS, languages have avoided a contravariance requirement on arguments. Then LineLenS can use the new definition of length defined in LineLenS only, thus making the samelen in LineLenS more specific (and possibly more efficient).

In this thesis we look at this problem of avoiding a contravariance requirement without introducing any type insecurity from a different point of view. We try to view a subtype definition as information on how to specialize the supertype. This information is generally available in the environment and a message passing mechanism can harness the information to specialize an argument at runtime, if necessary — thus the effect of an uninformed object being adopted by the context which is aware of the intended specialization. This provides a way of relaxing the contravariance rule in many subtype definitions while avoiding run-time type errors.

1.4 Thesis overview

The constructions given for inheritance [15, 51] seem to be restrictive and we suggest thinking of delegation as a more useful mechanism even for "inheritance based languages". A suitable construction is provided in Chapter 2 that can be used for our purposes, and also in a more general context where mixing delegation and templates would be necessary. The mechanisms of SmallTalk is an example where this is necessary, since classes are first class objects. Chapter 2 is thus a discussion of the limitations of the formulations of inheritance to extend it to delegation or to better describe the inheritance mechanisms of SmallTalk. Pairing instances with their templates, and using the message passing mechanism to handle recursion, results in obtaining delegation in a general setting. Problems in accommodating the notion of recursion with delegation in the presence of templates are discussed. With a discussion of the desirable features of an object-oriented language, some constructs are provided for developing an applicative version. The general formulations developed in this chapter are the basis for the rest of the thesis.

Chapter 3 demonstrates the use of the formulations developed in Chapter 2 by proposing a message time delegation in a context-sensitive manner, so that code reuse can be achieved in many contexts where it is restricted now by the type environments. Chapter 4 adopts the general idea of Chapter 3 to the restricted context of the language Eiffel. A more restricted language that only identifies "inherited types" [17, 7] as subtypes is considered in detail in Chapter 6.

Chapter 5 gives an algebraic specification of a subset of the mechanisms developed in Chapter 2 using ASF+SDF. Many features that make an object oriented language interesting, both from the point of its dynamic semantics and type-checking issues are discussed in more detail. Appendix C contains the complete specification of the resulting language used in Chapter 6. Chapter 6 demonstrates the use of adoption, by specifying a language similar to Eiffel that identifies its sub-type hierarchy with the inheritance hierarchy but by only identifying inherited types with subtypes.

CHAPTER 2

MECHANISMS OF DELEGATION AND INHERITANCE

2.1 Introduction

We give a quick introduction to the notion of inheritance and delegation. For this introduction, we construct an example based on [15] which generalizes the notion of object in object oriented programming to accept natural numbers as messages (an infinite number of messages).

Lets say we have a functional F, whose least fixed point models the notion of an object (which has only one internal state) that takes the messages 0, 1, 2, $\cdots$ and returns a natural number, defined as (the symbol $\longleftarrow$) follows[1]:

$$F \longleftarrow \lambda self.\lambda n.if\ n = 0\ then\ 1\ else\ n * (self(n-1)).$$

The fixed-point of F is the factorial function defined by $fact \longleftarrow Y\ F$ where Y is the call-by-value fixpoint operator [57, 2, 46], (i.e., $F\ fact = fact$). The function $fact$, so defined, is equivalent to the recursive definition:

$$fact \longleftarrow \lambda n.if\ n = 0\ then\ 1\ else\ n * (fact(n-1)).$$

The explicit use of fixpoint operator and the definition of a functional will help us understand inheritance (defined in Section 2.2). Now, if a user instead desires, for example, a function

$$f \longleftarrow \lambda n.if\ n = 3\ then\ 3\ else\ if\ n = 0\ then\ 1\ else\ n * (f(n-1)),$$

how may the user specify this function as an "incremental modification" of fact's specification? What is meant by incremental modification is that, the structure of

[1]For λ-calculus and denotational semantics we use [57, 2, 27]

f and $fact$ are the same except for the additional condition, $if\ n = 3\ then\ 3\ else\ if\ \cdots$, and the goal is to specify f by just specifying this difference. This is achieved by specifying another functional F', which explicitly uses "its parent" F for its default behavior

$$F' \longleftarrow \lambda self.\lambda n.if\ n = 3\ then\ 3\ else\ (F\ self)\ n,$$

the fixed-point of which is f.

Explicitly naming F, in F' leads to a static hierarchy between F and F'. Instead if the incremental modification can be specified as a composable function, then it could be used to modify function definitions dynamically (at run-time). Inheritance insists that such bindings be recognized statically, while delegation allows for it to happen at run-time.

F' can be obtained instead, by giving the differences without explicitly naming F a-priori: Let

$$G \longleftarrow \lambda self.\lambda n.if\ n = 3\ then\ 3\ else\ self\, n,$$

then $F' = G \circ F$, where $G \circ F \longleftarrow \lambda self.G(F\ self)$ is the usual function composition.

In the above, F and F' can be considered as specifying classes[2] since $fact$ and f are their unique instances respectively and G as specifying the incremental modification (we will call it "delta") necessary to obtain F' from F using inheritance. $fact\ n$, can then be read as passing a message n to object $fact$. However, there is sense in considering F and F' as objects themselves with a message passing operator, $\ll$, defined as (using infix notation): $g \ll x \longleftarrow (\mathrm{Y}\ g)x$. Now, $fact\ n$ is the same as $F \ll n$ and $f\ n$ is the same as $F' \ll n$ and G defines incremental modification. One can, in the jargon of delegation, read the definition

[2]In general $Fclass(x) \longleftarrow \lambda self.\lambda n.if\ n = 0\ then\ x\ else\ n * self(n-1)$, can be used to define several instances, of which $fact \longleftarrow Y(Fclass\ 1)$ is one. We refer to these as class *templates*

of F' as: G handles the message 3 while other messages are delegated to F.

We can also, define the factorial differently, using what is referred to as an implementation oriented view, by obtaining the desired recursion using a message passing operator defined as $rcvr \ll_i msg \longleftarrow (rcvr\ msg)\ rcvr$. Then with

$$
\begin{aligned}
Fact \longleftarrow \lambda n.if\ n = 0\ then\ &\lambda self.1, \\
else\ &\lambda self.n * (self \ll_i (n-1))
\end{aligned}
$$

$fact\ n$ will be equivalent to $Fact \ll_i n$. Equivalence of these two different ways of viewing an object (effects $\ll$ and $\ll_i$) is discussed by Kamin and Reddy [34].

With this prelude to help understand object, class, inheritance and delegation in a general setting, we proceed to describe what objects are in object oriented programming. Unlike $fact$, objects typically understand a finite number of messages and for each message a function, better known as "method", is defined. These methods may be defined in a mutually recursive manner, using *self* to refer to other methods and, in a delta, *super* (Section 2.2) to refer to methods of the parent.

We begin with the compositional model of inheritance, as given by Cook [15]. Changes are then introduced which incrementally work toward achieving delegation and inheritance in a general setting.

The concepts of delegation and inheritance are widely used in the object-oriented community [60], since they enable reuse of existing code without modifying it. The key feature of delegation is that "self-reference" in an ancestor dynamically denotes the delegating object, not the ancestor, thereby allowing the ancestor's behavior to be part of the extended identity of the delegating object. Dynamic binding of self-reference realizes sharing and reusability by allowing the resources of an ancestor to be part of the extended identity of different delegating objects at different points of execution [41].

The notion of inheritance is similar to delegation except that inheritance is used in the context of extending classes, which define templates for object instantiation [26]. This interpretation of inheritance gives it a static structure, where as delegation carries with it an ability to specify incremental changes to existing objects at runtime, an effect better understood by thinking of delegation as a mechanism to use classes as objects with an ability to “inherit” dynamically.

The presentation of the chapter begins with the semantic constructions of inheritance as given by Cook [15] and discusses the changes necessary to extend the language to facilitate delegation, templates, and the mix of both delegation and templates, as shown below.

$\mathcal{C}_1$, Inheritance with simple classes
(point and circle example)

$\mathcal{C}_2$, Adds delegation
(movable point, using updating)

$\mathcal{C}_3$, Adds templates
(movable point, using template)

$\mathcal{C}_4$, Combines delegation and templates
(movable point by using both updating and templates)

The presentation uses simple examples, based on examples given by Cook [15], which demonstrate the use of delegation and templates for creating other objects *like* self. Although the examples demonstrate the utility of delegation for updating objects, in Chapter 3 delegation is used in a more general sense.

2.2 A language with inheritance

We assume a knowledge of λ-calculus [57], which we use to give the semantics of the languages we define.

The semantics of the constructs are given by a translation to λ-calculus. We consider the λ-calculus with syntax for records, conditionals and "let" declarations (initialized definition of new names [57, page 226]) and some primitive domains (eg., Booleans, Identifiers, Naturals) and their operations [57, page 156]. The general convention in λ-calculus is that application ($F_1\ F_2$ below) associates to the left, while abstraction ($\lambda I.F$) associates to the right.

Let For and Rec be the domains:

For	the domain of formulas	F
Rec	the domain of Records	R

with primitive domains:

VIde	the domain of variable identifiers	I
Lab	the domain of labels	$\bar{I}$
Bas	the domain of constants	B
BOP	the domain of basic operators	O

The over bar on $\bar{I}$ is not used when it is clear from the context that it is a label.

The syntactic clauses are:

$$
\begin{aligned}
F \quad ::= \quad & I \mid\ \mid B \mid O\ F \mid F_1\ O\ F_2 \mid F_1\ F_2 \mid (F) \\
& \mid \text{if } F_1 \text{ then } F_2 \text{ else } F_3\ \mid \lambda I.F \mid \mathbf{Y}F \mid \text{let } I = F_1 \text{ in } F_2 \\
R \quad ::= \quad & [] \mid [\bar{I}\ \mapsto F] \mid R_1 \oplus R_2 \mid R.\bar{I}
\end{aligned}
$$

where the conditional is strict and

let $I = F_1$ in $F_2 \longleftarrow (\lambda I.F_2)F_1$.

We use a small language $\mathcal{C}_1$ to aid in discussing the notion of inheritance. The definitions for inheritance used in this section are from Cook [15].

2.2.1 Basic definitions

Definition 2.2.1 *A* record *is a finite mapping from a set of labels into a set of values. A record is denoted by* $\begin{bmatrix} x_1 & \mapsto & v_1 \\ & \vdots & \\ x_n & \mapsto & v_n \end{bmatrix}$ $(n \geq 0)$ *with labels* x_i *and values* v_i*. All labels that are not in the list are mapped onto* $\perp$*. The empty record is denoted by [].*

Definition 2.2.2 *For records, let* $dom(m) = \{x | m(x) \neq \perp\}$*. The* left-preferential combination *of two records m and n is defined by:*

$$(m \oplus n)(s) = \begin{cases} m(s) & \textit{if } s \in dom(m) \\ n(s) & \textit{if } s \in dom(n) - dom(m) \\ \perp & \textit{otherwise} \end{cases}$$

An object can be thought of as a record with functions as values. The messages that an object accepts will then be the labels of the record that are not mapped to $\perp$. When an object receives a message it executes the method associated with the message name, which will then be the application of the function associated with the corresponding label.

A *generator* is a function to which a fixed point operator can be applied. Its first formal parameter represents self-reference. A simple *class* is a generator that creates objects by means of a fixed point operation. Classes in general have formal parameter lists that should be instantiated before they can be termed simple classes. In this chapter, we use the word *template* to distinguish the general form from a simple class and *class* to mean simple class.

Inheritance is the derivation of a new class from an existing one, so that the self reference is shared between these classes. A *wrapper* is a function that modifies a class (generator) by defining a new self reference that wraps around the self reference of the class. A wrapper has a parameter for self-reference and a parameter for the generator it modifies. Definition 2.2.3 and definition 2.2.4 will show how wrappers operate on classes:

Definition 2.2.3 *Let* $*$ *be a binary operator on values, then the* distributive version *of* $*$ *is denoted by* $\boxed{*}$. *It operates on functions and is defined by:*

$$G_1 \boxed{*} G_2 \longleftarrow \lambda s.G_1(s) * G_2(s)$$

The $\boxed{*}$ *is generally used over the generators.*

Definition 2.2.4 *The* inheritance function $\boxed{\triangleright}$ *applies a wrapper* W *to a class* C *and returns a class.* $\triangleright$ *is defined by:* $w \triangleright c \longleftarrow (w\ c) \oplus c$. *Since conjunction (white space) denotes the binary application operator, we get*

$$W \boxed{\triangleright} C = (W \boxed{} C) \boxed{\oplus} C$$

The domain of wrappers is $Wrapper = Object \rightarrow Object \rightarrow Object$.

2.2.2 Syntactic domains and clauses

The language $\mathcal{C}_1$ is described using a translation to λ-calculus. Primitive syntactic domains,

Ide	the domain of identifiers	I
Bas	the domain of constants	B
BOP	the domain of operators on B	O

compound syntactic domains,

Pro	the domain of Programs	P

Exp	the domain of expressions	E
Cla	the domain of class declarations	K
Meth	the domain of method declarations	M

and the syntactic clauses:

$$
\begin{aligned}
P \quad &::= K\ E \\
K \quad &::= \textbf{class}\ I\ M \mid \textbf{class}\ I\ \textbf{subclass of}\ I_1\ M \mid K_1\ K_2 \mid \epsilon \\
E \quad &::= B \mid I \mid E \bullet I(E_1, \cdots, E_n) \mid \textbf{self} \mid \textbf{super} \\
&\quad \mid E \bullet I \mid \textbf{new}\ I \mid O\ E \mid E_1\ O\ E_2 \mid \textbf{if}\ E\ \textbf{then}\ E_1\ \textbf{else}\ E_2 \mid (E) \\
M \quad &::= \textbf{meth}\ I(I_1, \cdots, I_n)\ = E \mid M_1\ M_2 \mid \epsilon
\end{aligned}
$$

2.2.3 The translation function

The translation function is $\langle\!\langle \_ \rangle\!\rangle : \mathcal{C}_1 \rightarrow \lambda$-calculus.

$$
\begin{aligned}
\langle\!\langle K_1, \cdots, K_n\ E \rangle\!\rangle &= \text{let } \langle\!\langle K_1 \rangle\!\rangle \text{ in } \cdots \text{ let } \langle\!\langle K_n \rangle\!\rangle \text{ in } \langle\!\langle E \rangle\!\rangle \\
\langle\!\langle \textbf{class}\ I_1\ M \rangle\!\rangle &= I_1 = \lambda \mathit{self}. \langle\!\langle M \rangle\!\rangle \\
\langle\!\langle \textbf{class}\ I_1\ \textbf{subclass of}\ I_2\ M \rangle\!\rangle &= I_1 = (\lambda \mathit{self}. \lambda \mathit{super}. \langle\!\langle M \rangle\!\rangle)\ \boxed{\triangleright}\ I_2 \\
\langle\!\langle M_1, \cdots, M_n \rangle\!\rangle &= [] \oplus \langle\!\langle M_1 \rangle\!\rangle \oplus \cdots \oplus \langle\!\langle M_n \rangle\!\rangle \\
\langle\!\langle \textbf{meth}\ I(I_1, \cdots, I_n)\ E \rangle\!\rangle &= [\bar{I}\ \mapsto \lambda I_1. \cdots \lambda I_n. \langle\!\langle E \rangle\!\rangle] \\
\langle\!\langle \textbf{if}\ E\ \textbf{then}\ E_1\ \textbf{else}\ E_2 \rangle\!\rangle &= \text{if } \langle\!\langle E \rangle\!\rangle \text{ then } \langle\!\langle E_1 \rangle\!\rangle \text{ else } \langle\!\langle E_2 \rangle\!\rangle \\
\langle\!\langle \textbf{new}\ I \rangle\!\rangle &= \mathbf{Y}\ \mathrm{I} \\
\langle\!\langle E_1\ O\ E_2 \rangle\!\rangle &= \langle\!\langle E_1 \rangle\!\rangle\ \langle\!\langle O \rangle\!\rangle\ \langle\!\langle E_2 \rangle\!\rangle \\
\langle\!\langle O\ E \rangle\!\rangle &= \langle\!\langle O \rangle\!\rangle\ \langle\!\langle E \rangle\!\rangle \\
\langle\!\langle E \bullet I \rangle\!\rangle &= \langle\!\langle E \rangle\!\rangle . \bar{I} \\
\langle\!\langle E \bullet I(E_1, \cdots, E_n) \rangle\!\rangle &= (\langle\!\langle E \rangle\!\rangle . \bar{I})\ \langle\!\langle E_1 \rangle\!\rangle \cdots \langle\!\langle E_n \rangle\!\rangle \\
\langle\!\langle (E) \rangle\!\rangle &= \langle\!\langle E \rangle\!\rangle
\end{aligned}
$$

class Point
  meth $x = 3$
  meth $y = 4$
  meth *distFromOrig* = sqrt($\mathbf{self}{\bullet}x^2 + \mathbf{self}{\bullet}y^2$)
  meth *closerToOrig*(p) = $\mathbf{self}{\bullet}\mathit{distFromOrig} < \mathrm{p}{\bullet}\mathit{distFromOrig}$

Example 2: Class Point with *distFromOrig*

$$\langle\!\langle \mathbf{self} \rangle\!\rangle = \mathrm{self}$$

$$\langle\!\langle \mathbf{super} \rangle\!\rangle = \mathrm{super}$$

The expression before the inheritance operator in the third clause is a wrapper. *self* and *super* are λ-abstracted. The methods and method calls are curried, when translated. The names chosen for classes should be unique and should not be captured by the method argument names.

2.3 Inheritance

A class of Cartesian points can be described as shown below, using the behavior of four methods. The locations of points are specified by x and y methods. The *distFromOrig* method computes their distance from the origin. The *closerToOrig* method takes another point object and compares their distances from the origin.

A point at (3, 4) is specified by the class in Example 2.3.

The *self* reference in the class definition is captured by the abstracted self variable (second equation of the translation function and as formalized in [9] and [15]):

class Circle **subclass of** Point
  meth radius = 5
  meth distFromOrig = max((**super**.distFromOrig − **self**.radius),0)

Example 3: Class Circle modifies *distFromOrig*

$$\text{Point} \longleftarrow \lambda \mathit{self}.\left[\begin{array}{lcl} x & \mapsto & 3, \\ y & \mapsto & 4, \\ distFromOrig & \mapsto & sqrt(\mathit{self}.x^2 + \mathit{self}.y^2), \\ closerToOrig & \mapsto & \lambda p.(\mathit{self}.distFromOrig < p.distFromOrig) \end{array}\right]$$

A point at (3, 4) is created by "new Point" which is (Y Point), and defines the record with "." being the usual field selector.

[x $\mapsto$ 3, y $\mapsto$ 4, distFromOrig $\mapsto$ 5,
closerToOrig $\mapsto$ λp.(5 < p.distFromOrig)]

closerToOrig method takes an argument that is assumed to be a point. Actually, it need only be a record with a distFromOrig method whose value is a number.

Inheritance allows a new class to be defined, by adding or replacing methods in an existing class. For example, the Point class may be inherited in defining a class of Circles (Example 2.3). Circles have a different notion of distance from origin and the method *distFromOrig* is therefore redefined.

Circle is defined by using Point for both the default behavior and for super behavior (in the wrapper), while making sure that self of Point is bound to the self of Circle.

This is expressed in the inheritance function by a three step process, defined by CircleWrapper $\boxed{\triangleright}$ Point. First the difference (we call it delta) is specified by a CircleWrapper defined with no reference to Point. Second the self of Point is bound to self of Circle using the $\boxed{}$ operator (Definition 2.2.3). Third, a left preferential combination, $\oplus$, is done to yield Circle:

$$\text{CircleWrapper} \longleftarrow \lambda \mathit{self}.\lambda \mathit{super}.$$
$$\begin{bmatrix} radius & \mapsto & 5, \\ distFromOrig & \mapsto & max(super.distFromOrig - \mathit{self}.radius, 0) \end{bmatrix}$$

$$\text{CircleResolveSuper} \longleftarrow \text{CircleWrapper} \boxed{} \text{Point}$$
$$\text{Circle} \longleftarrow \text{CircleResolveSuper} \boxed{\oplus} \text{Point}$$

The above is equivalent to defining Circle as:

$$\text{Circle} \longleftarrow \lambda \mathit{self}.$$
$$\begin{bmatrix} x & \mapsto & 3, \\ y & \mapsto & 4, \\ radius & \mapsto & 5, \\ distFromOrig & \mapsto & max(sqrt(\mathit{self}.x + \mathit{self}.y) - \mathit{self}.radius, 0), \\ closerToOrig & \mapsto & \lambda p.(\mathit{self}.distFromOrig < p.distFromOrig) \end{bmatrix}$$

Note that the resulting definition of distFromOrig has a syntactic substitution of the method definition from "super" class and this kind of substitution redefines the old semantics to the new context. This syntactic substitution can be done explicitly even before the $\langle\!\langle \_ \rangle\!\rangle$ function is applied, as is done in the algebraic specification (Chapter 5).

The classes as definable in this language are in fact specifying a unique object[3]. Although we used the definitions of class and inheritance to demonstrate the effect of redefining the "self" of an object, it is an analogous effect in delegation. A language that supports delegation would then provide a form of the "inheritance function" in its expression syntax, thus allowing for the redefinition of "self" in a more dynamic sense, than the static hierarchy obtained from the class inheritance relation provided in $\mathcal{C}_1$.

2.4 Delegation

In this section we modify the syntax of $\mathcal{C}_1$ presented in Section 2.2, to $\mathcal{C}_2$, and include a delegation operation (;) in its expression syntax, and discuss the necessary modifications that provide us the desired behavior.

We extend the domain of expressions of $\mathcal{C}_1$ to include the ability to delegate to another expression (effectively another object).

$E \quad ::= E \; ; \; E \mid \cdots$

This immediately tells us that we need to alter the way we are thinking of objects since in the translation function given for $\mathcal{C}_1$, the translation of "new I" is a fixed point of a generator and the self is fixed (completely defined) during object creation. Delegation needs redefinition of self and therefore an apparent strategy would be to delay applying the $\mathbf{Y}$ operator to a later stage. An appropriate place might be to fix the self before invoking a method. The following changes to the translation function defines our intention:

$$\langle\!\langle E{\bullet}I(E_1, \cdots, E_n) \rangle\!\rangle \quad = \quad ((\mathbf{Y} \; \langle\!\langle E \rangle\!\rangle).\mathrm{I}) \; \langle\!\langle E_1 \rangle\!\rangle \cdots \langle\!\langle E_n \rangle\!\rangle$$

This modification implies that we need to distinguish calling a method of

[3]Generalization of this notion of class to template classes is considered in Section 2.5

self, from sending a message to others, since self will be a fixed-point (some method is invoked) and others are not fixed. So invoking a method of self is simple field selection of a record, while sending a message to others is as defined above.

We also need to define $E;E$. The desirable translation would be to use the inheritance function $\boxed{\triangleright}$ to get the effect of delegation. This requires us maintain wrappers and generators which gets restrictive since in effect we cannot use the delegation operation between two arbitrary[4] expressions.

For example, a desirable modification of the closerToOrig method, so that it specializes the argument to Circle using the CircleWrapper would be:

meth closerToOrig(p) = self.distFromOrig <

((CircleWrapper ; p)$\bullet$ distFromOrig)

where CircleWrapper is defined as before. Note that using p on the left of the ; is not well defined and hinders in uniform treatment of "objects".

2.4.1 Wrappers for delegation

Definition 2.4.1 *The* extended inheritance function $\boxed{\succ}$ *applies a wrapper to a wrapper and returns a wrapper.* $\succ$ *is defined by:* $w_1 \succ w_2 = (w_1 \circ w_2) \boxed{\oplus}\, w_2$ *and hence:*

$$W_1 \boxed{\succ} W_2 \longleftarrow (W_1 \boxed{\circ} W_2) \boxed{\boxed{\oplus}} W_2$$

Definition 2.4.2 *We can achieve uniformity by using an everywhere undefined super parent* Top *(a trivial generator), which can be used to cap off the open end of these wrappers during a message send with the* $\tilde{Y}$ *function:*

$Top \longleftarrow \lambda\text{self}.\lambda msg.\bot$

[4]We do not consider the case where an expression evaluates to the domain of constants.

$\tilde{Y}\ rcvr \longleftarrow \mathbf{Y}(rcvr \boxed{}\ Top)$

This uniformity is necessary to achieve a flexible system with delegation[5].

Changes to the domain of expressions (page 21):

$E \quad ::= \cdots \mid E\ ;\ E$

with translation of class and message passing redefined and the translation of new construct given as:

$$
\begin{array}{lcl}
\langle\!\langle \mathbf{class}\ I_1\ M \rangle\!\rangle & = & I_1 = \lambda self.\lambda super.\langle\!\langle M \rangle\!\rangle \\
\langle\!\langle \mathbf{class}\ I_1\ \mathbf{subclass\ of}\ I_2\ M \rangle\!\rangle & = & I_1 = (\lambda self.\lambda super.\langle\!\langle M \rangle\!\rangle) \boxed{\succ} I_2 \\
\langle\!\langle \mathbf{new}\ I \rangle\!\rangle & = & \mathrm{I} \\
\langle\!\langle E \bullet I(E_1, \cdots, E_n) \rangle\!\rangle & = & ((\tilde{Y}\ \langle\!\langle E \rangle\!\rangle).\mathrm{I})\ \langle\!\langle E_1 \rangle\!\rangle \cdots \langle\!\langle E_n \rangle\!\rangle, \text{ when } E \neq \mathbf{self} \\
\langle\!\langle \mathbf{self} \bullet I(E_1, \cdots, E_n) \rangle\!\rangle & = & (self.\mathrm{I})\ \langle\!\langle E_1 \rangle\!\rangle \cdots \langle\!\langle E_n \rangle\!\rangle \\
\langle\!\langle E_1\ ;\ E_2 \rangle\!\rangle & = & \langle\!\langle E_1 \rangle\!\rangle \boxed{\succ} \langle\!\langle E_2 \rangle\!\rangle
\end{array}
$$

Translation of message send has two cases above depending on whether E is the variable self or not, which informs that the structure of self is different from other objects.

For completeness we consider the Point and Circle examples:

$$
\mathrm{Point} \longleftarrow \lambda self.\lambda \mathrm{super}.
\left[\begin{array}{lcl}
x & \mapsto & 3, \\
y & \mapsto & 4, \\
distFromOrig & \mapsto & sqrt(self.x^2 + self.y^2), \\
closerToOrig & \mapsto & \lambda p.self.distFromOrig < (\tilde{Y}\ p).distFromOrig
\end{array}\right]
$$

$\mathrm{Circle} \longleftarrow$

[5]Some novelties of the associative property of it are discussed in Chapter 3.

let cdelta $\longleftarrow \lambda \mathit{self}.\lambda$super.
$$\left[\begin{array}{lcl} radius & \mapsto & 5, \\ distFromOrig & \mapsto & max(super.distFromOrig - \mathit{self}.radius, 0) \end{array}\right]$$
in cdelta $\boxed{\succ}$ Point

2.4.2 Updating with delegation

The power of delegation can be illustrated by the following two simple examples. First, let us consider modifying the method closerToOrig to compute closer to origin as before, but assuming that the argument point is on the same y axis as *self*. We can change the method to:

closerToOrig $\mapsto$ λp.(*self*.distFromOrig < (($[\![$ x $\mapsto$ *self*.x $]\!]$ $\boxed{\succ}$ p)$\bullet$distFromOrig))

where, $[\![\ \_ \mapsto \_\ ]\!]$ is defined as follows:

Definition 2.4.3 $[\![\ x \mapsto v\ ]\!] \longleftarrow \lambda\text{self}.\lambda super.[x \mapsto v]$

Note that in the above example, the method can specify some assumptions over the argument, and distFromOrig method will now use this value of x and not the one in p! This is basically the effect of updating an object and we use this technique to add ":=" to the domain of expressions later.

For a second example, we insist on some default behavior for the argument p, by saying that the default value for distFromOrig method is 0, so that it is always guaranteed that distFromOrig exists:

closerToOrig $\mapsto$ λp.(*self*.distFromOrig <

((p $\boxed{\succ}$ $[\![$ distFromOrig $\mapsto$ 0 $]\!]$)$\bullet$distFromOrig))

Such default behavior is utilized in Chapter 3 to harness subtype information. This facility also illustrates that an effect of extending the class hierarchy chains in the other direction can be achieved (Section 3.7).

2.4.3 Limitations

The above mechanisms are not sufficiently powerful, since an object cannot return it's self, since self with our mechanisms will be a "fixed object", and is not a wrapper. Also updating an instance variable, considered a very common-place happening, cannot be accomplished since this requires returning a variant of self.

We could add updating to the language, even without the ability to update self using the ":=" operator:

$$E \quad ::= \cdots \mid E{\bullet}I := E$$

with the translation of ":=" being:

$$\langle\!\langle E_1{\bullet}I := E_2 \rangle\!\rangle \quad = \quad [\![\ \mathrm{I} \mapsto \langle\!\langle E_2 \rangle\!\rangle\ ]\!]\ \boxed{\succ}\ \langle\!\langle E_1 \rangle\!\rangle$$

Note that the update is not well defined (rather not possible) when E_1 is self. This feature, however, has added expressivity to the language since now, although a class defines a unique object, update allows one to obtain other objects *like* the one defined by class. For example, a point at (2, 4) is $[\![\ \mathrm{x} \mapsto 2\ ]\!]\ \boxed{\succ}\ \mathit{Point}$. Although delegation allows us to redefine arbitrarily method labels in this manner, in a statically-typed context this flexibility makes it hard to analyze the type of an expression. In such cases, it would help to differentiate between instance variables and methods and allow only updating of instance variables [11].

2.5 Templates and inheritance

The class construct of language $\mathcal{C}_1$ (Section 2.2) is restrictive in expressivity since it can only define a unique object instance, and an update facility was looked at in Section 2.4.3 which increased the expressivity by allowing other objects *like* the one defined by the class. In this section we extend the language $\mathcal{C}_1$ with templates and obtain the power of having many different object instances from a class specification. This more general notion of a class is what is available in most commonly available object-oriented languages.

The syntactic domains (for $\mathcal{C}_3$) is an extension to that of $\mathcal{C}_1$ as follows:

$$\begin{aligned} K \quad &::= \textbf{class } I_1(I_2,\ \cdots,\ I_n)\ M \mid K_1\ K_2 \mid \epsilon \\ &\qquad \mid \textbf{class } I_1(I_2,\ \cdots\ I_n)\ \textbf{subclass of } I'(E_1,\ \cdots\ E_{n'})\ M \\ E \quad &::= \cdots \mid \textbf{new } I(E_1, \cdots, E_n) \end{aligned}$$

The translation function is $\langle\!\langle \_ \rangle\!\rangle : \mathcal{C}_3 \rightarrow \lambda$-calculus.

$$\begin{aligned} &\langle\!\langle \textbf{class } I_1(I_2,\ \cdots\ I_n)\ M \rangle\!\rangle \qquad = \quad I_1 = \lambda I_2. \cdots \lambda I_n.\lambda \mathit{self}.\langle\!\langle M \rangle\!\rangle \\ &\langle\!\langle \textbf{class } I_1(I_2,\ \cdots\ I_n)\ \textbf{subclass of } I'(E_1,\ \cdots\ E_{n'})\ M \rangle\!\rangle = \\ &\qquad\qquad I_1 = \lambda I_2. \cdots \lambda I_n.((\lambda \mathit{self}.\lambda \text{super}.\langle\!\langle M \rangle\!\rangle)\ \boxed{\triangleright}\ (I'\langle\!\langle E_1 \rangle\!\rangle \cdots \langle\!\langle E_{n'} \rangle\!\rangle)) \\ &\langle\!\langle \textbf{new } I(E_1, \cdots, E_n) \rangle\!\rangle \qquad\quad = \quad \mathbf{Y}\ (\mathrm{I}\langle\!\langle E_1 \rangle\!\rangle \cdots \langle\!\langle E_n \rangle\!\rangle) \end{aligned}$$

The **new** construct now supports the notion of classes as templates by accomplishing the template instantiation during object creation.

The Point and Circle examples (Section 2.3) could be defined using templates as in Example 2.5. A point at (3, 4) is now created by "**new** Point(3,4)".

2.6 Recursive objects

Informally, recursive objects are ones that return themselves, or those that create different objects that are like themselves[6] as a result of evaluating a

[6]Either by updating self, or by creating other instances by using their own templates.

class Point(a,b)
  meth x = a
  meth y = b
  meth *distFromOrig* = sqrt(**self**•x^2 + **self**•y^2)
  meth *closerToOrig*(p) = (**self**•*distFromOrig* < p•*distFromOrig*)

class Circle(a,b,r) **subclass of** Point(a,b)
  meth radius = r
  meth distFromOrig = (**super**•distFromOrig − **self**•radius) max 0

Example 4: Classes Point and Circle using templates

class MovablePoint(a,b)
    meth x =a
    meth y =b
    meth *movex*(*p*) =**new** MovablePoint(p•x, **self**•y)

class MovableCircle(a,b,r) **subclass of** MovablePoint(a,b)
    meth *radius* =r

Example 5: Classes MovablePoint and MovableCircle

method. In this section, we consider the semantics of this as it is known for inheritance, and also model the desired semantics for delegation by a straight forward use of the semantics developed in Section 2.4. We then explain how the desired semantics can be achieved, in a simple manner.

The need for an additional level of recursion when an object definition itself is recursive is illustrated by Cook [15, page 26]. Consider a class of points with a movex method, that defines MovablePoint (the examples of MovablePoint and MovableCircle are borrowed from [15], but simplified and adopted to our context).

The movex method creates a point just like itself, but with a different x value.

However, the MovablePoint example does not quite illustrate returning the modified self in an object-oriented sense when the effect of inheritance is considered as seen by the MovableCircle example. The inherited movex method in MovableCircle will return a MovablePoint, which is not a modified self, as this results in loss of information (radius in this case).

The recursion of the MovablePoint is opened up further by explicitly defining template as a generator [15]. Assuming that we have such a translation function properly defined, we get:

$$\text{MovablePoint} \longleftarrow \lambda\text{make-self}.\lambda\text{a}.\lambda\text{b}.\lambda \textit{self}.\lambda\text{super}.\begin{bmatrix} \text{x} & \mapsto & a, \\ \text{y} & \mapsto & b, \\ \text{movex} & \mapsto & \lambda p.\text{Y}(\text{make-self}(p.\text{x}, \textit{self}.\text{y})) \end{bmatrix}$$

using which one could define (Y ((Y MovablePoint)(2,3))) as the desired movable point.

And MovableCircle can then be realized by:

$$\text{MovableCircle} \longleftarrow \lambda\text{make-self}.\lambda\text{a}.\lambda\text{b}.\lambda\text{r}.$$

$$[\![\text{ radius} \mapsto \text{r }]\!] \boxed{\succ} \text{MovablePoint}(\lambda\text{a}'.\lambda\text{b}'.\text{make-self}(\text{a}',\text{b}',\text{r}))$$

The notion of returning a modified self, as intended by delegation, has different expressivity (updatability for instance) that should be captured. The above complex construction models returning an instance like self using the template for a statically defined hierarchy of classes (inheritance) and requires a-priori knowledge of the parent (delegatee) template. When we adopt these to the context of delegation as discussed in Section 2.4, we will be able to modify templates dynamically and thus this changed information should be reflected when

class MovablePoint(a,b)
  meth x = a
  meth y = b
  meth $movex(p)$ = x := p•x ; **self**

Example 6: Class MovablePoint with updating

an object is instantiated using "make-self" later.

This calls for appropriately changing the structure of an object (say by using another level of recursion — make-make-self). Now this is a circular argument, since the structure that will be returned by a recursive call, will have an incompatible structure. The circularity in the argument can be explained as follows (in the Smalltalk context):

> First we desired for the ability of modifying an object specification at runtime. Since a class specifies an object, we used classes themselves as "objects" by properly redefining what message passing meant. Then we desired that certain messages be able to return the object itself or a variation of itself, which required us to have a reference to a class. Now this is provided by a metaclass whose recursive structure can be explained by using a metametaclass ... *ad infinitum* as done in [26].

In a language that supports delegation, the need for making a new object like self, is much less than the need for the ability to return itself or an ability to use self as a delegator or a delegatee. The ability to return itself can be achieved by using the message passing mechanism to pass a copy of the receiver as an argument to the methods, and each method will assume that the first argument is itself (denoted by variable bigself below).

Consider the case of movable point example (Example 2.6), by assuming that the language allows updating of self (Page 35). We can read the movex method definition as "update the method x to the new value of p$\bullet$x and return the modified self" — or had we distinguished instance variables and methods of a class we could read this as updating the instance variable x. By defining the $\bullet$ so it passes the receiver of a message as first argument and modifying the class translation appropriately we can specify the MovablePoint and MovableCircle as:

$$\text{MovablePoint(a,b)} \longleftarrow \lambda \textit{self}.\lambda\text{super}.$$
$$\begin{bmatrix} x & \mapsto & \lambda\text{bigself}.a, \\ y & \mapsto & \lambda\text{bigself}.b, \\ movex & \mapsto & \lambda\text{bigself}.\lambda p.\ [\![\ \text{x} \mapsto \text{p}\bullet\text{x}\ ]\!]\ \boxed{\succ}\ \text{bigself} \end{bmatrix}$$

When evaluating a method of an object, say rcvr, bigself will be rcvr while self will be ($\tilde{Y}$ rcvr).

It can be seen that for the above case the effect of movex is to create a movable point equivalent to (MovablePoint(p.x, *self*.y)). This is because, the first argument of MovablePoint becomes the value of the method x, and the second argument remains the same. A value of method x can be changed using delegation to get effectively the same result. The explanation is not meant to imply that one can get the effect of make-self by delegation, as described above, but to give an insight at the difficulty of getting this effect in general.

$$\text{MovableCircle} \longleftarrow \lambda\text{make-self}.\lambda\text{a}.\lambda\text{b}.\lambda\text{r}.$$
$$[\![\ \text{radius} \mapsto \text{r}\ ]\!]\ \boxed{\succ}\ \text{MovablePoint}(\lambda\text{a}'.\lambda\text{b}'.\text{make-self}(\text{a}',\text{b}',\text{r}))$$

In the MovableCircle which is defined like before, the effect of movex is to return a MovableCircle object as desired and not a MovablePoint object. This is achieved without the complex semantic mechanism needed to properly create a new MovableCircle, that we studied earlier. The expressiveness of delegation is clear: "Change the attribute value of the x method to a different value, while keeping the rest like before". The expressiveness of make-self is entirely different: "Create something like myself, but by using the new set of parameters". In this example, it just happens that these do the same thing. Before considering the effects of mixing these two in detail (in the next section), we formulate the idea of passing self as first argument here. This idea has been used by the people who have implemented object-oriented languages [52, 12] as noted in [47, 33].

Although the constructions using the explicit fixpoint operator seemed elegant for the case of inheritance, as we have noted above, it looses its elegancy for delegation. A simpler translation can be obtained for delegation that encompasses the expressivity of the constructions of inheritance. The notion of passing a copy of the receiver is quite powerful and as a result we do not need to distinguish between self and bigself. The translation function can thus be defined by pushing the variable self into the method definition, and eliminating the **Y** combinator:

The syntax of $\mathcal{C}_3$ can be translated instead to:

$$\langle\!\langle \mathbf{class}\ I_1(I_2,\ \cdots\ I_n)\ M \rangle\!\rangle = I_1 = \lambda I_2.\cdots\lambda I_n.\lambda super.\langle\!\langle M \rangle\!\rangle$$

$$\langle\!\langle \mathbf{class}\ I_1(I_2,\ \cdots\ I_n)\ \mathbf{subclass\ of}\ I'(E_1,\ \cdots\ E_{n'})\ M \rangle\!\rangle =$$

$$I_1 = \lambda I_2.\cdots\lambda I_n.((\lambda super.\langle\!\langle M \rangle\!\rangle) \succ I'\langle\!\langle E_1 \rangle\!\rangle \cdots \langle\!\langle E_{n'} \rangle\!\rangle)$$

$$\langle\!\langle \mathbf{meth}\ I(I_1,\cdots,I_n)\ E \rangle\!\rangle = [I \mapsto \lambda self.\lambda I_1.\cdots\lambda I_n.\langle\!\langle E \rangle\!\rangle]$$

$$\langle\!\langle \mathbf{new}\ I(E_1,\cdots,E_n) \rangle\!\rangle = \mathrm{I}\langle\!\langle E_1 \rangle\!\rangle \cdots \langle\!\langle E_n \rangle\!\rangle$$

$$\langle\!\langle E_1\ ;\ E_2 \rangle\!\rangle = \langle\!\langle E_1 \rangle\!\rangle \succ \langle\!\langle E_2 \rangle\!\rangle$$

$$\langle\!\langle E \bullet I(E_1,\cdots,E_n) \rangle\!\rangle = ((\langle\!\langle E \rangle\!\rangle\ \mathrm{Top}).\mathrm{I})\ \langle\!\langle E \rangle\!\rangle\ \langle\!\langle E_1 \rangle\!\rangle \cdots \langle\!\langle E_n \rangle\!\rangle$$

$$\langle\!\langle \mathbf{super}\bullet I(E_1,\cdots,E_n) \rangle\!\rangle = \mathrm{super.I}\ \langle\!\langle E_1 \rangle\!\rangle \cdots \langle\!\langle E_n \rangle\!\rangle$$

The translation of message send again has two cases, now depending on whether E is the variable super or not. This is not restrictive as the case when self had a different structure (Page 27), since "sending a message to super class" is a misnomer and "invoking a super class method" is more appropriate, since sending a message to super would loose the current self (like in C++ and Simula).

We look at the MovablePoint and MovableCircle (Example 2.6) in their entirety (movex method uses delegation) by defining

Definition 2.6.1 $[\![\ x \mapsto v\ ]\!] \longleftarrow \lambda s.[x \mapsto v]$

class Point(a,b)
  meth x = a
  meth y = b
  meth *movex*(p) = x := p$\bullet$x ; **self**
  meth *distFromOrig* = sqrt(**self**$\bullet x^2$ + **self**$\bullet y^2$)
  meth *closerToOrig*(p) = (**self**$\bullet$*distFromOrig* < p$\bullet$*distFromOrig*)

class Circle(a,b,r) **subclass of** Point(a,b)
  meth *radius* = r
  meth *distFromOrig* = max(**super**$\bullet$*distFromOrig* − **self**$\bullet$*radius*, 0)

Example 7: Classes Point and Circle revisited

$$\text{Point(a,b)} \longleftarrow \lambda\text{super}.
\begin{bmatrix}
x & \mapsto & \lambda self.a, \\
y & \mapsto & \lambda self.b, \\
movex & \mapsto & \lambda self.\lambda p.[\![\ \text{x} \mapsto \text{p.x}\ ]\!] \succ self, \\
distFromOrig & \mapsto & \lambda self.sqrt((self \bullet x)^2 + (self \bullet y)^2), \\
closerToOrig & \mapsto & \lambda self.\lambda p.((self \bullet distFromOrig) < (p \bullet distFromOrig))
\end{bmatrix}$$

Circle(a,b,r) $\longleftarrow$

let cdelta $\longleftarrow$ λsuper.

$$\begin{bmatrix} radius & \mapsto & \lambda \mathit{self}.r, \\ distFromOrig & \mapsto & \lambda \mathit{self}.max(super.distFromOrig - (\mathit{self}\bullet radius), 0) \end{bmatrix}$$

in cdelta $\succ$ (Point(a,b))

which yields the desired circle behavior.

The above constructions are a simplified version of [15]. These simple mechanisms are sufficient to describe delegation. These are also sufficient to describe inheritance of non-recursive objects. However, the delegation semantics is more appropriate to model object-oriented languages since incremental changes to self can be specified in a natural manner. A simple example would be a class like Point above with both x and y methods defined initially as value a. In this case the movex method will never be able to achieve its goal without delegation semantics, since using make-self it would not be possible to get the effect of modifying the x component while keeping the rest like before.

2.7 Templates and delegation

The coexistence of delegation with templates (the ability to make-self) is an interesting issue, since they seem to have different goals. The *treaty of Orlando* [56] emphasizes this issue by identifying these (called empathy and template respectively) as distinct features that object-oriented programs must provide. In this section, we illustrate the difference between these using examples and attempt to defend the existence of both.

The distinctive property of templates is their parameters, and the effect of the parameters cannot be achieved just by delegation. Likewise, even for the case when delegation is strict[7], its effect cannot be achieved by use of the same template the object was created from, as can be seen by a simple example:

Dpoint(a) $\longleftarrow$ Point(a,a), defines a template for Points with a constraint that both its x and y attributes be defined identically. None of the methods in Dpoint(a) can violate this constraint by using make-self(_). However, delegation is a facility to violate this constraint as any of its methods can return ($\llbracket$ x $\mapsto$ c $\rrbracket \succ$ self). If a method in this object returns the object make-self(d), the intended object should have the similar history as the object that returned this, and it will very likely not be Dpoint(d).

To incorporate the facility of templates with delegation, we need to be able to reciprocate the alteration of an object's constraints to its template. This can be achieved by considering *instance, template* pairs as first class entities ("objects") in the system. Delegation will then be pairwise with some explicit information regarding the intended resulting templates.

Definition 2.7.1 *The* paired inheritance function $\bowtie$ *applies an instance, template pair to another instance, template pair and returns an instance, template pair. (i.e., Delegation is handled component wise)*

$\bowtie$ *is defined by:*

$$(i_1 \,\rangle\, t_1) \bowtie (i_2 \,\rangle\, t_2) \longleftarrow \lambda\tau_1.\lambda\tau_2.\lambda\gamma_1.\lambda\gamma_2.\begin{pmatrix} i_1 \succ i_2 & \rangle & \lambda map.\lambda arg. \\ & & (t_1(map \circ (\gamma_1(\tau_2(arg))))(\tau_1 arg)) \\ & & \succ (t_2(map \circ (\gamma_2(\tau_1(arg))))(\tau_2 arg)) \end{pmatrix}$$

and: $(i_1 \,\rangle\, t_1) \bowtie' (i_2 \,\rangle\, t_1) \longleftarrow ((i_1 \,\rangle\, t_1) \bowtie (i_2 \,\rangle\, t_1))\ id\ id\ id\ id$

and: $id \longleftarrow \lambda x.x$

[7] "If an object may not gain or loose attributes once it is defined, we call the template strict" [56, page 38].

The ability to create objects using one's template can be added to the domain of expressions (language $\mathcal{C}_4$):

$$E \quad ::= \cdots \mid \textbf{new}\ \textit{self}\ \textbf{class}(E_1, \cdots\ E_n)$$

with the following translation:

$$
\begin{array}{lcl}
\langle\!\langle \textbf{class}\ I_0(I_1,\ \cdots\ I_n)\ M \rangle\!\rangle & = & I_0 = \lambda \text{map}.\lambda(I_1, \cdots I_n).\lambda \textit{super}.\langle\!\langle M \rangle\!\rangle \\
\multicolumn{3}{l}{\langle\!\langle \textbf{class}\ I_0(I_1,\ \cdots\ I_n)\ \textbf{subclass of}\ I'(I_1,\ \cdots\ I_{n-k})\ M \rangle\!\rangle =} \\
\multicolumn{3}{l}{\qquad \text{let}\ \ I_\delta = \langle\!\langle \textbf{class}\ I_0(I_{n-k+1},\ \cdots\ I_n)\ M \rangle\!\rangle} \\
\multicolumn{3}{l}{\qquad \text{in}\ \ \lambda \text{map}.\lambda(I_1, \cdots I_n).} \\
\multicolumn{3}{l}{\qquad\qquad (I_\delta\ (\langle\!\langle \text{map} \rangle\!\rangle \circ (\lambda(I_{n-k+1}, \cdots I_n).(I_1, \cdots \langle\!\langle I_n \rangle\!\rangle)))\ (\langle\!\langle I_{n-k+1} \rangle\!\rangle, \cdots \langle\!\langle I_n \rangle\!\rangle))} \\
\multicolumn{3}{l}{\qquad\qquad \succ (I'\ (\langle\!\langle \text{map} \rangle\!\rangle \circ (\lambda(I_1,\ \cdots\ I_{n-k}).(\langle\!\langle I_1 \rangle\!\rangle, \cdots \langle\!\langle I_n \rangle\!\rangle)))(\langle\!\langle I_1 \rangle\!\rangle, \cdots \langle\!\langle I_{n-k} \rangle\!\rangle))} \\
\langle\!\langle \textbf{meth}\ I(I_1, \cdots, I_n)\ E \rangle\!\rangle & = & [I \mapsto \lambda \textit{self}.\lambda I_1. \cdots \lambda I_n.\langle\!\langle E \rangle\!\rangle] \\
\langle\!\langle \textbf{new}\ I(E_1, \cdots\ E_n) \rangle\!\rangle & = & (\langle\!\langle I(E_1, \cdots\ E_n) \rangle\!\rangle\ _{\rangle}\ \langle\!\langle I \rangle\!\rangle) \\
\langle\!\langle \textbf{new self class}(E_1, \cdots\ E_n) \rangle\!\rangle & = & (\pi_2\ \langle\!\langle self \rangle\!\rangle\ id\ \langle\!\langle (E_1, \cdots\ E_n) \rangle\!\rangle\ _{\rangle}\ \pi_2\ \langle\!\langle self \rangle\!\rangle) \\
\langle\!\langle E_1\ ;\ E_2 \rangle\!\rangle & = & \langle\!\langle E_1 \rangle\!\rangle \bowtie' \langle\!\langle E_2 \rangle\!\rangle \\
\langle\!\langle E\bullet I(E_1, \cdots, E_n) \rangle\!\rangle & = & (\pi_1\ \langle\!\langle E \rangle\!\rangle\ Top).I)\langle\!\langle E \rangle\!\rangle\langle\!\langle E_1 \rangle\!\rangle \cdots\ \langle\!\langle E_n \rangle\!\rangle \\
\langle\!\langle \textbf{super}\bullet I(E_1, \cdots, E_n) \rangle\!\rangle & = & \textit{super}.I\langle\!\langle E_1 \rangle\!\rangle \cdots\ \langle\!\langle E_n \rangle\!\rangle
\end{array}
$$

where π_i is a projection of ith element. Also, note that the argument list is not curried in the templates while it is curried in the method translation as before. A tuple structure of arguments in the templates help define proper mapping of the arguments during inheritance with the general strategy of extending the argument list of the superclass template, in a subclass template, to the right.

The MovablePoint and MovableCircle examples are revisited to observe the effect of delegation and "**new self class**(_)" (or make-self) on the templates.

Another argument, map, is needed to properly initialize the parameters [17]. We can rewrite the examples with some variations (using *movey* that uses the template to create an object like itself):

Definition 2.7.2 $[\![\ m \mapsto v\ ]\!]$ *is an (instance, template) pair defined as:*

$$[\![\ m \mapsto v\ ]\!] \longleftarrow$$
$$(\ \lambda super.[m \ \mapsto\ \lambda \mathrm{self}.v],$$
$$\lambda map.\lambda arg.\lambda super.[m \ \mapsto\ \lambda \mathrm{self}.v])$$

$$\text{MovablePoint} \longleftarrow \lambda\text{map}.\lambda(\text{a,b}).\lambda\text{super}.$$
$$\begin{bmatrix} x & \mapsto & \lambda self.\text{a}, \\ y & \mapsto & \lambda self.\text{b}, \\ movey & \mapsto & \lambda self.\lambda p.\langle\!\langle \textbf{new self class}(\text{a}, p\bullet y)\rangle\!\rangle \end{bmatrix}$$

$$\text{MovableCircleDelta} \longleftarrow \lambda map.\lambda r.\lambda super.$$
$$\begin{bmatrix} radius & \mapsto & \lambda self.\text{r}, \\ changeR & \mapsto & \lambda self.\lambda\text{r}'.\langle\!\langle \textbf{new self class}(\text{r}'))\rangle\!\rangle \end{bmatrix}$$

A MovableCircle can now be defined, with mappings that transform the parameters to appropriate ones for MovableCircleDelta and MovablePoint. Transformations needed are

$\tau_1(\text{a,b,r}) \longleftarrow (\text{r})$,

$\tau_2(\text{a,b,r}) \longleftarrow (\text{a,b})$,

$\gamma_1(\text{a,b}) \longleftarrow \lambda\text{r}.(\text{a,b,r})$ and

$\gamma_2(\text{r}) \longleftarrow \lambda\text{a,b}.(\text{a,b,r})$.

$$\text{MovableCircle} \longleftarrow$$

$$\lambda\text{map}.\lambda\text{arg}.\ (\text{MovableCircleDelta}\ ((\text{map} \circ (\gamma_1(\tau_2(\text{arg}))))\ (\tau_1(\text{arg})))$$
$$\succ (\text{MovablePoint}\ ((\text{map} \circ (\gamma_2(\tau_1(\text{arg}))))\ (\tau_2(\text{arg}))))$$

Or equivalently:

$$\text{MovableCircle} \longleftarrow$$

$$\lambda\text{map}.\lambda(\text{a,b,r}).\ (\text{MovableCircleDelta}\ ((\text{map} \circ (\lambda\text{r}.(\text{a,b,r})))\ \text{r})$$
$$\succ (\text{MovablePoint}\ ((\text{map} \circ (\lambda(\text{a,b}).(\text{a,b,r})))\ (\text{a,b})))\ )$$

Therefore, a generalization of delegation over templates can be achieved by representing recursive objects as an instance and its template pair. This allows for an object's template to be able to have the same "history" (delegation) information as the object. Every object in the system will effectively be a $(\mathit{instance}\ {}_{\rangle}\ \mathit{template})$, where

$$\mathit{instance} \longleftarrow \lambda\text{super}.[\text{m}_i \ \mapsto\ \lambda \mathit{self}.\ \cdots\ ]$$

$$\mathit{template} \longleftarrow \lambda\text{map}.\lambda\text{arg}.\mathit{instance}$$

We consider the MovablePoint and MovableCircle, with both *movex* and *movey*:

$$\text{MovablePoint} \longleftarrow \lambda\text{map}.\lambda(\text{a,b}).\lambda\text{super}.$$
$$\begin{bmatrix} x & \mapsto & \lambda \mathit{self}.\text{a}, \\ y & \mapsto & \lambda \mathit{self}.\text{b}, \\ \mathit{movex} & \mapsto & \lambda \mathit{self}.\lambda p.(\llbracket\ \text{x} \mapsto \text{p}_{\bullet}\text{x}\ \rrbracket \bowtie' \mathit{self}) \\ \mathit{movey} & \mapsto & \lambda \mathit{self}.\lambda p.\langle\!\langle \textbf{new self class}(x, p_{\bullet} y))\rangle\!\rangle \end{bmatrix}$$

$$\text{MovableCircle} \longleftarrow \lambda\text{map}.\lambda(\text{a,b,r}).\lambda\text{super}.$$

$$\begin{bmatrix} x & \mapsto & \lambda \mathit{self}.\mathrm{a}, \\ y & \mapsto & \lambda \mathit{self}.\mathrm{b}, \\ \mathit{radius} & \mapsto & \lambda \mathit{self}.\mathrm{r}, \\ \mathit{changeR} & \mapsto & \lambda \mathit{self}.\lambda \mathrm{r}'.\langle\!\langle \mathbf{new\ self\ class}(a, b, \mathrm{r}') \rangle\!\rangle, \\ \mathit{movex} & \mapsto & \lambda \mathit{self}.\lambda p.([\![\ \mathrm{x} \mapsto \mathrm{p}\bullet\ \mathrm{x}\ ]\!] \bowtie' \mathit{self}), \\ \mathit{movey} & \mapsto & \lambda \mathit{self}.\lambda p.\langle\!\langle \mathbf{new\ self\ class}(x, p\bullet y, \mathrm{r}) \rangle\!\rangle \end{bmatrix}$$

A MovableCircle object will be of form

(MovableCircle(a,b,r) $\rangle$ MovableCircle). Now, an expression of the form:

Let mc1 $\longleftarrow$ (MovableCircle(10,20,30) $\rangle$ MovableCircle)

and mc2 $\longleftarrow$ (MovableCircle(40,50,60) $\rangle$ MovableCircle)

in (((mc1$\bullet$ movex mc2)$\bullet$ changeR 70)$\bullet$ movey mc2)

will yield the movable circle, as desired (with x $\mapsto$ 40, y $\mapsto$ 60 and radius $\mapsto$ 70).

This development is a generalized semantics of inheritance, delegation and recursive templates.

2.8 Related work

The notion of objects being fixed points of mutually recursive functions over self was explained by Cardelli [9]. The notion of inheritance was made clear by Reddy [51] and Cook [15] who showed that inheritance required the redefinition of self, and also explained the meaning of super to refer to a parent's method. Mitchell [47], however, could not use these mechanisms to explain the updating of instance variables. He noticed that, considering objects as fixed points a-priori is too strong and does not allow for any modifications whatsoever. Instead he preferred the implementation oriented view of using the message passing mechanism to send the receiver of a message as the first argument of the method, thereby obtaining the desired recursion. Bruce [6], Kamin and Reddy [34] have shown that these two different views of the object are equivalent.

Mitchell's model is extended here by providing for super and also providing for the notion of templates. The notion of templates (or classes) adds an additional level of recursion [17] in Cook's semantics but extending classes in an analogous manner of extending objects is not possible. Stein [55] has shown that accommodating templates as first class objects (classes — like in SmallTalk), will blur the difference between delegation and inheritance, and also makes delegation a special case of inheritance and has thereby demonstrated the power of classes as first class objects (and the notion of metaclass) in SmallTalk. Stein, Lieberman and Ungar [56] have later agreed that some form of delegation and templates are two necessary facilities that an object-oriented language must provide, since these model orthogonal issues.

In this chapter, we have differentiated inheritance as a facility to extend or modify behavior of a template, while delegation is a facility to extend or modify the behavior of an object. This notion corresponds with languages claiming inheritance as their fundamental mechanism [58, 43, 53] and those that claim delegation as their fundamental mechanism [41, 59]. Languages like SmallTalk [26] have provision for both inheritance and delegation in this sense, as noted by [55]. The mechanisms presented in this chapter provide for SmallTalk like facilities, as desired by Cook, with updating of instance variables as desired by Mitchell.

2.8.1 Cook's and Mitchell's semantics

The semantic mechanisms for Inheritance, of Cook, as in [17] and those for delegation, of Mitchell, as in [47] are two special cases of the mechanisms given in the previous section.

We have generalized Mitchell's semantics, in Section 2.6, by providing for the notion of *super* and in the previous section by providing for templates. Thus by not using *super* and templates, our mechanisms specialize to Mitchell's.

The following observations can be made about Cook's semantics[8]. The only means of modifying an object (that might use "**new** myclass" which is analogous to our *make-self*) is by accessing its template generator. A template is a fixed point of its generator and an instance of a template is defined as a fixed point of its template applied to some arguments. Objects can therefore be extended, only by accessing the template generator which cannot exist as a first class entity in the system. As also noted by Kim Bruce [6], "myclass" in Cook's semantics cannot be extended and therefore Cook's semantics cannot model the class notion of SmallTalk, but does model the class notion of languages like Eiffel where the classes are only available as "object generating things".

To reduce our mechanisms to what is modeled by Cook's mechanisms, we need to disallow the use of " ; ". Thus, the only means of modifying a template is by using the **subclass of** construct. The **subclass of** construct is a special case of the $\bowtie$ operator. With this reduction, we can use the arguments for equivalence as given by [34], who compares Reddy's mechanisms [51] (which uses fixed point operators) and Kamin's [33] mechanisms which in turn are those used by Cook and Mitchell, respectively.

[8]By saying "Cook's semantics", we are referring to the section on class inheritance of [17] — which also involves the modification of recursive templates.

SmallTalk and *SELF* have many capabilities that cannot be modeled by the mechanisms provided by Cook. Our mechanisms are general enough to model these, with slight variations of $\bowtie$ and by forgetting the effect of aliasing of variable names ("side-effect" aspects) since we are modeling in a functional programming context.

CHAPTER 3

FLEXIBLE SUBTYPING AND INHERITANCE

3.1 Introduction

The inheritance mechanisms of SmallTalk [26] and its derivatives can be considered as rigid since class specification can only be extended in one direction. This rigidness is relaxed in this chapter in order to propose an alternative way of perceiving the problems of identifying subtype hierarchies with subclass hierarchies, and the need for specializing entire hierarchies as the system evolves. A controlled use of delegation provides the needed expressivity and flexibility, while maintaining the desirable properties of inheritance found in class based languages. One the results of using delegation in an inheritance based language is to accommodate some of the useful "inherited types" [17] as subtypes (Chapter 6).

Subtype definition is usually restricted to the contravariance of method argument types [9] as explained in Chapter 1. Practical limitations of such a subtype definition were discussed in Section 1.3 and are illustrated by Eiffel's [43] decision to allow the flexibility of defining inherited types as subtypes, at the risk of complicating the language [44] in order to avoid the resulting type insecurities [16].

In this chapter, we present a flexible message passing mechanism that allows an accommodation of covariance of method argument types (Section 1.3), when desired. This facility will also illustrate when covariance might be desired, and its peculiarities. The notion of delegation is extended by defining an associative delegation operator, which provides the ability to use any object as a delegator or a delegatee. This operator is used to define inheritance hierarchies, which are extendible in both directions to a certain extent. A simple language is defined using these operators to help understand the utility of these mechanisms.

In traditional OOPLs, the objects are fixed at creation time in the sense that the messages that the objects understand during their lifetime are the ones they can understand immediately after their creation. The perspective suggested here is that objects during their lifetime can gain additional methods in certain contexts, over the ones that they were born with. An object specification details only the minimal number of methods that the object must understand, and allows an object to acquire knowledge of other methods in certain contexts. This context sensitive method acquisition by objects can be understood as "the adaptation of the object to a context", a special case of which we recognize as "the context adopting an object".

Section 3.2 introduces a simple language that is modeled by λ-calculus. The following sections discuss some solutions to the problems mentioned above, and identify some of their peculiarities: In Section 3.4, the subtype rules are extended for a couple of examples to allow covariance for some of their method types. Sections 3.5 and 3.6, fortify Section 3.4 by using type information to resolve ambiguity. Section 3.7 discusses the issue of extending a type hierarchy without modifying its code. Section 3.8 uses this flexible message passing mechanism to let the user harness subtype information.

3.2 A small language

In the familiar semantics of inheritance [15] [51], an object is literally "fixed" during its creation by taking the fixed-point of its generator (Section 2.3). Our semantics is an extension of Mitchell's [47], where the necessary recursion is obtained by using a message passing mechanism (called messager) which passes the receiver of the message as the first argument to the method (as in Section 2.6). This can be described as a form of delegation [41] [59], as indicated by Mitchell.

In our generalized notion of delegation (as noted in Section 2.4), any object can delegate to any other object. Therefore, when an object is passed as an argument of a message, it can delegate to a specified object in that context. This can be achieved by manipulating the message passing mechanism. The message passing mechanism, which applies the appropriate method of the object (that receives the message) to the arguments, is usually a simple record field selection [9], but in our case this action is manipulable by the user to a certain extent and in a prescribed manner.

A small object-oriented language that defines the basic characteristics of a typical object-oriented language (similar to [43, 58]) is described. This language is a functional language with updatable instance variables (but no aliasing) and some predefined types like Booleans and Natural numbers. The objective for defining the language is to discuss the utility of certain flexible features and the resulting type checking problems, that typically arise in object oriented languages due to bounded and F-bounded quantification of the type expressions [7]. The language is flexibly defined so that certain techniques can be discussed that cannot be modeled by some of the accepted semantics [15, 51] that attempted to describe [26] and are usually not permitted [43, 58]. However, the language does not provide the templates facility (classes as first class objects) that was discussed in Section 2.7.

The denotational semantics is expressed using λ-calculus as in Chapter 2. Later (Chapter 5) we specify and develop the initial algebra semantics of such a language. Certain modifications to the language are discussed and considered (in Chapter 6), thereby resulting in an object oriented language with familiar semantics which facilitates both type-checking and adoption (Chapter 6).

3.2.1 Syntax of SOOL

The primitive syntactic domains are as follows:

Ide	the domain of identifiers	I
Bas	the domain of constants	B
BOP	the domain of operators	O

Ide, in our case is partitioned into variable identifiers and labels to avoid capture! We use $\bar{I}$to explicitly indicate that it is a label, and when there is no confusion we will not use the over bar. Also, "self" is used as a reserved identifier in SOOL.

The compound syntactic domains are as follows:

Pro	the domain of Programs	P
Exp	the domain of expressions	E
CExp	the domain of compound expressions	C
Var	the domain of variable declarations	V
Cla	the domain of class declarations	K
Meth	the domain of method declarations	M

The syntactic clauses:

$P \quad ::= K\ E$

$K \quad ::= \textbf{class}\ I\ V\ M \mid \textbf{class}\ I\ \textbf{subclass of}\ I_1\ V\ M \mid K_1\ K_2 \mid \epsilon$

$C \quad ::= E \mid C^A$

$$C^A \ ::= I := E \mid C_1^A \ ; C_2^A \mid \ \textbf{if } E \textbf{ then } C_1^A \textbf{ else } C_2^A$$

$$E \ ::= B \mid I \mid E\bullet I(E_1, \cdots, E_n) \mid E\bullet I \mid \textbf{new } I \mid E_1 \ O \ E_2 \mid O \ E \mid (E) \mid \textbf{self}$$

$$\mid \ \textbf{if } E \textbf{ then } E_1 \textbf{ else } E_2 \ \mid E\bullet I := E_1 \mid \textbf{super} \ \ I(E_1, \cdots, E_n)$$

$$V \ ::= \textbf{var } I := E \mid V_1 \ V_2 \mid \epsilon$$

$$M \ ::= \textbf{meth } I(I_1, \cdots, I_n) \ \ C \mid M_1 \ M_2 \mid \epsilon$$

Input and Output are not considered. Note that a keyword **super** is provided as part of the syntax, to suggest that the semantics of invoking the method of super is distinct from sending a message to an object (including self).

3.2.1.1 Syntax of λ-calculus

The semantics of the constructs are given by a translation to λ-calculus. Identifiers (Ide) used in SOOL syntax can be considered as disjoint union of the set of labels and variables used in the lambda calculus, there by distinguishing the two different uses of the identifiers.

For	the domain of λ-calculus formulas	F
VIde	the domain of variables	I
Lab	the domain of labels	$\bar{I}$

The syntactic clauses of the formulas are:

$$F \ \ ::= \ \ \bar{I} \mid I \mid B \mid O \ F \mid F_1 \ O \ F_2 \mid F_1 \ F_2 \mid \underline{\text{if}} \ F_1 \ \underline{\text{then}} \ F_2 \ \underline{\text{else}} \ F_3$$

$$\mid \lambda I.F \mid \underline{\text{let}} \ I = F_1 \ \underline{\text{in}} \ F_2 \mid (F)$$

Although, we used the domain of records in Chapter 2 to help define the meaning of objects, here we build the effect of records and the left-preferential operation using the auxiliary functions in Definition 3.2.1, since this gives us a uniform use of the composition (∘) operator to define **class**, **meth**, **var** and ; features of SOOL. Also, we assume that binary operations have a higher priority than application and also use parenthesis for disambiguation.

We use the following auxiliary function definitions:

Definition 3.2.1 $obj(\bar{I},v) \longleftarrow \lambda super.\lambda I'.\ \underline{if}\ \bar{I} = I'\ \underline{then}\ v\ \underline{else}\ super\ I'$

$Top \longleftarrow \lambda I.\bot$

$x \circ y \longleftarrow \lambda z.x\ (y\ z)$

$x \ll \bar{I} \longleftarrow (x\ Top)\ \bar{I}\ x$

$obj(\bar{I},v)$ is the basic construction in the translation function that maps a given label to a value. It also prescribes a search strategy for selection, by forwarding the argument to super for any other labels. We can think of these as partially defined records. *Top* defines an everywhere undefined mapping for all labels, and when $obj(\bar{I},v)$ is applied to *Top*, we get a record (Definition 2.2.1). The composition ($\circ$) serves as the record building operator, analogous to the *left-preferential combinator.* $x \ll \bar{I}$ gives the effect of record field selection, by capping off the open endedness of the partial records and then applying the resulting record to the identifier.

3.2.2 The translation function

The meaning function $\langle\!\langle \_ \rangle\!\rangle$: SOOL $\rightarrow$ λ-calculus.

$$
\begin{array}{lcl}
\langle\!\langle K_1, \cdots, K_n\ E \rangle\!\rangle & = & \underline{\text{let}}\ \langle\!\langle K_1 \rangle\!\rangle\ \underline{\text{in}} \cdots \underline{\text{let}}\ \langle\!\langle K_n \rangle\!\rangle\ \underline{\text{in}}\ \langle\!\langle E \rangle\!\rangle \\
\langle\!\langle \textbf{class}\ I_1\ V\ M \rangle\!\rangle & = & I_1 = \langle\!\langle V\ M \rangle\!\rangle \\
\langle\!\langle \textbf{class}\ I_1\ \textbf{subclass of}\ I_2\ V\ M \rangle\!\rangle & = & I_1 = \langle\!\langle V\ M \rangle\!\rangle \circ \langle\!\langle new\ I_2 \rangle\!\rangle \\
\langle\!\langle V_1, \cdots, V_n\ M \rangle\!\rangle & = & \langle\!\langle V_1 \rangle\!\rangle \circ \cdots \circ \langle\!\langle V_n \rangle\!\rangle \circ \langle\!\langle M \rangle\!\rangle \\
\langle\!\langle M_1, \cdots, M_n \rangle\!\rangle & = & \langle\!\langle M_1 \rangle\!\rangle \circ \cdots \circ \langle\!\langle M_n \rangle\!\rangle \\
\langle\!\langle C_1^A; \cdots; C_n^A \rangle\!\rangle & = & \langle\!\langle C_n^A \rangle\!\rangle \circ \cdots \circ \langle\!\langle C_1^A \rangle\!\rangle \\
\langle\!\langle I := E \rangle\!\rangle & = & \text{obj}(\bar{I}, \langle\!\langle E \rangle\!\rangle) \\
\langle\!\langle \textbf{var}\ I := E \rangle\!\rangle & = & \text{obj}(\bar{I}, \langle\!\langle E \rangle\!\rangle)
\end{array}
$$

$$
\begin{array}{lcl}
\langle\!\langle \mathbf{meth}\ I(I_1,\cdots,I_n)\ E \rangle\!\rangle & = & \mathrm{obj}(\bar{I},\lambda self.\lambda I_1.\cdots\lambda I_n.\langle\!\langle E \rangle\!\rangle) \\
\langle\!\langle \mathbf{meth}\ I(I_1,\cdots,I_n)\ C^A \rangle\!\rangle & = & \mathrm{obj}(\bar{I},\lambda self.\lambda I_1.\cdots\lambda I_n.\langle\!\langle C^A \rangle\!\rangle \circ self) \\
\langle\!\langle \mathbf{if}\ E\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2 \rangle\!\rangle & = & \underline{\mathrm{if}}\ \langle\!\langle E \rangle\!\rangle\ \underline{\mathrm{then}}\ \langle\!\langle E_1 \rangle\!\rangle\ \underline{\mathrm{else}}\ \langle\!\langle E_2 \rangle\!\rangle \\
\langle\!\langle \mathbf{if}\ E\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \rangle\!\rangle & = & \underline{\mathrm{if}}\ \langle\!\langle E \rangle\!\rangle\ \underline{\mathrm{then}}\ \langle\!\langle C_1 \rangle\!\rangle\ \underline{\mathrm{else}}\ \langle\!\langle C_2 \rangle\!\rangle \\
\langle\!\langle \mathbf{new}\ I \rangle\!\rangle & = & \mathrm{I} \\
\langle\!\langle O\ E \rangle\!\rangle & = & \langle\!\langle O \rangle\!\rangle\ \langle\!\langle E \rangle\!\rangle \\
\langle\!\langle E_1\ O\ E_2 \rangle\!\rangle & = & \langle\!\langle E_1 \rangle\!\rangle\ \langle\!\langle O \rangle\!\rangle\ \langle\!\langle E_2 \rangle\!\rangle \\
\langle\!\langle E\bullet I \rangle\!\rangle & = & \langle\!\langle E \rangle\!\rangle\ \mathrm{Top}\ \bar{I} \\
\langle\!\langle E\bullet I(E_1,\cdots,E_n) \rangle\!\rangle & = & \langle\!\langle E \rangle\!\rangle \ll \bar{I}\ \langle\!\langle E_1 \rangle\!\rangle \cdots \langle\!\langle E_n \rangle\!\rangle \\
\langle\!\langle E\bullet I := E_1 \rangle\!\rangle & = & \mathrm{obj}(\bar{I},\langle\!\langle E_1 \rangle\!\rangle) \circ \langle\!\langle E \rangle\!\rangle \\
\langle\!\langle (E) \rangle\!\rangle & = & \langle\!\langle E \rangle\!\rangle \\
\langle\!\langle \mathbf{super}\ \ I(E_1,\cdots,E_n) \rangle\!\rangle & = & \mathrm{super}\ \bar{I}\ \mathrm{self}\ \langle\!\langle E_1 \rangle\!\rangle \cdots \langle\!\langle E_n \rangle\!\rangle \\
\langle\!\langle \mathbf{self} \rangle\!\rangle & = & \mathrm{self}
\end{array}
$$

The translation of primitives (constants and binary operations) is not listed here.

3.2.3 An example

class point
   var $x := 0$ – – initially
   var $y := 0$
   meth *eq*(apt) (**self**•x = apt•x) & (**self**•y = apt•y)
   meth *move*(i,j) (x := (**self**•x + i)) ;
                        (y := (**self**•y + j))
class colorpoint **subclass of** point
   var c := red
   meth *cmove*(i,j,k) (c := k) ; (**super** *move*(i,j))
(**new** point)•*move(2,3)*

Example 8: Colorpoint subclass of point

A class of Cartesian points (**class** point) with methods x, y, eq and *move* can be defined as in Example 3.2.3. Methods x and y return a constant value. The message passing operator "$\bullet$" is used as a default. The eq method compares two points by comparing the x and y components of itself (**self**) and its argument. To elaborate, ($\mathbf{self}{\bullet}x = \mathrm{apt}{\bullet}x$), compares the value returned by ($\mathbf{self}{\bullet}x$) with that of apt's x value ($\mathrm{apt}{\bullet}x$), and "&" is the *Boolean* "and" operation. The *move* method specifies a replacement object that is a modification of *self* by providing replacement behavior [41] for x and y methods. The replacement behavior specifies a replacement method x defined as the "old x value" + i. The $m := v$, operation creates a simple object that returns v for message m, while delegating those not recognized (i.e., all others) to the object that follows the ";", and the $o_1 \; ; \; o_2$ operation creates an object that delegates the messages not recognized by o_1 to o_2. The operators := (create simple object) and ; (delegation) provide the expressiveness needed to specify the *move* method in a natural way (with the use of *self* in the translation of methods with ";"). The semantics of these operators and the language constructs are as given with **self** and **super** in the method body replaced by free variables self and super respectively.

The **class** colorpoint inherits the **var** and **meth** information from **class** point. Note that invoking a method of self requires a $\bullet$ while invoking a method of super is by application (**meth** *cmove*).

The expression following the class definitions is intended to express what the program does, by starting off a chain of method invocations. In this example, a point object is created and moved to location (2, 3), briefly:

$\langle\!\langle \mathbf{new}\ \mathit{point} \rangle\!\rangle = \lambda \mathrm{super}.\lambda I.$

$\quad \underline{\mathrm{if}}\ I = x\ \underline{\mathrm{then}}\ 0\ \underline{\mathrm{else}}$

$\underline{\text{if}}\ I = y\ \underline{\text{then}}\ 0\ \underline{\text{else}}$

$\underline{\text{if}}\ I = eq\ \underline{\text{then}}\ \lambda\text{self}.\lambda\text{apt}.((\text{self}\ll x = \text{apt}\ll x)\ \&\ (\text{self}\ll y = \text{apt}\ll y))\ \underline{\text{else}}$

$\underline{\text{if}}\ I = move\ \underline{\text{then}}\ \lambda\text{self}.\lambda\text{i}.\lambda\text{j}.(\underline{\text{let}}\ \text{v1} = \text{self}\ll x + \text{i}\ \underline{\text{in}}$

$\qquad\underline{\text{let}}\ \text{v2} = \text{self}\ll y + \text{j}\ \underline{\text{in}}$

$\qquad(\lambda\text{super}.\lambda I.\underline{\text{if}}\ I = x\ \underline{\text{then}}\ \text{v1}\ \underline{\text{else}}$

$\qquad\qquad\underline{\text{if}}\ I = y\ \underline{\text{then}}\ \text{v2}\ \underline{\text{else}}\ \text{super}\ I) \circ \text{self})$

$\underline{\text{else}}\ \text{super}\ I$

Then the message *move* translates to:

$$\langle\!\langle(\mathbf{new}\ point)\bullet\ move(2,3)\rangle\!\rangle = \langle\!\langle(\mathbf{new}\ point)\rangle\!\rangle \ll\ move\ \langle\!\langle 2\rangle\!\rangle\ \langle\!\langle 3\rangle\!\rangle$$

$$\langle\!\langle(\mathbf{new}\ point)\rangle\!\rangle \ll\ move = \langle\!\langle(\mathbf{new}\ point)\rangle\!\rangle\ Top\ move\ \langle\!\langle(\mathbf{new}\ point)\rangle\!\rangle$$

The $\langle\!\langle(\mathbf{new}\ point)\rangle\!\rangle Top\ move$ is the method move desired that gets the actual arguments $\langle\!\langle(\mathbf{new}\ point)\rangle\!\rangle$, $\langle\!\langle 2\rangle\!\rangle$ and $\langle\!\langle 3\rangle\!\rangle$ for *self*, i and j respectively.

In this language, we do not have instance variables that affect the state of the objects (no side-affects), but do have a controlled form of delegation that enables a method to return modified self by providing a replacement behavior for instance variables [47]. Also we do not provide for the ability to recursively specify a class, in order to keep the semantics simple. All objects in this system have the same form, including those that are formed using delegation. In Cook's mechanism [15], the form of an object is not the same as that of an object modifier (wrapper) and this results in $(b\ ;\ c)$ having a different form than b, but the same form as c. In our mechanisms, the operation ; is composable (if $(a\ ;\ (b\ ;\ c))$ is well formed, then so is $((a\ ;\ b)\ ;\ c))$ and the constructions in this chapter use this fact[1].

[1]These also result in equivalent formulas.

3.3 Type information

The type of an object is given by a signature of its methods (Section 1.3). Type A is said to be a subtype ($\leq$) of type B, if A has all the methods B has and for each method of B the corresponding method of A conforms ($\leq'$). The method conformance is defined by the contravariance of the arguments and the covariance of the results [9]:

$$f : C \rightarrow D \ \leq' \ f : C' \rightarrow D' \quad \Longleftrightarrow \quad C' \leq C \ \wedge \ D \leq D'$$

We sometimes abbreviate "covariance of argument types of the methods" to just covariance and similarly contravariance, as this results in no confusion.

Types could be specified separately. In this chapter we discuss how a type checking system could help assist in achieving the flexibility we propose, but we do not confine ourselves to specifying any particular type system. One such system is presented completely in Chapter 6.

The type of the point class could be specified by defining the signatures of the methods of type *Point* and declaring that the class point is of type *Point* as follows:

Type *Point*

x: $\rightarrow$ *Number*

y: $\rightarrow$ *Number*

eq: *Point* $\rightarrow$ *Boolean*

move: *Number*, *Number* $\rightarrow$ *Point*

point : *Point*

Similarly, colorpoint can be given a type declaration:

Type *ColorPoint*

c: $\rightarrow$ *Color*

x: $\rightarrow$ *Number*

y: $\rightarrow$ *Number*

eq: *Point* $\rightarrow$ *Boolean*

move: *Number*, *Number* $\rightarrow$ *Point*

cmove: *Number*, *Number*, *Color* $\rightarrow$ *ColorPoint*

colorpoint : *ColorPoint*

Although it is clear from the type signatures that *ColorPoint* is a subtype of *Point* (Section 1.3), we require that the user make the declaration that *ColorPoint* type objects may be substituted where *Point* type objects were expected by a declaration of the form:

ColorPoint **subtype of** *Point*

This kind of explicit information can be useful to think of alternate semantics when a type signature does not conform according to the subtype rules. A subtype hierarchy is created by explicitly specifying the **subtype of** relations. The peculiarities and the flexibility of the language is illustrated by several examples in the following sections.

3.4 Flexible subtyping

First we consider the problem of "covariance or contravariance?" for method arguments. The language Eiffel [43] requires covariance, while Trellis [53] and several others, require contravariance. It is well understood that contravariance is safe, while covariance is desirable in practice for certain tasks. Languages like CLOS [36] provide a compromise by multiple method dispatching wherein methods

class line: *Line*

var p := new point

var q := new point

meth*samelen*(aline) ...compute Manhattan

lengths of self and aline, and compare ...

Example 9: Class line, uses a "Manhattan length" definition

are selected by considering the types of all the arguments and not just the method name. In this section, we use delegation to give one interpretation for covariance of method arguments. Consider a line class[2] (Example 3.4) of type *Line*, which uses the point defined earlier, with *Line* defined as:

Type *Line*

p: $\rightarrow$ *Point*

q: $\rightarrow$ *Point*

samelen: *Line* $\rightarrow$ *Boolean*

Lets say, that the Manhattan length [51] is not the desired semantics for computing the length of a line object (line object is an instance of class line). An Iowan attempts to fix this problem by defining a subclass of line class by computing the length, using the Iowa definition. Besides, the Iowan has a need to add an additional method *length* in the subclass and therefore it is natural to use the *length* method to redefine *samelen* in the subclass linelen (Example 3.4).

The type *LineLen* is defined as:

[2]Although not SOOL syntax, we abbreviate by mixing type declaration with class definition.

class linelen: *LineLen* **subclass of** line

meth*length* ...compute Iowa length using p and q ...

meth *samelen*(aline) (**self**•*length*) = (aline•*length*)

Example 10: Class linelen, defines length and redefines samelen

Type *LineLen*

p: $\rightarrow$ *Point*

q: $\rightarrow$ *Point*

length: $\rightarrow$ *Real*

samelen: *LineLen* $\rightarrow$ *Boolean*

Notice that, since the *samelen* method in class linelen sends a *length* message to its argument aline, it cannot accept objects that do not understand the *length* message. An object of linelen class cannot be substituted for an object of line class and hence the type *LineLen* is not a subtype of *Line*. Eiffel's solution is to grant *LineLen* a "subtype" status, but with a promise to catch culprits that try to send the *samelen* message to an object of the linelen class with an argument object of line class. Such an approach effectively enforces contravariance during use time rather than during definition time. Trellis and other languages admit that they are not flexible enough to handle *LineLen* as a subtype of *Line*. CLOS looks at all the arguments during runtime to select either line's *samelen* method or linelen's *samelen* method, with preference given to linelen's method. This is the most desirable of all the above solutions, since it does not lead to type insecurities, but the resulting behavior for our example is undesirable (results in the same behavior the subclass is trying to fix!).

If the role of the linelen class was to replace the line class in the entire system, then this would be no problem (covariance is acceptable for the method *samelen*). All the line class objects would be linelen class objects and therefore there is no chance of a line object being passed as an argument to *samelen* method. However, this is not the intention of linelen class in object-oriented programming, since the linelen class specification is an intention of the Iowan to coexist with others. One can instead desire that the line object arguments to the method *samelen*, in linelen class, be specialized in this context. A way of achieving this is by providing a facility for the argument to delegate the messages that the argument does not understand back to the sender, and with this as the intended semantics, type *LineLen* can be used as a subtype of *Line*.

The type system can extend the language SOOL in order to use delegation in a more general manner.

$$E \quad ::= \cdots \mid E_1 ; E_2$$

with ; defined as delegation:

$$\langle\!\langle E_1 \ ; \ E_2 \rangle\!\rangle \qquad = \ \langle\!\langle E_1 \rangle\!\rangle \circ \langle\!\langle E_2 \rangle\!\rangle$$

The message passing operator, $\bullet$, which is defined as sending a message (mesg) to an object (rcvr), is defined as first selecting the appropriate method and then applying it to the arguments: rcvr$\bullet$mesg arg $\longleftarrow$ rcvr$\ll$ mesg arg

This definition could be modified using the type system to provide alternate semantics for the case of *Line* and *LineLen*, so that it behaves differently for the *samelen* message by specifying the intention for this special case (**with** gives additional information):

LineLen **subtype of** *Line*, **with**

$$\text{rcvr}\bullet \mathit{samelen}\ \text{arg} \longleftarrow \text{rcvr} \ll \mathit{samelen}\ (\text{arg}\ ;\ \text{rcvr})$$

Sending the message *samelen* to rcvr, will now execute the method *samelen* with argument (arg ; rcvr). (arg ; rcvr) specifies an object that delegates the messages not understood by arg to rcvr. Also, note that this form of delegation is only possible since our delegation operator is associative.

We can identify the class of "inherited types" [17], as subtypes that fall into this category where the message can be interpreted by the receiver if the message is not understood by the argument (if the user so desires). Knowing that the message samelen is interpreted in this manner, we can safely accept *LineLen* to be a subtype of *Line*, as the argument to the method *samelen* is always guaranteed to have the *length* method defined.

Peculiarities: Note that if all the messages sent to arg are understood by arg, then sending the messages to (arg ; rcvr) would have the same semantics as just sending the message to arg. However, if the method returned its argument, either directly or by some modification to it, the returned object would be the specialized one. A more elaborate system can circumvent this problem, although in general this may not be a discomfort.

The following example illustrates a need for a new perspective, which is to view covariance as a desire to specialize the objects of superclass. Consider the point and colorpoint classes defined earlier. If the *eq* method in colorpoint is modified to check the equality of x, y and c attributes, then the signature of *eq* method in *ColorPoint* type must be modified to: *eq*: ColorPoint $\rightarrow$ *Boolean*

The specialization of a point in a particular context can use the color in that context by defining:

ColorPoint **subtype of** *Point*, **with**

$$\text{rcvr}\bullet eq\ \text{arg} \longleftarrow \text{rcvr} \ll eq\ (\text{arg}\ ;\ \text{rcvr})$$

Now, let P1 be a point object, CP1 and CP2 be colorpoint objects with the same coordinates as P1, then the following scenario is possible:

(CP1•*eq* (CP2))	evaluates to false
(CP1•*eq* (P1))	evaluates to true
(CP2•*eq* (P1))	evaluates to true

The justification for this is simple. Since the color in the point object is not yet defined, it can get defined differently in different contexts.

3.5 Resolving ambiguity

The solution described above needs some additional fortification. In general, since we have a hierarchy of types, we might have additional subtypes of *Line* other than *LineLen* which might provide a length message that might be incompatible or even undesirable to be evaluated in the context of the *samelen* method. This problem can be solved by uniquely naming the first definition of a method (when it is introduced, not redefined) in an inheritance chain that corresponds to a subtype chain.

The classes in an inheritance chain have with them an associated type[3]. We qualify every method in a class by its type name. However, if the method is redefined in a subclass which has a type that is declared as subtype of the type of the class where the method was first defined, we keep the same qualification as before.

For example, let *LineBug* **subtype of** *Line*, be defined in addition to *LineLen* **subtype of** *Line*. Assume that the class linebug : *LineBug* also defines a length

[3]This need not mean that there exists a subtype chain corresponding to inheritance chain.

method, but incompatible with the length method in class linelen : *LineLen*.

To understand the use of such qualification of method names with type names, let us consider L and LL to be objects with types *Line* and *LineLen* respectively with the message *samelen* being sent to LL with argument L, (LL•*samelen* L). If L is a line object, it will be specialized to the context of LL, as indicated by the message passing mechanism. And, if L is a linelen object, L's own *length* method is used. When L is a linebug object, the length method defined in L is not used since the tagged *length* method in L is LineBug·length method, whereas *samelen* sends a LineLen·length method to L. Thus, in all these cases, the *length* method of linelen class is interpreted as desired.

The resulting class definitions of line and linelen, with the above type declarations will be:

class line: *Line*

  var Line·p := new point

  var Line·q := new point

  meth Line·samelen(aline : *Line*) ...compute lengths of self and
      aline, and compare...

class linelen: *LineLen* **subclass of** line

  meth LineLen·length ...compute distance between Line·p and Line·q...

  meth Line·samelen(aline: *LineLen*) (**self**•LineLen·length)
      = (aline•LineLen·length)

and the effective message passing for *samelen* will be

$$\text{rcvr}\bullet\text{Line}\cdot\text{samelen arg} \longleftarrow \text{rcvr} \ll \text{Line}\cdot\text{samelen (arg ; rcvr)}$$

class int: *Integer*
  var *nume* := 0
  meth *prod*(aN) (*nume* := ((**self**•*nume*) ∗ (aN•*nume*))) ; **self**
  meth *eq*(aN) (**self**•*nume*) = (aN•*nume*)

class fract: *Fraction* **subclass of** int
  var *denom* := 1
  meth *prod*(aN) (*denom* := (**self**•*denom*) ∗ (aN•*denom*))
    ; (**super** *prod*(aN))
  meth *eq*(aN) ((**self**•*denom*) = (aN ≪ *denom*)) & (**super** *eq*(aN))

Example 11: Fractions specialize Integers

3.6 More on flexible subtyping

Sometimes it is not desirable to just ask the receiver to interpret a message that is not understood by the argument. Some default behavior might be more appropriate, as is depicted example 3.6.

The *Integer* and *Fraction* types can be defined as:

Type *Integer*

$nume$: $\rightarrow IntBase$ – – assume $IntBase$

eq: $Integer \rightarrow Boolean$

$prod$: $Integer \rightarrow Integer$

Type *Fraction*

$nume$: $\rightarrow IntBase$

$denom$: $\rightarrow IntBase$

eq: $Fraction \rightarrow Boolean$

$prod$: $Fraction \rightarrow Fraction$

If the intention is to assert that *Fraction* is a subtype of *Integer*[4], a default

[4]Leavens [40] constructs Integer as a subtype of Fraction. However, since Integer forgets attributes of Fraction, the subtype consideration in our object-oriented

interpretation for the *denom* message should be provided for the arguments of *eq* and *prod*. This can be done by considering the default *denom* value to be 1. The message interpretation can be defined as:

Fraction **subtype of** *Integer*, **with**

$\text{rcvr}_\bullet \textit{prod}\ \text{arg} \longleftarrow \text{rcvr} \ll \textit{prod}\ (\text{arg}\ ;\ (\text{denom} := 1))$

$\text{rcvr}_\bullet \textit{eq}\ \text{arg} \longleftarrow \text{rcvr} \ll \textit{eq}\ (\text{arg}\ ;\ (\text{denom} := 1))$

This translates to:

$\text{rcvr}_\bullet \text{Integer}\cdot\text{prod arg} \longleftarrow \text{rcvr} \ll \text{Integer}\cdot\text{prod}\ (\text{arg}\ ;\ (\text{Fraction}\cdot\text{denom} := 1))$

$\text{rcvr}_\bullet \text{Integer}\cdot\text{eq arg} \longleftarrow \text{rcvr} \ll \text{Integer}\cdot\text{eq}\ (\text{arg}\ ;\ (\text{Fraction}\cdot\text{denom} := 1))$

Peculiarities: Note that this might result in an overloaded definition, which might seem that at run time we need to arbitrate depending on the dynamic type. For example, another subclass of int, say fracbug : *FracBug* might define a *denom* method and require:

FracBug **subtype of** *Integer*, **with**

$\text{rcvr}_\bullet \textit{prod}\ \text{arg} \longleftarrow \text{rcvr} \ll \textit{prod}\ (\text{arg}\ ;\ (\text{denom} := \delta)\ )$

However, the overloading can be collapsed, without loss of information, to either

$\text{rcvr}_\bullet \text{Integer}\cdot\text{prod arg} \longleftarrow$

$\quad \text{rcvr} \ll \text{Integer}\cdot\text{prod}\ (\text{arg}\ ;\ (\text{Fraction}\cdot\text{denom} := 1)\ ;\ (\text{FracBug}\cdot\text{denom} := \delta))$

or

$\text{rcvr}_\bullet \text{Integer}\cdot\text{prod arg} \longleftarrow$

$\quad \text{rcvr} \ll \text{Integer}\cdot\text{prod}\ (\text{arg}\ ;\ (\text{FracBug}\cdot\text{denom} := \delta)\ ;\ (\text{Fraction}\cdot\text{denom} := 1))$

---

perspective, is different.

which gives the effect of having both the defaults. A Fraction *prod* will look for Fraction·denom and the ability to access FracBug·denom is only illusionary since a fract *prod* will in no way be able to send a FracBug·denom message to its argument.

3.7 Altering hierarchies

Another haunting problem with current inheritance hierarchies is the lack of the ability to modify an entire hierarchy by a client, without resorting to modification of the existing code [56]. This arises in such cases as when a black and white terminal is changed to a color terminal, resulting in the entire hierarchy of shapes needing the addition of color attributes which are usually orthogonal to the existing attributes.

The subclassing mechanism allows one to specialize an existing class by some additive extensions. However, there is no mechanism to specialize a given hierarchy in a similar manner. From *Treaty of Orlando* [56] we quote: "no existing object-oriented language provides a mechanism for this that does not require the modification of previously existing code. Yet conceptually, we should be able to perform these changes by some sort of additive extension". In this section, some progress is made towards this.

Suppose that we have a shape hierarchy, that is designed for a black and white terminal, and hence color attributes were not considered in the design.

This shape hierarchy is a type hierarchy, meaning that it is possibly a part of a larger inheritance hierarchy. To add the color attributes to the entire shape hierarchy all we need is a prototype color object, say pcolor, defined by a **class** construct that takes no arguments. Then we can redefine the meaning of passing a message to an object of this hierarchy as: rcvr$\bullet$*shape* mesg arg $\longleftarrow$ (rcvr ; pcolor)$\ll$ mesg arg

As a result of such a definition, we have changed the type hierarchy effectively to the one defined with color attributes. It would thus be possible to send "colored messages" to objects in this hierarchy or to delegate "colored objects" to objects of this hierarchy in a type safe manner.

The above solution works fine only for simple color attributes and attributes that use self. However for attributes that use super, like in the case of a display method [56] which needs to use the display method of shape in order to display a colored shape, the solution may not be satisfactory. To correct this, we separate the methods of color prototype into overridable methods, like simple attributes, and the non-overridable methods, like the display method. Let ncolor be an object that defines the non-overridable methods and pcolor be one that defines overridable methods, then we can change the messager to instead:

$\text{rcvr} \bullet_{shape} \text{mesg} \longleftarrow (\text{ncolor} ; \text{rcvr} ; \text{pcolor}) \ll \text{mesg}$

Peculiarities: This has created some handicap on the ease of extensibility of this hierarchy in the future. Subclasses defined later will assume they are inheriting color attributes, but with one big handicap of not being able to modify the color display method! However, a uniform modification of this method can be achieved by modifying the messager, if so desired. Modifying the messager gives a new perspective on the ability of modify existing code, especially existing hierarchies of code, and also a possibility of having non-modifiable (uniformly modifiable) code in the system. A variation of this technique can also be used to update certain outdated persistent objects.

3.8 Harnessing subtype information

The elements of a list of shapes may be polygons or circles. Circles have diameter defined but polygons do not. In traditional languages it is difficult to ask a shape anything about diameter since there is no guarantee that it would be defined for the shape object (with out resorting to checking the type information at runtime). Here we use a technique we would like to call transient attribution. Using this technique, we can send messages to objects in a type safe manner.

For this example, let us assume that there are no circles with diameter less than zero and create a simple object which when used as a delegatee will allow a shape object to be type safe for this particular message.

Let S be a list of shapes and *cdia* be defined as (Circle·diameter := -1), then we can construct an expression that checks if the diameter is greater than zero:

if (((S•head ; *cdia*)•Circle·diameter) > 0) then ...

This simple solution can be very effective in reducing this limitation of traditional languages. However, it needs a sentinel value that can be used in a type safe manner (-1 in the case of Circle·diameter).

CHAPTER 4

EIFFEL EXAMPLES RECONSIDERED

4.1 Introduction

Eiffel, an object oriented language, has been developed with the intentions of providing the flexibility that no other strongly typed object oriented language has provided for. The type-checking of Eiffel has thus evolved to be a very complex process, which involves system validity as a whole [45, chapter 22]. The versions of Eiffel implementations available to date have not been able to provide for such thorough checking. These holes in the implementation of the type-checking rules were not acknowledged by the Eiffel implementors and is therefore the general opinion that Eiffel, a language intended to be statically type-checkable has had design oversight. This language has thus become the infamous example for many theoreticians working on developing type-safe object-oriented languages [7, 11], ever since Cook illustrated [16] these type-checking holes and made a proposal of removing the "extra flexibility" that Eiffel provides in favor of the safety of type-checking. The Eiffel designers, however, have not forsaken this flexibility as they believe that this flexibility provides for the most natural evolution of software systems. Instead, the Eiffel designers have developed a complex set of type-checking rules inorder to provide for the complete check of the whole system [44, 45] The main problem we see in this is that such "system validity checking" effectively defeats the advocated flexibility of the language, since adding a new class to the evolving system could result in a type error in a class that was type-checked as "OK" earlier. Loosing the monotonicity of the type correctness of a class is not only very confusing to the system developers, but also prevents evolution of the system as advocated. It has been noted by others [36, 25] that the most desirable solution to this issue is to reinterpret the semantics of message passing, which allows the user to indicate the method to be executed for the potentially problematic cases. The proposal made in this chapter also suggests a reinterpretation (ala, Chapter 3) of the message passing semantics of Eiffel. The effect of this proposal is to provide for desirable semantics for the messages that would not be understood in the current implementations. We advocate them as "desirable" by considering system evolution, efficiency and upwards compatibility of the existing Eiffel programs.

The introduction of the first proposal, from William Cook [16], notes that[1]:

> Statically type-correct Eiffel programs may produce run-time errors because (1) attributes may be redeclared during inheritance, invalidating assignments in the superclass, (2) a formal method argument type may be restricted in violation of the contravariance of function types, and (3) two applications of a generic class are assumed to conform if the actual arguments conform.

The first proposal to make Eiffel type-safe has been basically to eliminate the flexibility of Eiffel for most part. The second proposal, from Bertrand Meyer [44], is a clarification that Eiffel can be type-checked with all its features if subtyping does not imply substitutability in all contexts. Here, we propose another interpretation for Eiffel programs and a compile time translation that would enable Eiffel programs to become type-safe and give a flexible interpretation to

[1]Text from Cook's paper [16] is indented.

many problem cases, while maintaining substitutability once type-checked. Our goal is to give an interpretation that is upwards compatible with the dynamic semantics of Eiffel, i.e., we suggest an alternate behavior only where the current Eiffel semantics would lead to a runtime error.

We follow Cook's paper, which demonstrated the problems of Eiffel type conformance rules, and suggest a fix for the examples used. The fixes suggested in this chapter are indicative of our approach and some limitations for generalizing this is seen in Chapter 6.

4.2 The Eiffel type system

> Eiffel is a strongly-typed object-oriented programming language with a rich type system and a conformance (subtype) relation linked to the inheritance hierarchy. A type in Eiffel is either a simple type (REAL, INTEGER, etc.), a class type $P[U_1, \ldots, U_n]$ where P is a generic class with n parameters and $U_1, \ldots, U_n$ are types, a formal parameter T inside a parameterized class definition, or an association type of the form `like` *anchor*. When a class does not have generic parameters (i.e. $n = 0$) it is written as P (instead of $P[\,]$).

> A class definition specifies the *features* of the class. Features may be either *attributes* or *routines*.

Attributes are called *instance variables* and routines are called *methods* in many object-oriented languages.

> A class must export a feature in order for it to be used outside the class definition. The special symbol *Current* plays the role of *self* in SMALLTALK and of *this* in C++.

The following two classes: `Base` and `Extra`, serve as basis for many examples presented in the paper[2].

```
class Base feature
   base(n : Integer) : Integer is
      do Result := n * 2 end;
end

class Extra inherit Base feature
   extra(n : Integer) : Integer is
      do Result := n * n end;
end
```

In Eiffel, *type conformance* is a relation intended to capture the notion of one type being immediately compatible with another, in the sense that in a context where a value of some type is expected, any value of a conforming type can be used. Conformance is often called *subtyping*.

Type compatibility is essentially same as type conformance, except that it defines the primitive type *INTEGER* to be compatible with *REAL*.

Types of the form `like` *anchor* represent *declaration by association*, and will be called *association types*. Informal meaning of the type expression `like` *anchor* is that it represents the type of *anchor*, where *anchor* is an attribute or the pseudo-variable `Current`.

[2]We assume that all the `features` are exported, unless explicitly specified otherwise.

4.3 Delegation

Delegation when used in a controlled manner can help develop flexible statically type-checkable object-oriented languages (Chapter 3). Delegation is used in the following sections to provide default behavior for potentially problematic variable usage. The default behavior should be the most desirable one, and it is reasonable for it to be user-definable as indicated in Chapter 3. In this paper, since we are not proposing any extensions to the Eiffel language, we only consider a particular *desirable* behavior as suggested by the context.

Delegation captures the effect of inheritance, at runtime, by redefining *Current* as inheritance does. When an object x delegates to another object y, denoted by $x \bullet y$, the resulting object z understands the messages x understands while delegating the messages that x does not understand to y. Since the pseudo-variable *Current* in z, which is $(x \bullet y)$, refers to z, the messages sent to *Current* in the methods of x or y will be sent to z instead. This means that every message sent to $(x \bullet y)$, whether from outside or from inside, will be first attempted as a message to x, and only if this does not succeed it will be attempted as a message to y.

For example, let `b : Base` and `e1,e2 : Extra`. Then, when the variables are used as in a statically typed language (as in Eiffel), `b` $\bullet$ `e1` understands all the messages that could be sent to `e1` (see Section 4.5), and `e2` $\bullet$ `e1` is equivalent to `e2`.

A special case of this is when `extra`δ is an unique instance of the incremental specification, say `Extra`δ, that defines `Extra`, given `Base`. Then, `b` $\bullet$ `extra`δ understands all the messages that could be sent to `e1` (an instance of `Extra`), and `e2` $\bullet$ `extra`δ is equivalent to `e2`.

```
class Extraδ feature
   extra(n : Integer) : Integer is
      do Result := n * n end;
end
```

These observations on delegation (modulo Section 4.5) are used in the following sections to provide default behavior when it is known that a variable of type `Base` can potentially receive an `extra` message.

4.4 Attribute type redefinition

The example below illustrates the problem of assignment to an attribute that is redeclared. The class `P1` defines an attribute `a` of type `Base`, and a method `setup` assigns a new instance of `Base` to an attribute `a`. The subclass `C1` redeclares the attribute to have type `Extra`, which conforms to `Base` in accordance with the type redefinition rule. It also defines a `problem` method which calls the inherited `setup` method to initialize `a`, and then accesses the `extra` feature of `a`.

```
class P1 feature
   a : Base ;
   setup is
      local
         x : Base ;
      do
         x.Create ;    -- step2: create an instance of Base
         a := x;   -- step3: assign new instance to a
```

```
      end;
end

class C1 inherit P1 redefine a feature
   a :  Extra;
   problem :  Integer is
      do
         setup;    -- step1:  call setup
         Result := a.extra(2);
              -- step4:  a is an instance of Base
      end;
end
```

The comments describe the steps that occur when the problem message is sent to an instance of C1; a dynamic violation of the type system is produced in step4.

The source of the problem is an interaction between assignment and attribute redeclaration. An assignment that is type-correct given the original declaration is no longer valid in the context of redeclaration. In the face of this very fundamental problem, most programming languages have been forced to prohibit redeclaration of attributes.

We consider a few alternate "solutions" to this problem, since the proposal to restrict this feature appears to make the language inflexible.

Solution 4.4.1: Let `extra`δ be an instance of the incremental specification that defines `Extra` given `Base` as discussed in Section 4.3. A type checker can detect the problem and rewrite the resulting code of `C1` to:

```
class C1 feature
   a :  Extra;
   setup is
      local
         x :  Base ;
         do
            x.Create ;
            a := x • extraδ;
            -- specialize Base with the Extra messages
      end;
   problem :Integer is
      do
         setup;
         Result := a.extra(2);
      end;
end
```

Solution 4.4.2: Equivalently, the `problem` method can instead be transformed to the following, since `a` in superclass's `setup` has "contravariant usage":

```
    ...
    problem :  Integer is
      do
         setup;
```

```
        Result := (a • extraδ).extra(2);
     end;
end
```

This solution transforms the `problem` method, and not the superclass methods. Thus, the superclass code can be reused as is.

Solution 4.4.3: Alternatively, a transformation of the type information of the superclass methods can be specialized for the subclass. The resulting class definition for this example will be

```
class C1 feature
   a : Extra;
   setup is
      local
         x : Extra ; -- type of x is changed
         do
            x.Create ;
            a := x ;
      end;

   problem : Integer is
      do
         setup;
         Result := a.extra(2);
      end;
end
```

which is type-safe. Basically, the user has thus specialized the class `P1` to the context of `C1` by the redeclaration of the attribute `a`. This kind of specialization is automatically done in [49]. Eiffel has an explicit way of defining this intention, using the declaration by association, which would result in a program that is type-safe and has the same semantics.

4.5 Using type tags

The "solutions" 4.4.1 and 4.4.2 suggested in the previous section will not result in type-safety unless some measures are taken to identify the intended messages in a given context. Consider the case when there are other subclasses of `Base` that also define a method named `extra`, but with conflicting type information. For example, in the class `Ebug` below, the type of the method `extra` does not conform to the type of the method `extra` in the class `Extra`. Note that the method `base`, although redefined, has to conform to the type of the method `base` in class `Base`:

```
class Ebug inherit Base redefine base feature
   extra(s : String) : String is
      do Result := s end;
   base(n : Integer) : Integer is
      do Result := n end;
end
```

If `b : Base` and `e: Extra`, then when a message `extra(x)` is sent to `b` • `e` the method invoked will depend on the object in `b` during runtime. Since `EBug` conforms to `Base`, it is possible that `b` has an instance of `Ebug`. When such is the case one cannot predict at compile time which of the `extra` methods will be invoked during runtime, and hence its type-safety cannot be assured. To make it possible to allow independent definitions of method names, it is necessary for the compiler to tag the method names with type information. This way $\texttt{extra}_{\texttt{Extra}}$ can be distinguished from $\texttt{extra}_{\texttt{Ebug}}$ and the context information can be used to send the appropriate message, say $\texttt{extra}_{\texttt{Extra}}$ in the context of `Extra`.

Methods of a superclass can be redefined by a subclass, and our tagging should honor redefinitions in such a manner that dynamic binding results in invoking the appropriate method. Therefore, a simple type tag is not sufficient. Instead, the tagged method name should be inherited from the superclass when a method is redefined in a subclass. Only the new method names should be given the subclass tag, while the inherited methods names should preserve the tag as defined in the superclass.

For example, the class `EBug` will effectively be:

```
class Ebug
   extra_EBug(s :  String) :  String is
      do Result := s end;
   base_Base(n :  Integer) :  Integer is
      do Result := n end;
end
```

and a message `base` sent to an object of type `EBug` will effectively be $\texttt{base}_{Base}$, while a message `extra` sent to it will be $\texttt{extra}_{EBug}$. With the above simple technique of tagging the messages, and the methods, using the type information, only the intended method will be invoked.

4.6 Method argument type redefinition

The restriction of a formal method argument during inheritance, as permitted by the type redefinition rule, introduces the potential for dynamic type-errors within statically type-correct Eiffel programs. The problem arises when a subclass instance with a redefined argument is used where a superclass value is expected. In this situation an argument may be passed to the method that is not compatible with the redefinition, and a dynamic type-error may occur.

In the following example,

class `P2` is declared to have a method `get` which calls the `base` method of its argument. The subclass `C2` redeclares the `get` method and redefines it to call the `extra` method instead:

```
class P2 feature
   get(arg :  Base) :  Integer is
      do Result := arg.base(1) end;   -- compute twice 1

class C2 inherit P2 redefine get feature
   get(arg :  Extra) :  Integer is
      do Result := arg.extra(2) end;   -- compute square of 2
end
```

`C2` conforms to `P2` because `C2` inherits from `P2`. This means that objects created by `C2` may be assigned to variables of type `P2`. Manipulating a variable of type `P2` which refers to an instance of type

`C2` may lead to errors, as illustrated below:

```
local
   a : Base;
   v : P2;
   b : C2;
do
   a.Create;
   b.Create;
   v := b;  -- statically type correct
          -- because C2 conforms to P2
   v.get(a);  -- call C2.get which calls a.extra(2)
end
```

Since `v` has static type `P2`, `v.get` has type `Base` $\rightarrow$ `Integer`, and `v.get(a)` is statically type-correct. But `v` contains an instance of `C2` and the methods are selected dynamically, so `v.get` refers to the `get` method of `C2`. This method, when passed `a` as a parameter will access the `extra` method of `a`, which does not exist.

Solution 4.6.1: Our solution is again to consider `C2` as information for specializing the class `P2`. This will allow a translation of the above `C2` by a compiler to the following:

```
class C2 inherit P2 redefine get feature
   get(arg : Base) : Integer is
      do Result := (arg • extraδ).extra(2) end;
```

```
    -- arg gets a default ‘extra’
end
```

If `arg` above is of type `Extra` at runtime, the `extra`δ in (`arg` • `extra`δ) will be ignored. However, if `arg` is of type `Base`, this method specializes it with `Extra` information.

However, a solution similar to ‘solution 4.4.3’ will need global information of all uses of `C2`’s `get` and therefore is undesirable from the point of a compiler being able to transform the user defined code appropriately.

4.7 Methods declared by association

When attributes are declared by association, it results in redeclaration of attributes indirectly in the derived classes.

> In particular, if `like Current` is used, then the method argument is effectively redeclared in any subclass leading to the error outlined in Section 4.6.
>
> The problem of using `like Current` is illustrated in the following example.
>
> For this example the features of `Base` and `Extra` are combined with the classes illustrating method redeclaration in Section 4.6. This combination allows the use of a `like Current` argument in the method `get`:

```
class P3 feature
   base(n : Integer) : Integer is
      do Result := n * 2 end;
```

```
   get(arg :  like Current) :  Integer is
      do Result := arg.base(1) end;
   end

class C3 inherit P3 redefine get feature
   extra(n :  Integer) :  Integer is
      do Result := n * n end;
   get(arg :  like Current) :  Integer is
      do Result := arg.extra(2) end;
end
```

The example behaves just like the one in Section 4.6, and it produces a similar dynamic type-error.

Since this is an essential feature to define derived classes that are useful, Cook's proposal does not prohibit the use of `like Current`, but prohibits a subclass that thus results from being conforming to its superclass:

> We propose to eliminate the conformance relation that allowed the assignment `v := b`, an essential step in producing the dynamic error. We propose that if a class has an argument declared by association, then an inheriting class does not conform to that parent if the type of the anchor is redefined. In particular, the type of `Current` is always redefined so formal arguments of type `like Current` prevent conformance.

Solution 4.7.1: Let `C3`δ denote the additional behavior that derives `C3`, given `P3`. Solution 4.6.1 works.

```
class C3 inherit P3 redefine get feature
   ...
   get(arg : like Current) : Integer is
      do Result := (arg • C3δ).extra(2) end;
        -- arg gets a default 'extra'
end
```

A more desirable solution is a variation of the above, which uses the attribute values of the `Current` object dynamically. The `Current` object guarantees that it can handle the messages delegated to it. This can be thought of as adoption.

Solution 4.7.2: Use `Current` instead of `C3`δ, since `Current` has the information that can specialize the superclass. However, instance variables of `Current` can be modified inadvertently if this is implemented in a naive manner.

```
class C3 inherit P3 redefine get feature
   ...
   get(arg : like Current) : Integer is
      do Result := (arg • Current).extra(2) end;
        -- Current has default 'extra'
end
```

The subclass (as subtype) specification should conform to the contravariance rule in redefining method arguments. When a desire for covariance is noticed, the type-checker can interpret this as the user's desire for specializing the instances of the superclass appropriately. This can be achieved by interpreting the covariant declaration of arguments as suggestions by the user, like in solution 4.6.1 above. Delegation can be used to specialize objects by providing default behavior, for the objects in the system that need to change due to system evolution.

CHAPTER 5

AN ALGEBRAIC SPECIFICATION AND PROTOTYPE

5.1 Introduction

Object-Oriented Programming has become popular among the people who build large software, and has been intriguing many researchers in the area of semantics of programming languages [9, 14, 33, 15, 51, 21, 55, 17, 8, 32, 7, 34]. There are many Object-Oriented Programming Languages (OOPL) that have surfaced in the recent years [20, 26, 53, 59, 36, 58, 43, 5, 19, 38, 12, 61, 48]. Although many of these languages are used for programming in the large, the kernel of these languages is generally small [32, 50, 11, 7]. The semantic mechanisms of these languages have been studied mostly in a denotational setting [15, 18, 33, 32, 7, 51]. Although this research has produced interesting constructions for the denotational semantics community, among the community of programmers the mechanisms are still understood using informal terms. Algebraic specifications are not only easy to read and understand, since they are based on equations, they can provide a very different (often simplified) perspective on these denotational constructions to help bring to focus a number of issues that make these constructions interesting.

In this chapter, we intend to provide an appropriate perspective on the dynamic and static semantic issues present in object oriented languages, many of which was informally discussed in Chapter 1, and to provide a basis for the prototype implementation of our experimental language presented in Chapter 6.

The formalism used is ASF+SDF. The ASF+SDF system is a well-developed integrated system that makes the specification of languages simple through the use of syntax-directed editors and automatic generation of parsers. As these specifications are executable, we have an interpreter of the language we are specifying immediately available for experimentation.

In the next section (Section 5.2), we introduce the ASF+SDF formalism that is used to specify the languages presented here.

Sections 5.3–5.7, present the progressive design of an OOPL – using the languages KOL, KOOL, ASOOL. In this document we focus on a language based on objects (dynamic binding) with simple expressions that manipulate the objects and extend this language with inheritance, modification of self and type checking. These languages do not have side-effects, with the interpretation discussed in section 5.7 and can therefore be classified as functional object oriented programming languages. The language EOOL presented in Chapter 6 is an Eiffel [43] like language, but forces inherited types [17] to behave like subtypes using the ideas described in Chapter 4.

5.2 ASF+SDF

ASF, an acronym for Algebraic Specification Formalism, [4, 30] is a formalism supporting modularization and conditional equations. The ASF system, is able to generate rewriting systems from the specification, and to execute them.

SDF, Syntax Definition Formalism, has been developed [28] to support the definition of lexical, context-free and abstract syntax at the same time, and parsers can be incrementally generated from it.

These two formalisms have been combined into one algebraic specification formalism called ASF+SDF[1]. The ASF part takes care of the (conditional) equations, while the SDF part provides complete syntactic freedom for the specifier. A sentence according to an SDF grammar is mapped uniquely on an abstract syntax tree [28], and constitutes a term over the signature. Equations over these terms may have positive or negative conditions [4, p.306] giving rise to the quasi-initial models introduced by Kaplan [35]. For a complete description of the ASF+SDF formalism see [4, 31].

Specifying in the ASF+SDF formalism is supported by the ASF+SDF *system* [37] [31, Chapter 5]. This system is able to generate parsers from ASF+SDF specifications and to derive term rewriting systems for specifications in ASF+SDF, thus allowing to execute ASF+SDF specifications. Moreover, it can generate syntax-directed editors for modules in the specification as well as for terms over the signature. The system is able to perform several static and semantic checks on the specifications, and supports testing of specifications.

5.2.1 Lexical syntax

SDF allows the definition of lexical syntax, i.e., the definition of the elementary "words" of the syntax. There are two important corollaries for the specifier: First of all the `LAYOUT` for the equations must be defined, i.e., the specifier himself can (and must) define what symbols constitute white space, and how comments can be recognized. A typical layout definition is:

```
exports
  lexical syntax
    [ \t\n]           -> LAYOUT
    "%%" ~[\n]* "\n"  -> LAYOUT
```

A space, tab, or new line is a layout symbol, as well as everything between

[1]This section is included for completeness and is borrowed with permission of van Duersen [22]

two consecutive percent signs and a new line. If tokens of sort `LAYOUT` are detected in a text, they are ignored.

Secondly, variable declarations are treated as declarations of lexical syntax. This implies that any construct allowed in the lexical syntax definition is allowed in the variable definition section as well. Consequently it is possible to define the variables `p1`, `p2`, `p3`, ... all at once:

```
variables
  "p"[0-9]*    -> NAT
```

This declares all words starting with a `p` followed by zero or more characters in the range `0-9` to be variables of sort `NAT`.

5.2.2 List functions

An important feature of the ASF+SDF formalism is the existence of list functions and list variables. List functions have a variable number of arguments, and list variables may range over any number of arguments of a list function.

As an example, suppose we would like to have a function `[ ]` for the empty set, `[ E1 ]` for a set with one element, `[ E1, E2 ]` for a set with two elements, and so on.

The way to define this in ASF+SDF is as follows:

```
exports
  sorts SET ELEMENT
  context-free syntax
    "[" {ELEMENT ","}* "]"       -> SET
```

The asterisk `*` says that we want zero or more `ELEMENT`s, while the comma says that these should be separated by commas. Thus, a set consists of `ELEMENT`s, separated by commas and delimited by `[` and `]`.

This list notation is simply an abbreviation for the declaration of infinitely many functions `[ ... ]`, each with a different number of arguments. Likewise, the same (concrete) syntax could have been obtained by the following “normal” BNF grammar rules:

```
context-free syntax
   "[" ELEMENTS "]"      -> SET
   "[" "]"               -> SET
   ELEMENT               -> ELEMENTS
   ELEMENTS "," ELEMENT  -> ELEMENTS
```

In order to define equations over list functions, we need list variables:

```
exports
  sorts SET ELEMENT
  context-free syntax
    "[" {ELEMENT ","}* "]"   -> SET
  variables
    "Elts"[123]  ->  {ELEMENT ","}*
    "i"          ->  ELEMENT
equations
  [1]  [Elts1, i, Elts2, i, Elts3] =
          [Elts1, i, Elts2, Elts3]
```

`Elts1`, `Elts2`, and `Elts3` are list variables, ranging over list of zero or more `ELEMENT`s separated by commas.

Here we have specified in one single equation that elements of sets do not have multiplicity: Any set containing element `i` at least two times is equal to the set containing one occurrence less of `i`.

5.2.3 Disambiguation

SDF supports *priority* and *associativity* declarations, which are used to disambiguate terms which can otherwise be parsed in several ways. For instance,

```
imports Naturals
exports
  context-free syntax
    NAT "^" NAT     -> NAT  {right}
    NAT "-" NAT     -> NAT  {left}
priorities
  "-" < "^"
```

Here `^` (exponentiation) has a higher priority than `-`, thus stating that, for instance, the term `1 - 3 ^ 2` should be interpreted as `1 - ( 3 ^ 2 )` rather than as

`(1 - 3 ) ^ 2`.

Moreover, `-` is defined to be left associative, i.e., `1 - 2 - 3` is interpreted as `(1 - 2) - 3` rather than `1 - (2 - 3)`. Likewise, exponentiation is right associative.

5.2.4 Sections

A module in the ASF+SDF formalism may contain the following keywords:

- `imports`, listing the names of imported modules;
- `exports` or `hiddens`, listing the exported or hidden items;
- `sorts`, listing the sorts;
- `lexical syntax`, giving lexical syntax;
- `context-free syntax`, listing the function declarations. Attributes, between `{}` brackets, may be associated with functions, stating for instance that the function is `left` or `right` associative;
- `priorities`, giving priorities between functions;
- `variables`, declaring the variables used in the equations;
- `equations`, listing the (conditional) equations of the module.

5.3 A kernel object based language, KOL

The Object Based language presented in this section serves as the kernel of all the other languages specified in later sections. It is a functional language with copy semantics for ":=" and is therefore alias free.

5.3.1 KOL syntax

The KOL-syntax module given below is described in this section. An object oriented program is a *set* of class definitions and an expression that starts the chain of invocations through the instances of these classes. In KOL, a program (PROGRAM) is a *list* of classes followed by an expression (EXP). The *list* helps enforce a define-before-use condition during inheritance of code (Section 5.5). Since we do not have any predefined classes or primitive data types, we insist that there is at least one class definition (CLASS+) in the program. A class has a name (ID), instance variable declarations (VAR-DEC*) and method definitions (METHOD*). The instance variable declarations also insist that an initial value be specified (IEXP). The initialization of instance variables simplifies the definition of "new", as the only way an instance of a class can be created is by using "new ID". Thus we can work around the argument type translation [15] needed otherwise, during inheritance (Section 5.5). For simplicity, we keep the initialization expression (IEXP) trivial.

A method definition consists of a method name, formal arguments and a method body (COMM). For now, the method body consists of just (EXP) and in the definitions below the inclusion, EXP $\rightarrow$ COMM, is present for posterity. Also present for aesthetics are the declarations of CLASSES, METHODS and VAR-DECS, since these declarations help us write equations over them without having to give the context where CLASSES, METHODS or VAR-DECS might occur.

An expression can be an initialization expression (IEXP), *self* to refer to the current object environment, ID to refer to instance variables or formal arguments when defining a method, getting the contents of an instance variable of some object (EXP.ID), updating an instance variable of some object (EXP.ID := EXP) or sending a message to an object to invoke a particular method (EXP.ID({EXP,}*)). Updating an instance variable of an object is the only way, in this language, to obtain different instances of a class since "new ID" always returns the default instance. Updating the instance variable uses copy semantics to create an object with the updated value.

Every expression (EXP) in KOL evaluates to an object (Appendix B adds "nil" to expressions, making "nilvalue" the exceptional case). Message passing is used to invoke a method in the object, using the "." operation (with an argument list) and to access the value of instance variables (without an argument list in the message).

Modification of instance variables of an object is not possible in KOL (see Section 5.7), but there is a facility to obtain updated objects, from a given object and given update information (EXP.ID := EXP). For example, `anobj.avar := aval` will result in an object (an updated `anobj`) with its instance variable `avar` being assigned the value `aval`. The semantics of := is copy semantics and it is therefore impossible to construct cyclic data structures, which require aliasing indirectly (e.g. circular lists). Another peculiarity of copy semantics is that it prevents straight forward definitions of classes with cyclic initialization, as we will depict with examples later. This necessitates the availability of a nil value in the system, so that the user can delay initialization of variables that would require a value that would directly or indirectly refer to the class in which the variable is declared (Appendix B).

module KOL-syntax

imports Layout Identifiers

exports

sorts PROGRAM VAR-DEC METHOD EXP CLASS COMM IEXP CLASSES VAR-DECS METHODS
context-free functions

CLASS+ EXP	→ PROGRAM
class ID VAR-DEC∗ METHOD∗	→ CLASS
var ID ":=" IEXP	→ VAR-DEC
meth ID "(" {ID ","}∗ ")" COMM	→ METHOD
EXP "." ID	→ EXP
EXP "." ID ":=" EXP	→ EXP
EXP "." ID "(" {EXP ","}∗ ")"	→ EXP
ID	→ EXP
self	→ EXP
"(" EXP ")"	→ EXP {**bracket**}
new ID	→ IEXP
IEXP	→ EXP
EXP	→ COMM
CLASS∗	→ CLASSES
VAR-DEC∗	→ VAR-DECS
METHOD∗	→ METHODS

variables

$Meth$ $[0\text{–}9']*$	→ METHOD
$Meths$ $[0\text{–}9']*$	→ METHOD∗
$Iexp$ $[0\text{–}9']*$	→ IEXP
Exp $[0\text{–}9']*$	→ EXP

$$\begin{array}{ll} \mathit{Vdec}\ [\mathit{0}{-}\mathit{9}']{*} & \rightarrow \text{VAR-DEC} \\ \mathit{Vdecs}\ [\mathit{0}{-}\mathit{9}']{*} & \rightarrow \text{VAR-DEC}{*} \\ \mathit{Id\text{-}list}\ [\mathit{0}{-}\mathit{9}']{*} & \rightarrow \{\text{ID “,”}\}{*} \end{array}$$

equations

$$[\text{V0}]\quad \frac{\mathit{Vdec}_0 = \text{var } \mathit{Id} := \mathit{Iexp},\ \mathit{Vdec}_1 = \text{var } \mathit{Id} := \mathit{Iexp}'}{\mathit{Vdecs}\ \mathit{Vdec}_0\ \mathit{Vdecs}'\ \mathit{Vdec}_1\ \mathit{Vdecs}'' = \mathit{Vdecs}\ \mathit{Vdec}_0\ \mathit{Vdecs}'\ \mathit{Vdecs}''}$$

$$[\text{M0}]\quad \frac{\mathit{Meth}_0 = \text{meth } \mathit{Id}_0(\mathit{Id\text{-}list})\ \mathit{Exp},\ \mathit{Meth}_1 = \text{meth } \mathit{Id}_0(\mathit{Id\text{-}list}')\ \mathit{Exp}'}{\mathit{Meths}\ \mathit{Meth}_0\ \mathit{Meths}'\ \mathit{Meth}_1\ \mathit{Meths}'' = \mathit{Meths}\ \mathit{Meth}_0\ \mathit{Meths}'\ \mathit{Meths}''}$$

These equations syntactically reduce the program text in any class where the definition of instance variables ([V0]) or methods names ([M0]) is not unique, by only considering the first declaration.

The following classes describe the specification of Booleans and Naturals in this language. The class bool and nat are slightly modified versions of Ghelli's [24]. The `%%` symbol denotes that the rest of the line is a comment, and is specified in the Layout module.

```
class null


class bool
  meth cond(x,y) x
  meth and(y) self.cond(y,new false)
  meth or(y)  self.cond(new true,y)
```

```
  meth not()  self.cond(new false, new true)

class true
  meth cond(x,y) x
  meth and(y) self.cond(y,new false)
  meth or(y)  self.cond(new true,y)
  meth not()  self.cond(new false, new true)

class false
  meth cond(x,y) y
  meth and(y) self.cond(y,new false)
  meth or(y)  self.cond(new true,y)
  meth not()  self.cond(new false, new true)

class nat
  var isz := new true
  var pre := new null
  meth iszero() isz
  meth setto(x) x.iszero()
             .cond(((self.isz:=new true).pre:=self),
                   ((self.isz:=new false).pre:=(x.pred())))
  meth succ() new nat.setto(self).incr()  %% same as incr()
  meth pred() self.pre
  meth incr() (self.pre := self).isz := new false
  meth decr() self.iszero()
```

```
              .cond(self, (self.isz := (pre.iszero()))
                          .pre := (pre.pred()))
  meth less(i)
    self.iszero()
    .cond(new true, i.iszero()
    .cond(new false, self.pred().less(i.pred())) )


new nat.succ().iszero()
```

Example 12: Some basic classes in KOL

The evaluation of the above program (Section 5.3.3) would result in the following object, which is an instance of class false:

```
 [ ( cond : ( x,y ) y ),
   ( and  : ( y ) self . cond ( y, new false ) ),
   ( or   : ( y ) self . cond ( new true, y  ) ),
   ( not  : (   ) self . cond ( new false, new true ) ) ]
```

Note that the class nat will not be statically type-checkable since, for example, the initial value (new null) of instance variable pre is of a different type than nat (the type of values assigned later). The copy semantics prevents cyclic instantiation of instance variables, as replacing “new null” with “new nat” would loop during instantiation of nat (“new nat”). To compensate for this, the typed versions of this language should provide for a special value (say “nilvalue”) which the type–checker can then treat as a special case.

5.3.2 Value environment

The Value environment module is described in this section. The specification of value environment is similar to traditional value environments specification [3, 29]. The utility of the value environment is different, however, since in an object oriented language, the only interesting environment during program execution is the one associated with *self*. Another interesting difference specific to objects is that the method definitions also exist in the value environment since objects are dictionaries that map the variable names to values and method names to their values which are method definitions.

The Value environment module remains the same for all other languages specified later, unless otherwise mentioned, but with the respective language syntax module being imported instead.

A value pair (VPAIR) is an association of an identifier to a value. A value environment (VENV) is a list of value pairs. The objects are represented as value environments (VENV → VALUE). Since an identifier evaluates to an object, VENV is included in VALUE. However, the activation stack created for maintaining the history of formal to actual argument mappings during the recursive calls of methods also is a value environment. The "lookup ID in VENV" searches the value environment from left to right ("left prefer").

The & operation stacks the value environments as needed during a method invocation such that the left most list of identifier and value pairs, [{VPAIR,}*], is the top of the stack. And the @ operation removes any duplicate mappings by considering the left most ones (a property of [{VPAIR,}*]). Recursion is facilitated by using the &, value and pop operations (see module KOL-eval). The pop operation reduces the activation stack by removing the top VPAIR list when the stack has more than one VPAIR list, while the value operation only keeps the bottom VPAIR list. Updating a VPAIR list is by just inserting a VPAIR at the left, as this insertion will remove the right one (equation LP0), and updating a value environment in general (when considered as an activation record) is to replace the left most definition (when one exists). The (ID-LIST):(VALUE-LIST) is an auxiliary function that generates a VPAIR list, which is a VENV, that is useful to map the formal arguments to actual arguments during a method invocation.

module KOL-venv

imports KOL-syntax Booleans

exports

sorts VALUE VPAIR VENV ID-LIST VALUE-LIST

context-free functions

"(" ID ":" VALUE ")"	→ VPAIR
"[" {VPAIR ","}∗ "]"	→ VENV
VENV	→ VALUE
"(" ID-LIST ")" COMM	→ VALUE
VENV "@" VENV	→ VENV {**left**}
VENV "&" VENV	→ VENV {**left**}
value VALUE	→ VALUE
pop VALUE	→ VALUE
lookup ID in VENV	→ VALUE
update ID in VENV "by" VALUE	→ VENV
ID "isin" VENV	→ BOOL

{ID ","}∗	→ ID-LIST
{VALUE ","}∗	→ VALUE-LIST
"(" ID-LIST ")" ":" "(" VALUE-LIST ")"	→ VENV

variables

Value [*0*–*9*′]∗	→ VALUE
Value-list [*0*–*9*′]∗	→ {VALUE ","}∗
VP-list [*0*–*9*′]∗	→ {VPAIR ","}∗
Venv [*0*–*9*′]∗	→ VENV

equations

[IN0] $\mathit{Id}\ \text{isin}\ [] = \text{false}$

[IN1] $\mathit{Id}\ \text{isin}\ [(\mathit{Id} : \mathit{Value}), \mathit{VP\text{-}list'}] = \text{true}$

[IN2] $$\frac{\mathit{Id} \neq \mathit{Id'}}{\mathit{Id}\ \text{isin}\ [(\mathit{Id'} : \mathit{Value}), \mathit{VP\text{-}list'}] = \mathit{Id}\ \text{isin}\ [\mathit{VP\text{-}list'}]}$$

[IN3] $\mathit{Id}\ \text{isin}\ \mathit{Venv}\ \&\ \mathit{Venv'} = \mathit{Id}\ \text{isin}\ \mathit{Venv} \mid \mathit{Id}\ \text{isin}\ \mathit{Venv'}$

[LK0] $\text{lookup}\ \mathit{Id}\ \text{in}\ [(\mathit{Id} : \mathit{Value}), \mathit{VP\text{-}list}] = \mathit{Value}$

[LK1] $$\frac{\mathit{Id} \neq \mathit{Id'}}{\text{lookup}\ \mathit{Id}\ \text{in}\ [(\mathit{Id'} : \mathit{Value}), \mathit{VP\text{-}list}] = \text{lookup}\ \mathit{Id}\ \text{in}\ [\mathit{VP\text{-}list}]}$$

[LK2] $$\frac{\mathit{Id}\ \text{isin}\ \mathit{Venv} = \text{true}}{\text{lookup}\ \mathit{Id}\ \text{in}\ \mathit{Venv}\ \&\ \mathit{Venv'} = \text{lookup}\ \mathit{Id}\ \text{in}\ \mathit{Venv}}$$

$$\text{[LK3]}\quad \frac{\mathit{Id}\ \text{isin}\ \mathit{Venv} = \text{false}}{\text{lookup}\ \mathit{Id}\ \text{in}\ \mathit{Venv}\ \&\ \mathit{Venv}' = \text{lookup}\ \mathit{Id}\ \text{in}\ \mathit{Venv}'}$$

$$\text{[UP1]}\quad \text{update}\ \mathit{Id}\ \text{in}\ [\mathit{VP\text{-}list}]\ \text{by}\ \mathit{Value} = [(\mathit{Id} : \mathit{Value}), \mathit{VP\text{-}list}]$$

$$\text{[UP2]}\quad \frac{\mathit{Id}\ \text{isin}\ \mathit{Venv} = \text{true}}{\text{update}\ \mathit{Id}\ \text{in}\ \mathit{Venv}\ \&\ \mathit{Venv}'\ \text{by}\ \mathit{Value} = \text{update}\ \mathit{Id}\ \text{in}\ \mathit{Venv}\ \text{by}\ \mathit{Value}\ \&\ \mathit{Venv}'}$$

$$\text{[UP3]}\quad \frac{\mathit{Id}\ \text{isin}\ \mathit{Venv} = \text{false}}{\text{update}\ \mathit{Id}\ \text{in}\ \mathit{Venv}\ \&\ \mathit{Venv}'\ \text{by}\ \mathit{Value} = \mathit{Venv}\ \&\ \text{update}\ \mathit{Id}\ \text{in}\ \mathit{Venv}'\ \text{by}\ \mathit{Value}}$$

$$\text{[AP0]}\quad [\mathit{VP\text{-}list}_1]\ @\ [\mathit{VP\text{-}list}_2] = [\mathit{VP\text{-}list}_1, \mathit{VP\text{-}list}_2]$$

$$\text{[LP0]}\quad \begin{array}{l}[\mathit{VP\text{-}list}_1,(\mathit{Id} : \mathit{Value}), \mathit{VP\text{-}list}_2,(\mathit{Id} : \mathit{Value}'), \mathit{VP\text{-}list}_3] = \\ [\mathit{VP\text{-}list}_1,(\mathit{Id} : \mathit{Value}), \mathit{VP\text{-}list}_2, \mathit{VP\text{-}list}_3]\end{array}$$

$$\text{[EX0]}\quad () : () = []$$

$$\text{[EX1]}\quad (\mathit{Id},\mathit{Id\text{-}list}) : (\mathit{Value}, \mathit{Value\text{-}list}) = [(\mathit{Id} : \mathit{Value})]\ @\ (\mathit{Id\text{-}list}) : (\mathit{Value\text{-}list})$$

$$\text{[VO0]}\quad \text{value}\ [\mathit{VP\text{-}list}] = [\mathit{VP\text{-}list}]$$

$$\text{[VO1]}\quad \text{value}\ \mathit{Venv}\ \&\ \mathit{Venv}' = \text{value}\ \mathit{Venv}'$$

$$\text{[VO0]}\quad \text{pop}\ [\mathit{VP\text{-}list}] = [\mathit{VP\text{-}list}]$$

$$\text{[VO1]}\quad \text{pop}\ [\mathit{VP\text{-}list}]\ \&\ \mathit{Venv} = \mathit{Venv}$$

5.3.3 Dynamic semantics

The module KOL-c2o is used to make objects from class definitions. This module evaluates the initialization expression (IEXP), using the class definitions

(CLASSES) to new objects (object is a VENV). Keeping the IEXP simple enables us to keep this module logically separate from the KOL-eval module, but in EOOL (Appendix 6), we allow more complex expressions during initialization.

module KOL-c2o

imports KOL-venv

exports

context-free functions

$$\text{c2o “[” ID “]” CLASSES} \rightarrow \text{VALUE}$$

$$\text{c2o “[” IEXP “]” CLASSES} \rightarrow \text{VALUE}$$

$$\text{v2o “[” VAR-DEC}* \text{ “]” CLASSES} \rightarrow \text{VENV}$$

$$\text{m2o “[” METHOD}* \text{ “]” CLASSES} \rightarrow \text{VENV}$$

variables

$$\mathit{Fargs}\ [\mathit{0}-\mathit{9}']* \rightarrow \{\text{ID “,”}\}*$$

$$\mathit{Aargs}\ [\mathit{0}-\mathit{9}']* \rightarrow \{\text{VALUE “,”}\}*$$

$$\mathit{Cls}\ [\mathit{0}-\mathit{9}']* \rightarrow \text{CLASS}*$$

$$\mathit{Comm}\ [\mathit{0}-\mathit{9}']* \rightarrow \text{COMM}$$

equations

$$\text{[C2O0]}\quad \text{c2o[new } \mathit{Id}] \ \mathit{Cls} = \text{c2o}[\mathit{Id}] \ \mathit{Cls}$$

$$\text{[C2O1]}\quad \frac{\begin{array}{c}\mathit{Venv}_1 = \text{v2o}[\mathit{Vdecs}] \ \mathit{Cls} \text{ class } \mathit{Id} \ \mathit{Vdecs} \ \mathit{Meths} \ \mathit{Cls}', \\ \mathit{Venv}_2 = \text{m2o}[\mathit{Meths}] \ \mathit{Cls} \text{ class } \mathit{Id} \ \mathit{Vdecs} \ \mathit{Meths} \ \mathit{Cls}'\end{array}}{\text{c2o}[\mathit{Id}] \ \mathit{Cls} \text{ class } \mathit{Id} \ \mathit{Vdecs} \ \mathit{Meths} \ \mathit{Cls}' = \mathit{Venv}_1 \ @ \ \mathit{Venv}_2}$$

$$\text{[V2O0]}\quad \text{v2o[]} \ \mathit{Cls} = []$$

[V2O1] $\text{v2o}[\text{var } \mathit{Id} := \mathit{Iexp}\ \mathit{Vdecs}]\ \mathit{Cls} = [(\mathit{Id} : \text{c2o}[\mathit{Iexp}]\ \mathit{Cls})]\ @\ \text{v2o}[\mathit{Vdecs}]\ \mathit{Cls}$

[M2O0] $\text{m2o}[]\ \mathit{Cls} = []$

[M2O1] $\text{m2o}[\text{meth } \mathit{Id}(\mathit{Fargs})\ \mathit{Comm}\ \mathit{Meths}]\ \mathit{Cls} = [(\mathit{Id} : (\mathit{Fargs})\ \mathit{Comm})]\ @\ \text{m2o}[\mathit{Meths}]\ \mathit{Cls}$

The evaluation of (new bool) would result in the following object:

```
[ ( cond : ( x,y ) x ),
  ( and : ( y ) self . cond ( y,new false ) ),
  ( or : ( y ) self . cond ( new true,y ) ),
  ( not : (  ) self . cond ( new false,new true ) ) ]
```

In the module KOOL-eval presented below, the function [[_]] is the evaluation function. The module describes the evaluation of the program and in turn the evaluation of expressions in the program.

module KOL-eval

imports KOL-c2o

exports

context-free functions

"[[" PROGRAM "]]"	→ VALUE
"[[" COMM "]]e" VENV CLASSES	→ VALUE
"[[" {EXP ","}* "]]" VENV CLASSES	→ VALUE-LIST

variables

Self [0–9′]*	→ VENV
Exp-list [0–9′]*	→ {EXP ","}*

equations

[P1] $[[Cls\ \ Exp]] = [[Exp]]\mathrm{e}\ []\ Cls$

[I0] $[[\mathrm{self}]]\mathrm{e}\ Self\ \ Cls = Self$

[I2] $[[Id]]\mathrm{e}\ Self\ \ Cls = \mathrm{lookup}\ Id\ \mathrm{in}\ Self$

[Nw] $[[\mathrm{new}\ Id]]\mathrm{e}\ Self\ \ Cls = \mathrm{c2o}[Id]\ Cls$

[E0] $$\frac{Self' = [[Exp]]\mathrm{e}\ Self\ \ Cls}{[[Exp.\ Id]]\mathrm{e}\ Self\ \ Cls = [[Id]]\mathrm{e}\ Self'\ Cls}$$

[A0] $$\frac{Self' = [[Exp]]\mathrm{e}\ Self\ \ Cls}{\begin{array}{l}[[Exp.\ Id := Exp']]\mathrm{e}\ Self\ \ Cls \quad = \\ \mathrm{update}\ Id\ \mathrm{in}\ Self'\ \mathrm{by\ value}\ [[Exp']]\mathrm{e}\ Self\ \ Cls\end{array}}$$

[L0] $[[]]\ Self\ \ Cls =$

[L1] $$\frac{\begin{array}{c}Value = \mathrm{value}\ [[Exp]]\mathrm{e}\ Self\ \ Cls, \\ Value\text{-}list = [[Exp\text{-}list]]\ Self\ \ Cls\end{array}}{[[Exp, Exp\text{-}list]]\ Self\ \ Cls = Value, Value\text{-}list}$$

[M0] $$\frac{\begin{array}{c}Self' = [[Exp]]\mathrm{e}\ Self\ \ Cls, \\ \mathrm{lookup}\ Id\ \mathrm{in}\ Self' = (Id\text{-}list)\ Exp', \\ Aargs = [[Exp\text{-}list]]\ Self\ \ Cls, \\ Value = [[Exp']]\mathrm{e}\ (Id\text{-}list) : (Aargs)\ \&\ Self'\ Cls\end{array}}{[[Exp.\ Id(Exp\text{-}list)]]\mathrm{e}\ Self\ \ Cls = \mathrm{pop}\ \ Value}$$

The above **equations** reduce a program to its output. An evaluation of the

Program is an evaluation of the expression following the class list, assuming that the self environment is initially empty ([P1]). The evaluation of the expression *self* just returns the self environment (*Self*); the evaluation of *Id* is to get its value from the self environment and evaluation of "new *Id*" is to create a new object from the class definition of *Id*.

The evaluation of *Exp.Id* is to get the value of the instance variable *Id* of the object resulted by evaluating *Exp* (rule [E0]). The evaluation of *Exp.Id* := *Exp*′ returns an object with the instance variable *Id* mapped to the result of evaluating the expression *Exp*′. The role of "value" in rule [A0] is to ignore any history that was maintained in the self environment.

The rules [L0] and [L1] extend expression evaluation to expression lists, as this is needed to evaluate the list of actual arguments of a message send. The rule [M0] specifies the effect of message send. *Exp*′ is the body of the corresponding method and *Id-list* is the formal arguments list. The actual arguments, $Aargs$, are bound to the formal arguments ((*Id-list*):($Aargs$)) and the resulting VPAIR list is stacked on top of the current self environment. The body of the method is evaluated using this new "self environment" and pop forgets the most recent bindings of formal to actual arguments.

Recursion is facilitated in a very simple manner by using the &, value and pop operations. When the result of evaluation of an expression is to be used as a value in an object (a value environment), only the bottom of the stack is considered (the "value" operation), while during normal exit the most recent bindings are forgotten using the "pop" operation.

5.4 Type–checking with subtypes

The syntax of the language KOL is extended to TKOL to facilitate type declarations of formal arguments, instance variables and method return types. The name of the classes serve as type names. The classic subtyping rules presented in Section 1.3 provide us subtyping in TKOL. Informally, an object of a more defined class can be substituted where an object of a less defined class is expected (through type declaration).

5.4.1 The syntax module of TKOL

TKOL adds type declaration facility in the method definition. The method definitions now require return type information and the type information of formal arguments. The method definition signature of KOL is replaced in the module TKOL-syntax instead to be:

meth ID ":" ID "(" {IDID ","} $*$ ")" COMM $\rightarrow$ METHOD

The new sort IDID is simply a pair of IDs separated by ":".

ID ":" ID $\rightarrow$ IDID

Since the variable declarations insist on initial values, we can consider the type of the initial value as the intended type of the instance variable[2].

Assume the following **variables**, in addition to those in KOL-syntax module.

[2]This is not always desirable, however, it will not reduce the power of the language since more specific instances can be obtained after the initialization by explicitly changing the value of a variable.

$$\mathit{IdId}\ [\mathit{0}\text{–}\mathit{9}']* \quad \to \text{IDID}$$

$$\mathit{IdId\text{-}list}\ [\mathit{0}\text{–}\mathit{9}']* \quad \to \{\text{IDID ``,''}\}*$$

The equation [M0] is also restated appropriately as follows:

$$[\text{M0}] \quad \frac{\mathit{Meth}_0 = \text{meth } \mathit{Id}_0 : \mathit{Id}'(\mathit{IdId\text{-}list})\ \mathit{Exp}, \quad \mathit{Meth}_1 = \text{meth } \mathit{Id}_0 : \mathit{Id}''(\mathit{IdId\text{-}list}')\ \mathit{Exp}'}{\mathit{Meths}\ \mathit{Meth}_0\ \mathit{Meths}'\ \mathit{Meth}_1\ \mathit{Meths}'' = \mathit{Meths}\ \mathit{Meth}_0\ \mathit{Meths}'\ \mathit{Meths}''}$$

The Boolean classes defined in Section 5.3.1 is re–stated here with the type information, in the example program below:

```
class bool
  meth cond:bool(x:bool,y:bool) x
  meth and:bool(y:bool) self.cond(y,new false)
  meth or:bool(y:bool)  self.cond(new true,y)
  meth not:bool()  self.cond(new false, new true)

class true
  meth cond:bool(x:bool,y:bool) x
  meth and:bool(y:bool) self.cond(y,new false)
  meth or:bool(y:bool)  self.cond(new true,y)
  meth not:bool() self.cond(new false, new true)

class false
  meth cond:bool(x:bool,y:bool) y
```

```
  meth and:bool(y:bool) self.cond(y,new false)
  meth or:bool(y:bool)  self.cond(new true,y)
  meth not:bool()  self.cond(new false, new true)

new bool.or(new false)
```

Example 13: Boolean classes in TKOL

5.4.2 The type environment

The type environment module below specifies that in this system an identifier (ID) would be used for type names. The intention is to use the class name identifier as a named type in the system. A method type is also a TYPE. A method type is the return type of the method and the formal argument type identifier list, i.e., ID (ID-LIST).

During type checking, we need to keep track of the type environments of the variables. For this purpose, we need to associate with every type name, its declared type environment. This will be the information that can be extracted from a given class name (Section 5.4.3), by an analogous construction to that of the value environment of an object, by considering only the type tag information. Therefore the basic type information that would be associated with a type identifier would be a TENV, mapping variable identifiers to type names and method names to method types, i.e., [{IDT ","}*] → TENV.

The [[{IDID ","}*]] is a function that would be useful when type checking a method body. The declaration of formal arguments (with type information) is available as an {IDID ","}* and while type checking a method body, this information should be appropriately added to the current type environment. The function “@” provides the ability to merge two TENVs such that the mappings in the left is preferred over the ones in the right (property of “lookup _ in _” function).

The other functions defined are auxiliary ones. “lookup _ in _” function returns the type associated with the given identifier (ID) in a given type environment (TENV).

module TKOL-tenv

imports Booleans Identifiers TKOL-syntax

exports

sorts TYPE TENV IDT ID-LIST

context-free functions

ID	→ TYPE
ID “(” ID-LIST “)”	→ TYPE
ID “:” TYPE	→ IDT
“[” {IDT “,”}∗ “]”	→ TENV
lookup ID in TENV	→ TYPE
TENV “@” TENV	→ TENV
{ID “,”}∗	→ ID-LIST
“type” {IDID “,”}∗	→ ID-LIST
“[[” {IDID “,”}∗ “]]”	→ TENV
“IF” BOOL “THEN” BOOL “ELSE” BOOL	→ BOOL
empty TENV	→ BOOL

variables

Type [*0*−*9*′]∗ → TYPE

$Tenv\ [0\text{–}9']*$ $\rightarrow$ TENV

$Idt\text{-}list\ [0\text{–}9']*$ $\rightarrow$ {IDT ","}*

$Id\text{-}list\ [0\text{–}9']*$ $\rightarrow$ {TYPE ","}*

equations

[T1] lookup Id in $[Id : Type, Idt\text{-}list] = Type$

[T2] $$\frac{Id \neq Id'}{\text{lookup } Id \text{ in } [Id' : Type, Idt\text{-}list] = \text{lookup } Id \text{ in } [Idt\text{-}list]}$$

[A0] $[Idt\text{-}list]$ @ $[Idt\text{-}list'] = [Idt\text{-}list, Idt\text{-}list']$

[IDD0] $[[]] = []$

[IDD1] $[[Id : Id', IdId\text{-}list]] = [Id : Id']$ @ $[[IdId\text{-}list]]$

[TYO0] type $=$

[TYO1] $$\frac{Id\text{-}list = \text{type } IdId\text{-}list}{\text{type } Id : Id', IdId\text{-}list = Id', Id\text{-}list}$$

[IF0] IF true THEN $Bool$ ELSE $Bool' = Bool$

[IF1] IF false THEN $Bool$ ELSE $Bool' = Bool'$

[E0] empty $[] =$ true

[E1] empty $[Id : Type, Idt\text{-}list] =$ false

5.4.3 The static semantics

Given any two names, we need to be able to specify when one is a subtype of the other, to provide for substitutability of more specific objects where less specific ones were expected.

The basic strategy for the algorithm is to assume the intended subtype relationship and proceed by checking the associated type environments. This is similar to the idea used for checking the structural similarity of Pascal records by Fleck [23].

A list of assumed subtypes is maintained as necessary (SUBTYPES*). Note that the list comprises of only ID < ID assumptions, since we are interested in a lattice of types that results from the class definitions, like in [39].

The Boolean function "assumed __ in _" checks if the SUBTYPE list already has the given assumption (Equations [A0] to [A3]). Equations [T0] to [T2] provide the three cases where (i) two identifiers to be compared are same (subtyping is reflexive) (ii) they are different and it is already assumed, and (iii) they are different but not yet assumed (even the contrary is not assumed). The third case above adds a new assumption and proceeds to check if the associated type environments conform. The equations [T3] and [T4] handle the method types. Note that the arguments of the methods are checked for contra–variance (equation [T4]).

Equations [S0] to [S2] provide for checking the conformance of type environments. According to the rules provided in Section 1.3, every type environment (Tenv) should conform to an empty type environment ([S0]). The type of a variable or method name should conform to its corresponding type ([S1]).

The equation [S2] asserts that an empty type environment does not conform to any non–empty ones. This assertion helps recognize names for which no class are associated (Module TKOL-tc).

The function cid2t[_] gets the type environment associated with a class name from the class definitions (CLASSES) and is described in the module TKOL-tc.

module TKOL-sub

imports TKOL-tenv

exports

sorts SUBTYPE

context-free functions

assume ID “<” ID	→ SUBTYPE
assumed ID “<” ID in SUBTYPE∗	→ BOOL
TYPE “<” TYPE SUBTYPE∗ CLASSES	→ BOOL
TENV “<” TENV SUBTYPE∗ CLASSES	→ BOOL
cid2t “[” ID “]” CLASSES	→ TENV

variables

Stype [*0–9′*]∗	→ SUBTYPE
Stypes [*0–9′*]∗	→ SUBTYPE∗

equations

[T0] $Id < Id\ Stypes\ Cls = \text{true}$

[T1] $$\frac{Id \neq Id', \text{assumed } Id < Id' \text{ in } Stypes = \text{true}}{Id < Id'\ Stypes\ Cls = \text{true}}$$

$$\text{[T2]}\quad \frac{\begin{array}{c} Id \neq Id', \\ \text{false} = \text{assumed } Id < Id' \text{ in } Stypes \mid \text{assumed } Id' < Id \text{ in } Stypes, \\ Tenv = \text{cid2t}[Id]\ Cls,\ Tenv' = \text{cid2t}[Id']\ Cls \end{array}}{Id < Id'\ Stypes\ Cls = Tenv < Tenv' \text{ assume } Id < Id'\ Stypes\ Cls}$$

[T3] $Id() < Id'()\ Stypes\ Cls = Id < Id'\ Stypes\ Cls$

[T4] $Id(Id_0, Id\text{-}list) < Id'(Id0', Id\text{-}list')\ Stypes\ Cls =$
$Id0' < Id_0\ Stypes\ Cls\ \&\ Id(Id\text{-}list) < Id'(Id\text{-}list')\ Stypes\ Cls$

[S0] $Tenv < []\ Stypes\ Cls = \text{true}$

$$\text{[S1]}\quad \frac{Tenv \neq [],\ Tenv' = [Id : Type', Idt\text{-}list],\ Type = \text{lookup } Id \text{ in } Tenv}{\begin{array}{l} Tenv < Tenv'\ Stypes\ Cls \quad = \\ \text{IF } Type < Type'\ Stypes\ Cls \text{ THEN } Tenv < [Idt\text{-}list]\ Stypes\ Cls \text{ ELSE false} \end{array}}$$

[S2] $[] < [Id : Type, Idt\text{-}list]\ Stypes\ Cls = \text{false}$

[A0] assumed $Id < Id'$ in $\ =$ false

[A1] assumed $Id < Id'$ in assume $Id < Id'\ Stypes$ = true

$$\text{[A2]}\quad \frac{Id \neq Id_1}{\begin{array}{l} \text{assumed } Id < Id' \text{ in assume } Id_1 < Id1'\ Stypes \quad = \\ \text{assumed } Id < Id' \text{ in } Stypes \end{array}}$$

$$\text{[A3]}\quad \frac{Id' \neq Id1'}{\begin{array}{l} \text{assumed } Id < Id' \text{ in assume } Id_1 < Id1'\ Stypes \quad = \\ \text{assumed } Id < Id' \text{ in } Stypes \end{array}}$$

The module TKOL-tc described below checks all the classes (function tc[_]), one by one, and then if they all type check allright, it returns the type of the expression (type[_]) following the classes, provided that the expression is type correct ([P1]), with respect to the type information provided in the classes.

The equation [C0] asserts that an empty list of classes is type consistent. Type checking a class is basically to check if the method definitions are consistent ([C1]) using the type environment associated with the class.

A class is type consistent when it has no method definitions ([M0]). The method definition is type consistent when assuming the type declaration of formal arguments ([[IdId-list]] @ Tenv adds this information to the current environment), the method body type checks okay ([M2]).

The use of "self" in an expression forces maintaining "selftype", the name of the class that this environment represents, so that the type of "self" can be obtained when checking a method body (Equations [TE0] and [D2]).

Equations [D0] to [D2] specify the type environment of a given class and equations [CID0] to [CID2] specify the type environment associated with a class name. Equation [CID2] specifies that when a class is not defined an empty environment is considered as its associated type environment. This may not be preferred in languages that provide for incremental type checking, although here this is not of much significance.

The equations [TE0] through [TE5] specify type checking for the six different expression forms. The equations [EST0] and [EST1] are auxiliary equations that help specify equation [TE5], where the actual arguments for a method should be type checked with the formal argument types.

module TKOL-tc

imports TKOL-sub

exports

context-free functions

"tc" "[" PROGRAM "]"	$\rightarrow$ TYPE
"type" "[" EXP "]" TENV CLASSES	$\rightarrow$ TYPE
"tc" "[" CLASSES "]" CLASSES	$\rightarrow$ BOOL
"tc" "[" METHOD$*$ "]" TENV CLASSES	$\rightarrow$ BOOL
"c2t" "[" CLASS "]"	$\rightarrow$ TENV
{EXP ","}$*$ TENV CLASSES "<" ID-LIST	$\rightarrow$ BOOL

variables

Tenv $[0\!-\!9']*$	$\rightarrow$ TENV
Stype $[0\!-\!9']*$	$\rightarrow$ TENV
Cls "+"	$\rightarrow$ CLASS+

equations

[P1] $$\frac{\mathrm{tc}[\mathit{Cls}]\ \mathit{Cls} = \mathrm{true}}{\mathrm{tc}[\mathit{Cls}\ \mathit{Exp}] = \mathrm{type}[\mathit{Exp}]\ []\ \mathit{Cls}}$$

[C0] $\mathrm{tc}[]\ \mathit{Cls} = \mathrm{true}$

[C1] $\mathrm{tc}[\mathrm{class}\ \mathit{Id}\ \mathit{Vdecs}\ \mathit{Meths}]\ \mathit{Cls} =$
$\mathrm{tc}[\mathit{Meths}]\ \mathrm{c2t}[\mathrm{class}\ \mathit{Id}\ \mathit{Vdecs}\ \mathit{Meths}]\ \mathit{Cls}$

[C2] $$\frac{\mathrm{tc}[\mathit{Cl}]\ \mathit{Cls} = \mathrm{true}}{\mathrm{tc}[\mathit{Cl}\ \mathit{Cls}{+}]\ \mathit{Cls} = \mathrm{tc}[\mathit{Cls}{+}]\ \mathit{Cls}}$$

[M0] $\mathrm{tc}[]\ \mathit{Tenv}\ \mathit{Cls} = \mathrm{true}$

$$\text{[M1]}\quad \frac{\text{tc}[\mathit{Meth}]\ \mathit{Tenv}\ \mathit{Cls} = \text{true}}{\text{tc}[\mathit{Meth}\ \mathit{Meths}{+}]\ \mathit{Tenv}\ \mathit{Cls} = \text{tc}[\mathit{Meths}{+}]\ \mathit{Tenv}\ \mathit{Cls}}$$

$$\text{[M2]}\quad \frac{\text{type}[\mathit{Exp}]\ [[\mathit{IdId\text{-}list}]]\ @\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id}'\ \mathit{Cls} = \text{true}}{\text{tc}[\text{meth}\ \mathit{Id} : \mathit{Id}'(\mathit{IdId\text{-}list})\ \mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} = \text{true}}$$

$$\text{[TE0]}\quad \text{type}[\text{self}]\ \mathit{Tenv}\ \mathit{Cls} = \text{lookup selftype in}\ \mathit{Tenv}$$

$$\text{[TE1]}\quad \frac{\mathit{Id} \neq \text{self}}{\text{type}[\mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls} = \text{lookup}\ \mathit{Id}\ \text{in}\ \mathit{Tenv}}$$

$$\text{[TE2]}\quad \text{type}[\text{new}\ \mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls}'\ \text{class}\ \mathit{Id}\ \mathit{Vdecs}\ \mathit{Meths}\ \mathit{Cls}'' = \mathit{Id}$$

$$\text{[TE3]}\quad \frac{\mathit{Id}' = \text{type}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}'\ \text{class}\ \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}\ \mathit{Cls}''}{\begin{array}{l}\text{type}[\mathit{Exp}.\ \mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls}'\ \text{class}\ \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}\ \mathit{Cls}'' \quad = \\ \text{type}[\mathit{Id}]\ \text{c2t}[\text{class}\ \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}]\ \mathit{Cls}'\ \text{class}\ \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}\ \mathit{Cls}''\end{array}}$$

$$\text{[TE4]}\quad \frac{\begin{array}{c}\mathit{Type}' = \text{type}[\mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls}, \\ \mathit{Type} = \text{type}[\mathit{Exp}.\ \mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls}, \\ \mathit{Type}' < \mathit{Type}\ \mathit{Cls} = \text{true}\end{array}}{\text{type}[\mathit{Exp}.\ \mathit{Id} := \mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls} = \text{type}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}}$$

$$\text{[TE5]}\quad \frac{\begin{array}{c}\mathit{Id}_0 = \text{type}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}, \\ \mathit{Tenv}' = \text{cid2t}[\mathit{Id}_0]\ \mathit{Cls}, \\ \text{lookup}\ \mathit{Id}\ \text{in}\ \mathit{Tenv}' = \mathit{Id}_1(\mathit{Id\text{-}list}_1), \\ \mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id\text{-}list}_1 = \text{true}\end{array}}{\text{type}[\mathit{Exp}.\ \mathit{Id}(\mathit{Exp\text{-}list})]\ \mathit{Tenv}\ \mathit{Cls} = \mathit{Id}_1}$$

$$\text{[EST0]}\quad \mathit{Tenv}\ \mathit{Cls} < \ = \text{true}$$

$$[\text{EST3}]\quad \frac{\text{type}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id}\ \mathit{Cls} = \text{true}}{\mathit{Exp},\mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id},\mathit{Id\text{-}list} = \mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id\text{-}list}}$$

[D0] c2t[class Id var Id_0 := new Id' Vdecs Meths] =
[Id_0 : Id'] @ c2t[class Id Vdecs Meths]

[D1] c2t[class Id meth Id' : Id''($\mathit{IdId\text{-}list}$) Exp Meths] =
[Id' : Id''(type $\mathit{IdId\text{-}list}$)] @ c2t[class Id Meths]

[D2] c2t[class Id] = [selftype : Id]

[CID0] cid2t[Id] class Id Vdecs Meths Cls =
c2t[class Id Vdecs Meths]

$$[\text{CID1}]\quad \frac{\mathit{Id} \neq \mathit{Id}'}{\begin{array}{l}\text{cid2t}[\mathit{Id}]\ \text{class}\ \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}\ \mathit{Cls}\ \ =\\ \text{cid2t}[\mathit{Id}]\ \mathit{Cls}\end{array}}$$

[CID2] cid2t[Id] = []

The evaluation modules do not change much. The equations for evaluating the program remain the same. An additional function is added to forget the type information, in the module TKOL-c2o, when an object is created using the class definition. Thus the type tags have no utility in the "run–time" environment, after the type–checking is done at "compile–time".

5.5 A type–free object oriented language, KOOL

The Object Oriented language presented in this section adds the ability to inherit code from a class that was defined earlier. We develop a type-free version so that we can understand the several ways in which type declarations restrict the flexibility of an object oriented language. This language with inheritance is called KOOL.

module KOOL-syntax

imports Layout Identifiers KOL-syntax

exports

sorts RENAME

context-free functions

class ID subclass of ID RENAME∗ VAR-DEC∗ METHOD∗	→ CLASS
rename ID "to" ID	→ RENAME

variables

Cl [*0–9′*]∗	→ CLASS
Cls [*0–9′*]∗	→ CLASS∗
Renames [*0–9′*]∗	→ RENAME∗

equations

$$\text{[F0]}\quad \frac{\begin{array}{c} Cl_0 = \text{class } Id\ \ Vdecs\ \ Meths, \\ Cl_1 = \text{class } Id' \text{ subclass of } Id\ \ Vdecs'\ \ Meths', \\ Cl1' = \text{class } Id'\ \ Vdecs'\ \ Vdecs\ \ Meths'\ \ Meths \end{array}}{Cls\ Cl_0\ Cls'\ Cl_1\ Cls'' = Cls\ Cl_0\ Cls'\ Cl1'\ Cls''}$$

$$\text{[F1]}\quad \frac{\begin{array}{c} Cl_0 = \text{class } Id\ \ Vdecs\ \ Meths_0 \text{ meth } Id_0(Id\text{-}list)\ Exp\ \ Meths_1, \\ Cl_1 = \text{class } Id' \text{ subclass of } Id \text{ rename } Id_0 \text{ to } Id0'\ \ Renames\ \ Vdecs'\ \ Meths', \\ Cl1' = \text{class } Id' \text{ subclass of } Id\ \ Renames\ \ Vdecs'\ \ Meths' \text{ meth } Id0'(Id\text{-}list)\ Exp \end{array}}{Cls\ Cl_0\ Cls'\ Cl_1\ Cls'' = Cls\ Cl_0\ Cls'\ Cl1'\ Cls''}$$

The above equations modify the program text by copying the variables and method definitions as suggested by the construct "subclass of", thus flattening the class hierarchy. The equation [F1] specifies the renaming of methods while the method definitions are copied.

The effect of adding inheritance can be seen by the following recoding of the `boolean`, `true` and `false` classes:

```
class boolean
  meth cond(x,y) x
  meth and(y) self.cond(y,new false)
  meth or(y)  self.cond(new true,y)
  meth not()  self.cond(new false, new true)

class true subclass of boolean

class false subclass of boolean
  meth cond(x,y) y

%%example expression
new true.or(new false).cond(new false,new true)
```

Example 14: Boolean classes in KOOL

Renaming of the inherited methods accomplishes the same effect of using *super* to invoke the methods defined in the super class, as is done in the language Eiffel. We provide for renaming instead of *super* not only to keep the language simple, but also since this is less confusing for a user as to which method is referred to in a long inheritance chain. Bruce [7] notes that understanding inheritance as being equivalent to textual substitution breaks down due to the presence of the keyword *super*. Here we indicate that this intuitive understanding can be aided by eliminating the keyword *super* while maintaing the expressivity provided by super (through renaming as in Eiffel).

An example that renames the inherited methods:

```
%% assume definitions of nat and bool classes

class point
  var x := new nat
  var y := new nat
  meth topof(apoint)                %% whose y is greater
    (apoint.y).less(self.y).cond(self, apoint)

class zpoint subclass of point
  rename topof to ptopof
  var z := new nat
  meth topof(azpoint)               %% whose z and then y
    (azpoint.z).less(self.z)
    .cond(self, self.ptopof(azpoint))

%% example expression
(new zpoint.z := (new nat.incr()))
.ptopof(new zpoint)
```

Example 15: Renaming methods in KOOL

5.6 Type–checking object oriented languages

The following is a discussion of necessary features for a statically type checkable language that intends to accommodate the flexibility advocated by object orientation. We study this by incrementally adding features that account for flexibility.

5.6.1 Simple code copying

A naive copying of code by accordingly renaming some methods can be obtained, like in KOOL-syntax module but by adopting the equation [F1] to handle type tags:

$$[\text{F1}] \quad \frac{\begin{array}{c} Cl_0 = \text{class } Id\ \ Vdecs\ \ Meths_0 \text{ meth } Id_0 : Id_1(IdId\text{-}list)\ Exp\ Meths_1, \\ Cl_1 = \text{class } Id' \text{ subclass of } Id \text{ rename } Id_0 \text{ to } Id0'\ Renames\ Vdecs'\ Meths', \\ Cl1' = \text{class } Id' \text{ subclass of } Id\ Renames\ Vdecs' \\ Meths' \text{ meth } Id0' : Id_1(IdId\text{-}list)\ Exp \end{array}}{Cls\ Cl_0\ Cls'\ Cl_1\ Cls'' = Cls\ Cl_0\ Cls'\ Cl1'\ Cls''}$$

The example

```
%% assume definitions of nat and bool classes
class point
  var x := new nat
  var y := new nat
  meth movex:point(dx:nat)
    self.x := (x.add(dx))

class zpoint subclass of point
```

```
  var z := new nat


%% example expression
new zpoint.movex(new nat).z
```

reduces to the following

```
 class point
  var x := new nat
  var y := new nat
  meth movex : point ( dx : nat ) self . x := x . add ( dx )

 class zpoint
  var z := new nat
  var x := new nat
  var y := new nat
  meth movex : point ( dx : nat ) self . x := x . add ( dx )

 new zpoint . movex ( new nat ) . z
```

Example 16: Providing for type declarations

As a result the program will not type check, according to the rules discussed in Section 5.4.3, since the result of `movex` is a `point` which does not understand `z`.

5.6.2 Using "selftype"

A well understood problem in object oriented programming [43, 17, 47, 49], is that inheritance of code will, in general, require certain type declarations concerning the type of self be specialized accordingly. The commonly used technique is to provide a type declaration facility that can be used to specialize the type information – Eiffel [43] provides for declaration by association (Chapter 4), Palsberg and Schwartzbach [49] discuss the safety of specializing all type declarations accordingly, and Cook et. al. [17] have discussed the notion of F-bounded types that arise in the presence of inheritance. In effect all of them discuss the techniques for specializing subclass type declarations which here can be abstracted by the notion of "selftype".

Let "selftype" be a special type name, which is not associated to any class name like other type names. However, "selftype" will refer to the name of the class in which it is used after the inheritance hierarchy is flattened. A simple form is presented here, but an extension of this would allow "selftype" for formal argument declarations also.

[default-SelfT]

$$\begin{array}{l} \mathit{Cls}\ \mathrm{class}\ \mathit{Id}\ \ \mathit{Vdecs}\ \mathit{Meths}\ \mathrm{meth}\ \mathit{Id}_1 : \mathrm{selftype}(\mathit{IdId\text{-}list})\ \mathit{Exp}\ \mathit{Meths}'\ \mathit{Cls}' = \\ \mathit{Cls}\ \mathrm{class}\ \mathit{Id}\ \ \mathit{Vdecs}\ \mathit{Meths}\ \mathrm{meth}\ \mathit{Id}_1 : \mathit{Id}(\mathit{IdId\text{-}list})\ \mathit{Exp}\ \mathit{Meths}'\ \mathit{Cls}' \end{array}$$

The equation [default-SelfT] applies after other equations have been tried and therefore after the hierarchy of classes is flattened (Appendix C has more elaborate syntax module and this effect is obtained without default equation. Alternatively an auxiliary function can be used to get this effect).

The resulting reduction would be:

```
class point
  var x := new nat
  var y := new nat
  meth movex : point ( dx : nat ) self . x := x . add ( dx )
```

```
class zpoint
  var z := new nat
  var x := new nat
  var y := new nat
  meth movex : zpoint ( dx : nat ) self . x := x . add ( dx )

new zpoint . movex ( new nat ) . z
```

Example 17: Handling selftype return declarations

Note that the inherited `movex` method in `zpoint` returns a zpoint type instead of `point` type.

5.6.3 Rename with type information

Although using the "selftype" specializes the type information as needed, in general, it has a tendency to make the subclass definition either not conform to its super class or to even render the resulting subclass type inconsistent.

In the following example, the `point` class has `x` and `y` co–ordinates (instance variables) a method (`eq`) that checks the correspondence of an another `point`'s `x` and `y` values. `point` also has method that allows one to return a `point` instance. A method like this could be provided for posterity, where the subclass can, for efficiency reasons, return a mere point.

A typical subclass would add some new variable (as in `zpoint`) and modify its `eq` method appropriately. In our case, the `myxy()` value is sent to the argument to check for its equality. This is a contrived example, but in large software systems something like this is very likely for many reasons. The redefinition of the `eq` method would naturally intend to reuse the `eq` behavior specified in the super class and thus would rename the super class `eq` method to `peq`.

```
%% with appropriate definitions of nat and bool classes
class point
  var x := new nat
  var y := new nat
  meth eq:bool(pt:selftype)
      x.eq(pt.x).and(y.eq(pt.y))
  meth myxy:point()
      (new point.x := x).y := y

class zpoint subclass of point
  rename eq to peq
  var z := new nat
  meth eq:bool(zpt:selftype)
      z.eq(zpt.z).and(zpt.peq(self.myxy()))

%% example expression
(new point).eq(new zpoint)
```

Assuming that "selftype" declaration was supported for arguments of a method, the intended reduction would be:

```
class point
  var x := new nat
  var y := new nat
```

```
  meth eq:bool(pt:point)
      x.eq(pt.x).and(y.eq(pt.y))
  meth myxy:point()
      (new point.x := x).y := y

class zpoint
  var z := new nat
  var x := new nat
  var y := new nat
  meth eq:bool(zpt:zpoint)
      z.eq(zpt.z).and(zpt.peq(self.myxy()))
  meth peq:bool(pt:zpoint)
      x.eq(pt.x).and(y.eq(pt.y))
  meth myxy:point()
      (new point.x := x).y := y

(new point).eq(new zpoint)
```

Example 18: Handling selftype argument declarations

Firstly, we note that the argument type of `eq` as changed in a covariant manner. Thus, `zpoint` would not be a subtype of `point` and since this is a natural manner in which classes seem to evolve [43, 16, 17] subtype hierarchy should in general be separate from inheritance hierarchy (For example, as in our case here). Chapters 3, 4 and 6 discuss other techniques to help enforce these hierarchies to correspond.

The reason for contriving the example with method `myxy`, however, is to indicate that the resulting type tagging of class `zpoint` would render it type inconsistent! Note that the method `peq` expects a `zpoint` type argument, but in the redefined method `eq` it is invoked with a `point` type argument.

This example thus indicates that, although it might lead to a natural way of specializing super class definition by using "selftype" for argument declaration, one should be cautious reusing super class methods. An extreme way of handling this kind of specialization can be seen in [49], where all mention of "`point`" would be read as "`zpoint`" in the class `zpoint`. Although this would render such cases type safe, it would force specialization when not intended.

Therefore, instead of automatically specializing argument type declaration, we provide for renaming of methods with possibly changing the type declaration of arguments.

The language presented in Appendix 6 includes this facility. The above example can be made type safe as follows:

```
%% with appropriate definitions of nat and bool classes
class point
  var x := new nat
  var y := new nat
  meth eq:bool(pt:point)
      (self.x).eq(pt.x).and((self.y).eq(pt.y))
  meth myxy:point()
      (new point.x := self.x).y := self.y

class zpoint subclass of point
  rename eq to peq:point -> bool
```

```
  var z := new nat
  meth eq:bool(zpt:zpoint)
      (self.z).eq(zpt.z).and(zpt.peq(self.myxy()))

%% example expression
(new point).eq(new zpoint)
```

Example 19: Changing method type declarations

The reduced `zpoint` class will thus be:

```
class zpoint subclass of point
  var z := new nat
  var x := new nat
  var y := new nat
  meth eq:bool(zpt:zpoint)
      (self.z).eq(zpt.z).and(zpt.peq(self.myxy()))
  meth peq:bool(pt:point)
      (self.x).eq(pt.x).and((self.y).eq(pt.y))
  meth myxy:point()
      (new point.x := self.x).y := self.y
```

The type redeclaration for the method `peq` was not necessary for the above case, but was included for illustration.

5.7 Updating to language ASOOL

The EXP of KOOL (as in Section 5.3.1) is extended with $(ID := EXP)$ and $(EXP; EXP)$. With these additions to the syntax, we hope be able to write programs that resemble those of conventional object–oriented languages that use assignments and statements [43, 26]. The following additions to language KOOL results in ASOOL.

module ASOOL-syntax

imports KOOL-syntax

exports

context-free functions

ID ":=" EXP $\rightarrow$ EXP

EXP ";" EXP $\rightarrow$ EXP

priorities

";" < ":=" < {EXP "." ID "(" {EXP ","}* ")" $\rightarrow$ EXP } < "."

module ASOOL-eval

imports KOOL-eval

equations

$$\text{[A1]}\quad [[\mathit{Id} := \mathit{Exp}]]\text{e } \mathit{Self}\ \ \mathit{Cls} = [[\text{self. } \mathit{Id} := \mathit{Exp}]]\text{e } \mathit{Self}\ \ \mathit{Cls}$$

$$\text{[A2]}\quad \frac{\mathit{Self}' = [[\mathit{Exp}]]\text{e } \mathit{Self}\ \ \mathit{Cls}}{[[\mathit{Exp};\ \mathit{Exp}']]\text{e } \mathit{Self}\ \ \mathit{Cls} = [[\mathit{Exp}']]\text{e } \mathit{Self}'\ \ \mathit{Cls}}$$

The effect of adding assignments can be seen in the `incr()` and `decr()` methods of the class `nat` (Section 5.3.1).

```
class nat
  var isz := new true
  var pre := nil
  meth iszero() isz
  meth pred() pre
  meth incr() pre := self ;
      isz := new false
  meth decr() isz.cond(self, isz := pre.iszero();
                             pre := pre.pred() )
```

Example 20: Using assignments in class nat

The following example adds movement to points. The step in point takes an x increment, a y increment and updates the x and y of self to result in a point with the new x and y values. The zstep uses the step method to define a zstep which updates x, y and z.

```
class point
  var x := new nat
  var y := new nat
  meth step(dx,dy)
    x := x.add(dx) ;      %% assume add is defined in class nat
    y := y.add(dy)

class zpoint subclass of point
  var z := new nat
  meth zstep(dx,dy,dz)
    z := z.add(dz) ; self.step(dx,dy)
```

```
new zpoint.step(new nat.incr(), new nat)
```

Example 21: Using assignments to move points

The above examples and the equations demonstrate the syntactical utility of adding the ":=" and ";". Many resulting expressions can be thought of as a convenient alternative to its directly corresponding expression in KOOL (Also see Section 3.2 for a simple correspondence). For example, the body of method `step` can be rewritten as:

```
(self.x := x.add(dx)).y := y.add(dy)
```

and that of `zstep` as:

```
(self.z := z.add(dz)).step(dx,dy)
```

We should note, however, that the equation [A2] replaces the current *Self* environment with the modified version after evaluating the expression before ";". Therefore a method body like

```
x := f(y) ; z := g(x,x)
```

would be equivalent to

```
self.z := g((self.x := f(y)).x, (self.x := f(y)).x)
```

Since `self` is accessible to the user for manipulation, we can also view ";" as mechanism that gives the effect of a `let _ in` construct that redefines `self`. The meaning of *Exp ; Exp′* (equation [A2]) can thus be stated instead as:

`let self =` *Exp*

`in` *Exp′*

Also of interest is the behavior of how instance variables get modified verses how the formal arguments get modified, during a recursive call. The instance variables bindings are "permanently modified" where as the formal argument bindings are restored to those in its dynamic scope, when a recursive call returns (Section 5.3.3).

CHAPTER 6

AN EIFFEL LIKE LANGUAGE

The "subclass is subtype" notion and "like *Current*" facility provided in Eiffel are considered in the language presented in this chapter. The techniques described in Chapter 4 are considered for the language EOOL, to specify the algebraic semantics of such a language. The modules presented in Appendix C specify the process of tagging, type-checking, transforming the program and the components needed for the dynamic semantics.

We discuss two different methods of achieving adoption in such a language. *Intrusive-adoption* insists that the objects that are typed like self must be like self. *Non-intrusive-adoption* only donates the necessary behavior when an object that is supposed to be like self cannot understand certain messages. We also discuss the precautions and the limitations.

6.1 The user level syntax

The ASOOL language (Section 5.7) developed in Chapter 5 is extended with predefined sort "natural", to aid in writing concise example programs that illustrates the details of the specification modules used.

The EOOL-syntax module (Appendix C) describes the user level syntax of such a language. The syntax is similar to that of the language ASOOL, but the subtype conformance and use of selftype will be realized similar to that found in Eiffel. The following program illustrates a typical specialization. The class `use` is modified after `point` is specialized to `zpoint` to the class `newuse`.

```
class point
  var x := 10
  var y := 20
  meth gt:point (pt:selftype)
    if x > pt.x then
      if y > pt.y then self
      else pt fi
    else pt fi

class use
  meth test:point() new point

class zpoint subclass of point
  rename gt to pgt:zpoint -> zpoint
  var z := 1000
  meth gt:zpoint(zpt:selftype)
     if (z + 10) > zpt.z then zpt.pgt(self) else self fi

class newuse subclass of use
  meth test:point() new zpoint

new newuse. test(). gt (new point)
```

Example 22: A case for adoption

6.1.1 Type-checking

The word “selftype” can be used as type name where ever a type declaration is expected (for both argument types and return types of methods), unlike only for return types as in Chapter 5. The use of “selftype” in this manner results in the so-called inherited types [17] when the class gets inherited.

The conformance relation, similar to that of Eiffel, is realized in module EOOL-sub (Appendix C.1.4). The functions are defined similar to those in Chapter 5, except that the preorder information defined by the inheritance hierarchy is used instead the preorder relation that could be deduced. For A to conform to B, we now require in addition to earlier conformance rules (i) A inherits from B and (ii) if “selftype” is used to declare an argument of a method, then “selftype” should be used in a subsequent redeclaration/redefinition of the method.

Type-checking module is similar to those presented earlier (Sections 5.4.3). The function that checks if the types inferred for the actual arguments of a method are conforming to the type declaration also translates the identifier “selftype” to the appropriate type name.

Type-checking the earlier example program returns `point` indicating the resulting type of the expression, and thus indicating that the program is type correct.

The type-information could either be discarded at this point or not depending on whether the adoption technique needs it needs it or not. The adoption technique, as specified in the module in Section 6.4 does not use “selftype” or have any need for type information, while the one in Section 6.3 uses the selftype information.

6.2 Dynamic semantics with adoption

A program evaluated without proper compensation for relaxing contravariance for "selftype" arguments, will evaluate with the possibility of not understanding some messages (in-spite of it being type-checked) and is therefore comparable to the earlier implementations of Eiffel, as discussed in [16].

Using type information for providing alternate behavior has also been popularized, in the name of multi-methods [36, 25], but the resulting semantics of such process is complementary to what can be achieved through adoption. Also the process of adoption need not add the over-head of type-checking during run-time.

Evaluation of the program above (after transformation) results in the following, indicating that the semantics for the evaluation is not defined completely in the presence of "selftype" argument declaration.

```
value
[[ if z _ zpoint + 10 > zpt . z _ zpoint
  then zpt . pgt _ point ( self )
  else self fi ]]
  [ ( zpt : [ ( x _ point : 10 ),
              ( y _ point : 20 ),
              ( gt _ point : ( pt ) if x _ point > pt . x _ point
                                    then ...
                                    else pt fi ) ] ) ]
    &
    [ ( z _ zpoint : 1000 ),
      ( x _ point : 10 ),
      ( y _ point : 20 ),
```

```
    ( gt _ point : ( zpt ) if z _ zpoint + 10 > zpt . z _ zpoint
                           then zpt . pgt _ point ( self )
                           else self fi ),
    ( pgt _ point : ( pt ) if x _ point > pt . x _ point
                           then ...
                           else pt fi ) ]
class point ...  class newuse ...
```

Note that `zpt` object in the value environment does not have any method named `z _ zpoint`, since this is an instance of class point.

6.3 Intrusive adoption semantics

First we consider, a simple adoption where the meaning of word selftype is taken literally and the dynamic semantics changes the arguments appropriately to be like self. This interpretation would give an orphaned object exactly the inheritance semantics. So, if a point comes into the context of zpoint (where a zpoint is expected) the zpoint methods override point methods. In this case, it also imposes zpoint methods to an object (where a selftype object was expected) even if a more specialized object appears in the context.

Let us assume that a marker is placed in the argument list where a "selftype" declaration is found. In the following we have (adopt Id) to indicate that the formal argument Id was declared as "selftype". The semantics of message send has to interpret this appropriately to make the actual value appear like the receiver of the message. However, the instance variable values will be maintained as originally found. Thus we get an adoption where the basic values are retained while the methods are imposed by the context.

The following module assumes the familiar evaluation semantics except for the general message passing case. The message passing semantics will use the context information to change actual argument object that is passed where a selftype object is expected.

module EOOL-adopt$_I$

imports EOOL-common$_I$

exports

context-free functions

"(" ID-LIST ")" ":" "(" VALUE-LIST ")" context VENV	$\rightarrow$ VENV
VENV adopt VENV	$\rightarrow$ VENV

equations

$$[\text{EA-E2}]\quad \frac{\begin{array}{c} \mathit{Oenv} = [[\mathit{Exp}]]\ \mathit{Self}\ \ \mathit{Cls}, \\ \text{lookup } \mathit{Name} \text{ in } \mathit{Oenv} = (\mathit{Id\text{-}list})\ \mathit{Exp}', \\ \mathit{A\text{-}args} = [[\mathit{Exp\text{-}list}]]{*}\ \mathit{Self}\ \ \mathit{Cls}, \\ \mathit{Oenv}' = (\mathit{Id\text{-}list}) : (\mathit{A\text{-}args}) \text{ context value } \mathit{Oenv}, \\ \mathit{Value} = [[\mathit{Exp}']]\ \mathit{Oenv}'\ \&\ \mathit{Oenv}\ \ \mathit{Cls} \end{array}}{[[\mathit{Exp}.\ \mathit{Name}(\mathit{Exp\text{-}list})]]\ \mathit{Self}\ \ \mathit{Cls} = \text{value } \mathit{Value}}$$

$$[\text{EA-X0}]\quad () : () \text{ context } \mathit{Oenv} = []$$

$$[\text{EA-X1}]\quad \frac{Id = (\text{adopt } Id'),\ Venv = Value}{\begin{array}{l}(Id, Id\text{-}list) : (Value, Value\text{-}list) \text{ context } Oenv \quad = \\ [(Id' : Oenv \text{ adopt } Venv)] \mathbin{@} (Id\text{-}list) : (Value\text{-}list) \text{ context } Oenv\end{array}}$$

$$\begin{array}{ll}[\text{default-EA-X2}] & (Id, Id\text{-}list) : (Value, Value\text{-}list) \text{ context } Oenv = \\ & [(Id : Value)] \mathbin{@} (Id\text{-}list) : (Value\text{-}list) \text{ context } Oenv\end{array}$$

$$[\text{EA-A0}]\quad \frac{Oenv' = [(Id : Venv)] \mathbin{@} Oenv}{\begin{array}{l}Oenv \text{ adopt } [VP\text{-}list_3, (Id : Venv), VP\text{-}list_4] \quad = \\ Oenv' \text{ adopt } [VP\text{-}list_3, VP\text{-}list_4]\end{array}}$$

$$[\text{default-EA-A1}] Oenv \text{ adopt } Venv Oenv \mathbin{@} Venv$$

6.4 Non-intrusive adoption semantics

We now consider adoption that is non-intrusive, which prefers the methods of the orphan object over any of the methods in the context. An immediate effect of this is that an object that is already specialized enough for the context will retain its behavior. So the context provides only the necessary additional behavior needed for an object to exist in the specialized context.

The three cases that require handling messages (instance variable lookup, update and method invocation) are considered here. The basic strategy used is to donate a method or instance variable of *Self* environment when some message is not understood by an object. Although this appears to be too general, the type-checking section (Section C.1) confines this special behavior to the argument types labeled “selftype”, as this is the only place that the usual notion of subtype conformance rules is relaxed. Therefore, the type-checking rules with the conformance rules (Modules EOOL-tc and EOOL-sub) will ensure the messages sent to objects that are not "selftypes" will be understood.

The following module extends the evaluation semantics by defining additional semantics for the message not understood (lookup *Name* in *Oenv* = false) case.

module EOOL-adopt

imports EOOL-eval

equations

$$\text{[EA-Vfix]}\quad \frac{\mathit{Venv}' = \mathit{Venv}_0\ \&\ \mathit{Venv}_1,\ \mathit{Venv}_2 = \mathit{Venv}\ @\ \mathit{Venv}_1}{\mathit{Venv}\ @\ \mathit{Venv}' = \mathit{Venv}_0\ \&\ \mathit{Venv}_2}$$

$$\text{[EA-M0]}\quad \frac{\begin{array}{c}\mathit{Oenv} = [[\mathit{Exp}]]\ \mathit{Self}\ \ \mathit{Cls},\ \mathit{Name}\ \text{isin}\ \mathit{Oenv} = \text{false},\\ \text{lookup}\ \mathit{Name}\ \text{in}\ \mathit{Self} = (\mathit{Id\text{-}list})\ \mathit{Exp}',\\ \mathit{Self}' = \text{value}\ \mathit{Self},\ \mathit{Oenv}' = \mathit{Oenv}\ @\ \mathit{Self}',\\ \mathit{A\text{-}args} = [[\mathit{Exp\text{-}list}]]*\ \mathit{Self}\ \ \mathit{Cls},\\ \mathit{Value} = [[\mathit{Exp}']]\ (\mathit{Id\text{-}list}) : (\mathit{A\text{-}args})\ \&\ \mathit{Oenv}'\ \mathit{Cls}\end{array}}{[[\mathit{Exp}.\ \mathit{Name}(\mathit{Exp\text{-}list})]]\ \mathit{Self}\ \ \mathit{Cls} = \text{value}\ \mathit{Value}}$$

$$\text{[EA-E0]}\quad \frac{\begin{array}{c}\mathit{Oenv} = [[\mathit{Exp}]]\ \mathit{Self}\ \ \mathit{Cls},\\ \mathit{Name}\ \text{isin}\ \mathit{Oenv} = \text{false},\ \mathit{Self}' = \text{value}\ \mathit{Self}\end{array}}{[[\mathit{Exp}.\ \mathit{Name}]]\ \mathit{Self}\ \ \mathit{Cls} = [[\mathit{Name}]]\ \mathit{Oenv}\ @\ \mathit{Self}'\ \mathit{Cls}}$$

$$\text{[EA-A5]}\quad \frac{\begin{array}{c}\mathit{Oenv} = [[\mathit{Exp}]]\ \mathit{Self}\ \ \mathit{Cls},\\ \mathit{Name}\ \text{isin}\ \mathit{Oenv} = \text{false},\ \mathit{Self}' = \text{value}\ \mathit{Self}\end{array}}{\begin{array}{l}[[\mathit{Exp}.\ \mathit{Name} := \mathit{Exp}']]\ \mathit{Self}\ \ \mathit{Cls}\ \ =\\ \text{update}\ \mathit{Name}\ \text{in}\ \mathit{Oenv}\ @\ \mathit{Self}'\ \text{by}\ [[\mathit{Exp}']]\ \mathit{Self}\ \ \mathit{Cls}\end{array}}$$

All three equations ([EA-M0, EA-E0 and EA-A5]) use the technique of donating the *Self* environment for the object in context, when an object *does not understand* a message sent to it. The context provides an interpretation for the method that is not understood, and also offers all its information to be default for the orphaned object. These donated capabilities of *self* may be used by the object (*Oenv @ Self'* above) for the rest of its life-time. This particular adoption technique preserves the objects own attributes, while adding new ones only on a demand for specialization. This process also means that an object will need to adopt to a particular context only once, and the type-checking process eliminates meaningless adoptions in general.

Evaluating the example program with adoption semantics results in:

```
[ ( z _ zpoint : 1000 ),
  ( x _ point : 10 ),
  ( y _ point : 20 ),
  ( gt _ point : ( zpt )
     if z _ zpoint + 10 > zpt . z _ zpoint
     then zpt . pgt _ point ( self ) else self fi ),
  ( pgt _ point : ( pt )
     if x _ point > pt . x _ point
     then if y _ point > pt . y _ point then self else pt fi
     else pt fi ) ]
```

However, if the body of the method gt in zpoint was instead `zpt.z := 30` (update argument z and return result), then the result using adoption semantics is

```
[ ( z _ zpoint : 30 ),
```

```
( x _ point : 10 ),
( y _ point : 20 ),
( gt _ point : ( pt )
   if x _ point > pt . x _ point
   then if y _ point > pt . y _ point then self else pt fi
   else pt fi ),
( pgt _ point : ( pt )
   if x _ point > pt . x _ point
   then if y _ point > pt . y _ point then self else pt fi
   else pt fi ) ]
```

Note that the `zpoint` attributes have been acquired by the object, while the `gt` method remains to be that defined in `point` class. This indicates that only the minimal additional knowledge required to exist in a more specialized context is gained by the object in the process of this adoption. This results in objects that do not conform to being instances of any particular class, and therefore a proper understanding of the effects by the users is called for. A general strategy of defining new methods and using the new methods in altering the method definitions, and using proper default initial values appears to aid in specializing the classes in a useful manner.

6.4.1 Adoptability conditions

The non-intrusive adoption semantics requires careful reconsideration of the conformance rules. We need to make sure that an adopted object does not appear in a totally different context where a more speacilized object was expected.

The following example illustrates the problem:

```
class point
```

```
  var x := 10
  var y := 20
  meth gt:natural(pt:selftype) pt.x
  meth test0:selftype() new point

class puse
  meth some:natural(arg1:point,arg2:point) arg1.gt(arg2)
  meth other:natural(pt:spoint) pt.test0().z

class spoint subclass of point
  var z := 1000
  meth test0:selftype() new spoint
  meth gt:natural(spt:selftype)
       if z > spt.z then new puse.other(spt)
       else spt.test0().z fi

new puse . some(new spoint,new point)
```

Example 23: A case for adoptability check

The `other` method defined in the class `puse` requires that the argument be of type `spoint`. However, a `point` object, adopted to the `spoint` context will in general not conform to `spoint` type. In this case, after the non-intrusive adoption, the resulting object still prefers the `test0` method of `point` which returns a `new point` (which does not understand the message `z`). Therefore this case needs to be restricted and the argument of the `other` method should be at most a `point`. The module EOOL-adoptable in the Appendix C provides for these additional checks that are necessary to make the program safe in the presence of non-intrusive adoption.

We considered inheritance, with covariant arguments in the methods, as a mechanism to specialize the inherited class to a more specific context. The additional restriction on the type-checking rules intended to preserve this meaning of inheritance by requiring that the adopted object not be sent to a class where a specialized object was desired. The rules in module EOOL-adoptable check for potentially adopted objects going out of the context of adoption as a specialized object, and restrict this case.

With the above rules, the `other` method will at most have the argument type to be `point`:

```
meth other:natural(pt:point) pt.test0().x
```

CHAPTER 7

CONCLUSIONS

7.1 Summary

Among the implementors of object-oriented languages, it is known [52, 33, 47] that the receiver of the message can be used as the first argument of an object's methods to access its other methods. We have used this idea to develop the semantics of delegation with recursive templates. Delegation without templates is provided for in [47]. Templates and some form of inheritance are distinct features an object-oriented language must provide, as emphasized by [56] by comparing their individual works (DELEGATION [41], SELF [59] and [55]). The mechanisms we have provided are a generalization of all these facilities which includes those of inheritance as in SmallTalk [26] (given by [51] and [15]).

We have concluded that in the presence of delegation and templates, every object needs to be paired with its template. We have used [15] semantics of inheritance to discuss what the semantics of delegation should be. The resulting mechanisms use a more familiar interpretation of message passing than that developed in [15], and we have also shown the inadequacy of the constructions of [15] to accommodate both delegation and recursive templates found in object-oriented programming languages.

We then considered a class based language that provides for delegation in a limited manner (SOOL — Section 3.2) to only allow for updating of *self.* The examples provided in Chapter 3 illustrate the natural way of writing programs, that this language facilitates, by considering an example where updating of self is desired. Language SOOL is an instance that provides for delegation capabilities in a simple class based language. We then discussed a way to manipulate the message passing mechanism of SOOL to achieve flexibility that inheritance constructions alone cannot provide, using an associative delegation operation as the basis. Requirements are outlined for a type checking system that could control the nature and extent of manipulations of the message passing mechanism, with facilities of the language. This is similar to adding the required coercion functions by the type-checker during compilation in languages like Pascal [46]. Particular instances where some of the restrictions found in statically type checkable object oriented languages, could be relaxed are identified and discussed. This discussion, in turn, gives some insight to when covariance would be appropriate. Finally, a technique to modify an entire hierarchy without modifying the existing code is discussed.

In Chapter 4 we consider the examples provided by Cook [16] to show the type holes in Eiffel, and suggest an alternate way of resolving these problems using the general framework of Chapter 3. Cook's proposal to make Eiffel type-safe has been basically to eliminate the flexibility of Eiffel for most part. The effect of our proposal is to provide for the most desirable semantics for the messages that would not be understood in the current implementations. We advocate them as "most desirable" by considering system evolution, efficiency and upwards compatibility of the existing Eiffel programs. Only time would tell the actual implication of these proposals on real software system evolution.

We next concentrate on completely specifying a language, and thereby obtain a prototype implementation, that incorporates proposed facilities using the algebraic specification formalism ASF+SDF. The algebraic specification mechanism allows us to investigate alternative ways of describing inheritance and delegation with equational specifications. We develop inheritance in the intuitive sense of code substitution by providing for an alternate way of describing the effect of *super*. The ideas and problems addressed by statically typed object oriented languages are discussed using a set of progressive languages in Chapter 5. The intention is to provide an appropriate perspective on the dynamic and static semantic issues present in object oriented languages and to provide a basis for the prototype implementation of our experimental language presented in Chapter 6.

It is well understood that inheritance does not lead to subtypes, but to the so called *inherited types* [17]. We finish with a case of study analysis of this view. Since the language Eiffel provided for the use of inherited types as subtypes [43], we consider a language (EOOL) similar to Eiffel in that subclasses are subtypes. We then use the ideas developed in Chapter 3 to adopt the orphan objects created in the process of identifying subclasses and subtypes. Chapter 6 gives a prototype of such a language with examples demonstrating both the effects of using and not using the module EOOL-adopt (adoption semantics). These examples, and the language description, illustrates the usefulness of adoption and the care that must be taken to accomplish adoption.

7.2 Conclusions

The community of object oriented programmers can be partitioned into to two groups, depending on whether they would like to use subclasses as subtypes or not. The general tendency being that the subclass hierarchies must be separated from subtype hierarchies in general, since there have been many examples that are demonstrative of cases where subclasses are not subtypes. It also agreed in general that there are many examples where the most natural way to specialize a class does not conform to a subtype definition, but it is quite useful if it can be treated as a subtype in spite of the complications that this causes.

Therefore, an alternate way of perceiving the notion of subclasses as subtypes is desirable and is investigated in this thesis. This encourages one to understand the role of class specialization that can be achieved through inheritance, and reason for the desire of using *inherited types* as *subtypes*. The approach taken here of providing a desirable and possibly user specifyable semantics for how the orphan objects (that such an identification would be creating) should be adopted, will also ask of a covariant desiring user to think of what s/he desires. Thus we feel that this alternative is in the direction of a good compromise if and when there is a desire for *covariance*.

7.3 Future work

One of the adoption techniques investigated, in Chapter 6, can be termed as a non-intrusive non-imposing demand-driven adoption technique. Other alternative adopting and/or adapting techniques should be studied, that relax the adoptability conditions according to different needs (Appendix C).

The practicality of the techniques suggested in Chapter 3 should also be studied in its most general sense. The work done by Cardelli, Mitchell and Bruce [10, 13, 7] appears to be the direction to go in understanding the limitations of the proposal in Chapter 3.

A better understanding of how this can be achieved when the subtype hierarchies and subclass hierarchies are completely separate is necessary. Also, the necessary precautions that should be taken in the presence of mutation of state and in the presence of hidden methods needs to be investigated.

Development of large software in the presence of adoption and efficient implementation of adoption also need to be considered before such a facility can be advocated for a large programming community.

APPENDIX A

THE LAYOUT, BOOLEANS AND NAT MODULES

These modules are commonly used and are part of ASF+SDF library.

module Layout

exports

lexical syntax

"%%" ~[\n]∗ [\n]	$\rightarrow$ LAYOUT
[\t\n]	$\rightarrow$ LAYOUT

module Booleans

imports Layout

exports

sorts BOOL

context-free functions

true	$\rightarrow$ BOOL
false	$\rightarrow$ BOOL
BOOL "|" BOOL	$\rightarrow$ BOOL {**left**}
BOOL "&" BOOL	$\rightarrow$ BOOL {**left**}
not "(" BOOL ")"	$\rightarrow$ BOOL
"(" BOOL ")"	$\rightarrow$ BOOL {**bracket**}

variables

$Bool\ [0\text{–}9']*\quad \rightarrow$ BOOL

priorities

"|" < "&"

equations

[B1] true | $Bool$ = true

[B2] false | $Bool$ = $Bool$

[B3] true & $Bool$ = $Bool$

[B4] false & $Bool$ = false

[B5] not(false) = true

[B6] not(true) = false

module Nat

imports Layout Booleans

exports

sorts NAT-CON NAT INT

lexical syntax

$[0-9]+\quad \rightarrow$ NAT-CON

context-free functions

NAT-CON	$\rightarrow$ NAT
"(" NAT ")"	$\rightarrow$ NAT {**bracket**}
NAT "+" NAT	$\rightarrow$ NAT {**left**}
NAT ">" NAT	$\rightarrow$ BOOL
NAT	$\rightarrow$ INT

variables

$Nat\text{-}con$ $[0{-}9']*$	$\rightarrow$ NAT-CON
Nat $[0{-}9']*$	$\rightarrow$ NAT

hiddens

context-free functions

gt "(" NAT "," NAT ")"	$\rightarrow$ BOOL
NAT "$-//$" NAT	$\rightarrow$ NAT

variables

n $[0{-}9]*$	$\rightarrow$ NAT-CON
c $[0{-}9]*$	$\rightarrow$ CHAR
x $[0{-}9]*$	$\rightarrow$ CHAR*
y $[0{-}9]*$	$\rightarrow$ CHAR+

equations

[1] nat-con("0" y) = nat-con(y)

[2] $0 + Nat = Nat$

[3] $Nat + 0 = Nat$

[4] $1 + 1 = 2$	[5] $1 + 2 = 3$	[6] $1 + 3 = 4$
[7] $1 + 4 = 5$	[8] $1 + 5 = 6$	[9] $1 + 6 = 7$
[10] $1 + 7 = 8$	[11] $1 + 8 = 9$	[12] $1 + 9 = 10$
[13] $2 + 1 = 3$	[14] $2 + 2 = 4$	[15] $2 + 3 = 5$
[16] $2 + 4 = 6$	[17] $2 + 5 = 7$	[18] $2 + 6 = 8$
[19] $2 + 7 = 9$	[20] $2 + 8 = 10$	[21] $2 + 9 = 11$
[22] $3 + 1 = 4$	[23] $3 + 2 = 5$	[24] $3 + 3 = 6$
[25] $3 + 4 = 7$	[26] $3 + 5 = 8$	[27] $3 + 6 = 9$
[28] $3 + 7 = 10$	[29] $3 + 8 = 11$	[30] $3 + 9 = 12$
[31] $4 + 1 = 5$	[32] $4 + 2 = 6$	[33] $4 + 3 = 7$
[34] $4 + 4 = 8$	[35] $4 + 5 = 9$	[36] $4 + 6 = 10$
[37] $4 + 7 = 11$	[38] $4 + 8 = 12$	[39] $4 + 9 = 13$
[40] $5 + 1 = 6$	[41] $5 + 2 = 7$	[42] $5 + 3 = 8$
[43] $5 + 4 = 9$	[44] $5 + 5 = 10$	[45] $5 + 6 = 11$
[46] $5 + 7 = 12$	[47] $5 + 8 = 13$	[48] $5 + 9 = 14$
[49] $6 + 1 = 7$	[50] $6 + 2 = 8$	[51] $6 + 3 = 9$

[52] $6 + 4 = 10$ [53] $6 + 5 = 11$ [54] $6 + 6 = 12$

[55] $6 + 7 = 13$ [56] $6 + 8 = 14$ [57] $6 + 9 = 15$

[58] $7 + 1 = 8$ [59] $7 + 2 = 9$ [60] $7 + 3 = 10$

[61] $7 + 4 = 11$ [62] $7 + 5 = 12$ [63] $7 + 6 = 13$

[64] $7 + 7 = 14$ [65] $7 + 8 = 15$ [66] $7 + 9 = 16$

[67] $8 + 1 = 9$ [68] $8 + 2 = 10$ [69] $8 + 3 = 11$

[70] $8 + 4 = 12$ [71] $8 + 5 = 13$ [72] $8 + 6 = 14$

[73] $8 + 7 = 15$ [74] $8 + 8 = 16$ [75] $8 + 9 = 17$

[76] $9 + 1 = 10$ [77] $9 + 2 = 11$ [78] $9 + 3 = 12$

[79] $9 + 4 = 13$ [80] $9 + 5 = 14$ [81] $9 + 6 = 15$

[82] $9 + 7 = 16$ [83] $9 + 8 = 17$ [84] $9 + 9 = 18$

[85a] $$\frac{\text{nat-con}(c_1) + \text{nat-con}(c_2) = \text{nat-con}(x\ c),\quad \text{nat-con}(y_1) + \text{nat-con}(y_2) + \text{nat-con}(\text{“0”}\ x) = \text{nat-con}(y)}{\text{nat-con}(y_1\ c_1) + \text{nat-con}(y_2\ c_2) = \text{nat-con}(y\ c)}$$

[85b] $$\frac{\text{nat-con}(c_1) + \text{nat-con}(c_2) = \text{nat-con}(x\ c),\quad \text{nat-con}(\text{“0”}\ x_1) + \text{nat-con}(y_2) + \text{nat-con}(\text{“0”}\ x) = \text{nat-con}(y)}{\text{nat-con}(x_1\ c_1) + \text{nat-con}(y_2\ c_2) = \text{nat-con}(y\ c)}$$

$$\frac{\text{nat-con}(c_1) + \text{nat-con}(c_2) = \text{nat-con}(x\ c), \quad \text{nat-con}(y_1) + \text{nat-con}(\text{“0”}\ x_2) + \text{nat-con}(\text{“0”}\ x) = \text{nat-con}(y)}{\text{nat-con}(y_1\ c_1) + \text{nat-con}(x_2\ c_2) = \text{nat-con}(y\ c)} \quad [85c]$$

[86] $\text{nat-con}(c) -\!/\!/\ \text{nat-con}(c) = 0$

[87] $\text{nat-con}(c) -\!/\!/\ 0 = \text{nat-con}(c)$

[88] $2 -\!/\!/\ 1 = 1$

[89] $3 -\!/\!/\ 1 = 2$ [90] $3 -\!/\!/\ 2 = 1$

[91] $4 -\!/\!/\ 1 = 3$ [92] $4 -\!/\!/\ 2 = 2$ [93] $4 -\!/\!/\ 3 = 1$

[94] $5 -\!/\!/\ 1 = 4$ [95] $5 -\!/\!/\ 2 = 3$ [96] $5 -\!/\!/\ 3 = 2$

[97] $5 -\!/\!/\ 4 = 1$

[98] $6 -\!/\!/\ 1 = 5$ [99] $6 -\!/\!/\ 2 = 4$ [100] $6 -\!/\!/\ 3 = 3$

[101] $6 -\!/\!/\ 4 = 2$ [102] $6 -\!/\!/\ 5 = 1$

[103] $7 -\!/\!/\ 1 = 6$ [104] $7 -\!/\!/\ 2 = 5$ [105] $7 -\!/\!/\ 3 = 4$

[106] $7 -\!/\!/\ 4 = 3$ [107] $7 -\!/\!/\ 5 = 2$ [108] $7 -\!/\!/\ 6 = 1$

[109] $8 -\!/\!/\ 1 = 7$ [110] $8 -\!/\!/\ 2 = 6$ [111] $8 -\!/\!/\ 3 = 5$

[112] $8 -\!/\!/\ 4 = 4$ [113] $8 -\!/\!/\ 5 = 3$ [114] $8 -\!/\!/\ 6 = 2$

[115] $8 -\!/\!/\ 7 = 1$

[116] $9 -// 1 = 8$ [117] $9 -// 2 = 7$ [118] $9 -// 3 = 6$

[119] $9 -// 4 = 5$ [120] $9 -// 5 = 4$ [121] $9 -// 6 = 3$

[122] $9 -// 7 = 2$ [123] $9 -// 8 = 1$

[124] $10 -// 1 = 9$ [125] $10 -// 2 = 8$ [126] $10 -// 3 = 7$

[127] $10 -// 4 = 6$ [128] $10 -// 5 = 5$ [129] $10 -// 6 = 4$

[130] $10 -// 7 = 3$ [131] $10 -// 8 = 2$ [132] $10 -// 9 = 1$

[133] $$\frac{\text{nat-con}(c_1) -// \text{nat-con}(c_2) = \text{nat-con}(c), \text{nat-con}(c) \neq 0}{\text{gt}(\text{nat-con}(c_1),\ \text{nat-con}(c_2)) = \text{true}}$$

[134] $\text{gt}(\text{nat-con}(y_1\ \ c_1),\ \text{nat-con}(c_2)) = \text{true}$

[135] $\text{gt}(\text{nat-con}(y\ \ c_1),\ \text{nat-con}(y\ \ c_2)) = \text{gt}(\text{nat-con}(c_1),\ \text{nat-con}(c_2))$

[136] $$\frac{\text{gt}(\text{nat-con}(y_1),\ \text{nat-con}(y_2)) = \text{true}}{\text{gt}(\text{nat-con}(y_1\ \ c_1),\ \text{nat-con}(y_2\ \ c_2)) = \text{true}}$$

[158] $$\frac{\text{gt}(Nat_1,\ Nat_2) = \text{true}}{Nat_1 > Nat_2 = \text{true}}$$

[159] $$\frac{\text{gt}(Nat_1,\ Nat_2) \neq \text{true}}{Nat_1 > Nat_2 = \text{false}}$$

APPENDIX B

KERNEL LANGUAGE WITH NILVALUE

The Kernel language (section 5.3) is extended here with nilvalue. The language is called KOLN, for kernel object based language with nil. The syntax adds nil to the initialization expression (IEXP). This gives the ability for the language to have certain instance variables initially defined as "nil" and helps in avoiding cyclic instantiation of instance variables. As we saw in the `nat` example given in section 5.3, with the absence of such a "nil", one has to define a class on the lines of "`class nul`" to provide for classes like "class nat" that uses cyclic instantiation. Although this suffices for now, in the presence of type–checking this generally gets to be a big restriction in the language. Therefore most languages provide for "nil" and the type–checking rules treat its corresponding value "nilvalue" as an exceptional case.

In this appendix, we add "nil" to the syntax of KOL. We also add a facility to test for nil, which is the "if _ then _ else". The value environment adds the corresponding "nilvalue" to the value domain and the relavent equations.

The example of "`class nat`" is provided here for comparison.

module KOLN-syntax

imports KOL-syntax

exports

context-free functions

if EXP then EXP else EXP fi	→ EXP
nil	→ IEXP

module KOLN-venv

imports KOLN-syntax KOL-venv

exports

context-free functions

nilvalue → VALUE

equations

[VO2] value nilvalue = nilvalue

[VO2] pop nilvalue = nilvalue

module KOLN-c2o

imports KOLN-venv KOL-c2o

equations

[C2On] c2o[nil] *Cls* = nilvalue

module KOLN-eval

imports KOLN-c2o KOL-eval

equations

[Ni] $[[\mathrm{nil}]]\mathrm{e}\ \mathit{Self}\ \ \mathit{Cls} = \mathrm{nilvalue}$

$$[C0]\quad \frac{\mathrm{nilvalue} = [[\mathit{Exp}]]\mathrm{e}\ \mathit{Self}\ \ \mathit{Cls}}{[[\mathrm{if}\ \mathit{Exp}\ \mathrm{then}\ \mathit{Exp}_1\ \mathrm{else}\ \mathit{Exp}_2\ \mathrm{fi}]]\mathrm{e}\ \mathit{Self}\ \ \mathit{Cls} = [[\mathit{Exp}_2]]\mathrm{e}\ \mathit{Self}\ \ \mathit{Cls}}$$

$$[C2]\quad \frac{\mathrm{nilvalue} \neq [[\mathit{Exp}]]\mathrm{e}\ \mathit{Self}\ \ \mathit{Cls}}{[[\mathrm{if}\ \mathit{Exp}\ \mathrm{then}\ \mathit{Exp}_1\ \mathrm{else}\ \mathit{Exp}_2\ \mathrm{fi}]]\mathrm{e}\ \mathit{Self}\ \ \mathit{Cls} = [[\mathit{Exp}_1]]\mathrm{e}\ \mathit{Self}\ \ \mathit{Cls}}$$

```
class bool
  meth cond(x,y) x
  meth and(y) self.cond(y,new false)
  meth or(y)  self.cond(new true,y)
  meth not()  self.cond(new false, new true)

class true
  meth cond(x,y) x
  meth and(y) self.cond(y,new false)
  meth or(y)  self.cond(new true,y)
  meth not()  self.cond(new false, new true)

class false
  meth cond(x,y) y
  meth and(y) self.cond(y,new false)
```

```
  meth or(y)  self.cond(new true,y)
  meth not()  self.cond(new false, new true)

class nat
  var isz := new true
  var pre := nil
  meth iszero() isz
  meth setto(x) x.iszero()
               .cond(((self.isz:=new true).pre:=self),
                     ((self.isz:=new false).pre:=(x.pred())))
  meth succ()  new nat.setto(self).incr()
  meth pred()  self.pre
  meth incr() (self.pre := self).isz := new false
  meth decr() self.iszero()
              .cond(self,
                    (self.isz := (pre.iszero()))
                     .pre := (pre.pred()))
  meth less(i)
    i.isz.cond(new true, self.pred().less(i.pred()))

new nat.pred()
```

APPENDIX C

TRANSFORMATION MODULES OF EOOL

The modules presented in this chapter specify the process of tagging the visible identifiers, type-checking, transforming the program to program that uses tagged identifiers, and evaluation of the resulting transformed program using the technique of adoption. We discussed two different methods of achieving adoption in a language that allows "selftype" redeclaration for method arguments (Chapter 6). Here we specify the necessary modules for the *non-intrusive-adoption* that only donates the necessary behavior when an object that is supposed to be like self cannot understand certain messages.

C.1 Tagging, type-checking and transforming

The syntax and type-checking of a language that enforces conformance using the combination of user defined class hierarchy and *inherited types* as *subtypes* is presented in this chapter along with the module that transforms programs in such a language to ones that can be *adopted* is provided.

C.1.1 The user level syntax

The ASOOL language (Section 5.7) developed in Chapter 5 is extended with the predefined sort "natural", to aid in writing concise example programs while illustrating the details of the specification modules used.

The EOOL-syntax module describes the user level syntax of such a language. Although there is not much variation from the syntax of the language ASOOL, the subtype conformance and use of selftype will be realized in a similar way to that found in Eiffel.

module EOOL-syntax

imports Layout Identifiers Nat Booleans

exports

sorts PROGRAM CLASSES VAR-DEC METHOD EXP CLASS RENAME SCLASS IDID IDID-LIST
context-free functions

EXP	→ PROGRAM
CLASS+ SCLASS∗ EXP	→ PROGRAM
class ID VAR-DEC∗ METHOD∗	→ CLASS
class ID subclass of ID RENAME∗	
VAR-DEC∗ METHOD∗	→ SCLASS
rename ID “to” ID	→ RENAME
rename ID “to” ID “:” {ID “,”}∗ “− >” ID	→ RENAME
var ID “:=” EXP	→ VAR-DEC
meth ID “:” ID “(” IDID-LIST “)” EXP	→ METHOD
new ID	→ EXP
nil ID	→ EXP
EXP “.” ID	→ EXP
EXP “.” ID “(” {EXP “,”}∗ “)”	→ EXP
EXP “.” ID “:=” EXP	→ EXP
if EXP then EXP else EXP fi	→ EXP
ID	→ EXP
ID “:=” EXP	→ EXP
EXP “;” EXP	→ EXP {**left**}

ID "(" {EXP ","}∗ ")"	→ EXP
self	→ EXP
"(" EXP ")"	→ EXP {**bracket**}
NAT	→ EXP
EXP "+" EXP	→ EXP {**left**}
EXP ">" EXP	→ EXP
ID ":" ID	→ IDID
{IDID ","}∗	→ IDID-LIST
CLASS∗	→ CLASSES

variables

Cls [*0–9′*]∗	→ CLASS∗
Cl [*0–9′*]∗	→ CLASS
Meth [*0–9′*]∗	→ METHOD
Meths [*0–9′*]∗	→ METHOD∗
Exp [*0–9′*]∗	→ EXP
Exp-list [*0–9′*]∗	→ {EXP ","}∗
Vdec [*0–9′*]∗	→ VAR-DEC
Vdecs [*0–9′*]∗	→ VAR-DEC∗
Id-list [*0–9′*]∗	→ {ID ","}∗
IdId [*0–9′*]∗	→ IDID
IdId-list [*0–9′*]∗	→ {IDID ","}∗
Sl [*0–9′*]∗	→ SCLASS
Sls [*0–9′*]∗	→ SCLASS∗
Renames [*0–9′*]∗	→ RENAME∗

priorities

{“;” ,EXP “+” EXP → EXP ,EXP “>” EXP → EXP }

< {“:=” }

< {EXP “.” ID “(” {EXP “,”}∗ “)” → EXP ,EXP “.” ID → EXP }

Note that the equations that have usually accompanied the syntax module to flatten the inheritance hierarchy are absent here, since some pre-processing is necessary before flattening. For the purpose of pre-processing, which involves adding the type tag information for the method instance variable names of a class (names visible from the outside), we need the corresponding extension of the language at an internal level.

Note that the word “selftype” can be used as a type name anywhere a type declaration is expected (for both argument types and return types of methods), unlike only for return types as in Chapter 5. The use of “selftype” in this manner results in the so called inherited types [17] when the class gets inherited.

C.1.2 Tagging visible names

This module extends the language syntax to accommodate type tag (TAGID) information which involves the syntax for maintaining the preorder information (class Id : $\mathrm{Id}' \cdots$) as specified in the class hierarchy, the syntax for extending the method and instance variable names to be tagged (using TAGID), and allowance for “selftype” resolution as done before (Section 5.6.2).

module EOOL-tagid

imports EOOL-syntax

exports

sorts TAGID RENAMES

context-free functions

class ID ":" ID VAR-DEC∗ METHOD∗	→ CLASS
var TAGID ":=" EXP	→ VAR-DEC
meth TAGID ":" ID "(" IDID-LIST ")" EXP	→ METHOD
EXP "." TAGID "(" {EXP ","}∗ ")"	→ EXP
EXP "." TAGID	→ EXP
EXP "." TAGID ":=" EXP	→ EXP
TAGID ":=" EXP	→ EXP
TAGID "(" {EXP ","}∗ ")"	→ EXP
TAGID	→ EXP
ID "_" ID	→ TAGID
{IDID ","}∗ "/" {ID ","}∗	→ IDID-LIST
nometh ID in METHOD∗	→ BOOL
novar ID in VAR-DEC∗	→ BOOL
RENAME∗	→ RENAMES

variables

Tid [$0\text{–}9'$]∗ → TAGID

equations

[PreOrd] class Id $Vdecs$ $Meths'$ = class Id : none $Vdecs$ $Meths'$

$$\text{[LeftM]}\quad \frac{\text{nometh } \mathit{Id} \text{ in } \mathit{Meths}' = \text{true}}{\begin{array}{l}\text{class } \mathit{Id}' : \mathit{Id}'' \ \mathit{Vdecs} \ \mathit{Meths} \text{ meth } \mathit{Id} \_ \mathit{Id}_0 : \mathit{Id}_1(\mathit{IdId\text{-}list}_1) \ \mathit{Exp}_1 \\ \qquad\qquad \mathit{Meths}' \text{ meth } \mathit{Id} \_ \mathit{Id0}' : \mathit{Id}_2(\mathit{IdId\text{-}list}_2) \ \mathit{Exp}_2 \ \mathit{Meths}'' = \\ \text{class } \mathit{Id}' : \mathit{Id}'' \ \mathit{Vdecs} \ \mathit{Meths} \text{ meth } \mathit{Id} \_ \mathit{Id0}' : \mathit{Id}_1(\mathit{IdId\text{-}list}_1) \ \mathit{Exp}_1 \\ \qquad\qquad \mathit{Meths}' \ \mathit{Meths}''\end{array}}$$

$$\text{[LeftV]}\quad \frac{\text{novar } \mathit{Id} \text{ in } \mathit{Vdecs}' = \text{true}}{\begin{array}{l}\text{class } \mathit{Id}' : \mathit{Id}'' \ \mathit{Vdecs} \text{ var } \mathit{Id} \_ \mathit{Id}_0 := \mathit{Exp} \ \mathit{Vdecs}' \\ \qquad\qquad \text{var } \mathit{Id} \_ \mathit{Id0}' := \mathit{Exp}' \ \mathit{Vdecs}'' \ \mathit{Meths}' = \\ \text{class } \mathit{Id}' : \mathit{Id}'' \ \mathit{Vdecs} \text{ var } \mathit{Id} \_ \mathit{Id0}' := \mathit{Exp} \ \mathit{Vdecs}' \ \mathit{Vdecs}'' \ \mathit{Meths}'\end{array}}$$

$$\text{[TagM]}\quad \begin{array}{l}\text{class } \mathit{Id}' : \mathit{Id}'' \ \mathit{Vdecs} \ \mathit{Meths} \text{ meth } \mathit{Id} : \mathit{Id}_1(\mathit{IdId\text{-}list}_1) \ \mathit{Exp} \ \mathit{Meths}' = \\ \text{class } \mathit{Id}' : \mathit{Id}'' \ \mathit{Vdecs} \ \mathit{Meths} \text{ meth } \mathit{Id} \_ \mathit{Id}' : \mathit{Id}_1(\mathit{IdId\text{-}list}_1) \ \mathit{Exp} \ \mathit{Meths}'\end{array}$$

$$\text{[TagV]}\quad \begin{array}{l}\text{class } \mathit{Id}' : \mathit{Id}'' \ \mathit{Vdecs} \text{ var } \mathit{Id} := \mathit{Exp} \ \mathit{Vdecs}' \ \mathit{Meths}' = \\ \text{class } \mathit{Id}' : \mathit{Id}'' \ \mathit{Vdecs} \text{ var } \mathit{Id} \_ \mathit{Id}' := \mathit{Exp} \ \mathit{Vdecs}' \ \mathit{Meths}'\end{array}$$

$$\text{[Flat0]}\quad \frac{\begin{array}{c}\mathit{Cl}_0 = \text{class } \mathit{Id} : \mathit{Id}'' \ \mathit{Vdecs} \ \mathit{Meths}, \\ \mathit{Sl} = \text{class } \mathit{Id}' \text{ subclass of } \mathit{Id} \ \mathit{Vdecs}' \ \mathit{Meths}', \\ \mathit{Cl}_1 = \text{class } \mathit{Id}' : \mathit{Id} \ \mathit{Vdecs}' \ \mathit{Vdecs} \ \mathit{Meths}' \ \mathit{Meths}\end{array}}{\mathit{Cls} \ \mathit{Cl}_0 \ \mathit{Cls}' \ \mathit{Sl} \ \mathit{Sls}'' \ \mathit{Exp} = \mathit{Cls} \ \mathit{Cl}_0 \ \mathit{Cls}' \ \mathit{Cl}_1 \ \mathit{Sls}'' \ \mathit{Exp}}$$

$$\text{[Flat1]}\quad \frac{\begin{array}{c}\mathit{Cl}_0 = \text{class } \mathit{Id} : \mathit{Id}'' \ \mathit{Vdecs} \ \mathit{Meths}_0 \ \mathit{Meth}_0 \ \mathit{Meths}_1, \\ \mathit{Meth}_0 = \text{meth } \mathit{Id}_0 \_ \mathit{Id}_2 : \mathit{Id}_1(\mathit{IdId\text{-}list}) \ \mathit{Exp}', \\ \mathit{Sl}_0 = \text{class } \mathit{Id}' \text{ subclass of } \mathit{Id} \ \mathit{Renames}_0 \ \mathit{Vdecs}' \ \mathit{Meths}', \\ \mathit{Renames}_0 = \text{rename } \mathit{Id}_0 \text{ to } \mathit{Id0}' \ \mathit{Renames}_1, \\ \mathit{Meth}_1 = \text{meth } \mathit{Id0}' \_ \mathit{Id}_2 : \mathit{Id}_1(\mathit{IdId\text{-}list}) \ \mathit{Exp}', \\ \mathit{Sl}_1 = \text{class } \mathit{Id}' \text{ subclass of } \mathit{Id} \ \mathit{Renames}_1 \ \mathit{Vdecs}' \ \mathit{Meths}' \ \mathit{Meth}_1\end{array}}{\mathit{Cls} \ \mathit{Cl}_0 \ \mathit{Cls}' \ \mathit{Sl}_0 \ \mathit{Sls}'' \ \mathit{Exp} = \mathit{Cls} \ \mathit{Cl}_0 \ \mathit{Cls}' \ \mathit{Sl}_1 \ \mathit{Sls}'' \ \mathit{Exp}}$$

$$[\text{Flat2}]\quad \frac{\begin{array}{c} Cl_0 = \text{class } Id : Id'' \ Vdecs \ Meths_0 \ Meth_0 \ Meths_1, \\ Meth_0 = \text{meth } Id_0 \text{ \_ } Id_3 : Id_1(IdId\text{-}list) \ Exp', \\ Sl_0 = \text{class } Id' \text{ subclass of } Id \ Renames_0 \ Vdecs' \ Meths', \\ Renames_0 = \text{rename } Id_0 \text{ to } Id0' : Id\text{-}list -> Id_2 \ Renames_1, \\ Meth_1 = \text{meth } Id0' \text{ \_ } Id_3 : Id_2(IdId\text{-}list \ / \ Id\text{-}list) \ Exp', \\ Sl_1 = \text{class } Id' \text{ subclass of } Id \ Renames_1 \ Vdecs' \ Meths' \ Meth_1 \end{array}}{Cls \ Cl_0 \ Cls' \ Sl_0 \ Sls'' \ Exp = Cls \ Cl_0 \ Cls' \ Sl_1 \ Sls'' \ Exp}$$

$$[\text{SelfT}]\quad \frac{\begin{array}{c} Cl_0 = \text{class } Id : Id' \ Vdecs \ Meths \ Meth_0 \ Meths', \\ Meth_0 = \text{meth } Tid : \text{selftype}(IdId\text{-}list) \ Exp', \\ Meth_1 = \text{meth } Tid : Id(IdId\text{-}list) \ Exp', \\ Cl_1 = \text{class } Id : Id' \ Vdecs \ Meths \ Meth_1 \ Meths' \end{array}}{Cls \ Cl_0 \ Cls' \ Exp = Cls \ Cl_1 \ Cls' \ Exp}$$

$$[\text{Rdec1}]\quad / \ =$$

$$[\text{Rdec2}]\quad \frac{IdId\text{-}list' = IdId\text{-}list \ / \ Id\text{-}list}{Id : Id', IdId\text{-}list \ / \ Id'', Id\text{-}list = Id : Id'', IdId\text{-}list'}$$

$$[\text{NoM0}]\quad \text{nometh } Id \text{ in } \ = \text{true}$$

$$[\text{NoM1}]\quad \text{nometh } Id \text{ in meth } Id \text{ \_ } Id_0 : Id_1(IdId\text{-}list) \ Exp \ Meths = \text{false}$$

$$[\text{NoM2}]\quad \frac{Meth = \text{meth } Id' \text{ \_ } Id_0 : Id_1(IdId\text{-}list) \ Exp, Id \neq Id'}{\text{nometh } Id \text{ in } Meth \ Meths = \text{nometh } Id \text{ in } Meths}$$

$$[\text{NoM3}]\quad \text{nometh } Id \text{ in meth } Id : Id_1(IdId\text{-}list) \ Exp \ Meths = \text{false}$$

$$[\text{NoM4}]\ \frac{\mathit{Meth} = \text{meth } \mathit{Id}' : \mathit{Id}_1(\mathit{IdId\text{-}list})\ \mathit{Exp}, \mathit{Id} \neq \mathit{Id}'}{\text{nometh } \mathit{Id} \text{ in } \mathit{Meth}\ \mathit{Meths} = \text{nometh } \mathit{Id} \text{ in } \mathit{Meths}}$$

$$[\text{NoV0}]\ \text{novar } \mathit{Id} \text{ in } \ = \text{true}$$

$$[\text{NoV1}]\ \text{novar } \mathit{Id} \text{ in var } \mathit{Id}\ \_\ \mathit{Id}_0 := \mathit{Exp}\ \mathit{Vdecs} = \text{false}$$

$$[\text{NoV2}]\ \frac{\mathit{Vdec} = \text{var } \mathit{Id}'\ \_\ \mathit{Id}_0 := \mathit{Exp}, \mathit{Id} \neq \mathit{Id}'}{\text{novar } \mathit{Id} \text{ in } \mathit{Vdec}\ \mathit{Vdecs} = \text{novar } \mathit{Id} \text{ in } \mathit{Vdecs}}$$

$$[\text{NoV3}]\ \text{novar } \mathit{Id} \text{ in var } \mathit{Id} := \mathit{Exp}\ \mathit{Vdecs} = \text{false}$$

$$[\text{NoV4}]\ \frac{\mathit{Vdec} = \text{var } \mathit{Id}' := \mathit{Exp}, \mathit{Id} \neq \mathit{Id}'}{\text{novar } \mathit{Id} \text{ in } \mathit{Vdec}\ \mathit{Vdecs} = \text{novar } \mathit{Id} \text{ in } \mathit{Vdecs}}$$

The equation [PreOrd] indicates that the identifier “none” has a special meaning and can be considered as an empty class at the top of the hierarchy. It is assumed that no user class will be named “none”. The equation also suggests that the transformed subclasses will remember the name of the super class ([Flat0]). This information is used to realize conformance of classes only in the presence of inheritance.

Equations [TagM] and [TagV] tag the untagged instance variables and method names in a class. The equations [Flat0, Flat1 and Flat2] flatten the hierarchy as before by renaming method names if necessary. As this flattening operation can result in classes with more than one definition for a method name, the equations [LeftM and LeftV] pick the most recent definition for a name while preserving the inherited tag name. The utility of this tagging process was discussed in Chapter 4. The two auxiliary functions “nometh ID in _” and “novar

ID in _" help in picking the right tag information in the presence of many names with different tags, thus matching the oldest tag with the newest definition.

C.1.3 The type environment

A type environment will need to map tagged identifiers to their type. Both the untagged identifiers and tagged identifiers need to be considered due to the presence of untagged identifiers (formal arguments of methods) which are invisible from outside. This is generalized in the equations by using a special identifier "temp" to identify the local identifiers (equation [Tv-X1] below). The only untagged identifier will then be "selftype" which is a mapping to the type name of the current type environment.

As the identifier "selftype" can be used in the formal argument list[1] we now need to extend the function [[_]] defined in earlier Tenv modules to also replace "selftype" with an actual type identifier (equations [Tv-X0] - [Tv-X2]).

module EOOL-tenv

imports Booleans Identifiers EOOL-tagid

exports

sorts TYPE TENV IDT ID-LIST

context-free functions

ID	→ TYPE
ID "(" ID-LIST ")"	→ TYPE

[1]The identifiers "none", "temp", "selftype" and "oops" are reserved as keywords of the language

ID ":" TYPE	→ IDT
"[" {IDT ","}∗ "]"	→ TENV
lookup ID in TENV	→ TYPE
TENV "@" TENV	→ TENV
{ID ","}∗	→ ID-LIST
"type" {IDID ","}∗	→ ID-LIST
"[[" {IDID ","}∗ "selftype/" ID "]]"	→ TENV
"IF" BOOL "THEN" BOOL "ELSE" BOOL	→ BOOL
TAGID ":" TYPE	→ IDT
lookup TAGID in TENV	→ TYPE
istemp ID in TENV	→ BOOL

variables

Type [*0*–*9*′]∗	→ TYPE
Tenv [*0*–*9*′]∗	→ TENV
Idt-list [*0*–*9*′]∗	→ {IDT ","}∗

equations

[Tv-L1] $\text{lookup } \mathit{Id} \text{ in } [\mathit{Id} : \mathit{Type}, \mathit{Idt\text{-}list}] = \mathit{Type}$

[Tv-L2] $$\frac{\mathit{Id} \neq \mathit{Id}'}{\text{lookup } \mathit{Id} \text{ in } [\mathit{Id}' : \mathit{Type}, \mathit{Idt\text{-}list}] = \text{lookup } \mathit{Id} \text{ in } [\mathit{Idt\text{-}list}]}$$

[Tv-L3] $\text{lookup } \mathit{Id} \text{ in } [\mathit{Id} \mathbin{\_} \mathit{Id}_0 : \mathit{Type}, \mathit{Idt\text{-}list}] = \mathit{Type}$

[Tv-L4] $$\frac{Id \neq Id'}{\text{lookup } Id \text{ in } [Id' \_ Id_0 : Type, Idt\text{-}list] = \text{lookup } Id \text{ in } [Idt\text{-}list]}$$

[Tv-L5] lookup Tid in $[Tid : Type, Idt\text{-}list] = Type$

[Tv-L6] $$\frac{Tid \neq Tid'}{\text{lookup } Tid \text{ in } [Tid' : Type, Idt\text{-}list] = \text{lookup } Tid \text{ in } [Idt\text{-}list]}$$

[Tv-A0] $[Idt\text{-}list]$ @ $[Idt\text{-}list'] = [Idt\text{-}list, Idt\text{-}list']$

[Tv-X0] $[[\ \text{selftype}/\ Id_0]] = []$

[Tv-X1] $$\frac{Id' \neq \text{selftype}}{\begin{array}{l}[[Id : Id', IdId\text{-}list \text{ selftype}/\ Id_0]] \quad = \\ [Id \_ \text{temp} : Id'] \text{ @ } [[IdId\text{-}list \text{ selftype}/\ Id_0]]\end{array}}$$

[Tv-X2] $[[Id : \text{selftype}, IdId\text{-}list \text{ selftype}/\ Id_0]] =$
$[Id \_ \text{temp} : Id_0]$ @ $[[IdId\text{-}list \text{ selftype}/\ Id_0]]$

[Tv-Y0] type $=$

[Tv-Y1] $$\frac{Id\text{-}list = \text{type } IdId\text{-}list}{\text{type } Id : Id', IdId\text{-}list = Id', Id\text{-}list}$$

[Tv-IF0] IF true THEN $Bool$ ELSE $Bool' = Bool$

[Tv-IF1] IF false THEN $Bool$ ELSE $Bool' = Bool'$

[Tv-IM0] istemp Id in $[] =$ false

[Tv-IM1] istemp Id in $[Id \_ \text{temp} : Type, Idt\text{-}list] =$ true

$$\text{[Tv-IM2]}\quad \frac{Id \neq Id_0}{\text{istemp } Id \text{ in } [Id_0 \text{ \_ temp} : \mathit{Type}, \mathit{Idt\text{-}list}] = \text{istemp } Id \text{ in } [\mathit{Idt\text{-}list}]}$$

$$\text{[Tv-IM3]}\quad \frac{Id' \neq \text{temp}}{\text{istemp } Id \text{ in } [Id_0 \text{ \_ } Id' : \mathit{Type}, \mathit{Idt\text{-}list}] = \text{istemp } Id \text{ in } [\mathit{Idt\text{-}list}]}$$

$$\text{[Tv-IM4]}\quad \text{istemp } Id \text{ in } [Id_0 : \mathit{Type}, \mathit{Idt\text{-}list}] = \text{istemp } Id \text{ in } [\mathit{Idt\text{-}list}]$$

C.1.4 The conformance relation

The conformance relation, similar to that of Eiffel, is realized in this module. The functions are similar to those defined in Chapter 5, except for the function "stypes _ using _ _" which collects the preorder defined by the inheritance hierarchy if it is a conforming one. Note that the equations [ES-S2] and [ES-S3] check for contravariance of argument methods and equality of instance variable types. The equation [ES-S2] also effectively checks for "selftype" declarations to correspond, thus effectively allowing a covariant method argument redeclaration for the case of "selftype".

module EOOL-sub

imports EOOL-tenv

exports

sorts SUBTYPE SUBTYPES

context-free functions

assume TYPE "<" TYPE	$\rightarrow$ SUBTYPE
assumed TYPE "<" TYPE in SUBTYPE$*$	$\rightarrow$ BOOL
TENV "<" TENV SUBTYPE$*$ CLASSES	$\rightarrow$ BOOL
TYPE "<" TYPE SUBTYPE$*$ CLASSES	$\rightarrow$ BOOL
TYPE v TYPE CLASSES	$\rightarrow$ TYPE
cid2t "[" ID "]" CLASSES	$\rightarrow$ TENV
stypes CLASSES using CLASSES SUBTYPES	$\rightarrow$ SUBTYPES
SUBTYPE$*$	$\rightarrow$ SUBTYPES

variables

Stype [*0–9*$'$]$*$	$\rightarrow$ SUBTYPE
Stypes [*0–9*$'$]$*$	$\rightarrow$ SUBTYPE$*$

equations

[ES-P0] stypes using *Cls Stypes* = *Stypes*

$$\text{[ES-P1]}\quad \frac{\begin{array}{c} \mathit{Stypes}' = \text{assume } \mathit{Id} < \mathit{Id}' \ \mathit{Stypes}, \\ \mathit{Tenv} = \text{cid2t}[\mathit{Id}]\ \mathit{Cls},\ \mathit{Tenv}' = \text{cid2t}[\mathit{Id}']\ \mathit{Cls}, \\ \mathit{Tenv} < \mathit{Tenv}'\ \mathit{Stypes}'\ \mathit{Cls} = \text{true}, \\ \mathit{Stypes}'' = \text{stypes } \mathit{Cls}'\ \mathit{Cls}'' \text{ using } \mathit{Cls}\ \mathit{Stypes}' \end{array}}{\text{stypes } \mathit{Cls}' \text{ class } \mathit{Id} : \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}\ \mathit{Cls}'' \text{ using } \mathit{Cls}\ \mathit{Stypes} = \mathit{Stypes}''}$$

$$\text{[ES-S1]}\quad \frac{\mathit{Tenv} = [\mathit{Idt\text{-}list}', \text{selftype} : \mathit{Id}],\ \mathit{Tenv}' = [\text{selftype} : \mathit{Id}']}{\mathit{Tenv} < \mathit{Tenv}'\ \mathit{Stypes}\ \mathit{Cls} = \mathit{Id} < \mathit{Id}'\ \mathit{Stypes}\ \mathit{Cls}}$$

$$[\text{ES-S2}]\quad \frac{\begin{array}{c} \mathit{Tenv}' = [\mathit{Tid} : \mathit{Type}', \mathit{Idt\text{-}list}], \\ \mathit{Id}(\mathit{Id\text{-}list}) = \mathit{Type}', \\ \mathit{Bool}_0 = \text{lookup } \mathit{Tid} \text{ in } \mathit{Tenv} < \mathit{Type}' \ \mathit{Stypes}\ \mathit{Cls}, \\ \mathit{Bool}_1 = \mathit{Tenv} < [\mathit{Idt\text{-}list}]\ \mathit{Stypes}\ \mathit{Cls} \end{array}}{\mathit{Tenv} < \mathit{Tenv}'\ \mathit{Stypes}\ \mathit{Cls} = \text{IF } \mathit{Bool}_0 \text{ THEN } \mathit{Bool}_1 \text{ ELSE false}}$$

$$[\text{ES-S3}]\quad \frac{\begin{array}{c} \mathit{Tenv}' = [\mathit{Tid} : \mathit{Type}', \mathit{Idt\text{-}list}], \\ \mathit{Id} = \mathit{Type}', \\ \mathit{Bool}_0 = \text{lookup } \mathit{Tid} \text{ in } \mathit{Tenv} < \mathit{Type}', \\ \mathit{Bool}_1 = \mathit{Tenv} < [\mathit{Idt\text{-}list}]\ \mathit{Stypes}\ \mathit{Cls} \end{array}}{\mathit{Tenv} < \mathit{Tenv}'\ \mathit{Stypes}\ \mathit{Cls} = \text{IF } \mathit{Bool}_0 \text{ THEN } \mathit{Bool}_1 \text{ ELSE false}}$$

[ES-T0] $\mathit{Id} < \mathit{Id}\ \mathit{Stypes}\ \mathit{Cls} = \text{true}$

$$[\text{ES-T1}]\quad \frac{\mathit{Id} \neq \mathit{Id}', \text{assumed } \mathit{Id} < \mathit{Id}' \text{ in } \mathit{Stypes} = \text{true}}{\mathit{Id} < \mathit{Id}'\ \mathit{Stypes}\ \mathit{Cls} = \text{true}}$$

[ES-T3] $\mathit{Id}() < \mathit{Id}'()\ \mathit{Stypes}\ \mathit{Cls} = \mathit{Id} < \mathit{Id}'\ \mathit{Stypes}\ \mathit{Cls}$

[ES-T4] $\mathit{Id}(\mathit{Id}_0, \mathit{Id\text{-}list}) < \mathit{Id}'(\mathit{Id0}', \mathit{Id\text{-}list}')\ \mathit{Stypes}\ \mathit{Cls} =$
$\mathit{Id0}' < \mathit{Id}_0\ \mathit{Stypes}\ \mathit{Cls}\ \&\ \mathit{Id}(\mathit{Id\text{-}list}) < \mathit{Id}'(\mathit{Id\text{-}list}')\ \mathit{Stypes}\ \mathit{Cls}$

[ES-V0] $\mathit{Type} \text{ v } \mathit{Type}\ \mathit{Cls} = \mathit{Type}$

$$[\text{ES-V1}]\quad \frac{\mathit{Type}_1 \neq \mathit{Type}_2, \mathit{Type}_1 < \mathit{Type}_2\ \mathit{Cls} = \text{true}}{\mathit{Type}_1 \text{ v } \mathit{Type}_2\ \mathit{Cls} = \mathit{Type}_2}$$

$$[\text{ES-V2}]\quad \frac{\mathit{Type}_1 \neq \mathit{Type}_2, \mathit{Type}_2 < \mathit{Type}_1\ \mathit{Cls} = \text{true}}{\mathit{Type}_1 \text{ v } \mathit{Type}_2\ \mathit{Cls} = \mathit{Type}_1}$$

[ES-A0] assumed $\mathit{Type} < \mathit{Type}'$ in $= \text{false}$

[ES-A1] assumed $Type < Type'$ in assume $Type < Type'$ $Stypes$ = true

$$\text{[ES-A2]}\quad \frac{Type \neq Type_1}{\text{assumed } Type < Type' \text{ in assume } Type_1 < Type1' \ Stypes \ = \ \text{assumed } Type < Type' \text{ in } Stypes}$$

$$\text{[ES-A3]}\quad \frac{Type' \neq Type1'}{\text{assumed } Type < Type' \text{ in assume } Type_1 < Type1' \ Stypes \ = \ \text{assumed } Type < Type' \text{ in } Stypes}$$

$$\text{[ES-A4]}\quad \frac{Type' \neq Type''}{\text{assumed } Type < Type' \text{ in assume } Type < Type'' \ Stypes \ = \ \text{assumed } Type < Type'' \text{ in } Stypes}$$

The equation [ES-P1] non-deterministically chooses a class from the list of classes and checks if it conforms to the class it was inherited from. The inheritance information is encoded into the class information (Section C.1.2). In the process of checking for conformance, a list of conforming type pairs (assume TYPE < TYPE) is produced. The type-checker then uses this conformance list to check the method bodies, etc while type-checking the program.

The interesting check here is the check that enforces contravariance ([ES-T4]) while relaxing this to the case of “selftype”. This special case is achieved for free through equation ES-T0 which permits equality check. Thus, a type declaration of the formal arguments of a method in a super class matches the type declaration of the corresponding method of a subclass for the case of “selftype”. This also implies that the language only relaxes contravariance when the corresponding declarations are consistently "selftype". This intentional hole created in the conformance relation should be properly accounted for in the dynamic semantics (section C.2.3) of the language, as this violates *substitutability in all contexts* otherwise.

C.1.5 Type-checking

The type-checking module is similar to those presented earlier (Sections 5.4.3). The function that checks if the actual arguments of a method conform to the type declaration also translates the identifier "selftype" to the appropriate type name (equations [EST0], [EST1] and [EST3] below).

module EOOL-tc

imports EOOL-sub

exports

context-free functions

"tc" "[" PROGRAM "]"	→ TYPE
"type" "[" EXP "]" TENV CLASSES	→ TYPE
"tc" "[" CLASSES "]" CLASSES	→ BOOL
"tc" "[" METHOD∗ "]" TENV CLASSES	→ BOOL
"c2t" "[" CLASS "]"	→ TENV
"[" {EXP ","}∗ TENV CLASSES "<" ID-LIST "]selftype/" ID	→ BOOL
stypes CLASSES	→ SUBTYPES

equations

[ET-C0] tc[] *Cls* = true

$$\text{[ET-C1]} \quad \frac{\text{tc}[\mathit{Meths}]\ \text{c2t}[\text{class } \mathit{Id} : \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}]\ \mathit{Cls} = \text{true}}{\text{tc}[\text{class } \mathit{Id} : \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}\ \mathit{Cls}']\ \mathit{Cls} = \text{tc}[\mathit{Cls}']\ \mathit{Cls}}$$

$$\text{[ET-M0]} \quad \text{tc}[]\ \mathit{Tenv}\ \mathit{Cls} = \text{true}$$

$$\text{[ET-M1]} \quad \frac{\begin{array}{c}\mathit{Stypes} = \text{stypes } \mathit{Cls}, \mathit{Id}_0 = \text{lookup selftype in } \mathit{Tenv}, \\ \text{type}[\mathit{Exp}]\ [[\mathit{IdId\text{-}list}\ \text{selftype}/\ \mathit{Id}_0]]\ @\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id}'\ \mathit{Stypes}\ \mathit{Cls} = \text{true}\end{array}}{\text{tc}[\text{meth } \mathit{Tid} : \mathit{Id}'(\mathit{IdId\text{-}list})\ \mathit{Exp}\ \mathit{Meths}]\ \mathit{Tenv}\ \mathit{Cls} = \text{tc}[\mathit{Meths}]\ \mathit{Tenv}\ \mathit{Cls}}$$

$$\begin{array}{ll}\text{[ET-D0]} & \text{c2t}[\text{class } \mathit{Id} : \mathit{Id}'\ \text{var } \mathit{Tid} := \mathit{Nat}\ \mathit{Vdecs}\ \mathit{Meths}] = \\ & [\mathit{Tid} : \text{natural}]\ @\ \text{c2t}[\text{class } \mathit{Id} : \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}]\end{array}$$

$$\begin{array}{ll}\text{[ET-D1]} & \text{c2t}[\text{class } \mathit{Id} : \mathit{Id}'\ \text{var } \mathit{Tid} := \text{new } \mathit{Id0}'\ \mathit{Vdecs}\ \mathit{Meths}] = \\ & [\mathit{Tid} : \mathit{Id0}']\ @\ \text{c2t}[\text{class } \mathit{Id} : \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}]\end{array}$$

$$\begin{array}{ll}\text{[ET-D2]} & \text{c2t}[\text{class } \mathit{Id} : \mathit{Id}'\ \text{meth } \mathit{Tid} : \mathit{Id0}'(\mathit{IdId\text{-}list})\ \mathit{Exp}\ \mathit{Meths}] = \\ & [\mathit{Tid} : \mathit{Id0}'(\text{type } \mathit{IdId\text{-}list})]\ @\ \text{c2t}[\text{class } \mathit{Id} : \mathit{Id}'\ \mathit{Meths}]\end{array}$$

$$\text{[ET-D3]} \quad \text{c2t}[\text{class } \mathit{Id} : \mathit{Id}'\ ] = [\text{selftype} : \mathit{Id}]$$

$$\text{[ET-E0]} \quad \text{type}[\text{nil } \mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls}'\ \text{class } \mathit{Id} : \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}\ \mathit{Cls}'' = \mathit{Id}$$

$$\text{[ET-E1]} \quad \text{type}[\text{new } \mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls}'\ \text{class } \mathit{Id} : \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}\ \mathit{Cls}'' = \mathit{Id}$$

$$\text{[ET-E2]} \quad \text{type}[\mathit{Nat}]\ \mathit{Tenv}\ \mathit{Cls} = \text{natural}$$

$$\text{[ET-E3]} \quad \frac{\begin{array}{c}\text{type}[\mathit{Exp}_1]\ \mathit{Tenv}\ \mathit{Cls} = \text{natural}, \\ \text{type}[\mathit{Exp}_2]\ \mathit{Tenv}\ \mathit{Cls} = \text{natural}\end{array}}{\text{type}[\mathit{Exp}_1 + \mathit{Exp}_2]\ \mathit{Tenv}\ \mathit{Cls} = \text{natural}}$$

$$\text{[ET-E4]} \quad \frac{\begin{array}{c}\text{type}[\mathit{Exp}_1]\ \mathit{Tenv}\ \mathit{Cls} = \text{natural}, \\ \text{type}[\mathit{Exp}_2]\ \mathit{Tenv}\ \mathit{Cls} = \text{natural}\end{array}}{\text{type}[\mathit{Exp}_1 > \mathit{Exp}_2]\ \mathit{Tenv}\ \mathit{Cls} = \text{boolean}}$$

[ET-E5] $\text{type[self]}\ \mathit{Tenv}\ \mathit{Cls} = \text{lookup selftype in}\ \mathit{Tenv}$

$$\text{[ET-E6]}\quad \frac{\text{istemp}\ \mathit{Id}\ \text{in}\ \mathit{Tenv} = \text{true}}{\text{type}[\mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls} = \text{lookup}\ \mathit{Id}\ \_\ \text{temp in}\ \mathit{Tenv}}$$

$$\text{[ET-E7]}\quad \frac{\text{istemp}\ \mathit{Id}\ \text{in}\ \mathit{Tenv} = \text{false}}{\text{type}[\mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls} = \text{lookup}\ \mathit{Id}\ \text{in}\ \mathit{Tenv}}$$

$$\text{[ET-E8]}\quad \frac{\begin{array}{c}\text{type}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} = \text{boolean},\\ \mathit{Type}_1 = \text{type}[\mathit{Exp}_1]\ \mathit{Tenv}\ \mathit{Cls},\\ \mathit{Type}_2 = \text{type}[\mathit{Exp}_2]\ \mathit{Tenv}\ \mathit{Cls},\\ \mathit{Id} = \mathit{Type}_1\ \text{v}\ \mathit{Type}_2\ \mathit{Cls}\end{array}}{\text{type[if}\ \mathit{Exp}\ \text{then}\ \mathit{Exp}_1\ \text{else}\ \mathit{Exp}_2\ \text{fi]}\ \mathit{Tenv}\ \mathit{Cls} = \mathit{Id}}$$

$$\text{[ET-E10]}\quad \frac{\mathit{Id}_0 = \text{type}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}, \mathit{Tenv}' = \text{cid2t}[\mathit{Id}_0]\ \mathit{Cls}}{\text{type}[\mathit{Exp}.\ \mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls} = \text{type}[\mathit{Id}]\ \mathit{Tenv}'\ \mathit{Cls}}$$

$$\text{[ET-E11]}\quad \frac{\begin{array}{c}\mathit{Type}' = \text{type}[\mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls},\\ \mathit{Type} = \text{type}[\mathit{Exp}.\ \mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls},\\ \mathit{Stypes} = \text{stypes}\ \mathit{Cls}, \mathit{Type}' < \mathit{Type}\ \mathit{Stypes}\ \mathit{Cls} = \text{true}\end{array}}{\text{type}[\mathit{Exp}.\ \mathit{Id} := \mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls} = \text{type}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}}$$

[ET-E12] $\text{type}[\mathit{Id} := \mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} = \text{type[self.}\ \mathit{Id} := \mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}$

$$\text{[ET-E13]}\quad \frac{\mathit{Type}_1 = \text{type}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}, \mathit{Type}_2 = \text{type}[\mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls}}{\text{type}[\mathit{Exp};\ \mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls} = \mathit{Type}_1\ \text{v}\ \mathit{Type}_2\ \mathit{Cls}}$$

[ET-E15] $\text{type}[\mathit{Id}(\mathit{Exp\text{-}list})]\ \mathit{Tenv}\ \mathit{Cls} = \text{type[self.}\ \mathit{Id}(\mathit{Exp\text{-}list})]\ \mathit{Tenv}\ \mathit{Cls}$

$$\text{[ET-E16]}\quad \frac{\begin{array}{c} \mathit{Id}_0 = \text{type}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls},\ \mathit{Tenv}' = \text{cid2t}[\mathit{Id}_0]\ \mathit{Cls}, \\ \text{lookup}\ \mathit{Id}\ \text{in}\ \mathit{Tenv}' = \mathit{Id}_1(\mathit{Id\text{-}list}_1), \\ [\mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id\text{-}list}_1]\text{selftype}/\ \mathit{Id}_0 = \text{true} \end{array}}{\text{type}[\mathit{Exp}.\ \mathit{Id}(\mathit{Exp\text{-}list})]\ \mathit{Tenv}\ \mathit{Cls} = \mathit{Id}_1}$$

$$\text{[EST0]}\quad [\ \mathit{Tenv}\ \mathit{Cls} < \ ]\text{selftype}/\ \mathit{Id} = \text{true}$$

$$\text{[EST1]}\quad \begin{array}{l} [\mathit{Exp},\mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \text{selftype},\mathit{Id\text{-}list}]\text{selftype}/\ \mathit{Id} = \\ [\mathit{Exp},\mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id},\mathit{Id\text{-}list}]\text{selftype}/\ \mathit{Id} \end{array}$$

$$\text{[EST3]}\quad \frac{\begin{array}{c} \mathit{Id}' \neq \text{selftype},\ \mathit{Stypes} = \text{stypes}\ \mathit{Cls}, \\ \text{type}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id}'\ \mathit{Stypes}\ \mathit{Cls} = \text{true} \end{array}}{\begin{array}{l} [\mathit{Exp},\mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id}',\mathit{Id\text{-}list}]\text{selftype}/\ \mathit{Id}\quad = \\ [\mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id\text{-}list}]\text{selftype}/\ \mathit{Id} \end{array}}$$

$$\text{[CID0]}\quad \begin{array}{l} \text{cid2t}[\mathit{Id}]\ \text{class}\ \mathit{Id} : \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}\ \mathit{Cls} = \\ \text{c2t}[\text{class}\ \mathit{Id} : \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}] \end{array}$$

$$\text{[CID1]}\quad \frac{\mathit{Id} \neq \mathit{Id}',\ \mathit{Id} \neq \text{none}}{\begin{array}{l} \text{cid2t}[\mathit{Id}]\ \text{class}\ \mathit{Id}' : \mathit{Id}''\ \mathit{Vdecs}\ \mathit{Meths}\ \mathit{Cls}\quad = \\ \text{cid2t}[\mathit{Id}]\ \mathit{Cls} \end{array}}$$

$$\text{[CID2]}\quad \text{cid2t}[\mathit{Id}]\ \ = []$$

$$\text{[CID3]}\quad \text{cid2t}[\text{none}]\ \mathit{Cls} = [\text{selftype} : \text{none}]$$

$$\text{[STYPE]}\quad \text{stypes}\ \mathit{Cls} = \text{stypes}\ \mathit{Cls}\ \text{using}\ \mathit{Cls}$$

C.1.6 Checking adoptability

Type checking needs to be extended for non-intrusive adoption so that context switching after adoption is restricted under certain circumstances. The example on page 141 illustrates the need for this.

The strategy used is to check in every class where there is a covariant argument declaration (use of “selftype”), whether the outgoing methods will be safe. We consider these classes with selftype argument declarations as specializing the root class of the inheritance chain. Thus we restrict the type declarations of the methods defined for the outgoing messages where an adopted object might be sent to, to be the root (of the corresponding chain) class name.

module EOOL-adoptable

imports EOOL-tc

exports

context-free functions

adoptable “[” CLASSES “]” CLASSES	→ BOOL
check CLASS with ID ID	→ BOOL
topclass “(” ID CLASSES “)”	→ ID
“ctype” “[” EXP “]” TENV CLASSES “with” ID “<” ID	→ TYPE
“check” “[” CLASSES “]” CLASSES “with” ID “<” ID	→ BOOL
“check” “[” METHOD∗ “]” TENV CLASSES “with” ID “<” ID	→ BOOL
“[” {EXP “,”}∗ TENV CLASSES “<” ID-LIST “]selftype/” ID	
“with” ID “<” ID	→ BOOL
covaries CLASS	→ BOOL
covaries ID-LIST	→ BOOL

variables

$$\text{A} \quad \to \text{ID}$$

$$\text{B} \quad \to \text{ID}$$

equations

$$\text{[ET-P0]} \quad \text{tc}[\mathit{Exp}] = \text{type}[\mathit{Exp}]\ []$$

$$\text{[ET-P1]} \quad \frac{\text{tc}[\mathit{Cls}]\ \mathit{Cls} = \text{true}, \text{adoptable}[\mathit{Cls}]\ \mathit{Cls} = \text{true}}{\text{tc}[\mathit{Cls}\ \mathit{Exp}] = \text{type}[\mathit{Exp}]\ []\ \mathit{Cls}}$$

$$\text{[ETA-P0]} \quad \text{adoptable}[]\ \mathit{Cls} = \text{true}$$

$$\text{[ETA-P1]} \quad \frac{\begin{array}{c} \mathit{Cl} = \text{class}\ \mathit{Id} : \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}, \\ \text{covaries}\ \mathit{Cl} = \text{true}, \\ \text{check}[\mathit{Cl}]\ \mathit{Cls}\ \text{with}\ \mathit{Id} < \text{topclass}(\mathit{Id}\ \mathit{Cls}) = \text{true} \end{array}}{\text{adoptable}[\mathit{Cl}\ \mathit{Cls}']\ \mathit{Cls} = \text{adoptable}[\mathit{Cls}']\ \mathit{Cls}}$$

$$\text{[ETA-P2]} \quad \frac{\text{covaries}\ \mathit{Cl} = \text{false}}{\text{adoptable}[\mathit{Cl}\ \mathit{Cls}']\ \mathit{Cls} = \text{adoptable}[\mathit{Cls}']\ \mathit{Cls}}$$

$$\text{[ETA-T0]} \quad \frac{\mathit{Id}' \neq \text{none}}{\text{topclass}(\mathit{Id}\ \mathit{Cls}\ \text{class}\ \mathit{Id} : \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}\ \mathit{Cls}') = \text{topclass}(\mathit{Id}'\ \mathit{Cls}\ \mathit{Cls}')}$$

$$\text{[ETA-T1]} \quad \text{topclass}(\mathit{Id}\ \mathit{Cls}\ \text{class}\ \mathit{Id} : \text{none}\ \mathit{Vdecs}\ \mathit{Meths}\ \mathit{Cls}') = \mathit{Id}$$

$$\text{[ETA-C0]} \quad \text{check}[\text{class}\ \mathit{Id} : \text{none}\ \mathit{Vdecs}\ \mathit{Meths}\ \mathit{Cls}']\ \mathit{Cls}\ \text{with}\ \mathit{Id} < A = \text{true}$$

$$[\text{ETA-C1}]\quad \frac{Id' \neq \text{none}}{\begin{array}{l}\text{check}[\text{class } B : Id' \ Vdecs \ Meths] \ Cls \text{ with } B < A \quad = \\ \text{check}[Meths] \ \text{c2t}[\text{class } B : Id' \ Vdecs \ Meths] \ Cls \text{ with } B < A\end{array}}$$

$$[\text{ETA-M0}]\quad \text{check}[] \ Tenv \ Cls \text{ with } B < A = \text{true}$$

$$[\text{ETA-M1}]\quad \frac{\begin{array}{c} Stypes = \text{stypes } Cls, Id_0 = \text{lookup selftype in } Tenv, \\ Tenv' = [[IdId\text{-}list \ \text{selftype}/ \ Id_0]] \ @ \ Tenv, \\ \text{ctype}[Exp] \ Tenv' \ Cls \text{ with } B < A < Id' \ Stypes \ Cls = \text{true}\end{array}}{\begin{array}{l}\text{check}[\text{meth } Tid : Id'(IdId\text{-}list) \ Exp \ Meths] \ Tenv \ Cls \text{ with } B < A \quad = \\ \text{check}[Meths] \ Tenv \ Cls \text{ with } B < A\end{array}}$$

$$[\text{ETA-E0}]\quad \text{ctype}[\text{nil } Id] \ Tenv \ Cls \text{ with } B < A = Id$$

$$[\text{ETA-E1}]\quad \text{ctype}[\text{new } Id] \ Tenv \ Cls \text{ with } B < A = Id$$

$$[\text{ETA-E2}]\quad \text{ctype}[Nat] \ Tenv \ Cls \text{ with } B < A = \text{natural}$$

$$[\text{ETA-E3}]\quad \frac{\begin{array}{c}\text{ctype}[Exp_1] \ Tenv \ Cls \text{ with } B < A = \text{natural}, \\ \text{ctype}[Exp_2] \ Tenv \ Cls \text{ with } B < A = \text{natural}\end{array}}{\text{ctype}[Exp_1 + Exp_2] \ Tenv \ Cls \text{ with } B < A = \text{natural}}$$

$$[\text{ETA-E4}]\quad \frac{\begin{array}{c}\text{ctype}[Exp_1] \ Tenv \ Cls \text{ with } B < A = \text{natural}, \\ \text{ctype}[Exp_2] \ Tenv \ Cls \text{ with } B < A = \text{natural}\end{array}}{\text{ctype}[Exp_1 > Exp_2] \ Tenv \ Cls \text{ with } B < A = \text{boolean}}$$

$$[\text{ETA-E5}]\quad \text{ctype}[\text{self}] \ Tenv \ Cls \text{ with } B < A = \text{lookup selftype in } Tenv$$

$$[\text{ETA-E6}]\quad \frac{\text{istemp } \mathit{Id} \text{ in } \mathit{Tenv} = \text{true}}{\text{ctype}[\mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A = \text{lookup } \mathit{Id}\ \_\ \text{temp in } \mathit{Tenv}}$$

$$[\text{ETA-E7}]\quad \frac{\text{istemp } \mathit{Id} \text{ in } \mathit{Tenv} = \text{false}}{\text{ctype}[\mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A = \text{lookup } \mathit{Id} \text{ in } \mathit{Tenv}}$$

$$[\text{ETA-E8}]\quad \frac{\begin{array}{c}\text{ctype}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A = \text{boolean},\\ \mathit{Type}_1 = \text{ctype}[\mathit{Exp}_1]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A,\\ \mathit{Type}_2 = \text{ctype}[\mathit{Exp}_2]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A,\\ \mathit{Id} = \mathit{Type}_1 \text{ v } \mathit{Type}_2\ \mathit{Cls}\end{array}}{\text{ctype}[\text{if } \mathit{Exp} \text{ then } \mathit{Exp}_1 \text{ else } \mathit{Exp}_2 \text{ fi}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A = \mathit{Id}}$$

$$[\text{ETA-E9}]\quad \frac{\mathit{Id}_0 = \text{ctype}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A, \mathit{Tenv}' = \text{cid2t}[\mathit{Id}_0]\ \mathit{Cls}}{\text{ctype}[\mathit{Exp}.\ \mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A = \text{ctype}[\mathit{Id}]\ \mathit{Tenv}'\ \mathit{Cls} \text{ with } B < A}$$

$$[\text{ETA-E11a}]\quad \frac{\begin{array}{c}B \neq \text{ctype}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A,\\ B = \text{ctype}[\mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A,\\ A = \text{ctype}[\mathit{Exp}.\ \mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A\end{array}}{\begin{array}{l}\text{ctype}[\mathit{Exp}.\ \mathit{Id} := \mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A \quad = \\ \text{ctype}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A\end{array}}$$

$$[\text{ETA-E11b}]\quad \frac{B \neq \text{ctype}[\mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A}{\begin{array}{l}\text{ctype}[\mathit{Exp}.\ \mathit{Id} := \mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A \quad = \\ \text{ctype}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A\end{array}}$$

$$[\text{ETA-E11c}]\quad \frac{B = \text{ctype}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A}{\text{ctype}[\mathit{Exp}.\ \mathit{Id} := \mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A = B}$$

[ETA-E12] $\text{ctype}[\mathit{Id} := \mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}$ with $B < A =$
$\text{ctype}[\text{self. } \mathit{Id} := \mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}$ with $B < A$

$$\text{[ETA-E13]}\quad \frac{\mathit{Type}_1 = \text{ctype}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A, \quad \mathit{Type}_2 = \text{ctype}[\mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A}{\text{ctype}[\mathit{Exp};\ \mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A = \mathit{Type}_1 \text{ v } \mathit{Type}_2\ \mathit{Cls}}$$

[ETA-E14] $\text{ctype}[\mathit{Id}(\mathit{Exp\text{-}list})]\ \mathit{Tenv}\ \mathit{Cls}$ with $B < A =$
$\text{ctype}[\text{self. } \mathit{Id}(\mathit{Exp\text{-}list})]\ \mathit{Tenv}\ \mathit{Cls}$ with $B < A$

$$\text{[ETA-E15]}\quad \frac{\begin{array}{c} B = \text{ctype}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A, \\ \mathit{Tenv}' = \text{cid2t}[B]\ \mathit{Cls}, \\ \text{lookup } \mathit{Id} \text{ in } \mathit{Tenv}' = \mathit{Id}_1(\mathit{Id\text{-}list}_1), \\ [\mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id\text{-}list}_1]\text{selftype/ } B = \text{true} \end{array}}{\text{ctype}[\mathit{Exp}.\ \mathit{Id}(\mathit{Exp\text{-}list})]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A = \mathit{Id}_1}$$

$$\text{[ETA-E16]}\quad \frac{\begin{array}{c} \mathit{Id}_0 = \text{ctype}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A, \\ \mathit{Id}_0 \neq B,\ \mathit{Tenv}' = \text{cid2t}[\mathit{Id}_0]\ \mathit{Cls}, \\ \text{lookup } \mathit{Id} \text{ in } \mathit{Tenv}' = \mathit{Id}_1(\mathit{Id\text{-}list}_1), \\ [\mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id\text{-}list}_1]\text{selftype/ } \mathit{Id}_0 \text{ with } B < A = \text{true} \end{array}}{\text{ctype}[\mathit{Exp}.\ \mathit{Id}(\mathit{Exp\text{-}list})]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A = \mathit{Id}_1}$$

[ETA-EST0] $[\ \mathit{Tenv}\ \mathit{Cls} <\ ]\text{selftype/ } \mathit{Id}$ with $B < A = \text{true}$

[ETA-EST1] $[\mathit{Exp}, \mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \text{selftype}, \mathit{Id\text{-}list}]\text{selftype/ } \mathit{Id}$ with $B < A =$
$[\mathit{Exp}, \mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id}, \mathit{Id\text{-}list}]\text{selftype/ } \mathit{Id}$ with $B < A$

$$[\text{ETA-EST3}]\quad \frac{\begin{array}{c} \mathit{Id}' \neq \text{selftype}, \mathit{Stypes} = \text{stypes } \mathit{Cls}, \\ \mathit{Type} = \text{ctype}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A, \\ B \neq \mathit{Type}, \mathit{Type} < \mathit{Id}'\ \mathit{Stypes}\ \mathit{Cls} = \text{true} \end{array}}{\begin{array}{l} [\mathit{Exp}, \mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id}', \mathit{Id\text{-}list}]\text{selftype}/\ \mathit{Id} \text{ with } B < A \quad = \\ [\mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id\text{-}list}]\text{selftype}/\ \mathit{Id} \text{ with } B < A \end{array}}$$

$$[\text{ETA-EST4}]\quad \frac{B = \text{ctype}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} \text{ with } B < A}{\begin{array}{l} [\mathit{Exp}, \mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < A, \mathit{Id\text{-}list}]\text{selftype}/\ \mathit{Id} \text{ with } B < A \quad = \\ [\mathit{Exp\text{-}list}\ \mathit{Tenv}\ \mathit{Cls} < \mathit{Id\text{-}list}]\text{selftype}/\ \mathit{Id} \text{ with } B < A \end{array}}$$

[ETA-Cv0] covaries class $\mathit{Id} : \mathit{Id}'\ \mathit{Vdecs}\ = \text{false}$

[ETA-Cv1] covaries class $\mathit{Id} : \mathit{Id}'\ \mathit{Vdecs}$ meth $\mathit{Tid}_0 : \mathit{Id}_0(\mathit{IdId\text{-}list})\ \mathit{Exp}\ \mathit{Meths} =$
covaries type $\mathit{IdId\text{-}list}$ | covaries class $\mathit{Id} : \mathit{Id}'\ \mathit{Vdecs}\ \mathit{Meths}$

[ETA-Cv2] covaries $= \text{false}$

[ETA-Cv3] covaries selftype,$\mathit{Id\text{-}list} = \text{true}$

$$[\text{ETA-Cv4}]\quad \frac{\mathit{Id}' \neq \text{selftype}}{\text{covaries } \mathit{Id}', \mathit{Id\text{-}list} = \text{covaries } \mathit{Id\text{-}list}}$$

C.1.7 Transforming expressions

The tagging operation specified in this section is a homomorphism that maps the user level EOOL syntax (Section C.1.1) to the evaluation level essential syntax that is presented in the next section (Section C.2.1). In order to achieve this transformation the essential syntax for expressions is already included as part of the module EOOL-tagid (Section C.1.2).

module EOOL-ac

imports EOOL-tc

exports

sorts METHODS EXP-LIST

context-free functions

"tag" "[" PROGRAM "]"	→ PROGRAM
"tag" "[" CLASSES "]" CLASSES	→ CLASSES
"tag" "[" METHOD∗ "]" TENV CLASSES	→ METHODS
"tag" "[" EXP "]" TENV CLASSES	→ EXP
"tag" "[[" EXP-LIST "]]" TENV CLASSES	→ EXP-LIST
match ID in TENV	→ TAGID
METHOD∗	→ METHODS
{EXP ","}∗	→ EXP-LIST

equations

[AC-P0] $$\frac{Id = \text{type}[Exp]\ []}{\text{tag}[Exp] = \text{tag}[Exp]\ \text{cid2t}[Id]}$$

[AC-P1] $$\frac{Cls' = \text{tag}[Cls]\ Cls,\ Exp' = \text{tag}[Exp]\ []\ Cls}{\text{tag}[Cls\ \ Exp] = Cls'\ Exp'}$$

[AC-C0] $\text{tag}[]\ Cls =$

$$\text{[AC-C1]}\quad \frac{\begin{array}{c} \mathit{Cl} = \text{class } \mathit{Id} : \mathit{Id}' \ \mathit{Vdecs} \ \mathit{Meths}, \mathit{Cls}'' = \text{tag}[\mathit{Cls}'] \ \mathit{Cls}, \\ \mathit{Meths}' = \text{tag}[\mathit{Meths}] \ \text{c2t}[\text{class } \mathit{Id} : \mathit{Id}' \ \mathit{Vdecs} \ \mathit{Meths}] \ \mathit{Cls} \end{array}}{\text{tag}[\mathit{Cl} \ \mathit{Cls}'] \ \mathit{Cls} = \text{class } \mathit{Id} : \mathit{Id}' \ \mathit{Vdecs} \ \mathit{Meths}' \ \mathit{Cls}''}$$

$$\text{[AC-M0]}\quad \text{tag}[] \ \mathit{Tenv} \ \mathit{Cls} =$$

$$\text{[AC-M1]}\quad \frac{\begin{array}{c} \mathit{Id}_0 = \text{lookup selftype in } \mathit{Tenv}, \\ \mathit{Exp}' = \text{tag}[\mathit{Exp}] \ [[\mathit{IdId\text{-}list} \text{ selftype}/ \ \mathit{Id}_0]] \ @ \ \mathit{Tenv} \ \mathit{Cls}, \\ \mathit{Meths}' = \text{tag}[\mathit{Meths}] \ \mathit{Tenv} \ \mathit{Cls} \end{array}}{\begin{array}{l} \text{tag}[\text{meth } \mathit{Tid} : \mathit{Id}'(\mathit{IdId\text{-}list}) \ \mathit{Exp} \ \mathit{Meths}] \ \mathit{Tenv} \ \mathit{Cls} \quad = \\ \text{meth } \mathit{Tid} : \mathit{Id}'(\mathit{IdId\text{-}list}) \ \mathit{Exp}' \ \mathit{Meths}' \end{array}}$$

$$\text{[AC-E0]}\quad \text{tag}[\text{nil } \mathit{Id}] \ \mathit{Tenv} \ \mathit{Cls}' \ \text{class } \mathit{Id} : \mathit{Id}' \ \mathit{Vdecs} \ \mathit{Meths} \ \mathit{Cls}'' = \text{nil } \mathit{Id}$$

$$\text{[AC-E1]}\quad \text{tag}[\text{new } \mathit{Id}] \ \mathit{Tenv} \ \mathit{Cls}' \ \text{class } \mathit{Id} : \mathit{Id}' \ \mathit{Vdecs} \ \mathit{Meths} \ \mathit{Cls}'' = \text{new } \mathit{Id}$$

$$\text{[AC-E2]}\quad \text{tag}[\mathit{Nat}] \ \mathit{Tenv} \ \mathit{Cls} = \mathit{Nat}$$

$$\text{[AC-E3]}\quad \begin{array}{l} \text{tag}[\mathit{Exp}_1 + \mathit{Exp}_2] \ \mathit{Tenv} \ \mathit{Cls} = \\ \text{tag}[\mathit{Exp}_1] \ \mathit{Tenv} \ \mathit{Cls} + \text{tag}[\mathit{Exp}_2] \ \mathit{Tenv} \ \mathit{Cls} \end{array}$$

$$\text{[AC-E4]}\quad \begin{array}{l} \text{tag}[\mathit{Exp}_1 > \mathit{Exp}_2] \ \mathit{Tenv} \ \mathit{Cls} = \\ \text{tag}[\mathit{Exp}_1] \ \mathit{Tenv} \ \mathit{Cls} > \text{tag}[\mathit{Exp}_2] \ \mathit{Tenv} \ \mathit{Cls} \end{array}$$

$$\text{[AC-E5]}\quad \text{tag}[\text{self}] \ \mathit{Tenv} \ \mathit{Cls} = \text{self}$$

$$\text{[AC-E6]}\quad \frac{\mathit{Id} \neq \text{self}, \text{istemp } \mathit{Id} \text{ in } \mathit{Tenv} = \text{true}}{\text{tag}[\mathit{Id}] \ \mathit{Tenv} \ \mathit{Cls} = \mathit{Id}}$$

$$\text{[AC-E7]}\quad \frac{\text{istemp } \mathit{Id} \text{ in } \mathit{Tenv} = \text{false}}{\text{tag}[\mathit{Id}] \ \mathit{Tenv} \ \mathit{Cls} = \text{match } \mathit{Id} \text{ in } \mathit{Tenv}}$$

[AC-E8] $\text{tag}[\text{if } \mathit{Exp} \text{ then } \mathit{Exp}_1 \text{ else } \mathit{Exp}_2 \text{ fi}]\ \mathit{Tenv}\ \mathit{Cls} =$
$\text{if tag}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} \text{ then tag}[\mathit{Exp}_1]\ \mathit{Tenv}\ \mathit{Cls} \text{ else tag}[\mathit{Exp}_2]\ \mathit{Tenv}\ \mathit{Cls} \text{ fi}$

[AC-E9]
$$\frac{\mathit{Id}_0 = \text{type}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}, \quad \mathit{Tenv}' = \text{cid2t}[\mathit{Id}_0]\ \mathit{Cls}, \quad \mathit{Tid} = \text{tag}[\mathit{Id}]\ \mathit{Tenv}'\ \mathit{Cls}}{\text{tag}[\mathit{Exp}.\ \mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls} = \text{tag}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}.\ \mathit{Tid}}$$

[AC-E10]
$$\frac{\mathit{Id}_0 = \text{type}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}, \quad \mathit{Tenv}' = \text{cid2t}[\mathit{Id}_0]\ \mathit{Cls}, \quad \mathit{Tid} = \text{tag}[\mathit{Id}]\ \mathit{Tenv}'\ \mathit{Cls}}{\text{tag}[\mathit{Exp}.\ \mathit{Id} := \mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls} \ = \ \text{tag}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}.\ \mathit{Tid} := \text{tag}[\mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls}}$$

[AC-E11]
$$\frac{\text{istemp } \mathit{Id} \text{ in } \mathit{Tenv} = \text{false}, \mathit{Tid} = \text{match } \mathit{Id} \text{ in } \mathit{Tenv}}{\text{tag}[\mathit{Id} := \mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} = \mathit{Tid} := \text{tag}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}}$$

[AC-E12]
$$\frac{\text{istemp } \mathit{Id} \text{ in } \mathit{Tenv} = \text{true}}{\text{tag}[\mathit{Id} := \mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls} = \mathit{Id} := \text{tag}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls}}$$

[AC-E13] $\text{tag}[\mathit{Exp};\ \mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls} = \text{tag}[\mathit{Exp}]\ \mathit{Tenv}\ \mathit{Cls};\ \text{tag}[\mathit{Exp}']\ \mathit{Tenv}\ \mathit{Cls}$

[AC-E15]
$$\frac{\mathit{Exp\text{-}list}' = \text{tag}[[\mathit{Exp\text{-}list}]]\ \mathit{Tenv}\ \mathit{Cls}, \quad \mathit{Tid} = \text{tag}[\mathit{Id}]\ \mathit{Tenv}\ \mathit{Cls}}{\text{tag}[\mathit{Id}(\mathit{Exp\text{-}list})]\ \mathit{Tenv}\ \mathit{Cls} = \mathit{Tid}(\mathit{Exp\text{-}list}')}$$

$$\text{[AC-E16]}\quad \frac{\begin{array}{c} \textit{Exp-list}' = \text{tag}[[\textit{Exp-list}]]\ \textit{Tenv}\ \textit{Cls}, \\ \textit{Id}_0 = \text{type}[\textit{Exp}]\ \textit{Tenv}\ \textit{Cls}, \\ \textit{Tenv}' = \text{cid2t}[\textit{Id}_0]\ \textit{Cls}, \\ \textit{Tid} = \text{tag}[\textit{Id}]\ \textit{Tenv}'\ \textit{Cls} \end{array}}{\begin{array}{l} \text{tag}[\textit{Exp}.\ \textit{Id}(\textit{Exp-list})]\ \textit{Tenv}\ \textit{Cls} \quad = \\ \text{tag}[\textit{Exp}]\ \textit{Tenv}\ \textit{Cls}.\ \textit{Tid}(\textit{Exp-list}') \end{array}}$$

$$\text{[AC-X0]}\quad \text{tag}[[]]\ \textit{Tenv}\ \textit{Cls} =$$

$$\text{[AC-X1]}\quad \frac{\begin{array}{c} \textit{Exp-list}'' = \text{tag}[[\textit{Exp-list}]]\ \textit{Tenv}\ \textit{Cls}, \\ \textit{Exp}' = \text{tag}[\textit{Exp}]\ \textit{Tenv}\ \textit{Cls}, \\ \textit{Exp-list}' = \textit{Exp}', \textit{Exp-list}'' \end{array}}{\text{tag}[[\textit{Exp}, \textit{Exp-list}]]\ \textit{Tenv}\ \textit{Cls} = \textit{Exp-list}'}$$

$$\text{[AC-U0]}\quad \text{match } \textit{Id} \text{ in } [] = \textit{Id} \_ \text{oops}$$

$$\text{[AC-U1]}\quad \text{match } \textit{Id} \text{ in } [\textit{Id} \_ \textit{Id}' : \textit{Type}, \textit{Idt-list}] = \textit{Id} \_ \textit{Id}'$$

$$\text{[AC-U2]}\quad \frac{\textit{Id} \neq \textit{Id}_0}{\text{match } \textit{Id} \text{ in } [\textit{Id}_0 \_ \textit{Id}' : \textit{Type}, \textit{Idt-list}] = \text{match } \textit{Id} \text{ in } [\textit{Idt-list}]}$$

$$\text{[AC-U3]}\quad \text{match } \textit{Id} \text{ in } [\textit{Id}_0 : \textit{Type}, \textit{Idt-list}] = \text{match } \textit{Id} \text{ in } [\textit{Idt-list}]$$

The transformation involves tagging the classes and the expression following the classes ([AC-P0, AC-P1]). Tagging a class uses the corresponding type environment to facilitate access to the tagged instance variable and method names ([AC-C0, AC-C1]). Tagging a method involves tagging the method body (EXP) using the formal argument information ([AC-M1]). The type-information can be discarded at this point since the adoption technique, as specified in the module in Section C.2.3, does not used "selftype" or have any need for type information. However, the type information is not discarded since there is a potential for this information to be used if necessary to provide for alternate semantics for adoption.

The process of tagging an expression needs to separate the "temp" tag from other tags. The "temp" tag helps identify the local variables in a method ([AC-E6, AC-E7]), with preference given to local variables in case of conflict, which corresponds to the evaluation semantics (Section C.2.2).

The function "match ID in TENV" returns the tag corresponding to a given identifier. The combination of "istemp" and "match" help in resolving the issue of finding the right match in the presence of temporary variables.

Transforming the example program gives the following program. Note that all the visible methods and instance variable names have been appropriately tagged.

```
class point : none
  var x _ point := 10
  var y _ point := 20
  meth gt _ point : point ( pt : selftype )
    if x _ point > pt . x _ point
    then if y _ point > pt . y _ point
           then self else pt fi
    else pt fi

class use : none
```

```
  meth test _ use : point (  ) new point

class zpoint : point
  var z _ zpoint := 1000
  var x _ point := 10
  var y _ point := 20
  meth gt _ point : zpoint ( zpt : selftype)
    if z _ zpoint + 10 > zpt . z _ zpoint
    then zpt . pgt _ point ( self )
    else self fi
  meth pgt _ point : zpoint ( pt : zpoint)
    if x _ point > pt . x _ point
      then if y _ point > pt . y _ point then self else pt fi
    else pt fi

class newuse : use
  meth test _ use : point (  ) new zpoint

new newuse . test _ use (  ) . gt _ point ( new point )
```

C.2 Dynamic semantics with adoption

Most of this section is a reiteration of the ideas already discussed in Chapter 5.

A program evaluated at the EOOL-eval level will evaluate (with the possibility of not understanding some messages, in-spite of it being type-checked) without the adoption semantics and is therefore comparable to the earlier implementations of Eiffel, as discussed in [16]. The adoption semantics is presented in Section C.2.3.

C.2.1 The essential syntax

This section introduces a language syntax that is similar to that in Section C.1.1 and Section 5.3. The introduction of the syntax again is to indicate the potential clarity of separation between the type-checking aspects and the evaluation aspects involved in the process of adoption. Using type information for providing alternate behavior has also been popularized, in the name of multi-methods [36, 25], but the resulting semantics of such a process is complementary to what can be achieved through adoption. Also the process of adoption does not add the overhead of type-checking during run-time.

The sort NAME defined in the syntax module below indicates the possible use of both ID and TID in the expression. This is apparent in the presence of temporary variables, inclusive of formal arguments. A quick look at the module EOOL-ac (Section C.1.7) indicates that the possible result of the tagging operation in subsumed by this syntax.

module EOOL-esyntax

imports Layout Identifiers Nat Booleans

exports

sorts PROGRAM CLASSES VAR-DEC METHOD EXP CLASS NAME RENAME IDID IDID-LIST
context-free functions

EXP $\rightarrow$ PROGRAM

CLASS∗ EXP	→ PROGRAM
class ID “:” ID VAR-DEC∗ METHOD∗	→ CLASS
var NAME “:=” EXP	→ VAR-DEC
meth NAME “:” ID “(” IDID-LIST “)” EXP	→ METHOD
new NAME	→ EXP
nil NAME	→ EXP
EXP “.” NAME	→ EXP
EXP “.” NAME “(” {EXP “,”}∗ “)”	→ EXP
EXP “.” NAME “:=” EXP	→ EXP
if EXP then EXP else EXP fi	→ EXP
NAME	→ EXP
NAME “:=” EXP	→ EXP
EXP “;” EXP	→ EXP {**left**}
NAME “(” {EXP “,”}∗ “)”	→ EXP
self	→ EXP
“(” EXP “)”	→ EXP {**bracket**}
NAT	→ EXP
EXP “+” EXP	→ EXP {**left**}
EXP “>” EXP	→ EXP
ID “:” ID	→ IDID
{IDID “,”}∗	→ IDID-LIST
CLASS∗	→ CLASSES
ID	→ NAME
ID “_” ID	→ NAME

variables

Cls $[0\text{–}9']*$ $\rightarrow$ CLASS$*$

Cl $[0\text{–}9']*$ $\rightarrow$ CLASS

$Meth$ $[0\text{–}9']*$ $\rightarrow$ METHOD

$Meths$ $[0\text{–}9']*$ $\rightarrow$ METHOD$*$

Exp $[0\text{–}9']*$ $\rightarrow$ EXP

$Exp\text{-}list$ $[0\text{–}9']*$ $\rightarrow$ {EXP ","}$*$

$Vdec$ $[0\text{–}9']*$ $\rightarrow$ VAR-DEC

$Vdecs$ $[0\text{–}9']*$ $\rightarrow$ VAR-DEC$*$

$Id\text{-}list$ $[0\text{–}9']*$ $\rightarrow$ {ID ","}$*$

$IdId$ $[0\text{–}9']*$ $\rightarrow$ IDID

$Name$ $[0\text{–}9']*$ $\rightarrow$ NAME

$IdId\text{-}list$ $[0\text{–}9']*$ $\rightarrow$ {IDID ","}$*$

priorities

{";" ,EXP "+" EXP → EXP ,EXP ">" EXP → EXP}

< {":="}

< {EXP "." NAME "(" {EXP ","}∗ ")" → EXP ,EXP "." NAME → EXP }

C.2.2 Usual semantics

The modules in this section are presented for completeness and are analogous to those presented in Chapter 5.

module EOOL-venv

imports EOOL-esyntax

exports

sorts VALUE VPAIR VENV ID-LIST VALUE-LIST

context-free functions

"(" NAME ":" VALUE ")"	→ VPAIR
"[" {VPAIR ","}* "]"	→ VENV
VENV "@" VENV	→ VENV **{left}**
VENV "&" VENV	→ VENV
nilvalue	→ VALUE
NAT	→ VALUE
BOOL	→ VALUE
VENV	→ VALUE
"(" ID-LIST ")" EXP	→ VALUE
"IF" VALUE "THEN" VALUE "ELSE" VALUE	→ VALUE
lookup NAME in VENV	→ VALUE
update NAME in VENV "by" VALUE	→ VENV
NAME "isin" VENV	→ BOOL
{ID ","}*	→ ID-LIST
{VALUE ","}*	→ VALUE-LIST
"(" ID-LIST ")" ":" "(" VALUE-LIST ")"	→ VENV

variables

Value [*0*– *9′*]*	→ VALUE
Value-list [*0*– *9′*]*	→ {VALUE ","}*
VP-list [*0*– *9′*]*	→ {VPAIR ","}*
Venv [*0*– *9′*]*	→ VENV

equations

[Vv-IF0] IF true THEN Value ELSE $\mathit{Value}' = \mathit{Value}$

[Vv-IF1] IF false THEN Value ELSE $\mathit{Value}' = \mathit{Value}'$

[Vv-I0] Name isin [] = false

[Vv-I1] Name isin $[(\mathit{Name} : \mathit{Value}), \mathit{VP\text{-}list}']$ = true

[Vv-I2] $$\frac{\mathit{Name} \neq \mathit{Name}'}{\mathit{Name} \text{ isin } [(\mathit{Name}' : \mathit{Value}), \mathit{VP\text{-}list}'] = \mathit{Name} \text{ isin } [\mathit{VP\text{-}list}']}$$

[Vv-I3] Name isin Venv & $\mathit{Venv}' = \mathit{Name}$ isin Venv | Name isin Venv'

[Vv-L0] lookup Name in $[(\mathit{Name} : \mathit{Value}), \mathit{VP\text{-}list}] = \mathit{Value}$

[Vv-L1] $$\frac{\mathit{Name} \neq \mathit{Name}'}{\text{lookup } \mathit{Name} \text{ in } [(\mathit{Name}' : \mathit{Value}), \mathit{VP\text{-}list}] = \text{lookup } \mathit{Name} \text{ in } [\mathit{VP\text{-}list}]}$$

[Vv-L2] $$\frac{\begin{array}{c}\mathit{Value}_0 = \text{lookup } \mathit{Name} \text{ in } \mathit{Venv},\ \mathit{Value}_1 = \text{lookup } \mathit{Name} \text{ in } \mathit{Venv}',\\ \mathit{Value}_2 = \text{IF } \mathit{Name} \text{ isin } \mathit{Venv} \text{ THEN } \mathit{Value}_0 \text{ ELSE } \mathit{Value}_1\end{array}}{\text{lookup } \mathit{Name} \text{ in } \mathit{Venv} \,\&\, \mathit{Venv}' = \mathit{Value}_2}$$

[Vv-U0] update Name in [] by $\mathit{Value} = [(\mathit{Name} : \mathit{Value})]$

[Vv-U1] update Name in $[\mathit{VP\text{-}list}]$ by $\mathit{Value} = [(\mathit{Name} : \mathit{Value}), \mathit{VP\text{-}list}]$

$$[\text{Vv-U2}]\quad \frac{\begin{array}{c} \mathit{Venv}_0 = \text{update } \mathit{Name} \text{ in } \mathit{Venv} \text{ by } \mathit{Value} \ \&\ \mathit{Venv}', \\ \mathit{Venv}_1 = \mathit{Venv} \ \&\ \text{update } \mathit{Name} \text{ in } \mathit{Venv}' \text{ by } \mathit{Value}, \\ \mathit{Venv}_2 = \text{IF } \mathit{Name} \text{ isin } \mathit{Venv} \text{ THEN } \mathit{Venv}_0 \text{ ELSE } \mathit{Venv}_1 \end{array}}{\text{update } \mathit{Name} \text{ in } \mathit{Venv} \ \&\ \mathit{Venv}' \text{ by } \mathit{Value} = \mathit{Venv}_2}$$

$$[\text{Vv-A0}]\quad [\mathit{VP\text{-}list}_1] \ @\ [\mathit{VP\text{-}list}_2] = [\mathit{VP\text{-}list}_1, \mathit{VP\text{-}list}_2]$$

$$[\text{Vv-LP}]\quad \begin{array}{l} [\mathit{VP\text{-}list}_1,(\mathit{Name} : \mathit{Value}), \mathit{VP\text{-}list}_2,(\mathit{Name} : \mathit{Value}'), \mathit{VP\text{-}list}_3] = \\ [\mathit{VP\text{-}list}_1,(\mathit{Name} : \mathit{Value}), \mathit{VP\text{-}list}_2, \mathit{VP\text{-}list}_3] \end{array}$$

$$[\text{Vv-X0}]\quad () : () = []$$

$$[\text{Vv-X1}]\quad (\mathit{Id},\mathit{Id\text{-}list}) : (\mathit{Value}, \mathit{Value\text{-}list}) = [(\mathit{Id} : \mathit{Value})] \ @\ (\mathit{Id\text{-}list}) : (\mathit{Value\text{-}list})$$

module EOOL-c2o

imports EOOL-venv

exports

context-free functions

c2o "[" ID "]" CLASSES	→ VALUE
v2o "[" VAR-DEC∗ "]" CLASSES	→ VENV
m2o "[" METHOD∗ "]" CLASSES	→ VENV
eval EXP CLASSES	→ VALUE
varlist {IDID ","}∗	→ ID-LIST

variables

F-args [*0–9′*]∗	→ {ID ","}∗
A-args [*0–9′*]∗	→ {VALUE ","}∗

equations

[C2O0] c2o[natural] $Cls = 0$

$$\text{[C2O1]}\ \frac{\begin{array}{c} Id \neq \text{natural}, \\ Venv_1 = \text{v2o}[Vdecs]\ Cls\ \text{class}\ Id : Id'\ Vdecs\ Meths\ Cls', \\ Venv_2 = \text{m2o}[Meths]\ Cls\ \text{class}\ Id : Id'\ Vdecs\ Meths\ Cls' \end{array}}{\text{c2o}[Id]\ Cls\ \text{class}\ Id : Id'\ Vdecs\ Meths\ Cls' = Venv_1\ @\ Venv_2}$$

[V2O0] v2o[] Cls = []

[V2O1] v2o[var $Name$:= Exp $Vdecs$] Cls =
[($Name$: eval Exp Cls)] @ v2o[$Vdecs$] Cls

[M2O0] m2o[] Cls = []

$$\text{[M2O1]}\ \frac{F\text{-}args = \text{varlist}\ IdId\text{-}list}{\begin{array}{l} \text{m2o[meth}\ Name : Id'(IdId\text{-}list)\ Exp\ Meths]\ Cls\ = \\ [(Name : (F\text{-}args)\ Exp)]\ @\ \text{m2o}[Meths]\ Cls \end{array}}$$

[U2O0] varlist =

$$\text{[U2O1]}\ \frac{Id\text{-}list = \text{varlist}\ IdId\text{-}list}{\text{varlist}\ Id : Id', IdId\text{-}list = Id, Id\text{-}list}$$

module EOOL-eval

imports EOOL-c2o

exports

context-free functions

"[[" PROGRAM "]]"	$\to$	VALUE
"[[" EXP "]]" VENV CLASSES	$\to$	VALUE
"[[" {EXP ","}* "]]*" VENV CLASSES	$\to$	VALUE-LIST
"value" VALUE	$\to$	VALUE

variables

Oenv [*0–9*′]*	$\to$	VENV
Self [*0–9*′]*	$\to$	VENV
Program	$\to$	PROGRAM

equations

[EE-P0] $[[Exp]] = [[Exp]]\ []$

[EE-P1] $[[Cls\ \ Exp]] = [[Exp]]\ []\ \ Cls$

[EE-Ni] $[[\text{nil}\ \ Id]]\ \ Self\ \ Cls = \text{nilvalue}$

[EE-Nw] $[[\text{new}\ \ Id]]\ \ Self\ \ Cls = \text{c2o}[Id]\ \ Cls$

[EE-N0] $[[Nat]]\ \ Self\ \ Cls = Nat$

$$\text{[EE-N1]}\quad \frac{\begin{array}{c} Nat_1 = [[Exp_1]]\ \ Self\ \ Cls, \\ Nat_2 = [[Exp_2]]\ \ Self\ \ Cls \end{array}}{[[Exp_1 + Exp_2]]\ \ Self\ \ Cls = Nat_1 + Nat_2}$$

$$[\text{EE-N2}]\quad \frac{\begin{array}{c} Nat_1 = [[Exp_1]]\ Self\ \ Cls, \\ Nat_2 = [[Exp_2]]\ Self\ \ Cls \end{array}}{[[Exp_1 > Exp_2]]\ Self\ \ Cls = Nat_1 > Nat_2}$$

$$[\text{EE-I0}]\quad [[\text{self}]]\ Self\ \ Cls = Self$$

$$[\text{EE-I1}]\quad \frac{Name \neq \text{self}}{[[Name]]\ Self\ \ Cls = \text{lookup}\ Name\ \text{in}\ Self}$$

$$[\text{EE-C0}]\quad \frac{Bool = [[Exp]]\ Self\ \ Cls}{\begin{array}{l} [[\text{if}\ Exp\ \text{then}\ Exp_1\ \text{else}\ Exp_2\ \text{fi}]]\ Self\ \ Cls \quad = \\ \text{IF}\ Bool\ \text{THEN}\ [[Exp_1]]\ Self\ \ Cls\ \text{ELSE}\ [[Exp_2]]\ Self\ \ Cls \end{array}}$$

$$[\text{EE-E0}]\quad \frac{Oenv = [[Exp]]\ Self\ \ Cls}{[[Exp.\ Name]]\ Self\ \ Cls = [[Name]]\ Oenv\ \ Cls}$$

$$[\text{EE-E1}]\quad [[Name(Exp\text{-}list)]]\ Self\ \ Cls = [[\text{self}.\ Name(Exp\text{-}list)]]\ Self\ \ Cls$$

$$[\text{EE-E2}]\quad \frac{\begin{array}{c} Oenv = [[Exp]]\ Self\ \ Cls, \\ \text{lookup}\ Name\ \text{in}\ Oenv = (Id\text{-}list)\ Exp', \\ A\text{-}args = [[Exp\text{-}list]]*\ Self\ \ Cls, \\ Value = [[Exp']]\ (Id\text{-}list) : (A\text{-}args)\ \&\ Oenv\ \ Cls \end{array}}{[[Exp.\ Name(Exp\text{-}list)]]\ Self\ \ Cls = \text{value}\ Value}$$

$$[\text{EE-A1}]\quad \begin{array}{l} [[Name := Exp]]\ Self\ \ Cls = \\ \text{update}\ Name\ \text{in}\ Self\ \text{by}\ [[Exp]]\ Self\ \ Cls \end{array}$$

$$[\text{EE-A2}]\quad \frac{\begin{array}{c} Self_1 = [[Exp]]\ Self\ \ Cls, \\ Self_2 = [[Exp']]\ Self_1\ \ Cls \end{array}}{[[Exp;\ Exp']]\ Self\ \ Cls = Self_2}$$

$$\text{[EE-A3]}\quad \frac{\mathit{Oenv} = [[\mathit{Exp}]]\ \mathit{Self}\ \ \mathit{Cls}}{\begin{array}{l}[[\mathit{Exp}.\ \mathit{Name} := \mathit{Exp}']]\ \mathit{Self}\ \ \mathit{Cls} \quad = \\ \text{update } \mathit{Name} \text{ in } \mathit{Oenv} \text{ by } [[\mathit{Exp}']]\ \mathit{Self}\ \ \mathit{Cls}\end{array}}$$

$$\text{[EE-X0]}\quad [[]]*\ \mathit{Self}\ \ \mathit{Cls} =$$

$$\text{[EE-X1]}\quad \frac{\begin{array}{c}\mathit{Value} = [[\mathit{Exp}]]\ \mathit{Self}\ \ \mathit{Cls}, \\ \mathit{Value\text{-}list} = [[\mathit{Exp\text{-}list}]]*\ \mathit{Self}\ \ \mathit{Cls}\end{array}}{[[\mathit{Exp}, \mathit{Exp\text{-}list}]]*\ \mathit{Self}\ \ \mathit{Cls} = \mathit{Value}, \mathit{Value\text{-}list}}$$

$$\text{[EE-V0]}\quad \text{value } \mathit{Nat} = \mathit{Nat}$$

$$\text{[EE-V1]}\quad \text{value } \mathit{Bool} = \mathit{Bool}$$

$$\text{[EE-V2]}\quad \text{value } [\mathit{VP\text{-}list}] = [\mathit{VP\text{-}list}]$$

$$\text{[EE-V3]}\quad \text{value } \mathit{Venv}\ \&\ \mathit{Venv}' = \mathit{Venv}'$$

$$\text{[EE-Cx]}\quad \text{eval } \mathit{Exp}\ \ \mathit{Cls} = [[\mathit{Exp}]]\ []\ \ \mathit{Cls}$$

C.2.3 Adoption semantics

The three cases that require handling messages (instance variable lookup, update and method invocation) are considered here. The basic strategy used is to donate a method or instance variable of *Self* environment when some message is not understood by an object.

module EOOL-adopt

imports EOOL-eval

equations

$$[\text{EA-Vfix}]\quad \frac{\mathit{Venv}' = \mathit{Venv}_0 \ \&\ \mathit{Venv}_1,\ \mathit{Venv}_2 = \mathit{Venv}\ @\ \mathit{Venv}_1}{\mathit{Venv}\ @\ \mathit{Venv}' = \mathit{Venv}_0 \ \&\ \mathit{Venv}_2}$$

$$[\text{EA-M0}]\quad \frac{\begin{array}{c} \mathit{Oenv} = [[\mathit{Exp}]]\ \mathit{Self}\ \ \mathit{Cls},\ \mathit{Name}\ \text{isin}\ \mathit{Oenv} = \text{false}, \\ \text{lookup}\ \mathit{Name}\ \text{in}\ \mathit{Self} = (\mathit{Id\text{-}list})\ \mathit{Exp}', \\ \mathit{Self}' = \text{value}\ \mathit{Self},\ \mathit{Oenv}' = \mathit{Oenv}\ @\ \mathit{Self}', \\ \mathit{A\text{-}args} = [[\mathit{Exp\text{-}list}]]*\ \mathit{Self}\ \ \mathit{Cls}, \\ \mathit{Value} = [[\mathit{Exp}']]\ (\mathit{Id\text{-}list}) : (\mathit{A\text{-}args})\ \&\ \mathit{Oenv}'\ \mathit{Cls} \end{array}}{[[\mathit{Exp}.\ \mathit{Name}(\mathit{Exp\text{-}list})]]\ \mathit{Self}\ \ \mathit{Cls} = \text{value}\ \mathit{Value}}$$

$$[\text{EA-E0}]\quad \frac{\begin{array}{c} \mathit{Oenv} = [[\mathit{Exp}]]\ \mathit{Self}\ \ \mathit{Cls}, \\ \mathit{Name}\ \text{isin}\ \mathit{Oenv} = \text{false},\ \mathit{Self}' = \text{value}\ \mathit{Self} \end{array}}{[[\mathit{Exp}.\ \mathit{Name}]]\ \mathit{Self}\ \ \mathit{Cls} = [[\mathit{Name}]]\ \mathit{Oenv}\ @\ \mathit{Self}'\ \mathit{Cls}}$$

$$[\text{EA-A5}]\quad \frac{\begin{array}{c} \mathit{Oenv} = [[\mathit{Exp}]]\ \mathit{Self}\ \ \mathit{Cls}, \\ \mathit{Name}\ \text{isin}\ \mathit{Oenv} = \text{false},\ \mathit{Self}' = \text{value}\ \mathit{Self} \end{array}}{\begin{array}{l} [[\mathit{Exp}.\ \mathit{Name} := \mathit{Exp}']]\ \mathit{Self}\ \ \mathit{Cls}\quad = \\ \text{update}\ \mathit{Name}\ \text{in}\ \mathit{Oenv}\ @\ \mathit{Self}'\ \text{by}\ [[\mathit{Exp}']]\ \mathit{Self}\ \ \mathit{Cls} \end{array}}$$

All three equations ([EA-M0, EA-E0 and EA-A5]) use the technique of donating the *Self* environment for the object in context, when an object *does not understand* a message sent to it. This particular adoption technique preserves the objects own attributes, while adding new ones only on a demand for specialization and the type-checking process eliminates meaningless adoptions in general.

REFERENCES

[1] P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of Conference on European Conference on Object-Oriented Programming*, pages 234–242, 1987.

[2] H. P. Barendregt. *The lambda calculus, its syntax and semantics*. North Holland Publishing Company, Amsterdam, 1984. 2-nd revised edition.

[3] J. Bergstra, J. Heering, and P. Klint. Algebraic definition of a simple programming language. Report CS-R8504, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1985.

[4] J. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[5] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the emerald system. In *Proceedings of Conference on Object Oriented Programming, Systems and Languages*, pages 78–86, 1986.

[6] K. Bruce. *Private Communication*. Electronic mail, 1991.

[7] K. B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. Technical Report CS-92-01, Williams College, Williamstown, MA 01267, January 1992.

[8] P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded quantification for object–oriented programming. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[9] L. Cardelli. Semantics of multiple inheritance. In *Semantics of Datatypes, LNCS 173*, pages 51–68, 1984.

[10] L. Cardelli. Typeful programming. Technical report, DEC, May 1989.

[11] L. Cardelli. Typed foundations of object–oriented programming. Technical Report To appear, DEC Systems Research Center, 1992.

[12] L. Cardelli, J. Donahue, L. Galssman, M. Jordan, B. Kalsow, and G. Nelson. Modula–3 report. Technical Report SRC-31, DEC Systems Research Center, 1988.

[13] L. Cardelli and J. Mitchell. Operations on records. In *Proceedings of the Fifth Conference on Mathematical Foundations of Programming Language Semantics*, 1989.

[14] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[15] W. Cook. Denotational semantics of inheritance. Technical Report CS-89-33, Brown University, 1989.

[16] W. Cook. A proposal for making eiffel type-safe. In *Proceedings of Conference on European Conference on Object-Oriented Programming*, pages 57–73, 1989.

[17] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceedings of 19th ACM Symposium on Principles of Programming Languages*, pages 125–135, 1990.

[18] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings of Conference on Object Oriented Programming, Systems and Languages*, pages 433–443, 1989.

[19] B. J. Cox. *Object Oriented Programming.* Addison-Wesley, 1986.

[20] O.-J. Dahl and K. Nygaard. Simula – an algol-based simulation language. *Commun. ACM*, 9(9):671–678, 1966.

[21] S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1):29–72, 1988.

[22] A. v. Deursen. An algebraic specification for the static semantics of Pascal. In J. v. Leeuwen, editor, *Conference Proceedings of Computing Science in the Netherlands, CSN'91*, pages 150–164. SION, 1991.

[23] A. C. Fleck. A proposal for the comparison of types in pascal and associated semantic models. *Computer Languages*, 9(2):71–87, 1984.

[24] G. Ghelli. *Private Communication.* Electronic mail, 1991.

[25] G. Ghelli. A static type system for message passing. In *Proceedings of Conference on Object Oriented Programming, Systems and Languages*, pages 129–145, November 1991. Also in *SIGPLAN notices*, 26(11).

[26] A. Goldberg and D. Robson. *SmallTalk 80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[27] M. J. C. Gordon. *The Denotational Description of Programming Languages: An Introduction.* Springer–Verlag, New York/Heidelberg/Berlin, 1979.

[28] J. Heering, P. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[29] J. Heering and P. Klint. PICO revisited. In J. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 359–379. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 9.

[30] P. Hendriks. ASF system user's guide. Report CS-R8823, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1988. Extended abstract in: Conference Proceedings of Computing Science in the Netherlands, CSN'88 1, pp. 83-94, SION (1988).

[31] P. Hendriks. *Implementation of Modular Algebraic Specifications*. PhD thesis, University of Amsterdam, 1991.

[32] A. V. Hense. Denotational semantics of an object oriented language with explicit wrappers. Technical Report A 11/90, Universitat des Saarlandes, 1990.

[33] S. Kamin. Inheritance in smalltalk-80: A denotational definition. In *Proceedings of 15th ACM Symposium on Principles of Programming Languages*, pages 80–87, 1988.

[34] S. Kamin and U. Reddy. Equivalence of abstract and implementation semantics of inheritance. In *Submitted for Publication*. 1992.

[35] S. Kaplan. Positive/negative conditional rewriting. In S. Kaplan and J.-P. Jouannaud, editors, *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 129–143. Springer-Verlag, 1988.

[36] S. E. Keene. *Object-oriented programming in Common Lisp : a programmer's guide to CLOS*. Addison-Wesley, Reading, Massachusetts, 1989.

[37] P. Klint. Lazy scanner generation for modular regular grammars. Report CS-R9158, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1991.

[38] B. Kristensen, O. Madsen, B. Pedersen, and K. Nygaard. The beta programming language. In Shriver and Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. The MIT Press, Cambridge, Massachusetts, 1987.

[39] D. Lea. Technical note: Techniques for representing class sytems as type systems. Mailing list: objecttypes@daimi.aau.dk, October 1991.

[40] G. Leavens and W. Weihl. Reasoning about object-oriented programs that use subtypes. In *Proceedings of Conference on Object Oriented Programming, Systems and Languages/European Conference on Object-Oriented Programming*, pages 212–223, 1990.

[41] H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *Proceedings of Conference on Object Oriented Programming, Systems and Languages*, 1986.

[42] O. Madsen, B. Magnusson, and B. Moller-Pedersen. Strong typing of object-oriented languages revisited. In *Proceedings of Conference on Object Oriented Programming, Systems and Languages*, pages 140–149, 1990.

[43] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[44] B. Meyer. *Static typing for Eiffel.* USENET comp.object, July 1990.

[45] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1991.

[46] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[47] J. Mitchell. Towards a typed foundation for method specialization and inheritance. In *Proceedings of 19th ACM Symposium on Principles of Programming Languages*, 1990.

[48] S. M. Omohundro. Sather. *Eiffel Outlook*, 1991.

[49] J. Palsberg and M. Schwartzbach. What is type–safe code reuse? In *Proceedings of Conference on European Conference on Object-Oriented Programming*, 1991.

[50] J. Palsberg and M. Schwartzback. Static typing for object–oriented programming. Technical Report Aarhus University, Computer Science Department, 1991.

[51] U. Reddy. Objects as closures: Abstract semantics of object–oriented languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 289–297, 1988.

[52] J. Rees and N. Adams. T, a dialect of lisp, or lambda: the ultimate software tool. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 114–122, 1982.

[53] C. Schaffert, T. Coope, B. Bullis, M. Kilian, and C. Wilpolt. An introduction to trellis/owl. In *Proceedings of Conference on Object Oriented Programming, Systems and Languages*, pages 9–16, 1986.

[54] A. Snyder. Inheritance and the development of encapsulated software components. In Shriver and Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 165–188. The MIT Press, Cambridge, Massachusetts, 1987.

[55] L. A. Stein. Delegation is inheritance. In *Proceedings of Conference on Object Oriented Programming, Systems and Languages*, pages 138–146, 1987.

[56] L. A. Stein, H. Lieberman, and D. Ungar. A shared view of sharing: *Treaty of Orlando*. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, Frontier Series, pages 31–48. ACM Press, 1989.

[57] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, Massachusetts, 1977.

[58] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[59] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings of Conference on Object Oriented Programming, Systems and Languages*, pages 227–242, 1987.

[60] P. Wegner. Dimensions of object-based language design. In *Proceedings of Conference on Object Oriented Programming, Systems and Languages*, 1987.

[61] N. Wirth. The programming language oberon. *Software – Practice and Experience*, 18:671–690, 1988.