Polymorphism and Object-Oriented Languages

Mitch Cherniack

A Thesis

in

The Department

of

Computer Science

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

August, 1992

ISBN 0-315-80915-9

Abstract

Polymorphism and Object-Oriented Languages

Mitch Cherniack

Object-oriented languages are potentially polymorphic. Inheritance alone can achieve three of four polymorphisms identified by Cardelli and Wegner [CaWe85]: **overloading**, **genericity** and **inclusion**. Unfortunately, the use of inheritance to achieve multiple forms of polymorphism is problematic. Redefinition policies that make inheritance useful can also make inheritance unsafe. This is relevant to the design of production languages, where it must be guaranteed that objects understand all messages received (*strong typing*), and where it is desirable to make this guarantee at compile-time (*static typing*). Our objective is to establish inheritance redefinition policies that are flexible but that do not compromise static and strong typing.

This thesis has three components: analysis, survey and design. The **analysis** examines the Cardelli-Wegner polymorphism taxonomy and identifies three tensions of inheritance policies; one for each polymorphism potentially captured by inheritance. In the **survey** section, language proposals of Bruce ([Br92]) and Palsberg and Schwartzbach ([PaSc90a - PaSc90d, PaSc91]) are evaluated by how they resolve these tensions. We conclude that the latter proposal (PS theory) offers greater potential for flexible, type-safe polymorphism even though formalizations of the theory prohibit some useful classes and are incompatible with multiple inheritance and inclusion polymorphism. The **design** section addresses these limitations. An alternative interpretation is introduced that removes the restrictions on its use, and is easily extended to incorporate multiple inheritance. A prototype parser implementing alternative algorithms is described, and its output presented. Finally, preliminary work is shown extending this proposal to incorporate inclusion polymorphism, laying the groundwork for future study.

Table of Contents

List of Examples

List of Tables

List of Figures

List of Symbols

Sequences, Sets and Set Operations

$\{a_1, \ldots, a_n\}$	-	The set of items: $a_1, a_2, \ldots, a_n$
$\{x \mid p\}$	-	The set of all x's such that p holds
$«a_1, \ldots, a_n»$	-	The sequence of items: $a_1, a_2, \ldots, a_n$
$\{\}$ or $\emptyset$	-	The empty set
$S_1 \cup S_2$	-	The union of S_1 and S_2
$S_1 \cap S_2$	-	The intersection of S_1 and S_2
$S_1 \subseteq S_2$	-	S_1 is a subset of S_2
$S_1 \subset S_2$	-	S_1 is a proper subset of S_2
$S_1 \supseteq S_2$	-	S_1 is a superset of S_2
$S_1 \supset S_2$	-	S_1 is a proper superset of S_2
$i \in S$	-	i is an element of S
$i \notin S$	-	i is not an element of S

Symbols Defined in the Text of the Thesis:

$\mathtt{A}\,[\mathtt{B} \leftarrow \mathtt{C}]$	-	The class, A with C substituted for B
$\mathtt{A}\,[\mathtt{B} \leftarrow_{\mathrm{Deep}} \mathtt{C}]$	-	The class, A with C deeply substituted for B
$\mathtt{A} \lhd \mathtt{B}$	-	B is a subclass of A
$\mathtt{A} \lhd_I \mathtt{B}$	-	B is a subclass by inheritance of A
$\mathtt{A} \lhd_e \mathtt{B}$	-	B is an equivariant subclass of A
$\mathtt{A} \cong \mathtt{B}$	-	B is an equivalent class to A
$\mathtt{A} \ncong \mathtt{B}$	-	B is not an equivalent class to A
$\uparrow\mathtt{A}$	-	The set of all subclasses of A

$\mathsf{C} <_{\cap} T$	-	The class, C "narrowed by intersection" with type, T
Ω	-	The empty L-Tree
$\Diamond$	-	The recursive L-Tree

Chapter 1.

Introduction

Approaches to language design can be viewed as top-down or bottom-up. A top-down approach begins with some mathematical model that is given representation as a computer language. Functional and logic languages such as Lisp and Prolog were developed using this type of approach. Abstractions of machine operation constitute the bottom-up approach. The evolution of imperative languages from "low-level" machine languages to block structured and modular languages demonstrates the bottom-up approach carried out in increasing degrees.

As is true with their analogs in program design, top-down and bottom-up approaches should be viewed as ends of a spectrum rather than distinct categories. Mathematical models are compromised to add practical capabilities to functional and logic languages. Lisp supports functions with side-effects and Prolog includes control operations in the interests of efficiency and practicality. Imperative languages are buttressed by mathematical theory such as Hoare laws [Ho69], designed as a response to rather than as a basis for their designs.

A recurring theme of studies of the Object-Oriented Programming (OOP) paradigm is that it can be viewed in conflicting terms. In the context of language design, OOP languages can appear at either end of the top-down/bottom-up spectrum. The disparate views result from the definition of OOP being made only in retrospect. Simula is regarded as the first OOP language though its designers made no such claim in their original proposal [DaMyNy70]. Abstract data types (ADT's) are considered the mathematical model for the modular aspect of OOP languages, though ADT's were introduced 7 years after the appearance of Simula and without reference to OOP languages [LiZi74]. That Simula extended Algol 60 makes OOP seem a result of bottom-up design. That the Abstract Data Type is the mathematical basis for OOP modules or classes, makes the paradigm seem top-down.

1.1 Object-Oriented Terminology

The disparate views of object-oriented programming makes a consensus as to its meaning elusive. Typically, object-oriented languages support a number of features that allow source code to be modularized and easily reused. Common to all object languages is the anthropomorphic notion of an *object*; a run-time entity that can respond to messages and has identity. In many languages, the transmission of *messages* (or *invocations*) to objects resembles record selection in Pascal. Given some object, o that can respond to the message, m, one would *"send a message to o"* or *"invoke m on o"* by writing:

```
o.m
```

How an object responds to a message is dependent on whether it is defined in a *prototype-based* or *class-based* language [Ta91]. In a prototype-based language, each object has the capability of responding to a given message in a unique way. In a class-based language, an object's response depends on its *class*; a template that defines the protocol and behavior for some collection of conceptually related objects. Only class-based languages are discussed any further in this thesis.

An object is said to be *"of class C"* or *"an instance of C"* when the object responds to messages as the class definition for C dictates. A class definition can contain instance variable declarations, and method definitions. An *instance variable* is a variable defined in a class' definition, and *local* to all methods defined in the same class. The value of an object's instance variable is another object bound to the variable via an assignment statement. Frequently, languages consider instance variables to be *private* to a class' definition, and demand that access methods be defined if update and referencing privileges are to be granted externally. Some languages such as Eiffel [Me88] and Dee [Gr91] relax the reference restrictions on instance variables, and allow a value to be examined externally by sending the variable name as a message to the object which owns it. In such languages, instance variables are *public.*

A *method* is a procedure defined within a class definition. It can include *local variables* and *parameters,* and is understood to include an implicit parameter named `self`, which is bound at run-time to the object on which the method is invoked (the *receiver object*). The code of a method can include statements that send messages to instance variables from the same class, or to local variables and parameters (including `self`). The code of a method can also include statements that bind objects to instance variables from the same class and to local variables. Parameters are bound to arguments when the method is invoked. `Self` is automatically bound at runtime to the receiving object of the most recently processed message.

Signatures are associated with method and variable definitions, and their invocations. A *definition signature* specifies how a message should be sent to a receiving object to trigger invocation. The collection of definition signatures contained within a class is the class' *interface.* The invocation signature resembles the definition signature except that information about arguments rather than parameters is specified. It should be presumed in the object-oriented context that invocation signatures include information about the receiver object argument, and correspondingly, that definition signatures include information about the implicit parameter, `self`. We refer to *explicit* invocation and definition signatures when the implicit components of signatures are to be ignored.

A class' method definition need only include a signature and not code, provided that the code is defined elsewhere (see 1.3.2). A method defined without code is an *abstract method.* An *abstract class'* definition includes only abstract methods. A *partially abstract class* defines at least one abstract method. A *concrete class* includes code for all of its methods.

The amount of information that signatures reveal depends on whether the language involved is *typed* or *untyped.* A method signature in any language reveals the method name and the number of parameters/arguments demanded of the message invoking the method. Variable signatures reveal variable names in languages in which variables are public. A signature in a typed language such as Eiffel, Dee or C++ also associates a type with each variable, parameter, argument and method.

1.2 Types

In the context of programming languages, types offer information about values that are unknown until run-time. The *type of an expression* reveals what values it can take, and therefore what operations can be invoked on it. The operation, '+' for example, can be applied to the Integer values 3 and 4, but not to the Boolean values TRUE and FALSE. In object-oriented languages, the type of an expression can be expressed as a set of classes whose instances can be referred to by the expression.

A typed object-oriented language demands that every variable in a class be associated with a type that specifies the expressions that can be bound to the variable. As well, a method can be associated with a *return type* that specifies the type of the expression returned by the method. Variables and methods are associated with types by way of *type declarations*, usually of the form:

```
var v : C          or          method m : C
```

where typically, `C` is a class name that denotes a type. Since a class name can refer to a class or a type, the convention used in this thesis is to italicize type names, and to use a typewriter type style for class names. Thus, `C` refers to a class, while *C* refers to the type denoted by that class.

Languages differ on how class names denote types. In some languages, *C* would be a *class reference*, meaning that `v` could only be bound to objects of class `C`, and `m` could only return expressions of class `C`. Such languages support *singleton types*, since the set of classes denoted by *C* is the singleton set, {`C`}. The variable, `v` whose type is a singleton type is a *homogeneous variable*, since it can only name objects of a single class [PaSc90a].

In other languages, `C` would denote a type that includes more than one class and `v` would be a *heterogeneous* variable since it could refer to objects of more than one class [PaSc90b]. Such languages support *non-singleton types*. Languages that support both singleton and non-singleton types must distinguish between them in the type expression. For example, the declaration,

```
var v : ↑C
```

could declare v to be heterogeneous while the declaration,

```
    var v : C
```

would make v homogeneous. The statement that an expression, variable or method *"has type, C"* should be interpreted in light of the language involved. For example, in a language that interprets a class name to denote a singleton type, an expression of type, *C* is also of class, C.

Subtyping is defined in terms of substitutability. A type is a *subtype* of another type, if expressions of the first type can always be used where expressions of the second type are expected [CaWe85]. Naturally, criteria defining subtypes are dependent on the interpretation of types as a whole.

1.3 Class Relations

1.3.1 Supplier Relations

Relations on classes of typed object-oriented languages, can be defined on the basis of their signatures. A class B is said to *supply* a class A if any signature in A includes a reference to the type *B*. The reference to a *B* can appear anywhere in A's interface: in the declaration of an instance variable, a parameter, a local variable or in the return type of a method. The set of classes that are connected by the transitive closure of supplier, is referred to here as a *system* of classes. The inverse relation of supplier, is the *client* relation. Classes that are suppliers of themselves are *recursive classes*, and methods or variables have *recursive signatures* or *"are recursive"* if their signatures contain type declarations that make their containing class recursive [CoHiCa89]. Thus, a recursive method has a recursive signature, but not necessarily recursive code as the standard interpretation of recursion would imply.

1.3.2 Subclassing Relations

Class definitions can serve both as the basis for code reuse, and as a specification for other class definitions. Inheritance is the means by which both ends are served. Class definitions can list other classes which they inherit. The definition of a class that *inherits* another class includes all of the method and variable definitions of the inherited class. The class which inherits is a *subclass*. The class from which it inherits is its *superclass*.

Inheritance can be used to associate class specifications with their implementations. An abstract class can specify with signatures, a minimal set of methods to be provided by all subclasses. Inheritance acts as a contract, ensuring that concrete subclasses provide the code for the abstract methods of their superclasses. Inheritance also allows subclasses to reuse the code of their superclasses without demanding duplication or recompilation. Thus, a superclass that is concrete acts as a code library for its subclasses. A superclass that is abstract acts as a specification for its subclasses. A partially abstract class assumes both roles. Realizing the many potential uses of inheritance, some languages such as Eiffel, Dee and recent versions of C++, allow a class to have more than one superclass. Such languages are said to support *multiple inheritance*.

A subclass can add new methods and variables, and *redefine* the code and signatures of those inherited from the superclass. Languages differ by how inherited signatures of superclasses are constrained to be redefined. These differences can be characterized as the *direction* and *scope* of signature redefinition. Subclassing with redefinition is one of the ways that object-oriented languages achieve polymorphism. An operation is *polymorphic* if it can be invoked with varying numbers and/or types of arguments [CaWe85]. A language is polymorphic if it provides support for polymorphic operations. Polymorphism is not unique to object-oriented languages. However, polymorphic languages that are not object-oriented tend either to be monomorphic with polymorphic exceptions, or to support only selected forms of polymorphism.

1.4 Type Safety

1.4.1 Degrees of Language Typing

Type-safety is defined for object-oriented languages as the property of a class which guarantees that no statement in its code results in an object receiving a message that it does not understand [PaSc90d]. As an example, a class that has an instance variable of type *Integer*, would not be type-safe if the code for one of its methods included a statement that attempted to add a *Matrix* to that *Integer*. A language for which type-safety can be completely guaranteed is said to be *strongly typed*. A language for which at least some type-safety checks occur at compile-time is said to be *statically typed*. It is preferable for a language to be strongly typed so that "message not understood" run-time errors can be avoided. It is preferable for a language to be statically typed to as large a degree as possible, since *dynamic type checking* results in target code that is riddled with the inefficiencies of run-time type checks. The challenge of much of the current work in object-oriented type theory is to include strong and static typing in a language without sacrificing the language's flexibility and expressive power.

1.4.2 Type Checks

For a language to be both statically and strongly typed, the semantic analysis performed by its compiler must include three type checks: binding, signature and parameter binding checks. This classification is a variation of that of Palsberg and Schwartzbach [PaSc90a] who distinguish between "early" and "equality" type checks. We generalize "equality" checks to be "binding checks" (not necessarily demanding equality of types), and then divide them according to whether the bindings are via assignment statements or the passing of arguments. Our "signature checks" are indentical to the "early checks" of [PaSc90a].

• *Binding Checks* ensure the validity of assignments statements, (which bind objects denoted by expressions, to local variables and instance variables), and of statements that return method results (where the object returned must be able to be bound to any variable declared of the method's return type). Thus, binding checks ensure the validity of all bindings of objects to variables, save for the bindings of invocation arguments to method parameters.

The strictness of a binding check depends on how a type declaration, such as

```
a : A
```

is interpreted. For example, if in the above example A is interpreted to denote the singleton set of classes, {A}, then the binding check must be strict and ensure that any object bound to a will be of class A. On the other hand, if A is interpreted to denote the non-singleton set, {all subclasses of class A}, then the class of any object bound to a can be any subclass of A.

• *Signature Checks* ensure that all invocations will be understood by their receiver objects. Given the declaration of a above, any message m invoked on a with n

arguments must be mapped to a definition of m with n parameters, in every class denoted by A.

• *Parameter Binding Checks* are like binding checks, but monitor the binding of invocation arguments to method parameters. Parameter binding checks are performed after signature checks provide the mapping between arguments appearing in an invocation, and parameters declared with a method's definition.

1.4.3 The Tension of Type Checks

There is a natural tension between the flexibiiity of binding checks on the one hand, and the flexibilities of signature and parameter binding checks on the other. Signature and parameter binding checks succeed only if a given invocation can be guaranteed understood by all potential receiver objects. Naturally, more flexible binding checks mean that larger numbers of classes can in general define objects that can be bound to a variable. Thus, more flexible binding checks should result in stricter signature and parameter binding checks.

In practice, the tension between binding checks and signature checks does not pose a problem. Binding checks are usually defined so as to allow a given variable to be bound to objects from a set of related classes. These classes usually have as a common interface, the interface of one of the classes in the set. From the success of the signature check over this one class, success can be inferred over the entire set. For example, when the type A is interpreted as the set of all subclasses of A, then signature checks need only examine the definitions in A. The monotonic nature of subclassing is such that subclasses must include all of the declarations and definitions of their superclasses, and cannot redefine any method such that the number of parameters associated with the method changes. Thus, the success of a signature check over a set of subclasses of A can be inferred from its success over A alone. So as to keep type-checking simple and signature checks flexible, interpretations of types usually allow signature checks to be confined to a single class.

Ideally, binding and parameter binding checks should be identical. Whether or not this is possible depends on the nature of the binding check, and therefore on the interpretation of types as a whole. For example, let the variable a be defined as before, let P be any class, and let the definition for A include a method m, with signature:

```
method m (receiver_object : A, other_parm : P).
```

When the type declaration for a assumes its type to be singleton, the binding check ensures that any object bound to a will be of class A. In this case, it is known that a parameter of type *P* is required when m is invoked on a. On the other hand, if the type declaration for a assumes its type to be the set of all subclasses of class A, then the required type of any argument sent when m is invoked on a, will depend on policies regulating subclassing redefinition to which the language concerned subscribes.

In theory, the more classes that define objects that can be bound to an expression, the less behavior that can be statically inferred of the expression. This should result in a tension between flexible binding checks on one hand, and flexible signature and parameter binding checks on the other. In practice, type interpretations are such that bindings to a given expression are restricted to objects of related classes. This allows signature checks to be confined to a single class. The tension of binding checks and parameter binding checks is not easily reconciled and is further complicated by the tension with flexible subclassing redefinition when subclassing becomes the basis on which binding checks are decided.

1.5 Objectives

The objectives of this thesis are three-fold:

1. Analysis:

   To categorize all forms of polymorphism, identify the polymorphisms that can be achieved by inheritance, and identify the tensions associated with each of these forms of polymorphism,

2. Survey:

   To compare and evaluate existing languages and theories on how their proposals:

   a. provide each of the forms of polymorphism described above and
   b. guarantee type-safety while resolving the tensions of polymorphisms,

3. Design:

To design an object-oriented type and subclassing system that offers the most varied and most flexible forms of polymorphism, while resolving all tensions in the most practical way possible.

These objectives are addressed in the chapters that follow. Chapter 2 (Analysis: Discerning the Polymorphisms and Tensions of Inheritance) classifies the polymorphisms that can be achieved by inheritance, and identifies the associated tensions. Chapters 3 and 4, (Survey: An Examination of the Polymorphisms of the Bruce Language Proposal) and (Survey: An Examination of the Polymorphisms of the PS Language Proposal) examine tw recent language proposals in terms of what forms of polymorphism are supported, an how tensions associated with those polymorphisms are resolved. Included in this chapter is a extensive study of "PS subclassing": judged here to be the most promising because of its inherent flexibility. Shortcomings of this proposal are described in this chapter, and either remedied in Chapter 5 (An Interpretation and Extension of PS Subclassing) or explored in detail in Chapter 6 (Future Work: Marrying Dynamic Binding to PS Classes and PS Subclassing). Thus, Chapter 2 contains the analysis, Chapters 3 and 4 the survey, and Chapters 5 and 6, the original contributions to the field.

1.6 An Example OOP Language: Mini-Dee

A simple, generic object-oriented language is presented here. The language closely resembles Dee and is herein referred to as "Mini-Dee." Its purpose is to facilitate comparisons of various theories, and it is therefore used in this thesis as the language of all examples illustrating those theories.

The core language is informally described below, and is formally defined by way of a grammar in Appendix I-A. Various extended versions of Mini-Dee are used to illustrate proposals for typing and subclassing presented at later points in this thesis. Before each theory is presented, the appropriate extensions to the core language are described. The complete grammars of these Mini-Dee extensions are also found in Appendix I.

The core language is strongly and statically typed. A program in the core language consists of a system of classes. The Mini-Dee grammar demands that an entire system of class interfaces appear before the code of these classes. Such a separation is useful for theoretical purposes, but need not be enforced in an implementation since the separation can be achieved by a compiler. Classes included in a system can be base classes provided by the Mini-Dee library or classes defined by the user. The base classes are `Int`, `Bool`, `Float`, `String` and `Object` (the empty class that is a superclass of all classes). User-defined classes can be explicitly defined and can be said to inherit other classes.

The code of a method consists of statements contained within a "`begin` ...`end`" block as in Pascal. A statement can bind an expression to an instance variable or local variable, or can send a message to an object denoted by an expression. An expression can be anything that denotes an object: an instance variable, a method parameter (including `self`), or a result of an invocation made of another expression. Invocations are made using the record selection notation described earlier. Invocations without explicit receiver objects are assumed to be invoked on `self`, and are legal provided that the methods or variables invoked, are defined in the class containing the statement. Extensions to Mini-Dee differ both semantically and syntactically. Semantic differences between extensions include their interpretations of types, their restrictions governing redefinition in subclasses, and their binding, signature and parameter binding checks.* These attributes of Mini-Dee extensions are described informally as they are needed. Syntactic differences include whether the extended language supports single or multiple inheritance. The Mini-Dee grammar provides alternative rules for the nonterminal, IList (Inherits List), the first of which is applicable to single inheritance extensions and the second to multiple inheritance extensions. Descriptions of all Mini-Dee extensions in this thesis include indications of which form of inheritance is supported, and thus which of the two grammar rules applies.

For an example class in the Mini-Dee core language, the reader is referred to **Example 2.7**.

* Definitions of these type-checks vary, but the times at which they interrupt the parse do not. The checks are performed following the parse of rules indicated in the grammar of **Appendix II**.

Chapter 2.

Analysis: Discerning the Polymorphisms and Tensions of Inheritance

2.1 Classifications of Polymorphism

Polymorphic operations are those defined over varying numbers and types of arguments. The definition is broad as it captures those operations which share any of name, code, or signature. Discerning how inheritance achieves polymorphism is further complicated by the many options that implementations of inheritance tend to allow. Cardelli and Wegner propose a useful taxonomy of polymorphism [CaWe85], but suggest that the polymorphism of inheritance can be captured within a single category. It becomes clear when distinguishing these categories on the basis of mappings from invocations to code, that inheritance can realize three forms of polymorphism. Descriptions of the original and modified taxonomies follow, with tensions associated with each category identified.

2.1.1 The Cardelli-Wegner Classification of Polymorphism

Cardelli and Wegner divide polymorphism into four categories: overloading, coercion, parameterization and inclusion. The former two categories fall under the heading of "ad-hoc" polymorphism. *Overloading* is the polymorphism achieved when more than one operation shares the same name. For example, "3 + 2" and "3.2 + 2.7" are both legal expressions in otherwise monomorphic languages such as Pascal. The meaning of "+" is determined by context: it is an *Integer* operation in the first case and a *Float* operation in the latter case. The code for the two operations is distinct; only the name of the operation is shared. The expression "2 + 3.4" demonstrates *coercion.* The operation of adding *Integers* to *Floats* is meaningless. Languages supporting coercion would coerce the *Integer* value, 2 into the *Float* value, 2.0, and then pass it with 3.4 to the operation that adds two *Floats.* That coercion and overloading fall under the pejorative classification, "ad-hoc" is likely a result of their origins as polymorphic exceptions to monomorphic languages. As Danforth and Tomlinson point out [DaTo88], the disdain for overloading is unwarranted. Overloading is given new-found respectability under the guise of subclassing with code redefinition in object-oriented languages.

Parametric and inclusion polymorphisms fall under the heading of "universal" polymorphism. *Parametric polymorphism* is the polymorphism that requires a type parameter. An operation such as `swap`, which swaps the values bound to any two variables of the same type, would ideally be defined as a parametrically polymorphic operation. Monomorphic languages such as Pascal would require a separate `swap` operation over all types for which it was defined. A Pascal definition for swap, as defined over *Integers*, is shown in **Example 2.1**:

```
procedure swap (var a, b: Integer)
  var temp : Integer
  begin
      temp := a;
      a := b;
      b := temp;
  end
```

Example 2.1

Every Pascal swap definition would be identical to the one above, save for the types of the parameters, a and b, and the local variable temp. Languages such as Ada, that support parametric polymorphism, would allow a generic swap to be defined in terms of a type parameter. An Ada definition for swap is shown in **Example 2.2**.

```
generic
      type T is private;

procedure swap (a, b: in out T) is
      temp : T
  begin
      temp := a;
      a := b;
      b := temp;
  end swap
```

Example 2.2

Parametric polymorphism especially lends itself to operations that act on collection data structures such as lists, stacks and trees, since these operations tend to be independent of the types of items contained within the collections. For example, assuming some sort of reference semantics which makes the size of an item (as it appears in the collection) constant, the same procedure is used to pop a stack, no matter what the stack contains. That many related operations are all parameterized by the same type parameters allows an extra level of abstraction, and the data structure rather than the operation can become the parametric entity. Object-oriented languages supporting parametric polymorphism such as Eiffel and Dee, do so by parameterizing data structures with *parametric classes*; higher-order class generators with type parameters. Parametric class definitions resemble regular class definitions except that types are denoted both with class names and type variables. Type variables are defined as parameters to the class definition and in some languages are constrained. Acting as higher-order types, constraints limit the bindings to (or *instantiations* of) type variables.

An example parametric class is shown in **Example 2.3**. The class Pair (as in Ordered Pair) is borrowed from the Dee class library [Gr91] and translated into an extension of Mini-Dee that supports parametric classes.

```
class Pair [T : Order]

      var a : T
      var b : T

      method makepair (x : T, y : T)
            begin
                  a := x
                  b := y
            end

      method min : T
            begin
                  if a < b then
                        return a
                  else
                        return b
                  fi
            end

end Pair
```

Example 2.3

`Pair` has a single type parameter, `T`, constrained by the class, `Order`, the class of objects that belong to partial orders. `T` is used as the type of both instance variable components, and of the method, `min`.

Definitions of parametrically polymorphic operatio.. can be independent of the types over which they are defined and therefore can be defined over disjoint types. Conversely, inclusion polymorphism is the polymorphism of operations defined over types and their subtypes. Typically associated with object-oriented languages and inheritance, inclusion polymorphism is supported by any language that recognizes subtypes and allows their values to be bound to variables declared of their supertypes. It is the polymorphism that results when an operation requiring an argument of some type T can be interpreted as also allowing arguments of any subtype of T.

The Cardelli-Wegner taxonomy is useful, both as a conceptual classification of polymorphisms and as a timetable charting the development of polymorphism over several generations of languages. The "ad hoc" polymorphisms of overloading and coercion are considered the most primitive and in fact have the earliest origins. That coercion is even a form of polymorphism is debatable. Polymorphism has a connotation of flexibility on the part of the operation to accept arguments of varying types. Coercion can be viewed as a "backward polymorphism" since the onus is on the argument to conform, rather than the operation. The "universal" parametric and inclusion polymorphisms are considered more sophisticated, and products of languages of more recent generations. Parametric polymorphism is supported by languages such as Ada and ML [MiToHa90] and Object-Oriented languages such as Eiffel and Dee. Since inclusion is considered to capture the polymorphism of inheritance, it is thought to be supported by all object-oriented languages.

2.1.2 Mapping Invocations to Code

That the polymorphism of inheritance is presumed to be captured by a single category is where the Cardelli-Wegner taxonomy falls short. Inclusion only captures one type of polymorphism achieved by inheritance; the polymorphism of subtyping. The polymorphisms of inheritance also include overloading and a form of parametric polymorphism. The Cardelli-Wegner classification distinguishes between the various forms of polymorphism, but not in such a way as to identify all of the polymorphisms of inheritance.

The classification can be viewed in another way that helps to distinguish the polymorphisms of inheritance. Type-safe code has the property that every invocation can be associated with code whose execution it triggers. The mapping is from the signature of an invocation to code. Polymorphisms can be distinguished by the nature of the mapping. For example, a **one-to-one** mapping of invocation signatures to code is achieved both by overloading, and by subclassing with code redefinition. Overloading involves two operations sharing a name, but having distinct signatures and code. An inherited method whose code is redefined will share the same name as the method inherited, but will have a distinct signature (if only because of the automatic redefinition of `self`) and distinct code. Since the polymorphism of overloading and subclassing with code redefinition only involve the sharing of operation names, they are referred to here as *nominal polymorphism.*

The polymorphisms of **many-to-one** mappings are those where invocations of varying signatures can result in the execution of a single block of code. This category includes parametric polymorphism. A parametrically polymorphic operation such as `swap` can be called with two *Integer* arguments, or two *Boolean* arguments and so on. Each of the resulting invocation signatures results in the execution of the same block of code. The category also includes subclassing when inherited code is not redefined. Subclassing without code redefinition results in invocations of varying signatures triggering the execution of the same block of code. **Example 2.4** is written in the core language of Mini-Dee supporting either single or multiple inheritance. (Policies concerning type interpretations, type-checking and signature redefinition in subclasses are irrelevant to this example). As will be the case in many examples shown in this thesis, parts of some class and method definitions are replaced by ellipses, demonstrating that the actual contents of these definitions are not pertinent to the example.

```
class P
      method m
            begin
                  ...
            end
end P

class C inherits P
       ...  {m is not redefined}
end C

class example_class
      var c : C
       ...
      method example_method
            begin
                  c.m
            end
end example_class
```

Example 2.4

The invocation, `c.m` in `example_class` calls the method m defined in P. The invocation's signature,

```
m (receiver_object : C)
```

is one of many that could be used to invoke the code of `m`. The receiver object could also be of type *P*, or of any other type denoted by a subclass of `P` that does not redefine `m`.

The mapping of invocation to code of both parametric polymorphism and subclassing can be achieved at compile-time or link-time. This is because in both cases, the same code is executed no matter the types of the arguments in the invocations. For this reason, this form of polymorphism is referred to here as *static binding*.

Static binding is the polymorphism resulting from a many-to-one mapping from invocation signature to executed code. *Dynamic binding* on the other hand, results from a **one-to-many** mapping. Dynamic binding is the polymorphism that results when a single invocation can result in the execution of many different blocks of code. The classes of **Example 2.5** are written in an extended version of Mini-Dee which supports single or multiple inheritance, and defines binding checks assuming that an object can be bound to a variable if its type is denoted by a subclass of the class denoting the type of the variable.

```
class P
        method m
                begin
                        .
                end
end P

class C inherits P
        method m
                {Redefined version of m}
                begin
                        ..
                end
end C
```

```
class example_class
      var p : P
      var c : C
       ...
      method example_method
            begin
                  p := c
                  p.m
            end
end example_class
```

Example 2.5

Though the receiver object, p of invocation, p.m is declared to be of type *P*, the code whose execution the invocation triggers is defined in C. In general, depending on what is assigned to p, executed code could be that defined in any subclass of P including P itself. Thus, a single invocation can be mapped to many blocks of code. Note that the one-to-many polymorphism resembles the inclusion polymorphism of the Cardelli-Wegner taxonomy. Object-oriented inclusion allows the code of m to differ according to the class of the receiver object argument at run-time. Because the class of the receiver object cannot in general be inferred at compile-time, the mapping of invocation to code must be made dynamically.

2.2 The Tensions of Polymorphisms

When found in object-oriented languages, the polymorphisms above are subject to tensions that affect how type-safety is established or preserved. Nominal polymorphism can result in an invocation in the superclass being mapped to code in the subclass, and must be regulated such that subclass code is type-safe given this mapping. Static binding can result in an invocation made of objects of the subclass being mapped to code defined in the superclass and must be regulated such that superclass code is type-safe given this mapping. Dynamic binding can result in an invocation being mapped to any of a number of blocks of code, and must be regulated such that every block of code is type-safe with respect to each mapping.

To discuss these tensions, it is necessary to preview the sections on type-safety found in section 2.5. The tensions discussed are those evident, even given the most liberal possible binding and parameter binding checks that are found in typed object-oriented languages. Liberal type checks allow an object to be bound to an expression provided that the object's type is denoted by a subclass of the class denoting the expression's type. Accordingly, such binding and parameter binding checks are referred to here as *subclass-based*. To illustrate, statements 1 and 2 of the class `example` of **Example 2.6** would only be type-safe if defined in an extension of Mini-Dee that used subclass-based binding and parameter binding checks respectively. It should be assumed for this example, that `A'` is a subclass of `A`.

```
class example

        var a : A

        method m1 (x : A)
                begin .. end

        method m2
                var a2 : A'
                begin
                    a := a2                                        {1}
                    m1 (a2) {self is implicit receiver object}     {2}
                end

end example
```

Example 2.6

It can be inferred that tensions evident given subclass-based binding checks, are also problematic given more conservative binding checks. Thus for the examples of sections 2.3 and 2.4, it should be assumed that subclass-based binding and parameter binding checks establish the type-safety of code defined in any extension of Mini-Dee.

2.3 Nominal Polymorphism: Subclassing with Code Redefinition

Nominal polymorphism can be achieved by subclassing in two ways:

1. *Code Redefinition*: The code of an inherited method is redefined and the signature (save for the implicit redefinition of `self`) is inherited as is.

2. *Code and Signature Redefinition*: The code of an inherited method is redefined and explicit parts of its signature (parameters or return type) are also redefined.

The tension of nominal polymorphism and subclassing concerns what limits should be placed on redefinition to ensure that the type-safety of inherited code is maintained. Code redefinition is never a concern since redefined code must always be type-checked, and this redefinition does not affect the type-safety of other inherited methods. Signature redefinition can affect the type-safety of other inherited methods, since their code may include invocations of the original method. The class `Turtle` of **Example 2.7** is written in the core language of Mini-Dee, and illustrates how both forms of nominal polymorphism achieved by subclassing might be desired, and why signature redefinition constraints are necessary. For this example, some mathematical operations (+, -, √, `square`, mod) are assumed defined in class `Int`, and others (+, ≤, `round`, `sin`, `cos`) in class `Float`. The invocations of these operations are written in standard mathematical notation to improve readability.

```
class Turtle

      var x         : Int      {horizontal coordinate of Turtle}
      var y         : Int      {vertical coordinate of Turtle}
      var direction : Float    {direction Turtle facing (0 .. 359)}
```

```
method move (m : Int)                                            {1}
    {move m in current direction}

        begin
              x := x + round ((m) (cos (direction)))
              y := y + round ((m) (sin (direction)))
        end

method distance (a : Int  b : Int) : Float                       {2}
        {returns distance between Turtle and (a,b)}

        begin
              return $\sqrt{(x-a)^2 \ - \ (y-b)^2}$
        end

method turn (amt : Float)                                        {3}
        {turn Turtle clockwise, amt degrees}

        begin
              direction := direction + amt
        end

method set_pos (a : Int  b : Int)                                {4}
        {absclute move to (a,b) }

        begin
              x := a
              y := b
        end
    method within_one (a : Int  b : Int)                     {5}
        {returns true if turtle within one unit distance
         of (a,b)}

        begin
            return (distance (a,b) ≤ 1)
        end

end Turtle
```

Example 2.7

Nominal polymorphism might be desired when creating a subclass of `Turtle` such as `WrappingTurtle`, the class of *Turtles* whose movements are confined to appear on a screen. Assume for this example, that the screen in question is square, that (0,0) is the coordinate representing the bottom left of the screen, and $(k\text{-}1,\ k\text{-}1)$, the coordinate representing the top right. For this example it should be also be assumed that *ScreenRange* $(0 \ldots k)$ is a subrange and hence a subtype of *Int*.

The class `WrappingTurtle` might redefine the code (though not the signature) of the method, `move`, and redefine both code and signature of the method, `distance`, as described below:

(1) method `move` - Code Redefinition

The move of `WrappingTurtle` should ensure that the `WrappingTurtle` object appears at one edge of the screen when it disappears off the opposite edge. Thus, the redefined version of move could be:

```
    method move (m : Int)                    {1}
      {move m units in current direction}

        begin
            x := (x + round ((m) (cos (direction)))) MOD k-1
            y := (y + round ((m) (sin (direction)))) MOD k-1
        end
```

Note that it is not necessary to redefine the method's explicit signature since the move of a *WrappingTurtle* still requires an *Int* value to determine the degree of the move. Thus, this is an example of code redefinition.

(2) method `distance` - Code and Signature Redefinition

The method `distance`, in `WrappingTurtle` should return the smallest distance between the *Turtle* and the pixel, (*a*,*b*). Unlike the regular *Turtle*, the *WrappingTurtle* can reach (*a*,*b*) by traveling in two directions. The two trips, *trip_for* and *trip_bac*, are illustrated in **Figure 2.1**.

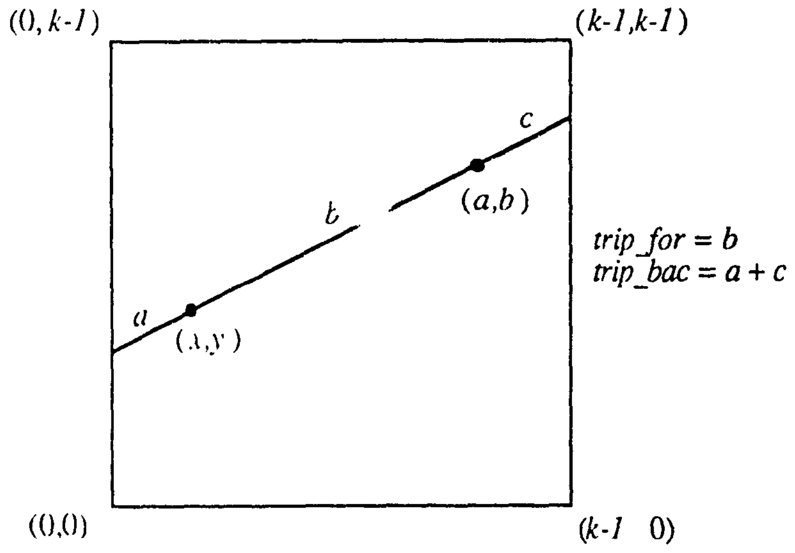


Figure 2.1

The new definition for `distance` should redefine the signature of the inherited method to ensure that the input coordinates constitute a point on the screen. The new definition should also redefine inherited code so that both possible journeys to (a,b) are measured, and the shortest distance returned. Thus the new method definition and signature might be:

```
method distance (a : ScreenRange b : ScreenRange) : Float     (2)
        {returns distance between Turtle and (a,b)}

        var trip_for, trip_bac : Float
        begin
                trip_for := $\sqrt{(x-a)^2 - (y-b)^2}$

                trip_bac := ..
                if trip_for ≤ trip_bac then
                        return trip_for
                else
                        return trip_bac
                fi
        end
```

The calculations determining the value of `trip_bac` are rather complicated and irrelevant to the example. Therefore, they are omitted from the code here. The important point is that the redefinition of the inherited method involves both code and signature.

It should be apparent from the examples above that there is a need to regulate signature redefinition to ensure that the type-safety of other inherited methods is retained, even when binding and parameter binding checks are liberally defined as subclass-based. If the redefinitions of `move` and `distance` above were the only redefined methods of class, `WrappingTurtle`, then the invocation statement of the method, `within_one` would no longer be safe since the parameter binding check (now binding a *Float* to a *ScreenRange*) would fail.

2.3.1 Directions of Signature Redefinition

Policies of signature redefinition can be characterized by direction and scope. The direction of signature redefinition determines how an individual type reference found in a signature can be redefined. Two direction guidelines are shown here.

Covariance

Covariant signature redefinition allows any type reference to be redefined to refer to a subtype (*specialized*). As applied to subclasses, it allows **any** variable declaration of a superclass, `v : A`, to be specialized in a subclass, whether the type declaration is for an instance variable, parameter, local variable or method return type. (Note that covariant signature redefinition would allow the method `distance` of class `WrappingTurtle`, above, to be redefined to accept parameters of type *ScreenRange*).

Covariance is the most flexible and natural form of signature redefinition for subclassing. A superclass usually defines a generalization in that it houses code that is applicable to specialized subclasses. It is most natural for generalized code to be defined in terms of general types, and for specialized versions to interpret the general code in terms of specific types.

Contravariance

As applied to subclassing, *contravariant* signature redefinition limits the type references that type:

- method return types to be specialized,
- method parameter types to be *generalized* (redefined to refer to supertypes),
- instance variable types to stay constant.

As a guideline for signature redefinition, contravariance arises from the work of Cardelli and Wegner [CaWe85], who attempt to justify the equation of subclasses with subtypes. In their record-based model of objects, (*label,value*) pairs model instance variables, functional record components model methods and record types are based on class interfaces. An example point record and record type are shown below:

Record:

point = {x_coord : 4.2, y_coord : -3.6}

Type of Above Record:

PointType = {x_coord : Int; y_coord : Int}

1. Subtyping Cardelli-Wegner Record Types

Record subtyping (≤) holds if all components of the supertype are also found in the subtype. The corresponding components of supertypes and subtypes need not have the same type. The type of a component in a record subtype can be a subtype of the type of its corresponding component in the supertype. For the example record types below, it should be assumed that *ColorType* is a type, and that *PosInt* is a subtype of *Int*.

ColorPointType = {x_coord : Int; y_coord : Int; color : ColorType}

PositivePointType = {x_coord : PosInt; y_coord : PosInt}

ColorPositivePointType =
{x_coord : PosInt; y_coord : PosInt; color : ColorType}

The subtyping relations below hold, given the above definitions and assumptions:

- *ColorPointType ≤ PointType*
- *PositivePointType ≤ PointType*
- *ColorPositivePointType ≤ PointType*
- *ColorPositivePointType ≤ ColorPointType*

Intuitively, *PointType* should be interpreted to include all records with at least the *x_coord* and *y_coord* Integer fields. Thus the set of records whose type is *PointType* includes all of those which are *ColorPoints*, *PositivePoints* and *ColorPositivePoints*.

2. Subtyping Records with Functions

Function subtyping is less intuitive than record subtyping. Given types A, B, A' and B', where $A' \leq A$ and $B' \leq B$, the following subtype ordering holds:

$$A \rightarrow B' \underbrace{\quad\leq\quad}_{(1)} A \rightarrow B \underbrace{\quad\leq\quad}_{(2)} A' \rightarrow B$$

$A \rightarrow B' \leq A \rightarrow B$ (1) since a function f_1, of type $A \rightarrow B'$ will behave in any context where a function of type $A \rightarrow B$ is expected. f_1 can accept any value of type A as an argument and returns a result of type B' which by the subtype relation is also a B. The ordering of (2) is less obvious because of the counterintuitive ordering of the function domain. But a function, f_2 of type $A \rightarrow B$ is substitutable in any context where a function of type $A' \rightarrow B$ is expected. Since f_2 is defined for values of type A it is trivially the case that it is defined for values of type A'. Note that by a combination of the reasoning of (1) and (2) it is also the case that:

$$A \rightarrow B' \quad \leq \quad A' \rightarrow B\,.$$

Cardelli and Wegner point out that since every class' signature includes the implicit parameter `self`, every class is implicitly recursive and a contradiction arises if it is assumed that subclasses denote subtypes. Given an arbitrary class, `C`, any signature found in `C`'s class definition will be of the form,

$$C \times C_1 \times \supset \times C_n \rightarrow C_{n+1}$$

where `C`$_i$ is a class name denoting a type. If we assume that subclasses denote subtypes, then any subclass of `C`, `C'` will inherit the above signature, and automatically redefine it to adjust the type of `self`, to be of type, C'. We arrive at a contradiction, since the inherited function signature is not a subtype of the original signature. Inherited signatures are automatically and covariantly redefined through the specialization of `self`. Because function subtypes must be contravariant versions of their supertypes, subclasses cannot denote subtypes.

3. The F-Bounded Model

Because every signature in a class has an implicit recursive parameter `self`, subclasses cannot denote subtypes. Cook et. al. modify the Cardelli-Wegner record model by removing the parameter representing `self`, from all signatures defined in a class, and making it a parameter to the class definition [CoHiCa89]. In their F-Bounded model, classes are record generating functions, and `self` is both a parameter and result. An object is created by application of a fixpoint operator to the record generating function, allowing the object's method definitions to refer to itself. Thus the F-Bounded model of classes and objects captures the self-referential aspect of objects while interpreting subclasses to denote subtypes, despite the automatic covariant redefinition of `self` in subclasses. Only the explicit signatures of subclasses must be contravariant versions of corresponding signatures in superclasses. Explicit parameters to methods can only be generalized in subclasses. Method return types can be specialized. Instance variables, as both parameters to their own implicit update methods, and expressions returned by implicit reference methods, cannot be redefined.

2.3.2 Scope of Signature Redefinition

The scope of signature redefinition determines whether the redefinition of a type reference must be applied *consistently* throughout a class or system of class definitions, or whether it can be applied *selectively*. As applied to subclassing, when a class contains two declarations, v_1 : A and v_2 : A, redefinition scope determines whether it is possible to redefine the type of v_1 in a subclass without redefining the type of v_2. Selective scope allows this, while consistent scope does not. Naturally, selective redefinition is more flexible than consistent redefinition. Most object-oriented languages support selective redefinition with nominal polymorphism.

2.3.3 The Tension of Nominal Polymorphism

The only tension of nominal polymorphism involves signature redefinition that is covariant in its direction and selective in scope. Languages that support both forms of signature redefinition can allow subclasses that do not preserve the type-safety of inherited code. The classes of **Example 2.8** are written in an extension of Mini-Dee that supports either single or multiple inheritance, uses subclass-based binding and parameter binding checks, and allows both covariant and selective redefinition in subclasses. It should be assumed for this example, that `A'` is a subclass of `A`.

```
    class P

          var a : A

          method m1 (x : A)
                begin ... end

          method m2
                begin
                      m1 (a)
                end

    end P


    class C inherits P
          {m1 redefined, m2 inherited as is}

          method m1 (x : A')
                begin .. end
    end C
```

Example 2.8

In the above example, the parameter binding check which succeeds in `P`, fails in `C`, since the invocation attempts to bind an object of type A, to a variable expecting an object of type A'.

It should be noted that though selective redefinition fails to cooperate with covariance, it can coexist with contravariance. In the above example, covariant redefinition of m_1 would demand that the type of the parameter x be a supertype of A. The code of m_2 could thus be safely inherited since an A can be bound to any variable declared of a supertype of A.

2.4 Static Binding

Static binding is achieved by subclassing in two ways:

1. *Prefixing*: A method or variable is inherited "as is". Because the implicit parameter, `self` is automatically redefined in subclasses, a new invocation signature is defined for existing code, and static binding is achieved.

2. *Genericity*: The explicit signature of an inherited method or variable is altered, while the associated code is inherited "as is".

Given the example `Turtle` class of **Example 2.7**, it might be desirable in the subclass `WrappingTurtle` to use prefixing and genericity to redefine the methods, `turn` and `set_pos` respectively.

(3) method `turn` - Prefixing

WrappingTurtles can turn just as *Turtles* do. That their movements are confined to the screen does not affect the directions in which they turn. Thus, `WrappingTurtle` should inherit the signature and code of turn, "as is".

(4) method `set_pos` - Genericity

The signature of `set_pos` should be restricted so that only coordinates that designate a pixel on the screen are accepted. The code of `set_pos` is unaffected. Thus the redefined version of `set_pos`, below, has redefined signature and code inherited "as is".

```
method set_pos (a : ScreenRange  b : ScreenRange) : Bool
```

The redefinition of this method is therefore, generic.

Prefixing and generic redefinition liken subclassing to parametric polymorphism, since the polymorphism in each case results from the creation of new signatures to substitute for the signature originally associated with some block of code. Subclassing differs from parametric polymorphism by the way in which new signatures are created.

Parametric definitions include type variables which are constrained by a type expression. Creation of new signatures results from instantiating type variables. Regular class definitions contain declarations with actual types but no type variables. Subclasses create new signatures by substituting for the original types.

As was the case with nominal polymorphism, type-safety issues with regard to static binding concern how signature redefinition must be regulated. The only concern of nominal polymorphism is how redefined signatures affect the invocations found in the code of other inherited methods. Static binding must also ensure the type-safety of the code whose signature is redefined.

2.4.1 Direction and Scope of Signature Redefinition of Static Binding

The direction and scope of signature redefinition guidelines can be characterized for static binding as they are for nominal polymorphism. Covariant and contravariant, and consistent and selective redefinition guidelines are applied to subclasses as described in Section 2.3, but can also be used to describe how parametric classes and operations are constrained to be instantiated. As applied to a parametric operations defined using some type parameter T constrained by type *A*, (t : A), direction and scope guidelines indicate the types to which T can be bound.

Direction

As applied to a parametric operation with the type parameter defined above, covariant instantiation would allow T to be bound to any subtype of *A*. Like superclasses, parametric classes and operations are generalizations and ideally house code defined over general types. Covariance is the most natural guide for instantiation of parametric classes and operations since it allows resulting classes and operations to apply general code to specialized types.

Parametric classes and operations rarely require contravariant redefinition. If they did, then the binding of types to T would depend on how T was used:

- if T were the type of an instance variable, then T could be bound only to A;
- if T were the type of a method parameter, then T could be bound only to a class denoting a supertype of *A*;
- if T were only used as a return type of methods, then T could be bound only to a class denoting a subtype of *A*.

Scope

As was the case with direction characterizations, the designations of selective and consistent scope can be applied to both subclassing redefinition and parameterization. With regard to the latter, given a parametric operation or class with type parameter, `T : A`, redefinition scope determines if it is possible to declare two variables:

v_1 : T and
v_2 : A.

If the above is possible, then the scope of redefinition is selective. If not, then scope is consistent. Just as parametric operations and classes are almost invariably covariant in direction, so too are they selective in scope. As will be shown below, this combination is problematic.

2.4.2 The Tensions of Static Binding

Covariant Direction and Selective Scope

Two tensions can be identified with static binding, both concerned with how signature redefinition affects the type-safety of inherited code. The first tension is identical to that of nominal polymorphism, and concerns how the type-safety of inherited code that invokes a redefined method or variable can be lost in subclasses. This tension concerns the combination of covariant direction and selective scope. The classes of **Example 2.8** illustrated this tension as it applied to the nominal polymorphism of subclassing, and these classes can easily be modified to show how this tension affects static binding and subclassing. **Example 2.9** is identical to **Example 2.8** except that the redefined method is generically redefined, rather than redefined by code and signature.

```
class P

        var a : A

        method m  (x : A)
                begin  ... end
```

```
      method m2
            begin
                  m1 (a)
            end

end P

class C inherits P
      {m1 generically redefined, m2 inherited as is}

      method m1 (x : A')

end C
```

Example 2.9

Example 2.10 illustrates this same tension as it applies to parametric classes and operations with covariant direction and selective scope. The parametric class, P is written in an extension of Mini-Dee that allows definitions of parametric classes. P is type-safe, but no instantiated version of P is.

```
class P [t : A]

      var a : A

      method m1 (x : t)
            begin .. end
```

```
          method m2
                begin
                      m1 (a)
                end

    end P
```

Example 2.10

The invocation statement of method m_2 would fail the parameter binding check, given any instantiation of `P`, `P [A']`.

Contravariant Direction and Genericity

The previous tension concerned how type-safety of invocations could be violated if the signatures of the code invoked were redefined. The tension of contravariant direction and any form of genericity concerns how the type-safety of code mapped to the redefined signature is violated. **Example 2.11** illustrates this tension and is written in an extension of Mini-Dee that supports single or multiple inheritance, and allows both contravariant and generic redefinition in subclasses.

```
class A
      method m
        begin  ... end
end A

class P
      method m2 (a : A)
            begin
                  a.m
            end
end P
```

```
class C inherits P
        {generically redefines m₂}

        method m₁ (a : Object)

end C
```

Example 2.11

The signature check of the invocation in method m_2 in class `P`, would be successful in `P` but not in `C`, since no methods can be invoked on objects of class `Object`.

The tension is applicable to parametric classes also, though it rarely affects them directly since parametric classes rarely demand contravariant instantiation. Given an extension of Mini-Dee allowing parametric classes with contravariant instantiation, class `P` below is type-safe, though instantiations of `P` such as `P [Object]` are not.

```
{class A as defined above}

class P [t : A]

        method m (x : t)
                begin
                        x.m
                end

end P
```

Example 2.12

Though parametric classes demanding contravariant instantiation are rarely found in object-oriented languages, this same tension can be revealed in languages with parametric classes allowing covariant instantiation, and subclasses that can be redefined using contravariant direction. This is illustrated in **Example 2.13** which is written in an extension of Mini-Dee supporting these features.

```
class A
      method m (i : Integer)
            begin  ... end
end A

class A' inherits A
      {redefines at least the signature of m}

      method m (i : Object)

end A'

class P [t : A]

      method m (x : t)
            begin
                  x.m (45)
            end
end P
```

Example 2.13

Note that though P should be considered type-safe, the parameter binding check of the invocation in method m will fail, given such instantiations of P as P [A']. Parametric classes can demand covariant instantiation, but can be undermined by contravariant subclassing.

2.5 Dynamic Binding

Dynamic Binding differs from static binding and nominal polymorphism not only by the nature of the mapping from invocation to code that defines it, but also by how it is related to subclassing and how its tension is related to type-safety. As a code and signature reuse mechanism, subclassing results in static binding and nominal polymorphism. Subclassing relations between classes only affect dynamic binding when these relations are used in some way to define types. The tensions of nominal polymorphism and static binding concern how type-safety is maintained when code and signatures are reused. The tension of dynamic binding concerns establishing type-safety in the first place, and is the tension between type checks discussed in Chapter 1.

2.5.1 Dynamic Binding, Non-Singleton Types and Subtyping

For a language to support dynamic binding, it must interpret types to be literally or effectively non-singleton. Effective non-singleton interpretations can result from support for subtyping, since this also allows objects of more than one class to be bound to any given variable. Languages lacking support for both non-singleton types and subtyping would demand that all objects bound to the variable a, with declaration,

```
a : A
```

be of class, A. Since every invocation on a would be mapped to a single definition or declaration found in the class definition for A (or one of its superclasses), the mapping of invocation to code would be either one-to-one or many-to-one, and the resulting polymorphism would either be nominal or static.

Interpretations of non-singleton types differ by what set of classes constitute a type. These interpretations must balance the tradeoff between flexible binding checks and flexible parameter binding checks. The policy on the former decides that of the latter. Languages that enforce disciplined binding checks can statically infer more of expressions, and thus can have relatively flexible parameter binding checks. At the other extreme, languages for which binding checks allow any object to be bound to any variable have to forbid argument passing or sacrifice strong, static typing. Interpretations that rely on subclassing relations to determine binding checks must also balance the flexibility of redefinition in subclasses.

2.5.2 Interpretations of Non-Singleton Types

Interpretations of non-singleton types can be divided into two categories according to the nature of the binding checks that result from the interpretation. As their names imply, subclass-based and subclass-independent type interpretations differ by how closely they are tied to the subclassing hierarchy.

Subclass-based binding checks were described briefly at the start of this chapter, as those binding checks that allow an object to be bound to a variable provided that the type of the object is denoted by a subclass of the class denoting the type of the variable. *Subclass-based type interpretations* use subclass-based binding checks, though not necessarily, subclass-based parameter binding checks, as will be shown shortly. Some subclass-based interpretations literally interpret a type denoted by a class name to be the non-singleton set of all subclasses of the named class. In other interpretations, the named class denotes a singleton type that is effectively non-singleton, since objects belonging to subtypes can be bound to expressions of that type. These latter interpretations are subclass-based when subclasses denote subtypes. The PS interpretation of types described in Chapter 3 is subclass-based and interprets types to be non-singleton. Type systems of such object-oriented languages as Eiffel, Dee and Trellis/Owl [ScCoBuKiWi86], interpret types to be singleton, but because these languages support subtyping, types effectively include more than one class and are therefore, effectively non-singleton.

Subclass-independent type interpretations were first advocated by Danforth and Tomlinson [DaTo88], and realized in a recent language proposal by Bruce [Br92]. In these interpretations, the determination of what objects can be bound to a variable does not depend on whether the classes denoting object and expression types are subclass-related. Subclass-independent type interpretations result in separate hierarchies for types and classes, but avoid the tension of dynamic binding and covariant subclassing redefinition, described below.

2.5.3 The Tension of Dynamic Binding

An explanation of the origins of contravariance in section 2.3.1 showed the problems associated with subtyping. When `self` is included as an implicit parameter to every signature in a class definition, it is impossible for subclasses to denote subtypes. Even when `self` is extracted from interface signatures and made a parameter to the class, redefinition of explicit signatures in subclasses must still respect contravariance. The tension can be more generally identified as that of subclass-based type interpretations and covariant subclassing redefinition. If covariant subclassing redefinition is supported in a strongly and statically typed language with subclass-based binding checks, then parameter binding checks can never succeed.

The conflict of subclass-based binding checks and covariance was first discussed in [GrBe89] and [Co89] and shown to have concrete implications in languages such as Eiffel. Grogono and Bennett named the problem, the "binary operator problem" and showed that languages which support both features must forbid the sending of meaningful arguments to methods, or forego static and strong typing. The following example is written in an extension of Mini-Dee with both subclass-based binding checks and covariant subclassing redefinition. Consider a variable, a : A, where A is defined by the class definition below, and where P is any class:

```
class A

        method m (p : P)
                begin
                        .
                end

end A.
```

Example 2.14

Since covariant redefinition is supported, subclasses of A can redefine m to accept an object of any type denoted by a subclass of P. Any invocation of m on a must be accompanied by an argument that can be bound to p, no matter which subclass of P denotes its type. Thus, if a can be bound to an object of any subclass of A, then m can be invoked on a only with an argument that can be bound to variables of all subtypes of *P*. Only a trivial value that can be bound to variables of all types satisfies this demand, and only this value can be sent as an argument to m.

Languages that use subclass-based binding checks differ by how they resolve this tension. Trellis restricts subclassing redefinition to be contravariant. This allows parameter binding checks to be identical to binding checks, but restricts subclassing significantly, denying potentially useful reuse of class definitions. Eiffel allows covariant subclassing redefinition, but sacrifices strong and static typing. Early versions of Eiffel allowed code such as that of **Example 2.14**, to compile and produce unpredictable and potentially disastrous results. More recent versions of Eiffel provide dynamic type-checking. Palsberg and Schwartzbach do not discuss this tension in their papers, and one must assume that their proposal does not allow meaningful arguments to sent with method invocations, or that their claim of strong and static typing is exaggerated.

2.6 Conclusion: The Conflicting Roles of Inheritance

Resolution of the tension of dynamic binding does not bode well for subclass-based type interpretations. Such interpretations are appealing because they allow flexible and intuitive binding checks, and demand that the programmer be aware of only one class and type hierarchy. Unfortunately, these interpretations also demand that one of:

- covariant subclassing redefinition,
- argument passing with method invocations, or
- strong or static typing

be sacrificed. Inheritance is a suitable basis for determining the mappings of nominal polymorphism and static binding since the tensions of the latter include that of the former, and both can be resolved while maintaining some flexibility. When used as the basis for dynamic binding also, the added tension demands that inheritance mechanisms be unsafe or too restrictive to be practical.

Subclass-independent type interpretations should be the basis for dynamic binding components of object-oriented languages. Subclassing should be viewed as a mechanism for reusing code and signatures, and should address the tensions of:

- covariant and selective signature redefinition and
- contravariant signature redefinition and genericity.

The design of an object-oriented type and subclassing system based on the above philosophy, is the focus of Chapters 5 and 6 of this thesis. These chapters borrow ideas found in Chapters 3 and 4 which are surveys of the language proposals of Bruce [Br92] and Palsberg and Schwartzbach [PaSc90a, PaSc90b, PaSc90c, PaSc90d, PaSc91] respectively. The proposals are compared and evaluated according to the kinds of polymorphisms that are supported, and how the tensions of polymorphisms are resolved.

Chapter 3.

Survey: An Examination of the Polymorphisms of the Bruce Language Proposal

3.1 Introduction

This chapter and the next examine the polymorphisms supported in the recently proposed languages of Bruce [Br92] and Palsberg and Schwartzbach [PaSc90a, PaSc90b, PaSc90c, PaSc90d, PaSc91]. The languages differ markedly. The emphasis of the Bruce approach is flexible and type-safe dynamic binding, while Palsberg and Schwartzbach are primarily concerned with nominal polymorphism and static binding. Appropriately, the strengths of each proposal are the polymorphisms emphasized, and the weaknesses, the polymorphisms ignored. The Bruce approach, while supporting a subclass-independent type interpretation, does not provide much support for static binding and only limited support for nominal polymorphism. The Bruce approach also places excessive limits on the composition of a class. The PS proposal allows more reasonable class composition, offers flexible and provably type-safe static binding and nominal polymorphism, but only in the absence of dynamic binding. A description of each proposal follows, broken down into the support offered for each form of polymorphism. Each proposal is evaluated for the choices made to resolve the tensions of polymorphism identified in Chapter 2. The survey sets the stage for an original desig.ı of a type system and subclassing system which borrows from both, and which is the subject of Chapters 5 and 6.

3.2 Basic Language Features of the Bruce Approach

The Bruce language is based on the F-Bounded model of objects of Cook et. al. [CoHiCa89] It might be remembered that the F-Bounded model of objects is an extension of the Cardelli and Wegner model, and uses recursive class definitions and fixpoints to resolve the difficulty of associating subclasses with subtypes. Despite the functional and subclass-based lineage, the Bruce language incorporates both state (allowing values of instance variables to be updated) and a subclass-independent interpretation of types.

Types are associated with specifications and classes can provide varying implementations of those specifications. Thus types are literally non-singleton. A type can be denoted by a class name, or by the special recursive name, *MT* (short for "My Type") which is the type of `self`. The type denoted by a class name is the set of all classes which share the same method interface as the named class. (Variables are private in the Bruce language, and considered to be relevant to type implementations (classes) but not to type specifications.) The type denoted by *MT* is the set of all classes sharing the same method interface as the class in which the type reference appears.

Subtyping is supported. One type is considered a subtype of another if its method interface is a contravariant version of that of its supertype. Resulting binding checks are therefore similar to those found in Trellis/Owl, though unlike Trellis/Owl, subclassing is not restricted by requirements of subtyping.

All example classes of the Bruce approach will be translated into an extension of Mini-Dee which extends the core language syntactically by:

- supporting single inheritance. Therefore, the first rule for IList (Inherits List) in the grammar for Mini-Dee of Appendix I-A applies.

- Allowing use of the type name, *MT*. Therefore, MT is added as a token symbol, and the following rule should be added for the nonterminal, T (Type) to the core Mini-Dee grammar of Appendix I-A:

  $$T \rightarrow \text{MT}$$

  For reasons that will be discussed in Section 3.6, the recursive type reference can only appear in a method signature and not in an instance variable signature. This should demand that the Mini-Dee grammar distinguish between variable and method types, but the distinction is made informally to keep the grammar simple. The grammar for the Bruce-based extension of Mini-Dee is presented in Appendix I-B.

Semantic extensions to the core language are described briefly here, and explored in greater detail in the sections on polymorphisms that follow. With regards to type-checking, binding and parameter binding checks are identical, and allow an object to be bound to a variable provided that the class denoting the type of the object is a contravariant version of the class denoting the type of the variable. (This reflects the definition of subtyping given above). Signature checks can be confined to the class denoting the type of a receiver object since all classes in the type and its subtypes will include the methods defined in that class.

3.3 Polymorphisms in the Bruce Approach

3.3.1 Nominal Polymorphism

Explicit subclassing redefinition in the Bruce approach is limited to that which results in nominal polymorphism. Both forms of nominal polymorphism described in 2.3 are supported. When a method's code is redefined, the new code must naturally be compiled and type-checked. A method's code and signature can be redefined provided that again, new code is type-checked, and signature redefinition has contravariant direction. That signature redefinition is contravariant allows redefinition to be selective in scope. It also means that instance variable types cannot be redefined and that method parameters can only be generalized in subclasses.

The inflexibility of contravariant signature redefinition is evident when examining how the Bruce approach constrains the definition of the class, `WrappingTurtle` from section 2.3. It might be remembered that it was desirable for this class to be a subclass of class *Turtle*, but redefining the inherited methods. In particular, it was desired that:

- the *code* of the method `move` be redefined and
- the *code and signature* of the method `distance` be redefined.

The Bruce approach would allow the first redefinition, but not the second. It is not a problem to modify code and signature of a method, but signature redefinition must respect contravariance. The method `distance` cannot be redefined as desired because to do so, the types of its parameters must be specialized from *Int* to *ScreenRange*.

An exception to contravariant and selective redefinition demands that a method parameter be specialized in subclasses, if its type is *MT*. *MT* always refers to the type denoted by the class in which it is used. An inherited signature containing a reference to *MT* cannot be redefined such that the reference to *MT* is replaced. Like `self`, any variable whose type is *MT* is automatically and covariantly redefined in subclasses. Thus, signature redefinition as a rule is contravariant and selective, but in the case of recursive signatures is covariant and consistent.

It was shown in Chapter 2 that the coexistence of covariance and selective redefinition is problematic. The tension is evident in the Bruce approach because the redefinition of recursive signatures, though covariant, is consistent only within a class. It will be shown in section 3.5 that classes that include bindings of recursively typed objects to non-recursively typed variables found in other classes cannot be subclassed in the Bruce approach, and thus must be considered invalid, despite satisfying all type-checks.

3.3.2 Static Binding

One weakness of the Bruce approach is its weak support of static binding. Prefixing is supported though because recursive signatures must remain recursive in subclasses, the explicit portion of a signature can only be inherited "as is" if it contains no recursive references. Genericity is not supported. The redefinition of the explicit portion of a signature must be accompanied by a recompilation of inherited code. The language also does not support parametric operations or classes. Extending the language by offering support for such forms of genericity is suggested in [Br92] as a possible direction of further research. It is not clear however, how the tension of genericity and contravariant signature redefinition in subclasses will be resolved given this addition to the language.

Static binding was required to redefine the inherited methods and variables of the class `Turtle`, in the class `WrappingTurtle` from section 2.3. In particular, it was desired that:

- no redefinition be made of the method, `turn` (*prefixing*) and
- the signatures of the method, `set_pos`, and the instance variables, x and y be redefined (*genericity*)

The Bruce approach would allow the first redefinition but not the second. The method `set_pos` cannot be redefined as desired because genericity is not supported in any form. The types of the instance variables x and y cannot be changed because of the demand for contravariant signature redefinition in subclasses.

3.3.3 Dynamic Binding

The dynamic binding component of the Bruce language is its greatest strength. The Bruce interpretation is subclass-independent since subtypes need not be denoted by subclasses. That both subtyping and subclassing rely on contravariance means that a subclass can denote a subtype provided that it inherits no methods with recursive parameters. Subtypes do not have to be denoted by subclasses however, since the inherited instance variables of a subclass must be typed and named as they are in its superclass, whereas the types and names of instance variables in classes denoting subtypes, need not have anything in common with the classes denoting supertypes. Because its type interpretation is subclass-independent, binding and parameter binding checks are identical, and do not interfere with the flexibility of subclassing redefinition. Because subtyping demands contravariance and contravariance implies name conformance, signature checks can be confined to the class which names the type.

3.4 Type-Safety is not Class Validity

It is shown in the Bruce paper [Br92] that the type-safety of a class is not sufficient for establishing the validity of a class. There are some classes which would type-check but are considered unsafe in the Bruce system because the guarantee cannot be made that their subclasses will be type-safe. [Br92] cites an example of this. Assuming the existence of classes `Student` and `Person` such that *Student* is a subtype of *Person*, Bruce claims that the class `Friend` of **Example 3.1** will not type-check:

```
class Friend
      method who () : Person
        begin ... end

      method same (other : Person) : Bool
        begin
             return (who () = other)
        end
end Friend.
```

Example 3.1

The reason given for this class' failure to type-check concerns the expression, `who() = other`. It is possible for a subclass of `Friend` to redefine the method `who` so that it returns a *Student* rather than a *Person* since *Student* is a subtype of *Person*. Thus, it is argued that the prefixed method `same` will no longer be type-safe since the comparison will be made of objects not belonging to the same type.

This example is flawed because of an assumption made about the '=' operation. Though the dot notation is not used to invoke '=', it appears that '=' is a method defined in class `Person`, and that it requires another *Person* object as an argument. (If on the other hand, '=' is applicable to objects of any class, then it is implicitly defined as a method in every class including `Person`, and the same argument follows). Because it is a method with a recursively declared parameter, any subclass of `Person` must inherit '=' and retain the recursive parameter. Thus any subclass of `Person` (including `Student`) will not be a subtype because contravariance is not respected in the signature of this method. Since *Student* cannot be a subtype of *Person*, the specialization of the return type of '`who`' in any subclass of `Friend` is invalid.

Though the above example is flawed, there are other classes whose subclasses do not retain the type-safety of inherited code. These classes are those with a method with a recursive parameter, and with a method whose code includes the binding of an object of recursive type to a variable of non-recursive type. An example of this kind of binding occurs with a *double dispatch*; a method invocation where `self` is sent as an argument. *Double dispatching* has been shown in [HeJo90] to be a useful means of defining arithmetic operations and other operations where the resolution of overloading should depend on the class of more than one argument. Any Bruce class containing both code that executes a double dispatch and a method with a recursive parameter, cannot guarantee the type-safety of its subclasses. The classes of **Example 3.2** demonstrate:

```
class Uses_A

    method m (a : A)
        begin  ... end

end Uses_A

class A

    var u : Uses_A;

    method p (x : MT)
        begin  ... end
        method q
              begin
                    u.m (self)
              end
  end A
```

Example 3.2

Because p has a recursive parameter, no subclass of A denotes a subtype of *A*. Therefore, the method q cannot be safely inherited since the binding of `self` to the parameter, a of type *A* will fail.

3.5 Cooperative Subclassing and Dynamic Binding: A Case Analysis

An exhaustive case analysis is presented that shows classes containing all possible forms of bindings and parameter bindings, and that considers how these classes are subclassed. Nine examples (ex13a - ex15c) are presented of classes that cannot be safely subclassed because of the binding of recursively typed objects to non-recursively typed variables. The exhaustive case analysis that follows examines the bindings and parameter bindings that can be type-safe in a superclass, to see if they are ever unsafe once inherited by a subclass. For each case, a potential superclass is shown and the type-safety of its potential subclasses is considered.

The cases are distinguished by the nature of the binding found in the potential superclass. Before categorizing the forms which a binding can take, preliminary discussion demands that the components of a bind be identified, and the forms that expressions can take, classified.

3.5.1 The Three Components of a Bind

All bindings (including parameter bindings) can be broken down into three components: the *expression bound*, the variable to which the expression is bound (*receiver*), and the expression which owns the receiver (*owner*). For example, given the binding of the constant, 45 to the instance variable, `i`:

```
i := 45
```

the expression bound is 45, the receiver is `i` and the owner is `self`, since `i` must be defined in the same class as this statement. As another example, let `a` be one expression, and `e : E` another expression where the class `E` includes a definition for the method, `m` with explicit signature:

```
method m (p : P).
```

Then, given the binding,

```
e.m (a),
```

the expression bound is `a`, the receiver is `p`, and the owner is `e`.

3.5.2 Classifying Expressions

There are two expressions involved in every bind: the bound expression and the owner expression. These expressions can be categorized by how their types can be modified in subclasses.

1. *Constant Expressions*: These expressions are those which denote objects whose type cannot be modified in subclasses. In the Bruce language, both constants (such as NIL, 17, TRUE etc.) and instance variables fall under this heading.

2. *Type Generalized Expressions*: These expressions denote objects whose types can be redefined to be supertypes in subclasses. Non-recursive method parameters fall under this heading.

3. *Type Specialized Expressions*: These expressions denote objects whose types can be redefined to be subtypes in subclasses. Since the return types of methods can be redefined in this way in subclasses, results of method invocations fall under this heading.

4. *Class Specialized Expressions*: This heading captures all expressions whose type is *MT*. This includes recursive method parameters, recursive method return types and self. Note that these expressions are not necessarily type specialized since subclasses do not necessarily denote subtypes.

3.5.3 The Structure of the Case Analysis

The case analysis considers the binding of an object expression of every possible category, to a variable owned by an expression of every possible category. The tree of **Figure 3.1** demonstrates all possible bindings that will be considered in the following section.

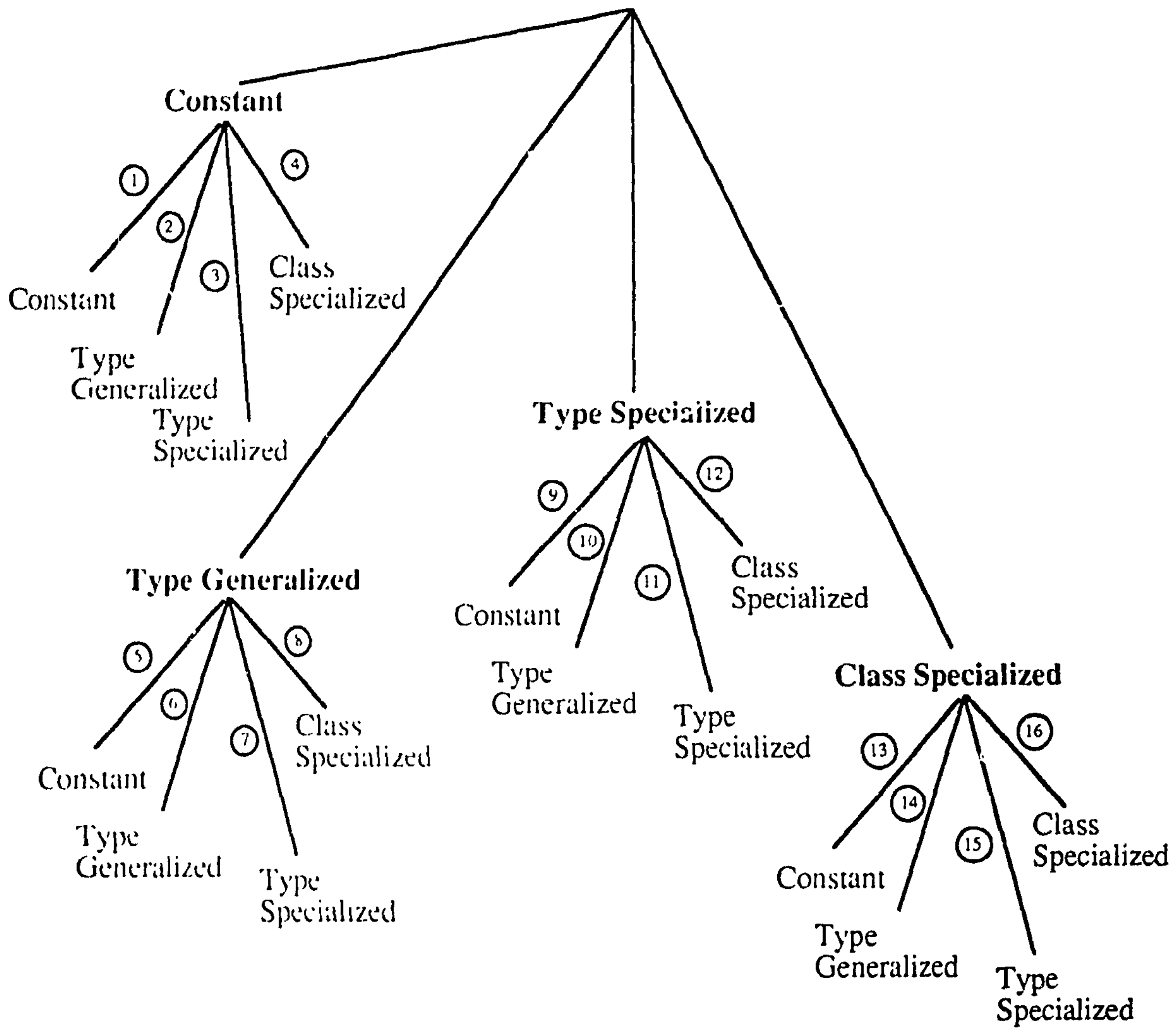


Figure 3.1: Case Analysis Tree

The boldfaced labels of the above tree concern the classifications of the bound expressions. The labels of all leaves concern the classifications of the owner expressions. Each leaf address is numbered, and each example in the case analysis is matched with the address of the leaf with which it is associated.

The 16 cases examined form a tedious analysis, though one that confirms that type-safety can be guaranteed of all inherited code, except for that involving the binding of recursively typed objects to non-recursively typed variables. (Cases 13a - 15c).

3.5.4 The Cases

For the following examples, it should be assumed that there exist classes V (short for Variable) and O (short for Object) such that the type *O* is a subtype of the type *V*. Every example class will include a binding of an *O* to a *V*, and all possible subclasses will be considered to see if the binding or parameter binding check will still hold in subclasses given all possible redefinition. For the sake of readability, some consistency in naming is employed. Every bound object is named o, and every variable to which it is bound is named v. When v is a method parameter, it is a parameter for a method, m_c with the explicitly defined signature:

```
method m  (v : V)
```

When m_c is a method located in another class besides the example class, then it is found in a class, C. The method or variable, c is always either of type *C* or *MT*, depending on whether C or the containing class contains a definition for m .

Bound Object is a Constant Type Expression

For these examples, the bound object could be a constant or an instance variable since the types of constants are constrained in subclasses in the same manner as instance variable types in the Bruce system. All of the examples here will only include bindings to instance variables, with the understanding that the conclusions drawn are applicable to examples with constants.

1. Owner Object is a Constant Type Expression (1)

```
class ex1
        var c : C
        var o : O
        method m
                begin
                        c.m  (o)
                end
end ex1
```

The types of both o and c will be the same in all subclasses. Since c has the same type in all subclasses, so too does the method parameter, v. Therefore, the parameter binding check of the invocation, "c.m_c (o)" will hold in all subclasses of ex1.

2. Owner Object is a Type Generalized Expression (2)

```
class ex2
        var o : O
        method m (c : C)
                begin
                        c.mc (o)
                end
end ex2
```

The type of o will not change in subclasses of ex2. If the signature of method, m is not redefined in a subclass of ex2, then the type of c (and therefore the type of the parameter, v also) will not change, and the parameter binding check of the invocation, "c.m_c (o)" will hold. If the signature of m is redefined, then m must be recompiled.

3. Owner Object is a Type Specialized Expression (3)

```
class ex3
        var o : O
        method c () : C
                begin    end
        method m
                begin
                        c ().mc (o)
                end
end ex3
```

In a subclass of ex3, the type of o will be *O*, and the return type of the method c will be a subtype of *C*. Because of the demand for contravariance in subtypes, the type of the parameter v, in the method m_c will be a supertype of *V*. Since O is a subtype of V, it is also a subtype of any supertype of V. Thus, the parameter binding check of the invocation, "`c () .m (o)`" will still hold in subclasses of `ex3`.

4. Owner Object is a Class Specialized Expression (4)

Class specialized expressions can be any whose type is *MT*. This includes `self`, any explicit recursive parameter, and any recursive return type of a method. Each possibility is considered here.

a. Owner Object is `Self`

Since the owner expression is `self`, binding need not involve passing an argument to a parameter, but could be the assignment of an expression to an instance (or local) variable. Both kinds of binds are considered.

binding

```
class ex4a
        var o : O
        var v : V
        method m
                begin
                        v := o
                end
end ex4a
```

parameter binding

```
class ex4b
        var o : O
        method m (v : V)
                begin  ... end
        method m
                begin
                        m (o)
                end
end ex4b
```

In any subclass of `ex4a`, the types of `o` and `v` will be *O* and *V* respectively. Thus, the binding check of the assignment, "`v := o`" will hold.

In any subclass of `ex4b`, the type of `o` will be *O*, and the type of the variable, v of method `m` will be some supertype of *V*. Since *O* is a subtype of *V*, *O* is a subtype of any supertype of *V*. Thus, the parameter binding check of the invocation, "`m (o)`" will hold.

b. Owner Expression is a recursive parameter

```
class ex4c
        var o : O
        method m_c (v : V)
                begin .. end

        method m (c : MT)
                begin
                        c.m_c (o)
                end
end ex4c
```

In any subclass of ex4c, c will still have type *MT*, o will still have type *O*, and the parameter v of method m_c will be of some supertype of *V*. Since *O* is a subtype of *V*, it is also a subtype of all supertypes of *V*. Thus, the parameter binding check of the invocation, "c.m_c (o)" will hold.

c. Owner Expression is a recursive return type

```
class ex4d
        var o : O

        method c () : MT
                begin    end

        method m_c (v : V)
                begin    end

        method m
                begin
                        c ().m_c (o)
                end
end ex4d
```

In any subclass of ex4d, the return type of c will be *MT*, the type of o will be *O*, and the type of the parameter v of m_c will be a supertype of *V*. Therefore, the parameter binding check of the invocation, "c () .m_c (o)" will hold in any subclass of ex4d.

Bound Object is a Type Generalized Expression

1. Owner Object is a Constant Type Expression (5)

```
class ex5
      var c : C
      method m (o : O)
            begin
                  c.m (o)
            end
end ex5
```

In any subclass of ex5, the type of c will be *C*. If the signature of m is not redefined, then the type of the parameter o will be *O* and the parameter binding check of the invocation, "c.m (o)" will hold in any subclass. If the signature of m is redefined, then m must be recompiled.

2. Owner Object is a Type Generalized Expression (6)

```
class ex6
      method m (o : O, c : C)
            begin
                  c.m (o)
            end
end ex6
```

In any subclass of ex6, if the signature of m is not redefined, then the types of parameters o and c will be *O* and *C* respectively, and the parameter binding check of invocation, "c.m (o)" will still hold. If the signature of m is redefined, then m must be recompiled

3. Owner Object is a Type Specialized Expression (7)

```
class ex7
      method c () : C
            begin ... end
      method m (o : O)
            begin
                  c ().m_c (o)
            end
end ex7
```

In any subclass of ex7, the return type of the method c will be some subtype of *C*. Because of the demand for contravariance in subtypes, the type of the parameter v in the method m in any subtype of *C*, will be a supertype of *A*. If the signature of method m is not redefined, the parameter binding check of the invocation, "c () .m_c (o)" will still hold since *O*, as a subtype of *V* is also a subtype of all supertypes of *V*. If the signature of m is redefined, the code of m will have to be recompiled.

4. Owner Object is a Class Specialized Expression (8)

Like the examples of case 4, the owner expression in these examples can be `self`, a recursive parameter or a recursive method return type. All possibilities are considered here.

a. Owner Object is `self`

Since the owner object is `self`, binding need not involve passing an argument to a parameter, but could be the assignment of an expression to an instance (or local) variable. Both kinds of binds are considered.

binding:

```
class ex8a
        var v : V

        method m (o : O)
                begin
                        v := o
                end
end ex8a
```

parameter binding:

```
class ex8b
        method m_c (v : V)
                begin ... end

        method m (o : O)
                begin
                        p (o)
                end
ex8b
```

In any subclass of ex8a, the type of v will be V. If the signature of m is not redefined, then the type of the parameter o will be O and the binding check of the assignment, "v := o" will hold. If the signature of m is redefined, then the code of m must be recompiled.

In any subclass of ex8b, the type of the parameter v of method m_c will be some supertype of V. If the signature of method m is not redefined, then the parameter binding check of the invocation, "p (o)" will hold because O, as a subtype of V, is a subtype of any supertype of V. If the signature of the method m is redefined, then the code of m must be recompiled.

b. Owner Expression is a recursive parameter

```
class ex8c
        method m_c (v : V)
                begin    end

        method m (c : MT  o : O)
                begin
                        c.m_c (o)
                end
end ex8c
```

In any subclass of ex8c, c will still have type *MT*, and the parameter, v of method m, will be of some supertype of *V*. If the signature of method m is not redefined, then the type of its parameter, o will be *O*, and the parameter binding check of the invocation, "c.m (o)" will hold because *O*, as a subtype of *V*, is a subtype of all supertypes of *V*. If the signature of m is redefined, then its code must be recompiled.

c. Owner Expression is a recursive return type

```
class ex8d

        method c () : MT
              begin     end

        method m  (v : V)
              begin  .. end

        method m (o : O)
              begin
                     c ().m  (o)
              end
end ex8d
```

In any subclass of ex8d, the return type of the method, c will be *MT*, and the type of the parameter v of m will be a supertype of *V*. If the signature of method m is not redefined, then the type of the parameter o will be *O* and the parameter binding check of the invocation, "m () .m (b)" will hold because *O*, as a subtype of *V*, is a subtype of all supertypes of *V*. If the signature of method m is redefined, then the code of m must be recompiled.

Bound Object is a Type Specialized Expression

1. Owner Object is a Constant Type Expression (9)

```
class ex9
      var c : C

      method o () : O
            begin ... end

      method m
            begin
                  c.m (o ())
            end
end ex9
```

In any subclass of ex9, the type of c will be C and the return type of the method, o will be a subtype of O. Since O is a subtype of V, any subtype of O is a subtype of V and the parameter binding check of the invocation, "c.m (o ())" will hold.

2. Owner Object is a Type Generalized Expression (10)

```
class ex10
      method o () : O
            begin . end

      method m (c : C)
            begin
                  c.m (o ())
            end
end ex10
```

In any subclass of ex10, the return type of the method, o will be some subtype of O. If the signature of m is not redefined, then the type of its parameter, c will be C. Since O is a subtype of V, any subtype of O is also a subtype of V and the parameter binding check of the invocation, "`c.m`$_c$ `(b())`" will hold. If the signature of method m is redefined, then its code must be recompiled.

3. Owner Object is a Type Specialized Expression (11)

```
class ex11
        method o () : O
              begin ... end

        method c () : C
              begin ... end

        method m
              begin
                     c ().m  (o ())
              end
end ex11
```

In any subclass of `ex11`, the return type of method o will be a subtype of *O*, and the return type of method c will be a subtype of *C*. By the rules of contravariance, the type of the parameter v of the method m$_c$ might be any supertype of *V*. Since *O* is a subtype of *V*, any subtype of *O* will be a subtype of any supertype of *V*. Thus, the parameter binding check of the invocation, "`c().m`$_c$`(o())`" will hold.

4. Owner Object is a Class Specialized Expression (12)

Like the examples of cases 4 and 8, the owner expression in these examples can be self, a recursive parameter or a recursive method return type. All possibilities are considered here.

a. Owner Object is `self`

Since the owner object is `self`, binding need not involve passing an argument to a parameter, but could be the assignment of an expression to an instance (or local) variable. Both kinds of binds are considered.

<u>binding</u>:

```
class ex12a
      var v : V

      method o () : O
            begin  ... end

      method m
            begin
                  v := o ()
            end
end ex12a
```

<u>parameter binding</u>:

```
class ex12b
      method m_c (v : V)
            begin  ... end

      method o () : O
            begin  ... end

      method m
            begin
              m_c (o ())
            end
ex12b
```

In any subclass of `ex12a`, the type of the variable `v` will be *V*, and the return type of the method `o` will be a subtype of *O*. Since *O* is a subtype of *V*, it is a subtype of any supertype of *V*. Thus, the binding check of the assignment, "`v := o()`" will hold.

In any subclass of `ex12b`, the type of the parameter `v` in method m_c will be a supertype of *V*, and the return type of the method `o`, will be a subtype of *O*. Since *O* is a subtype of *V*, any subtype of *O* will be a subtype of any supertype of *V*. Thus, the parameter binding check of the invocation, "m_c `(o ())`" will hold.

b. Owner Expression is a recursive parameter

```
class ex12c
      method m (v : V)
            begin  end

      method o () : O
            begin  end
```

```
        method m (c : MT)
              begin
                    c.m_c (o ())
              end
  end ex12c
```

In any subclass of ex12c, c will still have type *MT*, the parameter v of method m_c will be of some supertype of *V*, and the return type of the method o will be some subtype of *O*. Since *O* is a subtype of *V*, any subtype of *O* will be a subtype of any supertype of *V*. Thus, the parameter binding check of the invocation, "c.m$_c$ (o ())" will hold.

c. Owner Expression is a recursive return type

```
  class ex12d

        method c () : MT
              begin ... end

        method m_c (v : V)
              begin ... end

        method o () : O
              begin ... end

        method m
              begin
                    c ().m_c (o ())
              end
  end ex12d
```

In any subclass of ex12d, the return type of c will be *MT*, the type of the parameter v of m_c will be a supertype of *V*, and the return type of method c will be a subtype of *C*. Since *O* is a subtype of *V*, any subtype of *O* will be a subtype of any supertype of *V*. Thus, the parameter binding check of the invocation, "c ().m$_c$ (o())" will hold.

Bound Object is a Class Specialized Expression

This is the category of examples that involve binding recursively typed expressions to variables. The examples of this category are problematic when the examples involve binding to variables that are defined in different classes than the classes containing the obje``s bound, where such variables would not be recursively typed.

For each example, three bindings are shown:

- the binding of `self`,
- the binding of a recursive method parameter and
- the binding of a recursive return type.

For these examples, it is necessary to redefine the class `C` to include a method m_c with signature:

```
method mc (v : ex13)
```

where the type of the parameter v should be the name of whichever example class is being considered. Thus for the example of class `ex13b`, the type of v is *ex13b*. For the example of class `ex14c`, the type of v should be *ex14c* and so on.

1. Owner Object is a Constant Type Expression (13)

```
class ex13a
        method rec_method (any : MT)
                begin .. end

        var c : C

        method m
                begin
                        c.mc (self)
                end
end ex13a
```

```
class ex13b
        var c : C

        method m (o : MT)
               begin
                      c.m_ (o)
               end
end ex13b

class ex13c
        method rec_method (any : MT)
               begin     end

        var c : C

        method o () : MT
               begin     end

        method m
               begin
                      c.m_ (o ())
               end
end ex13c
```

Subclasses of each of the example classes above will not denote subtypes of the types denoted by these classes because each class has a method with a recursive (covariantly redefined) parameter. As a result, the parameter binding checks of the invocations of

- "`c.m_ (self)`", in `ex13a`,
- "`c.m_ (o)`", in `ex13b` and
- "`c.m_ (o ())`", in `ex13c`

will not hold in subclasses. In any subclass, the types of the objects bound (`self`, `o` and `o()`) are not subtypes of the types of the variables named v to which they are bound.

2. Owner Object is a Type Generalized Expression (14)

```
class ex14a
        method rec_method (any : MT)
                begin     end

        method m (c : C)
                begin
                        c.m  (self)
                end
end ex14a

class ex14b
        method m (o : MT  c : C)
                begin
                        c.m  (o)
                end
end ex14b

class ex14c
        method rec_method (any : MT)
                begin     end

        method o ()  : MT
                begin     end

        method m (c : C)
                begin
                        c.m  (o ())
                end
end ex14c
```

Subclasses of each of the example classes above will not denote subtypes of the types denoted by these classes, because each class has a method with a recursive (covariantly redefined) parameter. As a result, the parameter binding checks of the invocations of

- "`c.m`$_r$ `(self)`", in `ex14a`,
- "`c.m`$_r$ `(o)`", in `ex14b` and
- "`c.m`$_r$ `(o ())`", in `ex14c`

will not hold in subclasses. In any subclass, the types of the objects bound (`self`, `o` and `o()`) are not subtypes of the types of the variables named `v` to which they are bound.

3. Owner Object is a Type Specialized Expression (15)

```
class ex15a
      method rec_method (any : MT)
            begin  .. end

      method c () : C
            begin  .. end

      method m
            begin
                  c ().m_r (self)
            end
end ex15a

class ex15b
      method c () : C
            begin  ... end
```

```
        method m (o : MT)
            begin
                c ().m (o)
            end
    end ex15b

    class ex15c
        method rec_method (any : MT)
            begin . end

        method c () : C
            begin   end

        method o () : MT
            begin   end

        method m
            begin
                c ().m (o ())
            end
    end ex15c
```

Subclasses of each of the example classes above will not denote subtypes of the types denoted by these classes because each class has a method with a recursive (covariantly redefined) parameter. As a result, the parameter binding checks of the invocations of

- "`c ().m (self)`", in `ex15a`,
- "`c ().m (o)`", in `ex15b` and
- "`c ().m (o ())`", in `ex15c`

will not hold in subclasses. In any subclass, the types of the objects bound (`self`, `o` and `o ()`) are not subtypes of the types of the variables named `v`, to which they are bound.

4. Owner Object is a Class Specialized Expression (16)

Like the examples of cases 4, 8 and 12, the owner expression in these examples can be `self`, a recursive parameter or a recursive method return type. All possibilities are considered here.

a. Owner Object is `self`

Since instance variables cannot have recursive type, it is not possible to bind the expression, `self` to an instance variable. Therefore, it suffices for this example to show the effects of binding `self` to a parameter of a method defined in the same class.

```
class ex16a
      method m (v : MT)
            begin . end

      method m
            begin
                  m (self)
            end
end ex16a

class ex16b
      method m (v : MT)
            begin    end

      method m (o : MT)
            begin
                  m (o)
            end
end ex16b
```

```
class ex16c
      method m (v : MT)
            begin .. end

      method o () : MT
            begin    end

      method m
            begin
                  m (o ())
            end
end ex16c
```

The bindings of recursively typed expressions is not a problem here, since the variables to which they are bound are also recursively typed. Since signature redefinition is consistent and covariant within the class, all subclasses of the above class are type-safe.

b. Owner Expression is a recursive parameter

```
class ex16d
      method m (v : MT)
            begin . end

      method m (c : MT)
            begin
                  c.m (self)
            end
end ex16d

class ex16e
      method m (v : MT)
            begin    end
```

```
        method m (c : MT o : MT)
                begin
                        c.m_c (o)
                end
end ex16e

class ex16f
        method m_c (v : MT)
                begin . end

        method o () : MT
                begin . end

        method m (c : MT)
                begin
                        c.m_c (o ())
                end
end ex16f
```

The bindings of recursively typed expressions is not a problem here, since the variables to which they are bound are also recursively typed. Since signature redefinition is consistent and covariant within the class, all subclasses of the above class are type-safe.

c. Owner Expression is a recursive return type

```
class ex16g
        method m (v : MT)
                begin   end

        method o () : MT
                begin . end
```

```
          method m
                  begin
                          c ().m (self)
                  end
  end ex16g

  class ex16h
          method m  (v : MT)
                  begin  ... end

          method c () : MT
                  begin    end

          method m (c : MT)
                  begin
                          c ().m (o)
                  end
  end ex16h

  class ex16i
          method m  (v : MT)
                  begin    end

          method c () : MT
                  begin  .. end

          method o () : MT
                  begin  . end

          method m (c : MT)
                  begin
                          c ().m (o ())
                  end
  end ex16i
```

The bindings of recursively typed expressions is not a problem here, since the variables to which they are bound are also recursively typed. Since signature redefinition is consistent and covariant within the class, all subclasses of the above class are type-safe.

3.6 Choices of the Bruce Approach and their Consequences

The Bruce approach might appear to some to include a strange combination of language features. For example, dynamic binding is subclass-independent, but subclassing is still quite restrictive and contravariant in its direction. As well, no form of genericity is supported and no recursive instance variables are allowed. The combination of features in the Bruce approach can be traced to three language choices and their consequences. These choices and their corresponding consequences are listed below:

Choice 1: Signature Redefinition is Selective

Consequences:

1. Signature redefinition is contravariant in its direction, to resolve the tension of nominal polymorphism between covariant and selective redefinition.

2. Statements involving the binding of constant expressions to variables or parameters such as:

```
(1) i := 45        or
(2) e.m (45)
```

can be safely inherited since the types of instance variables (1) cannot change in subclasses and the type of parameters (2) can only be generalized. (We will see that the use of constant expressions is problematic in the covariant and consistent redefinition of PS subclassing).

3. No form of genericity is supported, to resolve the tension of static binding between contravariant signature redefinition and static binding.

Choice 2: Recursive type references are designated with the name, MT, and are bound to remain recursive in subclasses.

Consequences:

1. Since recursive signature redefinition is automatic and consistent, it can be covariant; an exception to the general rule demanding contravariant signature redefinition.

2. Subclasses do not necessarily denote subtypes of the types denoted by their superclasses. In particular, if a subclass inherits a method with a recursive parameter, then it cannot denote a subtype of the type its superclass denotes.

3. A class must be considered invalid if it includes both a method with a recursive parameter, and a method with code that binds a recursively typed object to a non-recursively typed variable.

Choice 3: Classes and objects in the Bruce language are modeled as in the F-Bounded model, but compromise the functional nature of this model by allowing updates of instance variables.

Consequence:

1. Classes cannot include recursive instance variables. This is because objects are modeled by recursive records that are generated by the application of a fixpoint operation on a record-generating function. Instance variables cannot be part of the records generated by the application of the fixpoint since this would imply that instance variable values must remain constant throughout the life of an object, and that these values are common to all objects belonging to the class. Thus, the scopes of the recursive parameter, `self` and its associated type, *MT* are limited to the methods of an object.

3.7 An Evaluation of the Bruce Approach

The strengths of the Bruce approach are its support for flexible, and subclass-independent dynamic binding, and its support for selective redefinition in subclasses.

Support for subclass-independent dynamic binding should allow subclassing redefinition to be quite flexible. Unfortunately, resolution of the tensions of nominal polymorphism and static binding in this approach make subclassing redefinition inflexible. The inflexibilities of the Bruce approach outweigh its strengths.

The inflexibilities of the Bruce approach concern class composition and subclassing redefinition. Regarding class composition, recursive instance variables are not supported and therefore it is impossible to represent recursive data structures such as lists and trees. With regard to subclassing redefinition, Bruce subclassing resolves the tension of nominal polymorphism by allowing selective but contravariant signature redefinition. To counter the restrictive nature of contravariant redefinition, recursive method parameters are allowed that must be covariantly and consistently redefined. That the covariant redefinition of recursive signatures must only be consistent within a class is problematic. Nine example bindings were shown (examples 13a - 15c of 3.5.4) that satisfy binding and parameter binding checks, but must be considered invalid because the checks would not succeed in subclasses.

Static binding support is severely limited. Prefixing is supported, but because of the tension with contravariant direction of signature redefinition, genericity is not. Thus, it is not possible to modify the signature of an instance variable in a subclass, nor to modify the signature of a method without redefining or recompiling the associated inherited code. Bruce comments that support for genericity might come with extensions to the language proposal. However, support for genericity will not come easily because of the tension with contravariant signature redefinition.

The complex and strict redefinition rules lead Bruce to conclude that inheritance is a mechanism to be avoided in general. Indeed as defined in this language, inheritance has limited uses. The type-safety of the Bruce subclassing system in the presence of dynamic binding is a worthy achievement. But the lack of support for genericity, contravariant restrictions on the modification of inherited methods, and restrictions on class validity above and beyond type-safety make the resulting system impractical in its present form.

Chapter 4.

Survey: An Examination of the Polymorphisms of the PS Language Proposal

4.1 Introduction

Over the course of several papers [PaSc90a, PaSc90b, PaSc90c, PaSc90d, PaSc91], Palsberg and Schwartzbach present a type system and accompanying proposal for subclassing. The papers include discussion of both subclassing and dynamic binding, but the proposals for each must be considered separately. PS subclassing, which achieves nominal polymorphism and static binding and is the focus of PS theory, is only well defined with respect to classes that only include signatures with singleton types, and hence no dynamic binding.

The description of basic language features which follows, defines a language on which PS subclassing theory can be described, where class interfaces only include singleton types. A description of PS subclassing and how it achieves nominal polymorphism and static binding follows. Finally, a section describing proposals for dynamic binding includes a description of a type system which allows types to be non-singleton, and a description of the problems associated with applying this type system to their subclassing proposal.

4.2 Basic Language Features - Classes with Singleton Types

PS subclassing is defined in terms of languages without dynamic binding. All types denoted in class definitions are singleton, and non-trivial subtyping is not supported. All examples of PS subclassing are translated here into an extension of Mini-Dee, that extends the core language syntactically by:

- supporting single inheritance. Therefore, the first rule for IList (Inherits List) in the grammar for Mini-Dee of **Appendix I** applies.

- allowing the types of type declarations to be derived. Class substitution (described in 4.3.2) allows a class to be denoted as indicated by the following grammar rule:

  $$\text{CName} \rightarrow \text{CName}_1 \text{ `[' } \text{CName}_2 \text{ `}\leftarrow\text{' } \text{CName}_3 \text{ `]'}$$

  This rule should be added to the rules for class names found in the Mini-Dee grammar of **Appendix I-A**.

- not allowing constant expressions. It will be shown in 4.3.4 that the use of constant expressions in classes that must be consistently redefined in subclasses, is problematic. In terms of the core Mini-Dee grammar, the rule, "E → Con" should be removed as should all rules for the nonterminal, Con.

The modified grammar is shown in its entirety in **Appendix I-C**.

As usual, semantic extensions of Mini-Dee concern definitions of type-checks and regulations concerning redefinition in subclasses. With regard to the former, binding and parameter binding checks are identical and demand that the class denoting the type of an object be the same as the class denoting the type of the variable to which it is bound. For any invocation, the signature check is naturally confined to the class denoting the type of the receiver object. Redefinition in subclasses has consistent scope, and resembles covariance in its direction.* More detailed descriptions regarding subclassing redefinition follow from the sections below on PS subclassing.

4.3 Nominal Polymorphism and Static Binding in the PS Approach

4.3.1 Informal Description

Flexible subclassing redefinition is the most impressive achievement of the PS approach. The subclassing mechanisms described in the PS approach achieve all forms of nominal polymorphism and static binding. Code redefinition in subclasses must be followed by compilation of the new code. Prefixing is supported, though like the Bruce approach, recursive components of signatures are constrained to remain recursive and therefore true prefixing is achievable only when inherited signatures are not recursive. Signature redefinition (with or without code redefinition) is supported and must be covariant and consistent. This results in flexible subclassing redefinition since instance variable, method parameter and method return types can all be specialized in subclasses. By avoiding contravariant and selective redefinition, the tensions of nominal polymorphism and static binding are resolved.

As a PS class, `WrappingTurtle` of section 2.3 can be derived in its entirety from class `Turtle`. Specifically,

- the *code* of the method `move` can be redefined,
- the *code and signature* of the method `distance` can be redefined, specializing the types of the parameters from *Int* to *ScreenRange*,
- no redefinition need be made of the method, `turn` (*prefixing*) and
- the signatures of the method, `set_pos`, and the instance variables, `x` and `y` can be redefined (*genericity*), specializing the types of the variables and method parameters from *Int* to *ScreenRange*.

* Because PS subclassing is only defined over classes with singleton types, it is not accurate to say that subclassing redefinition is covariant. Signatures can be redefined such that any type denoted by some class named C, can be replaced by a type denoted by a class named C' provided that C' is a <u>subclass</u> of C. Since types are singleton, the type *C'* is not a subtype of the type *C*. In languages that support non-singleton types however, the type *C'* would be a subtype of the type *C*. Therefore, the direction of subclassing redefinition in the PS scheme is referred to here as being covariant.

`WrappingTurtle` is an ideal PS subclass since all desired signature redefinition (specializing the type *Int* to the type *ScreenRange*) is consistently applied to all signatures. The only requirement of this redefinition is that `ScreenRange` be a subclass of `Int`.

The consistent scope of redefinition is achieved by viewing redefinition as an operation over a system of classes rather than on an individual signature. Subclasses result from the application of redefinition operations on their superclasses. The operations, and the subclassing relation over classes, are described below.

4.3.2 PS Subclassing

Representations of Classes: L-Trees

A structural subclassing relation is described in [PaSc90d] in terms of *L-Trees*: tree-based representations of class code. For any class, `C`, the L-Tree representing `C`, is named *TREE* (`C`). The root of *TREE* (`C`) is a node labeled with the untyped code of `C`. Branches extend from where type references occurred in the original code to the L-Trees representing the class naming the type.

The simplest tree is *TREE* (`Object`), denoted by Ω. Definitions for two classes are shown in **Example 4.1** showing less trivial L-Trees.

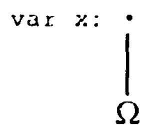
```
class A
      var x : Object
end A

class B
      var x : Object
      method set (y : Object) : B
        begin
            x := y
            return self
        end
end B
```

Example 4.1

Derived from A and B above, *TREE* (A) and *TREE* (B) are illustrated in **Figure 4.1a**.

TREE (A) =



TREE (B) =

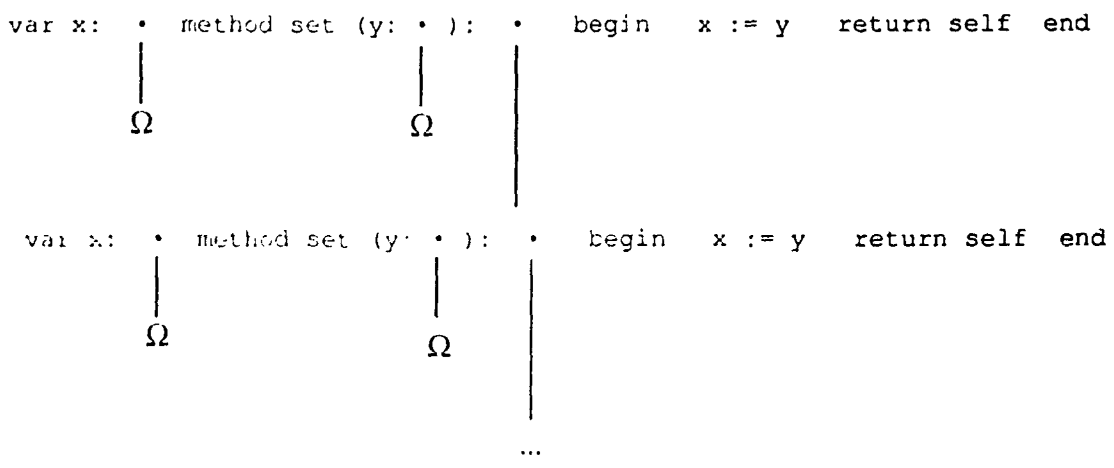

var x: • method set (y: •): • begin x := y return self end
Ω
Ω
var x: • method set (y: •): • begin x := y return self end
Ω
Ω
...

Figure 4.1a

Because of its recursive reference, *TREE* (B) is infinite. The symbol, ◊ is introduced to represent the entire tree, allowing some recursive trees to be finite. Trees that use this symbol are referred to as *generator trees* and are denoted by the function, *GEN*. *GEN* (*TREE* (B)) is as shown in **Figure 4.1b**.

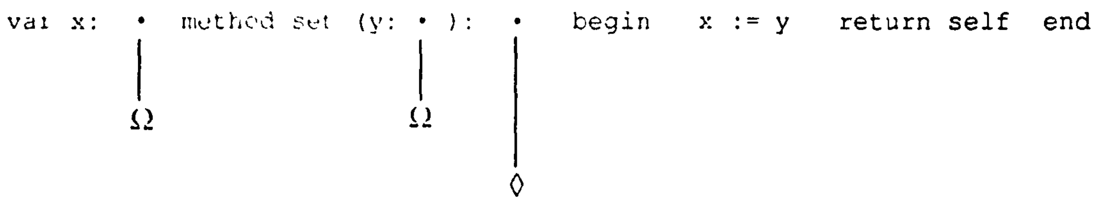


Figure 4.1b

Recursive classes have finite representation with the use of the special symbol, ◊. Clients of recursive classes still have infinite representation, since ◊ can only refer to the entire tree and not just to a subtree.

Subclassing: A Relation On L-Trees

Subclassing (◁) is described as a partially ordered relation on generator L-Trees. Informally, a class `C` is a subclass of another class, `P` if *GEN* (*TREE* (`C`)) has the same structure as *GEN* (*TREE* (`P`)), and differs only in the following ways:

1. *Monotonicity* : Labels in *GEN* (*TREE* (`C`)) must be extensions of, or equivalent to corresponding labels in *GEN* (*TREE* (`P`)). This ensures that signature checks are preserved in subclasses.

2. *Stability* : Equivalent subtrees of *GEN* (*TREE* (`P`)) must have correspondingly equivalent subtrees in *GEN* (*TREE* (`C`)). This requirement ensures that binding and parameter binding checks are preserved in subclasses.

3. *Recursive Retention* : It should be possible to map each of the recursive subtrees of *GEN* (*TREE* (`P`)) to corresponding recursive subtrees at corresponding positions in *GEN* (*TREE* (`C`)). This is necessary to ensure stability since recursive subtrees are equivalent to the implicit subtrees which represent the type of `self`. The subtrees of `self` are automatically kept recursive in subclasses.

The Mini-Dee classes of **Example 4.2** demonstrate how L-Trees can be used to determine subclassing relations.

```
class A
        var o : Object
        var o1 : Object
end A

class A'
        var o : Object
        var o1 : Object
        method m (o : Object)
                begin . end
end A'
```

Example 4.2

GEN (*TREE* (A)) and *GEN* (*TREE* (A')) are shown in **Figure 4.2**.

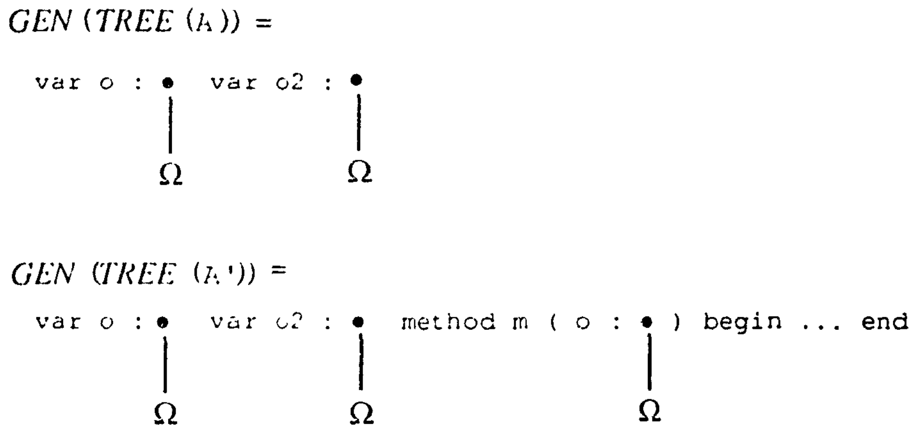


Figure 4.2

Monotonicity holds between *GEN* (*TREE* (A)) and *GEN* (*TREE* (A')) since the only label of the latter tree extends the only label of the former by the addition of text corresponding to the declaration of the method, m. *Stability* holds since the only equivalent subtrees of *GEN* (*TREE* (A)) (both being Ω) are also found at the same positions in *GEN* (*TREE* (A')). *Recursive retention* trivially holds since A is not a recursive class. Thus, A $\lhd$ A'.

Example 4.3 uses the definitions of A and A' of **Example 4.2**.

```
class C
        var c : C
        method m (o : Object  a : A)
                begin   end
end C
```

```
class C'
      var c : C'
      method m (o : Object  a : A')
            begin  .  end
end C'
```

Example 4.3

GEN (*TREE* (C)) and *GEN* (*TREE* (C')) are shown in **Figure 4.3**.

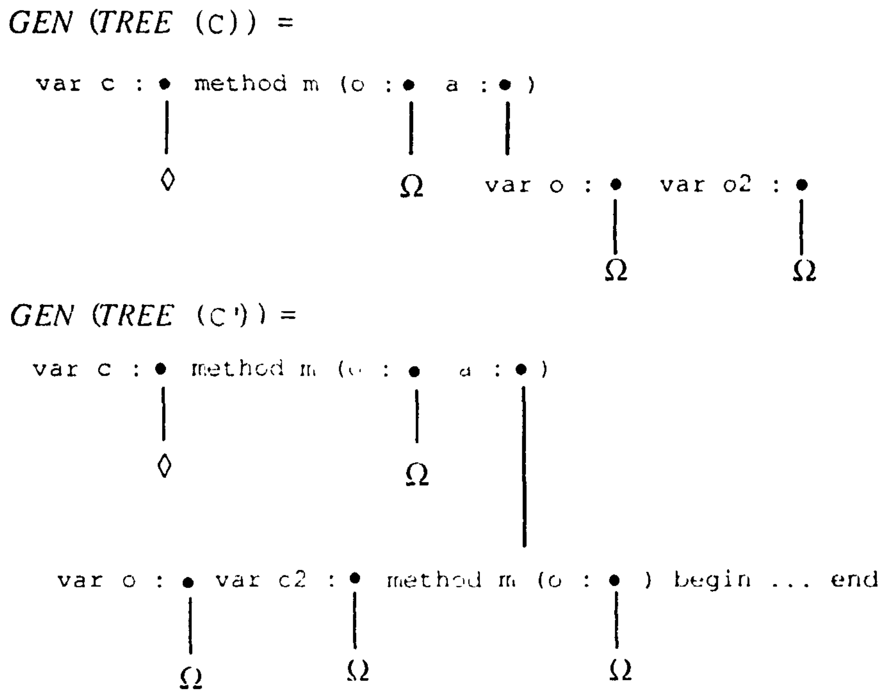

GEN (TREE (C)) =
var c : ● method m (o : ● a : ●)
◊
Ω
var o : ● var o2 : ●
Ω
Ω
GEN (TREE (C')) =
var c : ● method m (o : ● a : ●)
◊
Ω
var o : ● var c2 : ● method m (o : ●) begin ... end
Ω
Ω
Ω

Figure 4.3

Monotonicity holds between *GEN* (*TREE* (C)) and *GEN* (*TREE* (C')) because the root labels of the trees are identical and because the subtree label of *GEN* (*TREE* (C')) is an extension of the corresponding label in *GEN* (*TREE* (C)). *Stability* holds since the equivalent subtrees of *GEN* (*TREE* (C)) (all three being Ω) are found at the same positions in *GEN* (*TREE* (C')) and are equivalent at these positions also. *Recursive retention* holds since the only recursive subtree of *GEN* (*TREE* (C)) is found in *GEN* (*TREE* (C')) at the same position in the tree. Thus, C < C'.

4.3.3 Subclassing Mechanisms

The function *TREE* is only well-defined over classes whose definitions do not mention classes that are inherited. Classes with references to superclasses must first be converted to classes without mention of superclasses before being represented as L-Trees.

Inheritance

The conversion is referred to in [PaSc90d] as an "unfolding." A description of the unfolding algorithm is used to define the PS notion of inheritance. Input to the unfolding algorithm consists of:

- an initial system of classes whose definitions can include mention of inherited classes, and
- the systems of each inherited class.

All input classes are converted into a graph-based representation of class definitions. The output from the unfolding algorithm is another graph, equivalent to that generated of the input, but without indication of subclassing relations. Class definitions containing no mention of inherited superclasses can then be inferred from the resulting graph.

1. The Graph-Based Representation of Class Definitions

In this representation, every class name is a label for a node. Edges in the graph are of two types. "Has-a" edges connect client classes to their suppliers. "Is-a" edges connect subclasses to their superclasses. Definitions for sample classes, `U`, `V`, `W` and `R` are shown in **Example 4.4**. The graph representation for this set of classes is illustrated in **Figure 4.4**.

```
class U
        var a : U
end U
```

```
class V inherits U
      var b : W
end V

class W
      var c : V
end W

class R inherits V
      var d : W
end R
```

Example 4.4

Graph:

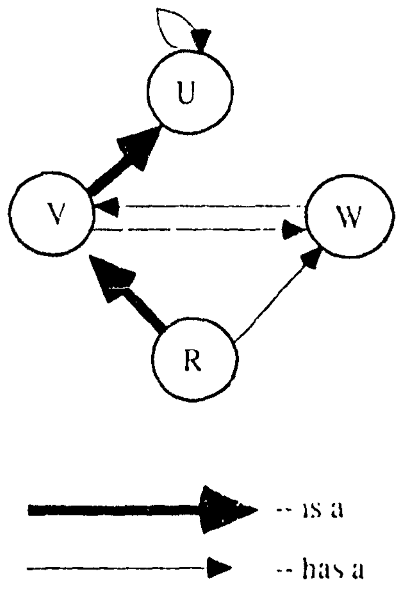


Figure 4.4

2. Temporal Cycles and Multiple Inheritance

The unfolding algorithm demands that input class definitions contain no temporal cycles and limits each class' definition to listing one inherited superclass. A temporal cycle exists when a cycle in the representation graph contains at least one "is-a" edge.

Intuitively, the temporal ordering corresponds to the order in which one would write classes. Ideally, a subclass is written after its superclass and thus should not be known to the superclass. In practice, the illegality of temporal cycles rules out some useful systems of classes including the `Voter` and `Candidate` classes, and the `Employee` and `Manager` classes of **Example 4.5**.

```
class Voter
      var backs : Candidate
end Voter

class Candidate inherits Voter

end Candidate

class Employee
      var works_for : Manager
end Employee

class Manager inherits Employee
 .
end Manager
```

Example 4.5

A *Candidate* is a specialized *Voter* and a *Voter* backs a *Candidate*. An *Employee* works for a *Manager* who is also an *Employee*.

Nodes in the representation graph are also limited to be sources only for one "is-a" edge, thus eliminating the possibility of multiple inheritance. It will be shown in the next chapter that restrictions forbidding systems with temporal cycles and multiple inheritance are unnecessary. The unfolding algorithm is limited to such systems, and therefore all input to the algorithm should be assumed free of temporal cycles and classes with more than one immediate superclass.

3. The Unfolding Algorithm Described

The unfolding algorithm unfolds the graph of a system of classes containing "is-a" and "has-a" edges into a graph containing only "has-a" edges. The algorithm processes the original graph in batches. At any point during the unfolding process, a node will fall in one of three categories:

- D (Done) = the set of nodes which have been processed,
- C (Current) = the set of nodes which are currently being processed, or
- T (To Do) = the set of nodes which have yet to be placed in a batch for processing.

An out-going "is-a" edge originating from a node in C will only lead to nodes in D, and an out-going "has-a" edge originating from a node in C will only lead to nodes in C and D. Because systems with temporal cycles or multiple inheritance are excluded as input, the algorithm is guaranteed to avoid deadlock.

A node in the current batch is processed by replacing its outgoing "is-a" edge (if it exists) with a set of "has-a" edges corresponding to the out-going "has-a" edges of the superclass (the receiving node of the "is-a" edge being replaced). If any of the "has-a" edges of the superclass lies on a cycle, then a copy must be made of each node in the cycle and all edges using any one of these nodes as a source. The new set of nodes and edges resembles the corresponding set of nodes and edges of the original graph, except that the superclass node is replaced by the subclass node.

4. An Example Application of the Unfolding Algorithm

The unfolding of the classes U, V, W and R of **Example 4.4** is described here. The expansion can be completed in three batches. Class U is temporally independent since it is a client only of itself and is a subclass of no class. Once the node for class U has been processed, classes V and W follow since V "is-a" U and "has-a" W and W "has-a" V. The node for class R is the sole member of the last batch, dependent on the processing of its superclass, V. **Table 4.1** shows how the graph is transformed in processing each batch.

Batch	Node Added	Edge Removed	"Has-a" Edge Added
U	-	-	-
V, W	-	(V,U) - "is-a"	(V,V)
R	W'	(R,V) - "is-a" (W,R) - "has-a"	(R, R) (R,W') (W',R)

Table 4.1

Processing of the first batch does not alter the graph since class U has no superclasses. Removal of the "is-a" edge, (V,U) requires the addition of the recursive variable, *a* to the class V and this the addition of the edge (V,V) to the graph. The removal of the "is-a" edge, (R,V) requires the addition of a new node, W' which represents a new implicit subclass of class W. This new node must be added to make sure that R has the same recursive structure as its superclass, V. V is a client of a class (W) which in turn is a client of V. It is necessary for class R to be a client of a class (W') which is in turn a client of R and not a client of V. Thus, the "has-a" edge, (R,W) is removed and replaced by (R, W'). The resulting graph and associated source code is shown below:

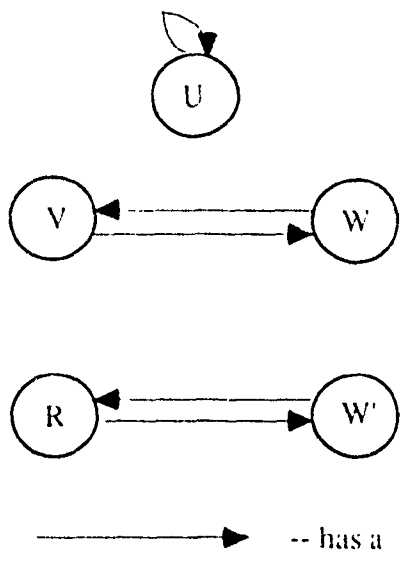


Figure 4.5

Code:

```
class U
        var a : U
end U

class V
        var a : V
        var b : W
end V

class W
        var c : V
end W

class R
        var a : R
        var b : W'
        var c : W'
end R

class W'
        var c : R
end W'
```

Example 4.6

The L-Trees of subclasses generated by the unfolding algorithm differ from the L-Trees of their superclasses by having a root label which is a superstring of the of the original tree's root label, and by having all (possibly deep) recursive subtrees of the superclass tree, adjusted in the subclass tree to refer to the subclass.

Class Substitution

Inheritance allows for prefixing (with automatic redefinition of recursive references), code redefinition and the addition of new methods and variables in a subclass.

Class substitution controls all signature redefinition involving non-recursive type references. In terms of L-Trees, inheritance results in the construction of subclass L-Trees which differ from the superclass L-Trees by the root label. Class substitution results in the construction of subclass L-Trees which differ from the superclass L-Trees by the labels of any nodes except the root. Since class substitution is associated with signature redefinition, it is an alternative mechanism to parametric classes.

Like inheritance, substitution is defined as a function over class definitions. Like the unfolding algorithm, the effect of the substitution algorithm is to translate classes denoted using the notation of substitutions, into classes that can be represented as L-Trees. Given classes D, B_i, and C_i, where C_i is a subclass of B_i ($B_i \lhd C_i$), one can denote a subclass of D, D' with the expression:

$$D' = D\ [B_1, \ldots, B_n \leftarrow C_1, \ldots, C_n]$$

As an example, given the class A:

```
class A
        var o1 : Object
        var o2 : Object
        method m
                begin
                        o1 := o2
                end
end A
```

the expression, A [Object ← Int] would be a shorthand for denoting the class A', shown in **Example 4.7**.

```
class A'
        var o1 : Int
        var o2 : Int
        method m
                begin
                        o1 := o2
                end
end A'
```

Example 4.7

Class substitution can be used to denote both a class and a type, since in the singleton interpretation of types, the two are roughly equivalent. A declaration can use class substitution to denote a type, as in:

```
var a : A [Object ← Int]
```

A class definition can use class substitution to denote a class, as in:

```
class A' inherits A [Object ← Int]
```

The latter example shows that inheritance and class substitution are complementary operations. They can and should be used together to denote subclasses of existing classes.

1. Class Substitution is Consistent Throughout A System of Classes

At first glance substitution appears similar to parametricity, differing by the use of every type reference as a potential type parameter and by the use of every class as a potentially generic class. While both of these statements are true, class substitution differs from parametricity in that it is inherently consistent throughout a system of classes. A parametric version of class A from **Example 4.7**, might not use a type variable to denote the types of both o_1 and o_2. **Example 4.8** shows a variation of A where the type of o_1 is defined by a type parameter and the type of o_2 is defined by a class name.

```
class A [T : Object]
        var o1 : T
        var o2 : Object
```

```
          method m
                    begin
                              o1 := o2
                    end
end A.
```

Example 4.8

As was mentioned in Chapter 2. such parametric classes do not resolve the tension of covariant and selective redefinition. Normally, a parametric class will type-check if the replacement of all type variables with their constraints results in a concrete class that type-checks. The parametric class, `A` should type-check since in this concrete version of `A`, the variable o_1 belongs to class `Object` and thus the assignment in method `m` is of an object of type *Object*, to a variable of type *Object*. Despite the type-safety of the parametric class itself, any instantiation of `A` save for `A [Object]` will not be type-safe.

The problem is not resolved by demanding consistent redefinition within a class. Consider the system of parametric classes of **Example 4.9**, similar to the classes of **Example 4.8**.

```
class B [T : Object]
          val o1 : T
          val o2 : T
          var o3 : C
          method m
                    begin
                              o1 := o3.p ()
                    end
end B

class C
          method p () : Object
                    begin .. end
end C
```

Example 4.9

Redefinition is consistent throughout the class in that there is no possible instantiation of `B` in which one variable is typed as an *Object*, and another is typed as some subtype of *Object*. But redefinition is consistent only within a class, and not throughout a system of classes. `B` and `C` satisfy all type-checks, but any instantiation of `B` besides `B [Object]` is unsafe because of the resulting failure of the binding check over the assignment statement, "o_1 `:= ` o_3`.p()`".

Class substitution uses a deeper form of redefinition than does parametricity. The equivalent PS classes to `B` and `C` above are as shown in **Example 4.10a**.

```
class B
        var o1 : Object
        var o2 : Object
        var o3 : C
        method m
                begin
                        o1 := o3.p ()
                end
end B

class C
        method p () : Object
                begin .. end
end C
```

Example 4.10a

and the subclass, `B [Object ← Int]` is the class `B'`, where `B'` is as defined in **Example 4.10b**.

```
class B'
        var o1 : Int
        var o2 : Int
        var o3 : C'
```

```
          method m
                    begin
                              o2 := o3.p ()
                    end
end B'

class C'
          method p ()  : Int
                    begin .. end
end C'
```

Example 4.10b

Deep redefinition results in the denotation of the implicit subclass of `C`, `C'`, that redefines the only method of `C` to return an *Int* instead of an *Object*.

2. Class Substitution is not just Deep Substitution

Deep substitution alone does not guarantee type-safe subclasses. In the above example, `B` $\lhd$ `B'`, `C` $\lhd$ `C'` and both subclasses retain the type-safety of their superclasses. But if class substitution were just deep substitution, then the class, `B [C` $\leftarrow$ `C']` would be equivalent to the class `B"` of **Example 4.11**.

```
class B"
          var o1 : Object
          var o2 : Object
          var o3 : C'
          method m
                    begin
                              o2 := o3.p ()
                    end
end B"
```

```
class C'
      method p () : Int
            begin .. end
end C'
```

Example 4.11

Note that B" does not safely reuse the code of B, since the assignment of "o$_1$:= o$_3$.p()" now binds an object of type *Int* to a variable of type *Object*. (Remember that binding checks in the PS scheme demand that the types of objects be the same as the types of the variables to which they are bound).

3. What then, is Class Substitution?

In their initial proposal, [PaSc90a] Palsberg and Schwartzbach contend that class substitution is not deep substitution, and that B [C ← C'] from **Example 4.10a** should be an alternative expression for denoting B [Object ← Int]. It can be inferred from [PaSc90a], that class substitution is deep substitution followed by some "cleaning up" where various type references are altered to preserve type-safety. In the above example, the "cleaning up" would demand that the types of the instance variables, o$_1$ and o$_2$ in class B" be redefined to be of type *Int*. This meaning of class substitution was not formalized.

In their later presentation of class substitution [PaSc90d], the algorithm for class substitution is a deep substitution followed by a check of the resulting class for type-safety. If the result is not type-safe, like B" above, then the algorithm reports failure and does not return a result. One effect of this latter resolution is to make unclear how it is that the substitution mechanism can be used safely. In the initial proposal, the result of any substitution, K [M ← N] produces a type-safe subclass of K provided that M $\lhd$ N. It is not made clear what should be the relationship between M and N to guarantee successful results when substitution is a deep substitution followed by a safety check.

The first proposed description of class substitution, though not formalized, is preferable to the second proposal. Not only can the success of the first version be determined in advance, but the second version waters down much of the expressive power of class substitution. Using the first interpretation of class substitution and given the example classes above, B [C ← C'] ≅ B [Object ← Int]. It is not always the case however, that complex substitutions are equivalent to simple ones. The example classes of **Example 4.12a** demonstrate.

```
class Animal
      var mate : Animal
      var owns : Object
      method lends () : Object
            begin
                  return owns
            end
end Animal

class Family
      var leader : Animal
      var has : Object
      method borrow ()
            begin
              has := leader.lends ()
            end
end Family

class Human inherits Animal [Object ← Int]
      method marry (other : Human)
            begin .. end
end Human
```

Example 4.12a

To define the class `Human`, inheritance and class substitution are used in conjunction to denote a subclass of `Animal`. The easiest way to think of the result of this class denotation is to think of the substitution first, and then the inheritance. The class expression, `Animal [Object ← Int]` is a shorthand for the class definition of a class, `Animal'` defined in **Example 4.12b**.

```
class Animal'
        var mate : Animal'
        var owns : Int
        method lends () : Int
                begin
                        return owns
                end
end Animal'.
```

Example 4.12b

As a result of inheriting this class and adding the method `marry`, the class definition for `Human` is equivalent to the class definition of **Example 4.12c**.

```
class Human
        var mate : Human
        var owns : Int
        method lends () : Int
                begin
                        return owns
                end
        method marry (other : Human)
                begin . end
end Human.
```

Example 4.12c

Thus, `Human` is a subclass both of `Animal` and of the denoted class `Animal'`.

A desirable subclass of `Family` might be `Human_Family`, denoted by the expression, `Family [Animal ← Human]`. To ensure type-safe code reuse of the code defined in `Family`, the class `Human_Family` should be defined as shown in **Example 4.12d**.

```
class Human_Family
        var leader : Human
        var has : Int
        method borrow ()
              begin
                has := leader.lends ()
              end
  end Human_Family1
```

Example 4.12d

Assuming the second definition of class substitution found in [PaScd], then `Family [Animal ← Human]` would fail since the class it would return is not type-safe because of the assignment in method `borrow` of an *Int* to an *Object*. This class is shown in **Example 4.12e**.

```
  class Human_Family2
        var leader : Human
        var has : Object
        method borrow ()
              begin
                has := leader.lends ()
              end
  end Human_Family2
```

Example 4.12e

The class denoted by the expression `Human_Family`$_3$ = `Family [Object ← Integer]` (**Example 4.12f**) is type-safe but is not equivalent to the class `Human_Family`$_1$, since the class `Human` is not equivalent to `Animal'`.

```
  class Human_Family3
        var leader : Animal'
        var has : Int
```

```
            method borrow ()
                  begin
                    has := leader.lends ()
                  end
      end Human_Family3
```

Example 4.12f

It will be shown in the next chapter that the class denoted by the deep substitution, `Human [Animal ← Human]` can only be type-safe if `Animal` ◁ `Human` by inheritance (ie: if `Human` can be formed by inheriting `Animal` and then adding any number of variables and methods). An alternative definition of substitution will also be presented that like the original PS proposal in [PaSc90a], accepts any class/type expression, `K [M ← N]` and returns a type-safe subclass of `K`, provided that `M` ◁ `N`. Given this new definition, the class expression `Family [Animal ← Human]` denotes `Human_Family`$_3$, the type-safe subclass of `Family` that was desired.

4.3.4 An Evaluation of PS Subclassing

The weaknesses of PS subclassing concern its incompleteness and the unintuitive nature of its formal definition. Its strengths are the applicability of subclassing mechanisms to all classes, and the flexible resolutions of the tensions of nominal polymorphism and static binding that do not impose weighty restrictions on class composition and subclassing redefinition. The strengths outweigh the weaknesses because most if not all of the weaknesses can be remedied with an alternative class representation to L-Trees. On the other hand, the irresolution of these tensions in languages with parametric classes, and the restrictive resolutions of these tensions in the Bruce approach, appear to be inherent to those approaches.

Strengths of the PS Approach: A Comparison with Languages Supporting Parametric Classes

1. Guaranteed Type-Safe Nominal Polymorphism and Static Binding

Perhaps the most important achievement of PS subclassing is its guarantee of type-safe and covariant signature redefinition. Class substitution does not provide as flexible signature redefinition as parametric classes, but it can guarantee the type-safety of mappings of new signatures to existing code.

Signature redefinition resulting from the instantiation of parametric classes is invariably covariant in direction while selective in scope. It is easy to see the appeal of selective redefinition. Selective redefinition offers flexibility to the class designer, allowing him/her to insist on the constancy of some type references in instantiated classes. That class substitution demands consistent redefinition means that the class designer cannot control as readily, the potential subclasses of the designed class. That class substitution demands that redefinition be consistent throughout a system of classes means that the system and not the individual class becomes the module for code reuse. Parametric classes offer more flexible forms of genericity, but as was shown in Chapter 2, the tension of covariant and selective redefinition makes it impossible to statically ensure the type-safety of instantiated classes.

2 A More General Approach to Subclassing

Another advantage of the PS approach over languages with parametric classes is that potentially *all* PS classes are superclasses that can be redefined in subclasses to realize nominal polymorphism and static binding. Conversely, languages that support parametric classes have three categories of class that differ by how they can be used: explicitly defined classes, parametric classes and instantiated parametric classes.

Explicitly defined classes are classes that are defined without the use of type variables. They can be inherited from, and their names can be used to denote variable types, and type parameter constraints. They cannot be used as templates for generic subclassing.

Parametric classes have definitions that include type parameters. They can be used as templates for generic static binding. Their names cannot be used to denote types nor type parameter constraints. Arguably, they can be used as superclasses to parametric subclasses, though this brings up the issue of how parameters must be defined in these specialized subclasses. It might be desirable for example, for a subclass to have a different number of parameters than its superclass. The parametric class `List` (of **Example 4.13**) might have a subclass `Indexed_List` with an extra parameter constrained to be a subclass of `Order`, the class of objects belonging to a partially ordered set.

```
class List [T : Contains]
 ...
end List

class Indexed_List [index : Order, T : Contains] inherits List
  ...
end List
```

Example 4.13

Whereas `List` is parameterized by the type of the object contained in the list `Indexed_List` would also be parameterized by the type of the index used to sort the *List.* A language that allows a parametric class to have subclasses must provide a means of matching up the type parameters of the subclass with those of the superclass. The issue becomes more complicated when multiple inheritance is introduced since it is possible that the same name might be used for a parameter in more than one parent class.

The third category of class is the instantiated class. An instantiated class is similar to an explicitly defined class in that it can be used as a basis for inheritance (eg: class `Alphabet` inherits `Set [Char]`), and its name can be used to denote a type in a variable declaration. Its use as a constraint for type parameters causes the same problems that were shown of class substitution when equated with deep substitution. To demonstrate this, a parametric version of an earlier example is shown in **Example 4.14**.

```
class C [T : Object]
      method p () . T
            begin .. end
end C'

 class B [T : C [Object]]
      var o· : Object
      var o, : Object
      var o; : T
```

```
    method m
        begin
            o2 := o3.p ()
        end
end B
```

Example 4.14

In this example, the class `B` is parameterized by the type parameter, `T` constrained to be a subclass of the instantiated class `C [Object]`. Any instantiation of `B` (save for `B [C [Object]]`) will not be type-safe despite the fact that the parametric class itself is type-safe. Symptoms of this problem are similar to those discussed earlier, of redefinition that is not consistent over a system of classes. It should be noted however, that deep and consistent substitution rules do not remedy the insecurity, just as they did not in the earlier example that showed that class substitution is not just deep substitution. The problem has no easy solution except for the restriction that instantiated parametric classes not be used as type parameter constraints, thus making this form of class fall in a different category than explicitly defined classes.

Class substitution is a more general mechanism than parametricity, allowing all classes to be subclassed via inheritance and genericity, allowing all type references to be replaced, and allowing the class name to be used as a type reference and to constrain how a type reference is replaced in subclasses. Parametric classes result in three categories of class: the explicitly defined class, the parametric class generators, and instantiated parametric classes. Each category defines a class that can be used in some but not all of the ways that a PS class can. Language implementations which support parametric classes such as Eiffel and Dee allow various categories of class to be used in contexts for which they are unsafe, and are thus subject to type insecurities. Even when restrictions are added to govern the use of each category of class, type-safety cannot be guaranteed of instantiated classes because of the selective nature of parametricity.

Strengths of the PS Approach: A Comparison with the Bruce Approach

1. Flexible Resolution of Subclassing Tensions

The flexibilities of the PS approach to subclassing are apparent when compared to the Bruce approach. Both approaches resolve the tensions of nominal polymorphism and static binding, but limitations on class composition and redefinition in subclasses are less restrictive in the PS system. Unlike Bruce classes, PS classes can include recursive instance variables, and the types of instance variables can be redefined in subclasses. Whereas Bruce subclassing does not support genericity, PS subclassing supports all forms of nominal polymorphism and static binding. Whereas Bruce subclassing demands that signature redefinition be contravariant, the redefinition of PS subclassing is covariant. Whereas Bruce classes cannot include bindings of recursively typed objects to non-recursive variables, PS classes have no problem with such bindings.

The inflexibilities of the PS approach in comparison to the Bruce approach are its demand for consistent signature redefinition, and its incompatibility with dynamic binding. Consistent signature redefinition is necessary because subclassing redefinition is covariant. Bruce subclassing suffers from contravariant signature redefinition and a lack of support for genericity to achieve selective redefinition. The incompatibility of PS subclassing with dynamic binding is the subject of current study, described in Chapter 6. **Table 4.2** summarizes the features of the PS and Bruce approaches. The last column of the table shows the features supported by the extended version of PS subclassing, presented in the next two chapters.

Feature	Bruce	PS	Extended PS
Polymorphisms			
Nominal Polymorphism			
Code Redefinition	√	√	√
Code and Signature Redefinition	contravariant and selective	covariant and consistent	covariant and consistent
Static Binding			
Prefixing	√	√	√
Genericity	no	covariant and selective	covariant and selective
Dynamic Binding	√	no	√[6]
Tension Resolution			
Covariance + Selective Redefinition	contravariant direction	consistent redefinition	consistent redefinition
Contravariance + Genericity	no genericity	covariant selective redefinition	covariant selective redefinition
Subclass-Based Binding + Covariance	subclass-independent binding	no dynamic binding	subclass-independent binding[6]
Language Features			
Constants	√	no	√[5]
Recursive Instance Variables	no	√	√
Temporal Cycles	no	no	√[5]
Multiple Inheritance	no	no	√[5]

[5] - Added Feature Described in Chapter 5
[6] - Proposed Feature Described in Chapter 6

Table 4.2

Weaknesses of the PS Approach

The weaknesses of the PS approach concern its complexity and incompleteness. Definitions of subclassing, inheritance and class substitution are complex and unintuitive. PS subclassing is incomplete because multiple subclassing and multiple inheritance are not supported and because the tension associated with dynamic binding is not resolved. The lack of incompatibility with dynamic binding is the most troublesome weakness, and issues regarding its resolution are presented in Chapter 6. All other weaknesses concern PS subclassing as applied to classes with singleton types. These weaknesses are identified here and remedied in Chapter 5.

1. No Multiple Subclassing and No Multiple Inheritance

Like the Bruce approach, no PS class can have more than one superclass (except for ancestral superclasses). The lack of support for multiple subclassing is a result of the use of L-Tree representations of classes as a basis for determining structural subclassing relations. L-Trees provide a name-free, structural class representation. However, the composition of an L-Tree is dependent on the order in which variables and methods appear in a class. As an example, L-Trees for the classes defined in **Example 4.15a** would be considered distinct.

```
class A
      var o : Object
      var i : Int
end A

class B
      var i : Int
      var o : Object
end B
```

Example 4.15a

Because subclassing is defined over L-Trees, it too is sensitive to the order in which variables and methods are declared and defined. Therefore, given the additional classes C and D in **Example 4.15b**, class A would only be a subclass of C, and class B would only be a subclass of D.

```
class C
        var o : Object
end C

class D
        var i : Int
end D
```

Example 4.15b

In fact, classes A and B should be considered equivalent and should be subclasses of both C and D. Multiple subclassing cannot be supported when L-Trees represent classes because the interfaces and code of a superclass must form a prefix for the interfaces and code for the subclass. For no class can multiple numbers of classes with distinct method and variable definitions satisfy this property.

Lack of support for multiple inheritance is related, not only to the lack of support for multiple subclassing, but to limitations of the unfolding algorithm. Input classes that inherit from more than one class can force the algorithm into deadlock.

2. No Temporal Cycles

We have seen already that temporal cycles cannot be handled by the PS unfolding algorithm and are therefore prohibited. This restriction not only rules out useful sets of classes, but also makes class validity an unintuitive concept; dependent on more criteria than just type-safety. It should be pointed out here that this weakness is a weakness of the Bruce approach also. The Bruce paper does not discuss temporal cycles, but does exclude recursive instance variables and does demand that all recursive type references found in method signatures be made with the type name, *MT*. Given these restrictions, it can be inferred that Bruce subclassing forbids temporal cycles. Suppose for example, that the Voter and Candidate examples of **Example 4.5** are programmed in a Bruce system. The class Voter has an instance variable, "backs" of type *Candidate*. Since Candidate is a subclass of Voter, it is constrained to keep the types of its inherited instance variables constant. Thus, class Candidate violates Bruce class composition rules in two ways: it has a recursive variable which is not declared to have type *MT*, and the recursive variable is an instance variable.

3. No Constants

Whereas the restriction forbidding temporal cycles is discussed in [PaSc90d], problems with constants are not discussed. The class of **Example 4.16** demonstrates that PS classes with constants also cannot be guaranteed to have type-safe subclasses.

```
class example
      var i : Int
      method m
            begin
                  i := 45
            end
end example
```

Example 4.16

Consider that one possible subclass of `Int` might be `MODULO_10`; the class of *Integers* between 0 and 9 inclusive. Because of the use of the constant 45, the method `m` of class `example` would not be safely prefixed in the subclass, `example [Int ← MODULO_10]`. This is because the statement of method `m` would, in the subclass, make the meaningless assignment of 45 to a *MODULO_10* object.

Classes whose instances are constant values (such as `Int`, `Bool` and `Char`) are herein called *base classes*. While Palsberg and Schwartzbach do not discuss the problems concerning base classes and type-safety, they do argue in [PaSc90d] that `NIL` should be the only constant in a PS-based language. In their approach, `NIL` is an instance of no class and therefore base classes are thought no longer to exist. Classes such as `Int`, `Bool` and `Char` are defined in the language like any other class. For example, [PaSc90a] contains a class definition for `Int` as shown in **Example 4.17**.

```
class Int
      var value : Object
      method plus (other : Int)
            begin ... end
      method times (other : Int)
            begin ... end
      method zero : Int
            begin ... end
end Int
```

Example 4.17

Class definitions such as the one above, do not eliminate constant expressions and base classes. The constant, 45 is an *Object* in this interpretation, and can be bound to the instance variable, `value`. `Object` is thus a base class, and subject to the same problems that beset all base classes. Any class that includes code that binds a constant to the instance variable, `value` will have unsafe subclasses when some other class is substituted for `Object`.

4.4 Dynamic Binding in the PS Approach

An interpretation of types as non-singleton sets can be found in [PaSc90d]. Discussion of this proposal has been separated from the discussion of PS subclassing because of the incompatibility of the two approaches.

4.4.1 The PS Interpretation of Types

Palsberg and Schwartzbach propose that a type can be any set of classes (singleton or non-singleton), and that subtyping is captured by set inclusion. Because types are sets, the universe of all types can be represented as a lattice. Since theirs is a single inheritance system, the class hierarchy is a tree. **Figure 4.6** shows portions of the universal class hierarchy and type lattice corresponding to class definitions for `Object`, `Vehicle`, and two `Vehicle` subclasses: `Bus` and `Car`. Note that the notation, $\uparrow$`A` means the set of all subclasses of `A`. Note also that $\uparrow$`Object` (the set of all classes) and $\emptyset$ (the set of no classes) serve as top and bottom not only of the partial lattice, but over the entire universal type lattice.

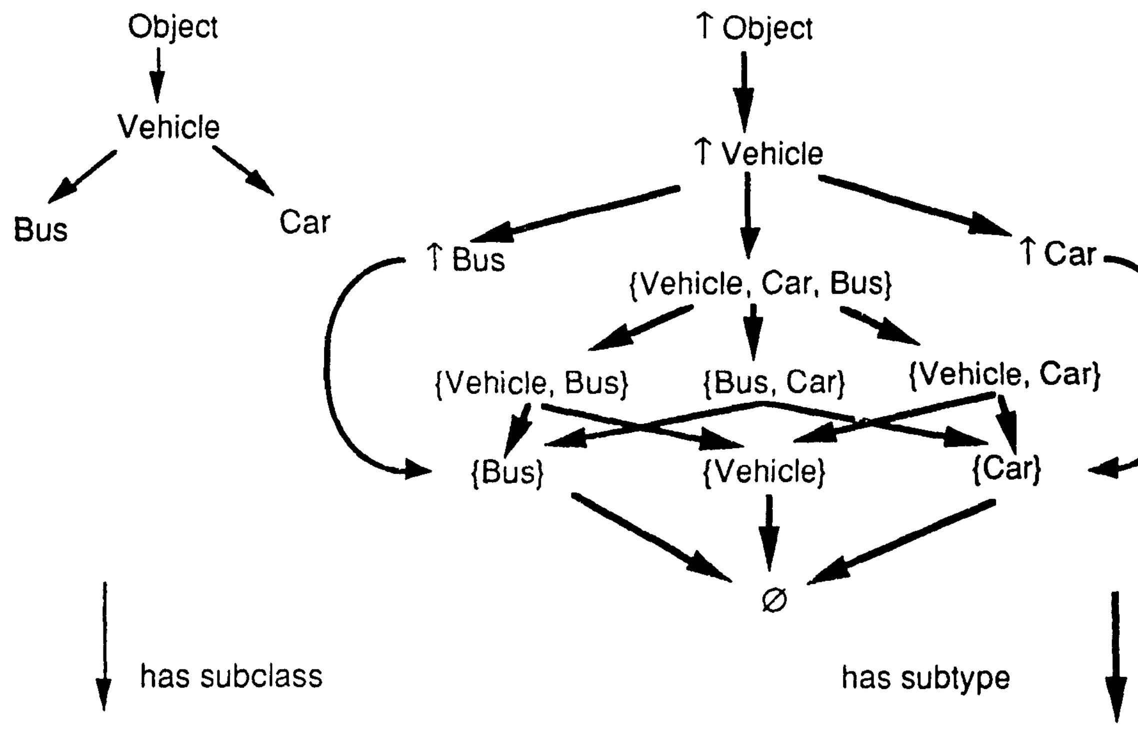


Figure 4.6

In practice, the PS interpretation of types is subclass-based. The PS interpretation is more general in theory than the interpretations of types that associate classes and their subclasses with types and their subtypes. A PS type can be any set of classes and not just a set of subclasses of some given class. But only sets of related classes form useful types, since only for these types can signature checks be confined to a single class. Since useful non-singleton PS types are sets of subclasses of common ancestry, the binding of an object to a heterogeneous variable is allowed if the class denoting the object's type is a subclass of the class denoting the variable's type. Thus, binding checks are subclass-based and the PS type interpretation results in equivalent type-checks in practice to those of traditional subclass-based interpretations.

4.4.2 Evaluating the PS Approach to Types

An Intuitive and Practical Model for Subclass-Based Type Interpretations

One advantage of the PS approach over the traditional subclass-based approaches is in its intuitive definitions of types and subtypes. The PS approach uses subclass-based binding without equating subclasses with subtypes and thus without having to incorporate the complications and ill-effects of the F-Bounded model. (As was shown in Chapter 3, recursive instance variables cannot be supported when the F-Bounded model is used to model languages where objects have state).

Set inclusion captures what is desired in subtyping. By definition, an object of a subtype can be substituted for an object of its supertype, and should therefore understand all messages presumed understood of objects of the supertype. The intersection of class interfaces included in a type define the messages that can be presumed understood by objects of that type. Thus, the common interface of any subset of classes (subtype) will be at least as large as the common interface of the superset (supertype). As is desired of subtyping, the subset ordering defines an implicit reverse ordering of understood messages.

NIL: The Value of All Types

One of the arguments that Palsberg and Schwartzbach contend to justify their type proposal, is that it provides a home for the constant value, `NIL`. Typically, `NIL` is used in object-oriented languages as a means of introducing state to object creation. In particular, cyclic structures typically are created with `NIL` used as a temporary value since `NIL` can be assigned to variables of any type. Suppose for example that classes `A` and `B` are clients of one another where `A` has an instance variable declared as a *B*, and `B` has an instance variable declared as an *A*. Instances of `A` can only be created once there exists an instance of `B` and vice-versa. In a language with `NIL`, one would create an instance of `A` by using `NIL` as a temporary value of its instance variable of type *B*, and then replacing that value with an instance of `B`, once the instance of `B` had been created.

In most object-oriented languages, `NIL` enjoys a special status. It can be bound to any variable though does not respond to any messages. Palsberg and Schwartzbach argue that `NIL`'s role can be justified as a member of no class, and with empty type. Because `NIL` belongs to no class, it cannot respond to any messages, and it has the set of no classes

(∅) as its only type. That the empty type is a subtype of all other types justifies the binding of NIL to any variable.

Unfortunately, the use of NIL as an expression that can be assigned to any variable but incapable of understanding any message is inherently contradictory, and demands that static guarantees of type-safety be sacrificed. The example core Mini-Dee class of **Example 4.18** shows that allowing NIL to be bound to a variable means that NIL should be able to respond to the messages that can be sent to that variable.

```
class example
      var a : example
      method m1
            begin  ... end
      method m2
            begin
                  a := NIL
                  a.m
            end
end example
```

Example 4.18

By the reasoning of Palsberg and Schwartzbach, the above class satisfies binding and signature checks. However, if NIL can respond to no messages, this class will fail when the method, m_2 is executed.

If strong and static typing is to be achieved, then an expression that can be bound to variables of all types must be able to respond to any message. This point follows from the desire that members of subtypes be able to respond at least to all messages to which members of the supertype can respond. If NIL can be bound to any variable, then its type must be a subtype of all types. Therefore, to maintain a consistent type system NIL must be able to respond to any message, even if the response is that the message is not understood. It would appear that strongly typed object-oriented languages cannot completely avoid the "message not understood" runtime errors of weakly typed object-oriented languages such as Smalltalk [GoRo83]. Fortunately, such errors in strongly typed object-oriented languages would only occur when messages are sent to objects as yet unconstructed.

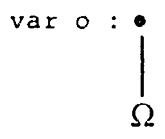
The Incompatibility of PS Types with PS Subclassing

Palsberg and Schwartzbach note in [PaSc90d] that their inheritance and class substitution definitions are incompatible with classes whose types are non-singleton. Their claim is that inheritance and class substitution do not denote type-safe subclasses of classes whose signatures contain types that are infinite sets of classes.

In fact, subclassing, inheritance and class substitution are not well defined in terms of classes with even finite, non-singleton types in their signatures. The problem mostly lies with the L-Tree representation of classes. L-Trees themselves are only well-defined over classes with signatures with singleton types. For example, while the L-Tree representation for the class `A` of **Example 4.19a** is easily determined, it is not clear what would be the representation tree for the non-singleton class, `B`, of **Example 4.19b**, even though the type of the instance variable `x` is a finite set of classes.

```
class A
      var o : Object
end A
```

L-Tree Representation for A:



Example 4.19a

```
class B
      var x : {Object, Int}
end B
```

Example 4.19b

Since subclassing is defined as a relation on L-Trees, it too is only well-defined over classes with signatures with singleton types . Every example subclass presented in [PaSc90a, PaSc90b, PaSc90d] is a subclass of such a class. Inheritance is not defined at all over classes with non-singleton types in their signatures, and it is not clear how the inheritance algorithm of [PaSc90d] could be adapted to apply to these classes. As well, contradictory assumptions are made in [PaSc90d] concerning how class substitution applies to these classes. (see Chapter 6)

Inheritance and class substitution can be defined so as to resolve the tensions of nominal polymorphism and static binding, even in denoting subclasses of classes with non-singleton types in their signatures. A presentation of an alternative representation of classes to L-Trees, is found in Chapter 5. The alternative representation:

- allows for intuitive definitions of subclassing, inheritance and class substitution,
- supports multiple subclassing and multiple inheritance,
- guarantees type-safe nominal polymorphism and static binding of classes whose code includes constant expressions, and
- allows most temporal cycles as input.

Chapter 6 contains a description of preliminary work done to incorporate PS subclassing and subclassing mechanisms in classes with non-singleton types in their signatures. It will be shown in this chapter that the incompatibility of PS subclassing with PS type interpretations concerns the tension of covariant subclassing redefinition with subclass-based interpretations of types, and not with the inapplicability of subclassing definitions to classes containing signatures with non-singleton types.

Chapter 5.

Design: An Alternative Interpretation and Extension of PS Subclassing

5.1 Introduction

PS theory provides a number of useful results but has significant practical limitations. L-Trees, the tree-based representations of class code, are free of class names but are not finite for all classes. As the basis for definitions of subclassing and subclassing mechanisms, L-Trees provide little intuitive insight. The lack of support for temporal cycles rules out useful sets of classes and makes class validity dependent on other criteria besides type-safety. The PS approach is also incompatible with such commonly used object-oriented language features as multiple inheritance, constants, and dynamic binding.

The purpose of my work thus far has been to interpret and to provide an intuitive understanding of PS Subclassing, and to address the shortcomings described above. An alternative representation of classes is presented which provides a finite representation for all classes and which is based on the interface rather than the code of a class. Type-safety is formally defined as a property of code with respect to a system of class interfaces. New definitions for subclassing and subclassing mechanisms follow. The resulting system is compared with the PS system and shown to be more flexible, allowing for constants, most temporal cycles, and supporting the use of multiple inheritance. (The incompatibility of PS subclassing with dynamic binding is addressed in the next chapter). In the end, the original proposals — novel though incomplete and somewhat unintuitive, have been reinterpreted and extended.

5.2 An Extension of Mini-Dee

The proposal presented here is described in terms of an extension to Mini-Dee that is identical to the PS-based extension described in 4.2, except that multiple inheritance is supported. The grammar for this extension can be found in **Appendix I-D**. Since the work described in this chapter does not address the incompatibility of PS subclassing with dynamic binding, types are still singleton and binding and parameter binding checks still demand that types of objects and variables to which objects are bound be identical. Signature checks are still confined to the class denoting the type of the receiver object. Signature redefinition in subclasses is still covariant and consistent though the operations which achieve consistent redefinition, inheritance and class substitution are defined according to an alternative representation of classes described in Section 5.3.

5.3 The Node Representation of Classes

The *"node representation"* of classes is presented here as an alternative representation to Palsberg and Schwartzbach's L-Trees. This representation will be used as the basis for alternative definitions of PS subclassing, inheritance and class substitution.

The representation is based on class interfaces rather than code. All instance variable, local variable and method signatures are included in a class' representation. Informally, a node represents a signature and contains all relevant information such as the kind of signature it represents, the name of the entity declared, its return type and so on. Formally, the node type is defined as the type of sextuples of the form (*kind, id, type, context, parms, locals*) where:

kind: {m,v} —
  is m when the signature is that of a method, and v when it is a variable

id : String —
  is the name of the method or variable

type : Class Identifier* —
  is the declared type of the variable, or the return type of the method†

context : Class Identifier —
  is the class in which the signature is declared

parms : Sequence of Node —
  is the sequence of nodes representing the parameters of the method (or ∅ for variables)

locals : Set of Node —
  is the set of nodes associated representing the local variables of the method (or ∅ for variables)

For any class C, the value of *Rep* (C) is the set of nodes obtained by parsing the interface of C.

* For the time being, it is useful to think of a Class Identifier as a string corresponding to the name of a class. Once inheritance and class substitution are reintroduced, Class Identifiers may refer to unnamed classes. Therefore, the representation is kept abstract.

† Following the convention employed by Palsberg and Schwartzbach [PaSc90d], and Cardelli and Wegner [CaWe85], methods which do not explicitly return a result will be represented as methods which return `self`.

5.3.1 Some Example Nodes

As an example, let C be a Mini-Dee class whose definition contains the example declarations of **Example 5.1**.

```
(a):  var q : R                        -- instance variable

(b):  method r (p1 : S1,   , .. : Sp)  -- method with parms, no return type, no
                                          locals

(c):  method s : T0                    -- method with no parms, return type, or
       var                                locals
           l1 : T1

           lk : Tk
```

Example 5.1

The nodes representing these three signatures are:

(a): (v, q, R, C, ∅, ∅)

(b): (m, r, C, C,
  « (v, self, C, C, ∅, ∅), (v, p1, S1, C, ∅, ∅), ..., (v, pm, Sm, C, ∅, ∅) »,
  ∅)

(c): (m, s, T0, C,
  « (v, self, C, C, ∅, ∅) »,
  { (v, l1, T1, C, ∅, ∅), ..., (v, lk, Tk, C, ∅, ∅) })

Note that the implicit method parameter, `self` is represented as a parameter of every method node.

5.3.2 An Example Class and its Node Representation

An entire Mini-Dee class interface with variable and method declarations, is given in **Example 5.2**.

```
class A
        var a : A
        method m (o : Object)
          var a : A
          begin  ... end
end A
```

Example 5.2

Given the class A in **Example 5.2**, the value of *Rep* (A) is:

```
{ (v, a, A, A, ∅, ∅),
 (m, m, A, A,
        «(v, self, A, A, ∅, ∅), (v, o, Object, A, ∅, ∅)»,
        {(v, a, A, A, ∅, ∅)})
}
```

Unlike L-Trees which provide an entirely structural representation of classes, the node representation is dependent on class names and individual nodes can be likened to symbol table entries. This approach allows all classes to have finite representation, including clients of recursive classes. As well, whether or not one class is a subclass of another, is dependent on the sets of signatures of each class and independent of the order in which these signatures appear. While the PS representation is free of class names, it does not capture the structural nature of subclassing that establishes the order of appearance of methods and variables to be irrelevant to determining subclass relationships. The node representation, preferable in this regard, is able to capture multiple subclassing relationships and supports the use of multiple inheritance.

5.4 Type Safety Defined Over Node Representations

The class is a module for code but does not define a program. Since a class' code can contain invocations of methods and references to variables found in other classes in its system, a system of classes comprises a program. Type-safety can be formally defined as a property of systems of Mini-Dee classes. A formal definition is given by way of an attribute grammar of extended Mini-Dee, found in **Appendix II.** A description of this formalization follows.

5.4.1 Formal Definitions of Supplier and System in terms of Nodes

The definition of a system of classes is dependent on a definition of the supplier relation between classes. Both definitions are given below. It is assumed beforehand, that there exists a function, *type_of*, that accepts a node argument and returns the type of the variable or return type of the method that the node represents. Thus:

$$type\_of(k, i, t, c, p, l) = t.$$

The Supplier Relation

The *supplier* relation (Class Identifier × Class Identifier) holds for classes B and A when *B* is used in a type declaration for an instance variable, method parameter, method result or local variable defined in A. In other words, *supplier* (B,A) holds when B is a supplier of A. The relation is defined formally in terms of node representations of classes:

$$\begin{aligned}
&supplier\,(\mathrm{B},\mathrm{A}) \Leftrightarrow \\
&\quad \exists\, w, x, y, p, l \bullet \\
&\qquad (w, x, y, \mathrm{A}, p, l\,) \in Rep\,(\mathrm{A}) \;\wedge \\
&\qquad [\mathrm{y} = \mathrm{B} \qquad \vee \\
&\qquad (\exists\, i \bullet 1 \le i \le |p| \bullet type\_of\,(p\,(i)) = \mathrm{B}) \qquad \vee \\
&\qquad (\exists\, n \in l \bullet type\_of\,(n) = \mathrm{B})]
\end{aligned}$$

Since a node representing `self` is found in every method representation, *supplier* (A,A) always holds when A includes a method definition.

The System Function

Intuitively, a system of classes is the set of classes that are direct or indirect suppliers of some given class. Formally, *system* is defined as a function (Class Identifier $\rightarrow 2^{\text{Class Identifier}}$), where given some class identifier, A:

$$system\,(\mathrm{A}) = \{\, x \mid supplier\,(x\,, \mathrm{A}) \;\vee\; (\exists y \bullet supplier\,(y\,,\mathrm{A}) \wedge x \in system\,(y)) \,\}$$

Again, because a node representing `self` is found in every method representation, A $\in$ *system*(A) when A includes a method definition.

5.4.2 The Attribute Grammar of Extended Mini-Dee

Type-safety is formally defined in **Appendix II** with respect to the extension of Mini-Dee described in 5.2, but without inheritance and substitution. It might be remembered that the grammar of Mini-Dee separates class interfaces from code. This allows the parse of a program to be divided into two distinct components: environment generation and type-checking. It also allows type-safety to be defined as a property held by class code with respect to the interfaces from a system of classes. Type-safety is formally defined as an attribute value synthesized from the parse of the attribute grammar of Mini-Dee. It is dependent on binding, parameter binding and signature checks, for which formal definitions are also given in **Appendix II.**

5.5 Subclassing Defined in terms of Nodes

In this section, structural subclassing relations between systems of classes will be defined in terms of nodes. A definition of subclassing requires a definition for *mapped systems*. Informally, one system, *system* (B) is a mapped system of another, *system* (A), if the code of *system* (A) can be safely reused with the interfaces of *system* (B). In terms of nodes, *system* (B) is a mapped system of *system* (A) if there exists a "mapping function", *f*, which maps all class references found in the classes of *system* (A) to corresponding references found in the corresponding classes of *system* (B). Note that the correspondence is applied in two ways. Given a single signature (node) with type *t* and context *c*, the same signature with type *f(t)* must be found in the representation of *f(c)*. That the corresponding signature has type *f(t)* guarantees binding and parameter binding checks in the mapped system. That the corresponding signature has context *f(c)* guarantees signature checks.

The Map Function

A formal definition of the *mapped system* relation depends on the definition of the higher-order function, *map*: (Class Identifier $\rightarrow$ Class Identifier) $\times$ Node $\rightarrow$ Node; a function which applies a mapping function, f to a node to produce a new node with each type reference t, replaced by $f(t)$. More formally, given the mapping function, f and node, $n = (k, i, t, c, «p_1, \ldots, p_m», \{l_1, \ldots, l_j\})$

$$\begin{aligned} map\ (f, n) = (k, i, f(t), f(c),\\ «map\ (f, p_1), \ldots, map\ (f, p_m)»,\\ \{map\ (f, l_1), \ldots, map\ (f, l_j)\}) \end{aligned}$$

Map is guaranteed to return a finite node despite the recursive nature of its definition since all nodes contained in the sequence, $«p_1, \ldots, p_m»$ and the set, $\{l_1, \ldots, l_j\}$ have empty parameter sequences and empty local variable sets. *Map* is extended to sets and sequences of nodes so that:

$map\ (f, \{l_1, \ldots, l_j\}) = \{map\ (f, l_1), \ldots, map\ (f, l_j)\}$ and

$map\ (f, «p_1, \ldots, p_m») = «map\ (f, p_1), \ldots, map\ (f, p_m)»$

The Mapped System Function

Given systems of classes S_1 and S_2, *mapped system*: $2^{\text{Class Identifier}} \times 2^{\text{Class Identifier}}$ is formally defined as the following relation:

$$mapped\ system\ (S_1, S_2) \Leftrightarrow \exists f : S_1 \rightarrow S_2 \bullet map\ (f, Rep\ (S_1)) \subseteq Rep\ (S_2)$$

5.5.1 Examples of Mapped Systems

An example system of classes, *system* (A) is defined in **Example 5.3**. Potential mapped systems of *system* (A) are then considered.

Let *system* (A) = {A, D} where A and D are defined as in **Example 5.3**.

```
class A
   var a : A
   method m (other : A) : D
      begin ... end
end A

class D
   var d : D
end D
```

Example 5.3

From the above declarations, we have:

Rep (*system* (A)) =

{ (v, a, A, A, «», ∅),
(m, m, D, A, « (v, self, A, A, «», ∅), (v, other, A, A, «», ∅) », ∅),

(v, d, D, D, «», ∅) }

Now consider the following potentially mapped systems of *system* (A):

Example 1: Recursive References Must be Retained in Subclasses

Let *system* (B_1) = {A, D, B_1} where B_1 is as defined in **Example 5.4** and classes A and D are as defined in **Example 5.3**.

```
class B₁
   var a : A
   method m (other : A) : D
      begin ... end
   var b : B₁
end B₁
```

Example 5.4

From the declarations of **Example 5.4**, we have:

Rep (*system* (B_1)) =

{ (v, a, A, B_1, «», ∅),

(m, m, D, B_1, « (v, self, B_1, B_1, «», ∅), (v, other, A, B_1, «», ∅) », ∅)

(v, b, B_1, B_1, «», ∅),

(v, d, D, D, «», ∅) }

Note that *system* (B_1) is not a mapped system of *system* (A), despite the fact that the interface of A is a prefix of the interface of B_1. An examination of the variable ***a*** alone shows that both A and B_1 would have to be images of A in any mapping function. Thus, it cannot be guaranteed that code which is type-safe with respect to *system* (A) will be type-safe with respect to *system* (B_1). The code for method m might for example, contain the assignment:

```
    a := self
```

which would be safe only in the context of the interfaces of *system* (A).

Example 2: A Minimal Recursive Mapped System

Let *system* (B_2) = {B_2, D} where B_2 is as defined in **Example 5.5** and and class D as defined in **Example 5.3**.

```
class B2
    var a : B2
    method m (other : B2) : D
        begin .. end
    var b : B2
end B2
```

Example 5.5

From the declarations of **Example 5.5**, we have:

Rep (*system* (B_2)) =

{ (v, a, B_2, B_2, «», ∅),
(m, [illegible], [illegible], B_2, « (v, [illegible], B_2, [illegible], «», ∅), (v, other, B_2, B_2, «», ∅) », ∅)
(v, b, B_2, B_2, «», ∅),

(v, a, D, D, «», ∅) }

System (B$_2$) is a mapped system of *system* (A) by the mapping function:

{ (A, B$_2$), (D,D) }.

Applied to the type field of a node, the first pair in the mapping assures that recursive references are retained (all type references to *A* in *system* (A) become type references to *B*). Applied to the context field of a node, the first pair in the mapping assures that *system* (B$_2$) defines all of the same methods and variables defined in *system* (A). The second pair of the mapping establishes that references to the type *D* in *system* (A) are maintained in *system* (B$_2$).

Example 3: All Mutual Client Relations must be Retained

Suppose that *system* (A) is redefined so that class D now is defined as in **Example 5.6**.

```
class D
   var d : A
end D
```

Example 5.6

and *Rep* (D) = { (v, [illegible], [illegible], [illegible], «», ∅) }

Now consider *system* (B$_2$) once more. Even though class B$_2$ is the same as class A with the recursive references retained, it is not the case that *system* (B$_2$) is a mapped system of *system* (A), since A must be replaced by B$_2$ in several nodes, but left unchanged in the node for the variable, d.

5.5.2 Mapped Systems Preserve Type-Safety of Reused Code

The proof of **Appendix III** shows that code that is type-safe with respect to the interfaces of any system of classes, *system*(A) will be type-safe with respect to the interfaces of any mapped system of *system* (A). The environment generated from the parse of some system, *system* (B) (which say is a mapped system of *system* (A) by some mapping function, *f*) is a superset of *map* (*f*, *Rep* (*system* (A)). A case analysis examining the attribute grammar of Mini-Dee shows that any statement which is type-safe with respect to *Rep* (*system* (A)), will also be type-safe with respect to *map* (*f*, *Rep* (*system* (A)) and hence to *Rep* (*system* (B)). This is b:cause binding, parameter binding and signature checks are still satisfied with respect to the mapped system. Binding and parameter binding checks are still satisfied because type references are replaced consistently throughout the mapped system, and therefore equal types remain equal. Signature checks are still satisfied because the class denoting the type of every expression in the mapped system defines at least the protocol of the class denoting the type that it replaced.

5.5.3 Subclassing Defined in Terms of Mapped Systems

Given the definitions for systems and mapped systems, it is possible to define subclassing relations and various functions over class identifiers. The functions are later used to formally define inheritance and class substitution.

• *superclass* ($\lhd$ — Class Identifier × Class Identifier)

Given classes A and B, A $\lhd$ B if the code of A can be safely reused in the context of class B. With respect to systems of classes, A is a superclass of B if the code of A is type-safe with respect to both the interface of *system* (A), and the interface of *system* (B).

A $\lhd$ B $\Leftrightarrow$ *mapped system* (*system* (A), *system* (B))

If A $\lhd$ B, there is a unique mapping function, *f* that maps *system* (A) to *system* (B). "A $\lhd$ B by *f*" is written when there is a need to mention this function explicitly.

• *equivalence* ($\cong$ — Class Identifier × Class Identifier)

Given classes A and B, A $\cong$ B if the code of A can be safely reused in the context of B and vice-versa. More formally:

$$A \cong B \Leftrightarrow A \lhd B \wedge B \lhd A$$

• *deep substitution*

$(\leftarrow_{\mathrm{Deep}}$ — (Class Identifier × Class Identifier × Class Identifier) → Class Identifier)

Given classes A, B and C, A [B $\leftarrow_{\mathrm{Deep}}$ C] denotes a system of classes which resembles *system* (A) except that all type references denoted by B and found in *system* (A) are replaced by references to C. More formally:

```
for any classes A, B and C, A [B $\leftarrow_{\mathrm{Deep}}$ C] =
    if A $\equiv$ B
        then C
    else if B $\equiv$ C or B $\notin$ system (A)
        then A
    else
        A ' where Rep (system (A ')) =
            { (k, i, t [B $\leftarrow_{\mathrm{Deep}}$ C], c [B $\leftarrow_{\mathrm{Deep}}$ C], p', l') |
                • (k, i, t , c, p, l) $\in$ Rep (system (A))
                • p' = { (x, (k', i', t' [B $\leftarrow_{\mathrm{Deep}}$ C], c' [B $\leftarrow_{\mathrm{Deep}}$ C], «», $\emptyset$)) |
                        p (x) = (k', i', t', c', «», $\emptyset$)) }
                • l' = { (k', i', t' [B $\leftarrow_{\mathrm{Deep}}$ C], c' [B $\leftarrow_{\mathrm{Deep}}$ C], «», $\emptyset$) |
                        (k', i', t', c', «», $\emptyset$) $\in$ l }
            }
```

$(\leftarrow_{\mathrm{Deep}}$ — (Node × Class Identifier × Class Identifier) → Node)

The deep substitution operation is extended over nodes such that:

for any node, (k, i, t, c, p, l) , and any classes, B and C, (k, i, t, c, p, l) [B $\leftarrow_{\mathrm{Deep}}$ C] =

(k, i, t [B $\leftarrow_{\mathrm{Deep}}$ C], c [B $\leftarrow_{\mathrm{Deep}}$ C], p', l') where

• p' = {(x, p (x) [B $\leftarrow_{\mathrm{Deep}}$ C]) | $1 \leq x \leq |p|$} and

• l' = {n [B $\leftarrow_{\mathrm{Deep}}$ C] | n $\in$ l}

Given the definition of $\leftarrow_{\text{Deep}}$ over nodes, the version defined over Class Identifiers can be rewritten in a more readable form:

($\leftarrow_{\text{Deep}}$ — (Class Identifier × Class Identifier × Class Identifier) → Class Identifier)

```
for any classes A, B and C, A [B $\leftarrow_{\text{Deep}}$ C] =
    if A $\equiv$ B
        then C
    else if B $\equiv$ C or B $\notin$ system (A)
        then A
    else
        A ' where Rep (system (A ')) = { n [B $\leftarrow_{\text{Deep}}$ C] | n $\in$ Rep (system (A) }
```

• <u>*parent*</u> ($\lhd_I$ — Class Identifier × Class Identifier)

A $\lhd_I$ B denotes a special superclass relationship between A and B, where the interfaces of *system* (B) form a superset of the interfaces of *system* (A) with all references to A in *system* (A) replaced by references to B in *system* (B). Formally:

For any classes, A and B:

$A \lhd_I B \Leftrightarrow A \lhd B$ by $\{(x, x\,[A \leftarrow_{\text{Deep}} B]) \mid x \in \textit{system}\,(A)\}$

5.5.4 Defining PS Inheritance and Class Substitution in terms of Nodes

Inheritance and substitution mechanisms provide shortcuts for denoting mapped systems of existing systems. Given classes A, B and C where B $\lhd$ C by *f*, A [B ← C] denotes the interface and code for an entire system. The interface of the mapped system has node representation $X \cup Y$, where:

- X = *map* (*f*, *Rep* (*system* (A))) and
- Y = *Rep* (*system* (C))

The code of the mapped system is found in *system* (A) (for methods with signatures in X) and in *system* (C) (for methods with signatures in Y). The expression, A [B ← C] can be used in both parts of the class interface where a class name can appear: in a variable declaration and as an argument to the inherits operator.

Inheritance allows for the denotation of mapped systems which include variables and/or methods not declared elsewhere. A class C, declared in its interface to "inherit B", consists of the same code and interface as B (with all references to B found in the system interfaces changed to C) as well as any additional methods and variables unique to C. Inheritance, like substitution, can be viewed as a higher-order function on systems of classes denoting mapped systems of those systems. Formally, we have:

- *inherits* (inherits — Class Identifier $\times 2^{\text{Node}} \rightarrow$ Class Identifier)

For any class A, and set of nodes N, *inherits* (A, N) denotes a mapped system of *system* (A) where all (possibly deep) recursive references are maintained. Thus:

$$\textit{inherits}\,(\mathrm{A}, N) = \mathrm{A}', \text{ where } \textit{Rep}\,(\textit{system}\,(\mathrm{A}')) = N \cup \{\mathrm{n}\,[\mathrm{A} \leftarrow_{\mathrm{Deep}} \mathrm{A}'] \mid \mathrm{n} \in \textit{Rep}\,(\textit{system}\,(\mathrm{A}))\}$$

The definition appears to be recursive but is well-defined. The deep substitutions resulting in the set, $\{\mathrm{n}\,[\mathrm{A} \leftarrow_{\mathrm{Deep}} \mathrm{A}'] \mid \mathrm{n} \in \textit{Rep}\,(\textit{system}\,(\mathrm{A}))\}$, involve symbolic manipulations only. As an example, let B be defined as in **Example 5.7a**.

```
class B
   var b : B
   method m (other : B) : Object
      begin  ... end
end B.
```

Example 5.7a

Then we have:

Rep (*system* (B)) =

```
{   (v, b, B, B, «», ∅),
    (m, m, Object, B,
            «(v, self, B, B, «», ∅),(v, other, B, B, «», ∅) »,
            ∅)
}
```

By application of the inherits and deep substitution definitions we can define a new class, `B'` which inherits the class definition of `B` and adds the instance variable, `i : B'`.

`B'` = *inherits* (B, { (v, i, Int, B', «», ∅) }).

As a result,

Rep (*system* (B')) =

```
{   (v, b, B', B', «», ∅),
    (m, m, Object, B',
            «(v, self, B', B', «», ∅),(v, other, B', B', «», ∅)»,
            ∅),
    (v, i, Int, B', «», ∅)
}
```

from which the class definition of **Example 5.7b** can be inferred.

```
class B'
   var b : B'
   method m (other : B') : Object
      begin {as defined in B} end
   var i : Int
end B'.
```

Example 5.7b

Note that for any class A, *inherits* (A, ∅) is an identity operation.

• *subst*

($\leftarrow$ — Class Identifier $\times$ Class Identifier $\times$ Class Identifier $\rightarrow$ Class Identifier | fail)

Given classes A, B and C where B $\triangleleft$ C by *f*, A [B $\leftarrow$ C] denotes a mapped system of *system* (A) where (at least) all references to B's have been replaced by references to C's. More formally:

```
For any classes, A, B and C:
   A [B ← C] =
      if B ◁ C by f
         then
            if A ≡ B
               then C
            else if B ≡ C
               then A
            else if B ◁_I C
               then A [B ←_Deep C]
            else --f = { ..., (D,E), ... } where E ≢ D [B ←_Deep C]*
               then A [D ← E] [B [D ← E] ← C]
      else
         fail
```

* Note that if B $\not\triangleleft_I$ C but B $\triangleleft$ C by *f*, then *f* must include some pair, (D,E) where E $\not\equiv$ D [B $\leftarrow_{Deep}$ C].

5.5.5 Results

It is necessary now to demonstrate that the inherits and substitution definitions above denote mapped systems, to ensure that their use leads to type-safe code reuse. The first result (for any class A and any set of nodes N, $A <_I$ *inherits* (A, N)), trivially follows from the definitions of $<_I$ and *inherits*. **Appendix IV** contains a proof that the definition of a substitution denotes a mapped system (ie: that $B < C \Rightarrow A < A[B \leftarrow C]$). This result is dependent on the result: $B <_I C \Rightarrow A < A[B \leftarrow_{Deep} C]$. Both results are summarized below:

(1) $B <_I C \Rightarrow A < A[B \leftarrow_{Deep} C]$

Intuitively, what is established here is that a deep substitution of one class for another (C for B) results in a mapped system provided that C is a subclass of B that can be created by inheriting B. This result is proven using a case analysis on the definition of $\leftarrow_{Deep}$. In the first case, we are given that $A \equiv B$, $B <_I C$ and $A[B \leftarrow_{Deep} C] = C$. It follows that $A < A[B \leftarrow_{Deep} C]$ since $A < B$, $B < C$ and $<$ is transitive. In the second case where $A[B \leftarrow_{Deep} C] = A$, the result trivially follows. In the third case we are given that $B \in$ *system* (A) and $A[B \leftarrow_{Deep} C] = A'$ where *Rep* (*system* (A')) = $\{n[B \leftarrow_{Deep} C] \mid n \in$ *Rep* (*system* (A))$\}$. That $A < A'$ can be shown by considering a partition of *Rep* (*system* (A)), that depends on the definition of a new function, *filter*.

• *filter*

Given some classes X and Y belonging to *system* (A), *filter* (X,Y) returns the class with the same interface and code as class X, minus any signature (and associated code) containing a reference to Y. As an example, the interface of class W' of **Example 5.8** is the same as the interface of *filter* (W,Z) [W $\leftarrow_{Deep}$ W'].

```
class W
      var w : W
      var w2 : Z
      method m ( i : Int) : W
             begin ... end
      method m2 (i : Z) : W
```

```
            begin  ... end
end W

class W'
     var w : W
     method m (i : Int) : W
            begin  ... end
end W'
```

Example 5.8

The *filter* function allows for a natural partition of *Rep* (*system* (A)). Since B ∈ *system* (A), it follows that *system* (B) ⊆ *system* (A), and therefore that:

- *Rep* (*system* (A)) = *Rep* (*system* (B)) ∪ *Rep* (*system 'filter* (A,B)))

Now consider any node, n ∈ *Rep* (*system* (A)) and the possibilities for its corresponding node, n' ∈ *Rep* (*system* (A')):

If n ∈ *Rep* (*system* (B)) and n ∉ *Rep* (*system* (*filter* (A, B))), then
n' = n [B $\leftarrow_{\text{Deep}}$ C] by the definition of $\leftarrow_{\text{Deep}}$.

If n ∉ *Rep* (*system* (B)) and n ∈ *Rep* (*system* (*filter* (A, B))), then
n' = n [B $\leftarrow_{\text{Deep}}$ C] since B $\lhd_I$ C.

If n ∈ *Rep* (*system* (B)) and n ∈ *Rep* (*system* (*filter* (A, B))), then corresponding nodes are found in *Rep* (*system* (A')) from B $\lhd_I$ C and from the definition of $\leftarrow_{\text{Deep}}$. Since in both cases, the corresponding node is n [B $\leftarrow_{\text{Deep}}$ C], then n has but one image and A $\lhd$ A' by { (t, t [B $\leftarrow_{\text{Deep}}$ C]) | t ∈ *system* (A)}.

Thus the result holds in the third case, and is proved.

Note that it is the third case that demonstrates that B $\lhd$ C by *f* is not a sufficient condition by itself, to guarantee that A $\lhd$ A [B $\leftarrow_{\text{Deep}}$ C], since it could be for some type, s ≠ t [B $\leftarrow_{\text{Deep}}$ C], that *f* (t) = s.

(2) $B \lhd C \Rightarrow A \lhd A[B \leftarrow C]$

This result follows from an inductive proof that depends on the previous result. For this proof it is necessary to define a function that given some classes X and Y, where $X \lhd Y$, returns the number of substitutions necessary to apply to denote a class that is a parent of Y.

• <u>*DOS*</u>

The function, *DOS* (or "degree of substitution") returns for any classes X and Y, where $X \lhd Y$, 0 if $X \lhd_1 Y$, and *n* if it takes *n* deep substitutions on X to denote a class which is a parent of Y. More formally, given that $X \lhd Y$ by *f*:

$$DOS(X, Y) = | \{(U,V) \in f \mid V \not\equiv U[B \leftarrow_{Deep} C]\} |$$

The proof of result (2) is an induction proof on the value of *DOS* (B,C):

In the base case where *DOS* (B,C) = 0, then either we have equivalent cases to cases 1 and 2 of the previous proof, or $B \lhd_1 C$ and the result of the previous proof applies.

In the inductive case, we are to show that $A \lhd A[D \leftarrow E][B[D \leftarrow E] \leftarrow C]$ given that $B \lhd C$ by some function *f*, but $B \not\lhd_1 C$. In this case, there exists some pair of classes, $(D,E) \in f$ where $E \not\equiv D[B \leftarrow_{Deep} C]$). Clearly,

• $DOS(B[D \leftarrow E], C) < DOS(B, C)$

Then, by the inductive hypothesis:

• $A[D \leftarrow E] \lhd A[D \leftarrow E][B[D \leftarrow E] \leftarrow C]$.

It is shown in the proof that $\mathtt{D} \lhd \mathtt{E}$ and that $DOS\ (\mathtt{D}, \mathtt{E}) < DOS\ (\mathtt{B}, \mathtt{C})$. By the inductive hypothesis,

- $\mathtt{A} \lhd \mathtt{A}\ [\mathtt{D} \leftarrow \mathtt{E}]$

and the result follows from the transitivity of $\lhd$.

5.6 The Additions to PS Subclassing

5.6.1 Multiple Inheritance and Multiple Subclassing

Unlike L-Trees, the node representation of classes allows for a class to have more than one unrelated superclass. This is because the node representation is based on the interface rather than the code of a class, and because subclassing relations do not demand that the labels of nodes of superclasses be prefixed strings of corresponding node labels in subclasses.

The unfolding algorithm of 4.3.3 can result in deadlock when multiple inheritance is allowed. The alternative inheritance algorithm of 5.5.4 can be extended in a natural way to support multiple inheritance. As is the case in all languages supporting multiple inheritance, type-safety of inherited code demands that no two variables or methods with the same name are inherited from different sources. Assuming that no two variables or methods with the same name are defined more than once in the set of classes denoted by the set of Class Identifiers, S and given some additional set of nodes, N:

inherits $(S,N) = \mathtt{A'}$, where $Rep\ (system\ (\mathtt{A'})) =$

$$N \cup \bigcup_{\mathtt{c} \in S} \{\mathtt{n}\ [\mathtt{C} \leftarrow_{\mathrm{Deep}} \mathtt{A'}] \mid \mathtt{n} \in Rep\ (system\ (\mathtt{C}))\}$$

5.6.2 Temporal Cycles

PS subclassing disallows sets of classes that comprise temporal cycles because the unfolding algorithm of section 4.3.3 results in deadlock given such classes as input. It is informally argued in [PaSc90d] that classes such as `Candidate` and `Voter` of **Example 4.5** should be irresolvable because `Voter` "has-a" `Candidate` and `Candidate` "is-a"

Voter, and thus there does not appear to be an order in which these classes should be parsed.

The definition of inheritance of section 5.5.4 allows the Candidate/Voter classes as input, as well as most other classes with temporal cycles. The system of classes denoted by the class definitions of **Example 4.5** are shown in **Example 5.9**.

```
class Voter
      var backs : Candidate
end Voter

class Candidate
      var backs: Candidate
end Candidate
```

Example 5.9

and Voter $\cong$ Candidate by the mapping function, {Candidate, Candidate}.

A more interesting example results from adding any method to the definition of class Voter. For example, if the class Voter were defined as:

```
class Voter
      var backs : Candidate
      method vote () : Candidate
            begin  ... end
end Voter
```

then class Candidate would be as defined in **Example 5.10**.

```
class Candidate
      var backs : Candidate
      method vote () : Candidate
            begin    end
end Candidate
```

Example 5.10

Because of the addition of a method to the superclass, (and thus the implicit parameter, `self`, to the node representations of both classes) the above two classes are no longer equivalent. The node representations for the classes in this system are:

Rep (*system* (Voter)) =

```
{    (v, backs, Candidate, Voter, «», ∅),
     (m, vote, Voter, Voter, «(v, self, Voter, Voter, «», ∅) », ∅),

     (v, backs, Candidate, Candidate, «», ∅),
     (m, vote, Candidate, Candidate,
         «(v, self, Candidate, Candidate, «», ∅) »,
         ∅)
}
```

and Voter ◁ Candidate by { (Voter, Candidate), (Candidate, Candidate)}. Note how the types of the inherited variables and methods of Candidate are determined. The return type of the method vote in class Candidate is Voter [Voter $\leftarrow_{\text{Deep}}$ Candidate], or Candidate. The class denoting the type of the variable backs is Candidate [Voter $\leftarrow_{\text{Deep}}$ Candidate], which is Candidate since Voter $\notin$ *system* (Candidate).

While the new interpretations of PS subclassing remove some of the restrictions detailed in the original proposal, they do not remove them all. In particular, node representations cannot be generated for systems containing two mutual supplier classes, where one is a subclass of the other. An example is *system* (Voter) = {Voter, Candidate} shown in **Example 5.11a**.

```
class Voter
      var backs : Candidate
      method vote () : Voter
            begin  . end
end Voter
```

```
class Candidate inherits Voter
      var influence : Voter
      method vote () : Voter
            begin  ... end
end Candidate
```

Example 5.11a

The system denoted by the class definitions above is the infinite system described in **Example 5.11b**.

```
class Voter
      var backs : Candidate
      method vote () : Voter
            begin  ... end
end Voter

class Candidate
      var backs : Candidate'
      method vote () : Candidate
            begin  ... end
      var influence : Voter
end Candidate

class Candidate'
      { ≅ Candidate [Voter ←Deep Candidate]}
      var backs : Candidate''
      method vote () : Candidate'
            begin  ... end
      var influence : Voter
end Candidate
```

```
class Candidate''
      (≡ Candidate' [Voter ←Deep Candidate])
      var backs : Candidate'''
      method vote () : Candidate''
            begin ... end
      var influence : Voter
end Candidate
```

...

Example 5.11b

Because both `Voter` and `Candidate [Voter` $\leftarrow_{\mathrm{Deep}}$ `Candidate]` belong to *system* (`Candidate`), *system* (`Candidate`) is infinite. Classes forming temporal cycles are acceptable as input to the alternative inheritance resolving algorithm of section 5.5.4, provided that they do not denote infinite systems.

5.6.3 Constants

It was shown in section 4.3.4 that PS subclassing mechanisms do not denote type-safe subclasses when classes contain constant expressions. **Example 4.16** contained a class named `example` that contained code assigning the constant value 45 to a variable of class (type) `Int`. It was shown that in any subclass of `example`, where `Int` had been substituted with some subclass of `Int`, this assignment would violate the binding check.

The PS proposal demanded that base classes be defined in the language like any other class. A sample definition from [PaSc90a], for the class, `Int` was shown in **Example 4.17**. It was shown that class definitions such as this did not eliminate constant expressions and base classes. The constant, 45 became an *Object* instead of an *Int* in this interpretation, as it could be bound to the instance variable, `value` of type *Object*. It was shown that any class that included code that bound a constant to the instance variable, `value` would have unsafe subclasses when another class was substituted for `Object`.

That `Int`, `Bool` and `Char` are not constant classes in the PS approach, is also problematic. Given the class definition for `Int` of **Example 4.17**, the class expression `Int [Object ← Bool]` denotes a distinct subclass of `Int`, equivalent to the class definition of **Example 5.12**.

```
class Int'
      var value : Bool
      method plus (other : Int')
            begin ... end
      method times (other : Int')
            begin ... end
      method zero : Int'
            begin ... end
end Int'
```

Example 5.12

Note that `Int'` is not structurally equivalent to `Int` because the instance variable, `value` has type *Bool* in `Int'`. While it is unlikely that the class expression, `Int [Object ← Bool]` would ever be used in a variable declaration, this class would inadvertently be a supplier to the class, `Array [Object ← Bool]`, given the class definition for `Array` that is borrowed from [PaSc90d] and shown in **Example 5.13**.

```
class Array
      method at (i : Int) : Object
            begin ... end
      method atput (j : Int, x : Object) : Array
            begin . end
      method init (size : Int) : Array
            begin ... end
      method arraysize () : Int
            begin .. end
end Array
```

Example 5.13

Because of the deep nature of substitution, the parameters `i` (of method `at`), `j` (of method `atput`) and `size` (of method `init`) would have its their types denoted by the class, Int [Object ← Bool] in the class, Array [Object ← Bool]. What was desired of course, was for these parameters to have the type *Int* in this subclass.*

Problems with constants arise if it is possible for base classes to have subclasses. Statements that bind constants to variables are not safe if the class of the constant (and hence of the variable) is replaced by a subclass. Subclasses of base classes can inadvertently become suppliers to the subclasses of base class' clients, because of the deep nature of subclassing mechanisms. The solution proposed here is to restrict base classes to have no subclasses, save for themselves. Thus, given a base class C and a set of nodes *N*,

inherits (C, *N*) = C

and given classes A and B where A ◅ B,

C [A ← B] = C.

Given this interpretation, Int [Object ← Bool] would be equivalent to Int, and therefore Int would be a supplier class for Array and also for all subclasses of Array. If Int also has no subclasses by inheritance, then the loss of binding checks in the presence of constants is countered since the types of variables that are bound to constants are guaranteed not to change in subclasses.

It is unfortunate but necessary to distinguish between regular and base classes. The division is a natural one, and while the aesthetic appeal of having only one form of class is lost, only classes in the base system are irregular. One of the next steps in my research will be prove that PS subclassing mechanisms ensure type-safety in subclasses, given this property of base classes, and given an extension of the PS-based Mini-Dee grammar that reintroduces rules allowing constant expressions.

5.7 A Prototype Parser of PS-Based Mini-Dee

Appendix V contains output from a prototype Mini-Dee parser, implemented as part of the preliminary work for this thesis. Complete systems of class interfaces (both

* This problem can be resolved by removing the instance variable, value from the class definitions for Int, Bool and Char. By doing this, any substitution applied to one of these classes results in a class that is structurally equivalent to the original class. This solution does not solve the initial problem that arises when constant values are bound to variables whose types can change in subclasses.

explicitly defined and those denoted by inheritance and substitution) are input to the parser. Node representations of these class interfaces are generated as output. Representations of systems denoted by inheritance and substitution are generated using the algorithms of sections 5.5.3 and 5.5.4. Also output is a list of every class given a representation, and every other class of which it is a subclass. For each subclass/superclass pair, the output includes the mapping function that establishes the pairing. The parser is implemented on a Macintosh using Think C; a variation of the hybrid object-oriented language, C++.

5.7.1 Data Structures

The important data structures for this implementation are defined using class definitions. The most fundamental data structure is the *node*. Collections of nodes are kept in *class representation objects*. Collections of class representation objects are kept in a *table*. Of these structures, the table is simplest. It contains an array of class representation objects as well as various operations that search and insert into the array. The class representation object contains a list of nodes comprising the representation of some given class. It also contains a list of strings ("aliases") by which the class is referred. For example, given any classes identified as A, B and C, where B $\lhd$ C and B $\notin$ *system* (A), the name "A [B ← C]" is an alias for the class A. Upon seeing a reference to A [B ← C], the parser generates its representation and seeing that it is equivalent to the representation for A, removes it from the table and adds the string, "A [B ← C]" as an alias associated with the representation for A.

Two data structures are used to implement the node. These data structures are implemented as separate subclasses of an abstract node class; one for variable nodes and the other for method nodes. Therefore, of the six components of the node tuple described in section 5.3, only five are explicitly represented. The *kind* component that distinguishes between a variable and method node, is unnecessary given the information provided by the subclass hierarchy. The *id* field, giving the name of the variable or method, is implemented as a string. The *parms* and *locals* fields only part of method nodes, and are implemented as lists of variable node objects. The *type* and *context* fields that show the declared class of the variable or method and the class in which the variable or method is declared, are implemented as indices into the table of class representation objects. The reasons for using table indices to represent class identifiers are outlined below.

5.7.2 Design Issue: Representing Class Identifiers

The Mini-Dee parser was designed in two stages. The original design was for a parser that generated node representations for systems of explicitly defined classes only. At this stage of the design, three representations of class identifiers were considered. Since the first parser did not allow the use of inheritance and substitution to denote systems of classes, it would have been possible to identify class references appearing in nodes with class names. However, it became clear that the addition of support for inheritance and substitution to the parser would result in some cases in the denotation of unnamed classes (see the footnote of section 5.3).

Another design considered was to represent class identifiers with the class representation objects themselves. As the mutual supplier classes of **Example 5.14** demonstrate, this choice would not have allowed the parser to terminate in some cases.

```
class A
      var b : B
end A

class B
      var a : A
end B
```

Example 5.14

If a class identifier were implemented as a class representation object, then the representation for class A would be incomplete until the representation for B was complete and vice-versa. Therefore, this approach was abandoned and the third approach was taken.

The third approach involved adding an extra level of indirection to allow the parser to generate representations for systems of classes such as the one above. In this scheme, a "dummy" representation is stored in the array while a class representation is being generated. Indices to these "dummy" representations are used to represent the references to the class whose representation is as yet incomplete. When the actual class' representation is complete, the "dummy" record is replaced by the completed representation object, keeping all index references found in other representations valid. **Table 5.1** shows the sequence of events occurring during the parse of the above system of classes, and the contents of the table of class representation objects at each stage of execution.

Step	Action	Table
1	Table is searched for representation for A.	—
2	A is not in table. A "dummy" representation for A is inserted.	1 | A | "dummy"
3	Parse of A begins.	"
4	Node for variable b of class A triggers table search for representation for B.	"
5	B is not in table. A "dummy" representation for B is inserted.	1 | A | "dummy" 2 | B | "dummy"
6	Parse of B begins.	"
7	Node for variable a of class B triggers table search for representation for A. A found at index, 1. 1 inserted into node for variable, a.	"
8	Parse of B ends. "Dummy" representation for B replaced by node representation for B. Completing step 4, index for B (2) inserted into node for variable, b.	1 | A | "dummy" 2 | B | { (v, a, 1, 2, ∅, ∅) }
9	Parse of A ends. "Dummy" representation for A replaced by node representation for A.	1 | A | { (v, b, 2, 1, ∅, ∅) } 2 | B | { (v, a, 1, 2, ∅, ∅) }

Table 5.1

5.7.3 Design Issue: Handling Compiler Dependencies (and Temporal Cycles)

The use of table indices to represent class identifiers allows explicitly defined classes to be parsed in any order. The order in which representations of classes can be generated cannot be random however, once inheritance and substitution can be used to necessary to generate the complete node representation of A completing the representation of A'. (Thus, given the temporally cyclic system of classes, {Voter, Candidate}, it is necessary to finish the parse of Voter before finishing the parse of Candidate). Similarly, it is necessary to generate representations of the classes C, D and E before generating the representation of C [D ← E].* These dependencies are referred to here as *compiler dependencies*. They not only determine the order in which classes must be parsed, but also what systems of classes cannot be resolved. A system of classes is irresolvable if there exists a cycle on the graph that has a node for every class, and edges indicating compiler dependencies. As a simple example, if A inherits B and B inherits A, then {A, B} is an irresolvable system. We saw earlier, that classes which are subclass-related and mutual clients must also be considered irresolvable, since such classes comprise infinite systems.

Two approaches to handling compiler dependencies were considered. One approach was to have a front-end preprocessor that would examine all classes in the input systems and establish an order in which these classes should be parsed. This approach was ruled out because it was incompatible with the design of the initial parser that generated node representations for explicitly defined classes. The order in which classes were parsed in the initial parser depended on the order in which references to them appeared in other classes. Thus, the parse of a class A that had a variable declared to be of type/class B, was interrupted to parse B if B had not yet been parsed.

The second approach to handling compiler dependencies was used in the final design. This approach introduces state to class representation objects. At any point during the parse, a class representation object can have been completely generated (*resolved*) or in the process of being generated (*unresolved*). Given some compiler dependency, A → B, the resolution of the class representation object for A is delayed until the state of the class representation object for B is resolved. Once B is resolved, the processing of A is triggered and subsequently the state of the class representation object for A can become resolved. This in turn, triggers the generation of representation objects for classes that depend on A and so on. This approach was compatible with the initial parser since the order in which classes were parsed still depended on the order in which references to classes appeared in other class definitions. (Though the order in which class parses were completed depended on compiler dependencies).

* In fact, it is only necessary to generate the complete node representation of C to generate the node representation of C [D $\leftarrow_{Deep}$ E]. (If this were not the case, then the inherits definition of 5.4 would not be valid). But C [D ← E] is only equivalent to C [D $\leftarrow_{Deep}$ E] when D ◁ E, and the latter relation can only be determined if the representations of D and E are complete.

The most interesting challenge of this approach was representing state in objects. A simple flag in the object that indicated its current state, was insufficient since the behaviors of an object also depended on its state. For example, a message to a class representation object to parse itself should do nothing if the object is resolved, but should initiate a parse if it is unresolved. The ideal object-oriented solution was to declare all methods whose implementations depended on state in an abstract class and to provide the varying implementations in separate subclasses. `Resolved` and `Unresolved` could name unique subclasses of some abstract representation class, and a class representation object would be able to change its class from `Unresolved` to `Resolved` dynamically. Untyped languages such as Smalltalk and CLOS support the dynamic change of class of an object; herein referred to as "*class metamorphosis*." Support for class metamorphosis in typed object-oriented languages is the subject of some current research [Ta91], but has yet to be implemented in any well-known typed object languages.

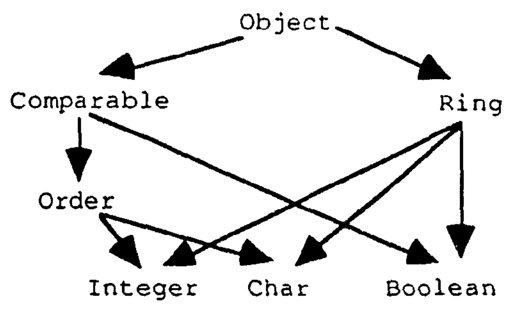
Class metamorphosis is simulated in my design using the supplier/client relationship between classes. Resolved and Unresolved states are implemented as subclasses of an abstract representation class, with varying implementations of inherited abstract methods. The class representation objects themselves do not belong to these classes but are clients of them. Each object has an instance variable `state`, declared of the abstract representation class. Methods whose implementations depend on state, are defined in the representation object to invoke the corresponding method in the state object. Thus, a class representation object can change its state by reassigning a new object to its state instance variable. The state object acts like the class of the representation object because it contains implementations of object methods. The state of the representation object though, can be changed dynamically.

This simulation does not work as well as true class metamorphosis should. At the implementation level, there is an overhead of an extra level of indirection in method calls, as well as some memory management necessary to dispose of discarded state objects when objects change their state. Also, self-reference can only be clumsily simulated. It is necessary to keep some variables and methods declared in the representation object since they are not altered by a change of state. Since some of the method definitions found in the state object must refer to these variables and methods, it is necessary for the state object to keep a reference to the class representation object that owns it. As a result, separate state objects are required for each class representation object, and self-reference that would be inherent if state were implemented as a class must be simulated instead as a clumsy network of interrelations that makes coding difficult and that violates encapsulation. A language's support for class metamorphosis should allow an object to change its class provided that it did not change its type. (By not allowing it to change its type, we can statically ensure that type-safety will not be violated). Class metamorphosis thus results in a form of dynamic binding, and its support is dependent on resolution of type issues that have been discussed here.

5.7.4 Sample Output from the Mini-Dee Parser

Appendix V contains sample output from the Mini-Dee parser. Twenty-one examples are given, illustrating many of the issues that have been discussed here.

The output from the first example (V.A) consists solely of the base classes of the Mini-Dee system and their superclasses. This class structure is borrowed somewhat from the Dee class library ([Gr91]) and presumably would constitute the minimal library of classes provided to the user of Mini-Dee. The graph of **Figure 5.1** shows how each class is related in the subclassing hierarchy:



A $\longrightarrow$ B indicates that A $\lhd$ B

Figure 5.1

`Object` is the superclass of all classes, defining no variables nor methods. `Comparable` is a partially abstract class that contains a method declaration for the comparison operator, '=' and a definition for '≠'. `Order` is also a partially abstract class containing declarations and definitions for other comparison operators such as '<', '≤' etc. `Ring` contains declarations for the operators "+" and "zero." `Int`, `Char` and `Bool` are base classes since their instances are constants. The definitions for all of these classes are shown in **Example 5.15**.

```
class Object
end Object

class Comparable
      method equals (other : Comparable) : Bool
            begin ... end
end Comparable

class Order inherits Comparable
      method lt (other : Order) : Bool
            begin ... end
      method gt (other : Order) : Bool
            begin ... end
      method lte (other : Order) : Bool
            begin ... end
      method gte (other : Order) : Bool
            begin . end
end Order

class Ring
      method plus (other : Ring) : Ring
            begin ... end
      method zero : Ring
            begin ... end
end Ring

class Bool inherits Ring Comparable

      method and (other : Bool) : Bool
            begin .. end
      method or (other : Bool) : Bool
            begin .. end
      method not : Bool
            begin . end
end Bool
```

```
class Char inherits Ring Order
      method asc : Int
            begin .. end
end Char

class Int inherits Order Ring
      method times (other : Int) : Int
            begin ... end
      method div (other : Int) : Int
            begin ... end
      method mod (other : Int) : Int
            begin .. end
end Int
```

Example 5.15

The output for this and other examples consists of three parts. The first part describes every unique class representation object generated during the parse. The second part shows for every class that was parsed, the superclasses of that class. For each class/superclass pair, also output is the mapping function that establishes the pairing. The last part shows for every class that was parsed, the parent classes of that class. Again, along with every class/parent pair, the mapping function establishing the pairing is also output.

The table output lists the name of each class given representation, the index of the representation in the table, the aliases for the represent. tion, the final state of the representation object, and if the final state is resolved, the nodes that comprise the representation. It is necessary to output the index of each class representation since the classes that are the type and context fields of a node are identified by indices.

As the base classes and their superclasses are always : vailable for use by the Mini-Dee user, they are always parsed before input systems of classes are parsed. As a result, they always appear in the same positions in the table. Therefore, the node representations of base classes are only provided in V.A.

The second example (V.B) consists of the `Turtle`, `WrappingTurtle` classes of **Example 2.7**. This example demonstrates that PS subclassing captures the form of subclassing redefinition usually desired by programmers.

The third example (V.C) consists of classes used to demonstrate the PS "unfolding" algorithm. Used as input to the Mini-Dee parser, it is evident that the alternative definition of inheritance presented in Section 5.5.4 results in the denotation of the same system of classes prduced by the "unfolding" algorithm.

The example classes of **V.D** are taken from **Example 5.7a** and **Example 5.7b** that show that the alternative definition of inheritance presented in Section 5.5.4 is well-defined.

The example classes of **V.E** are borrowed from section 4.3.3 (**Examples 4.10a** and **4.10b**), where it was shown that class substitution is consistent throughout a system of classes. Of interest in this example is the type of the variable, o_3, in class `B'`. If class substitution were not consistent over a system of classes, then the type of o_3 would be *C*, and not *C* [*Object* ← *Int*].

The example classes of **V.F** are also borrowed from section 4.3.3 (**Example 4.11**), where it was shown that class substitution is more than just deep substitution. Of interest in this example, are the types of o_1 and o_2 in class `B"` (`B [C ← C']`). If class substitution were only deep substitution, then the types of these variables would be *Object* rather than *Int*. As was demonstrated in Chapter 4, the resulting class from deep substitution could not be guaranteed type-safe. We can see by examining the aliases for the class representation of `B"`, that `B [C ← C']` is equivalent to `B [Object ← Int]`, and that the argument to the method `p` in this class must be of class `Int`. The classes of **V.G** are also borrowed from section 4.3.3, showing that a complex substitution is not necessarily equivalent to a simple substitution. In this example, the class `Fam [An ← Hum]` (HF_1) is not equivalent to `Fam [Object ← Int]` (HF_2).

The classes of **V.H** and **V.I** are taken from **Examples 5.3 - 5.5** that provide example "mapped systems." **V.J** shows that subclassing is structural and not just determined by the transitive application of explicitly defined inheritance relations. In this example, classes `E` and `F` are formed by multiply inheriting different and unrelated pairs of classes. Though class `E` inherits `A` and `B`, and class `F` inherits `C` and `D`, and `A`, `B`, `C` and `D` are not in any way related, `E` and `F` are structurally equivalent and hence given the same representation. **V.K** carries this idea a step further, showing eight very similar classes. Of the eight, only `I` is equivalent to `A` This is because classes `A` and `B` are mutual suppliers, and hence any subclass of `A` must preserve this relationship, and thus be both a client and supplier of some subclass of `B`. Classes `C` and `H` are also equivalent since each has a recursive variable named `a`, and a method `p` with one recursive parameter and one *Int* parameter, and returning a *B*.

The classes of **V.L**, **V.M** and **V.N** are taken from the exhaustive case analysis of Chapter 3, used to show that Bruce classes are not necessarily valid just because they are type-safe. Classes that are problematic to the Bruce system are valid as PS classes because of the consistent nature of signature redefinition that extends through a system of PS classes.

The classes of **V.O** and **V.P** illustrate how temporal cycles are handled. Both examples is a variation of the {`Candidate`, `Voter`} system first shown in **Example 4.5**. It should be pointed out that the classes of **V.P** comprise an irresolvable system. This example has as its input, the class interfaces first presented in section 5.6.2. It was shown in this section, that because this system contains two mutual supplie. classes, one of which is subclass of the other, that the denoted system is infinite.

V.Q and **V.R** also show irresolvable systems. These systems are irresolvable because of compiler dependencies. In both examples, `A` and `B` are mutually dependent. **V.S** demonstrates that multiple inheritance is supported in this subclassing scheme. Class `A` inherits three classes, two of which are themselves generated via substitutions.

V.T and **V.U** are borrowed from [PaSc90d]. Palsberg and Schwartzbach used the classes of **V.T** to show how class substitution might be used in a practical application. The classes `Array` and `Stack` are defined explicitly as collection classes whose items are *Objects*. `Bool` and `Int` stack specializations are denoted using substitution. This example shows how the deep nature of substitution can result in the denotation of classes that have not been explicitly named. `Stack` is implemented with a variable, `space`, that is an *array* of *objects*. Appropriately, in `Intstack`, `space` is an *array* of *Ints* and in `Boolstack`, an *array* of *bools*. These unnamed subclasses of `Array` are given representation at indices 16 and 23 respectively in the table.

V.U shows another system of classes involving *Arrays*. Two things should be noted about this example. Firstly, the variable, `r` in the class `Matrix`, is declared of a class denoted via substitution. (All previous examples only used substitution to denote classes that were arguments to the "inherits" class operator). Secondly, this example shows the structural nature of subclassing as defined on node representations of classes. The classes `BoolMatrix` and `MatrixMatrix` are subclasses (and also child classes) of the class `Ring`, even though they are not explicitly declared as such. Similarly, the class `DoubleRingArray` is a subclass and a child class of `Array`, though not defined in this way.

This example shows the transitive nature of subclassing. `DoubleRingArray` is a subclass of `DoubleArray` which in turn is a subclass of `Array`.

5.8 Conclusion

It was argued in Chapter 4, that despite its shortcomings, the PS approach to subclassing offers more potential as a subclassing approach than does the restrictive approach of Bruce. It was pointed out however, that the PS approach is as yet incomplete. The PS representation of classes and accompanying algorithms for inheritance and substitution, while useful for proving results, are not intuitive. PS subclassing is incompatible with classes containing constant expressions and is not well-defined for types that are anything but singleton sets. Classes forming temporal cycles are invalid in the PS approach, and thus some useful classes that directly or indirectly, are clients of their own subclasses are disallowed. Finally, PS subclassing only allows for single inheritance and single subclassing relations.

An original and alternative interpretation of PS subclassing has been presented in this chapter. The node representation of classes is based on interfaces rather than code, and gives a finite representation of all classes. Various new terms have been introduced. A mapped system denotes a set of classes that can safely reuse the code of some existing system of classes. If there is a correspondence between classes in one system, to classes with at least the same structure in another system, and all references to classes in the original system are replaced by references to corresponding classes, then the latter system is a mapped system of the former. Subclassing is not a relation between classes but between systems of classes, and is based on the mapped system relation. Inheritance and class substitution are defined as higher-order functions that denote mapped systems. Inheritance denotes a mapped system which resembles the original system except that recursive references in inherited signatures are deeply replaced. Substitution involves the replacement of one system of classes, S_B with a mapped system, S_C in some other system, S_A, by applying the mapping function which maps S_B to S_C, to S_A. These definitions are less restrictive than the algorithms presented in [PaSc90d], since they accept most systems containing temporal cycles as input and are compatible with multiple subclassing relations and multiple inheritance. Finally, a resolution to the problems with constants is presented, which establishes constant classes to have no subclasses.

The ideas presented in this chapter have been implemented in a prototype parser which is also described, and whose output is presented in **Appendix V.** These contributions interpret and extend the original PS subclassing proposals, though do not resolve all of their shortcomings. Thought has been given to extending PS subclassing so as to be compatible with dynamic binding. Preliminary ideas are described in the next chapter, which outlines the future work being considered.

Chapter 6.

Future Work: Marrying Dynamic Binding to PS Classes and PS Subclassing

It is claimed in [PaSc90d] that PS subclassing, inheritance and class substitution are well-defined over classes whose signatures include non-singleton types (such classes are herein referred to as *non-singleton classes*) provided that all non-singleton types are finite sets of classes. PS subclassing and subclassing operations, it is said, do not result in type-safe subclasses when applied to non-singleton classes whose types are infinite sets of classes.

This argument was refuted in Chapter 4. PS subclassing and subclassing operations are only well-defined over classes whose signatures include singleton types (Such classes are herein referred to as *singleton classes*). L-Trees are only well-defined over singleton classes and it is therefore not clear how subclassing, defined over L-Trees, applies to non-singleton classes, even when the types of such classes are finite. It is also not clear how the graph-based representation of class definitions (used in the unfolding algorithm described in 4.3.3) can be extended over non-singleton classes and therefore it is not clear how inheritance is defined over such classes. Finally, there are conflicting interpretations in [PaSc90d] of how class substitution is defined over non-singleton classes.

Alternative definitions of subclassing, inheritance and class substitution, based on node representations of classes, can be naturally extended over non-singleton classes. It will be shown in this chapter that the new definitions for inheritance and class substitution result in type-safe subclasses, even when applied to non-singleton classes. Still, two key problems remain, that must be addressed before PS theory can be put into practice:

- The classes that result from application of inheritance and class substitution operations to non-singleton classes are type-safe subclasses, but include types that are difficult to express, not very useful, and not easily predicted. One problem that remains is how to achieve useful and intuitive subclassing of non-singleton classes.

- The combination of PS subclassing and the PS interpretation of types results in the tension described in Chapter 2, of subclass-based type interpretations and covariant subclassing. This tension must still be resolved.

These two problems are the focus of our current research. A detailed description of the problems, and informal descriptions of possible solutions are given here.

6.1 Marrying Dynamic Binding to PS Classes

It was pointed out in Section 2.5.3 that object-oriented languages with subclass-based type interpretations must sacrifice one of the following features:

- covariant subclassing redefinition,
- argument passing with method invocations, or
- strong or static typing.

None of these choices is appealing. It was suggested in Section 3.7 that the Bruce approach to dynamic binding, which adopted a subclass-independent interpretation of types, is preferable to subclass-based interpretations. A proposal follows, for a variation of the PS type interpretation that is subclass-independent.

6.1.1 A Proposal: Subclass-Independent PS Types

The PS interpretation of types establishes a type to be any set of classes, and a subtype any subset. These are more general definitions than are typically used, but as was argued in Section 4.4.2, only certain sets of classes make for useful types and subtypes. It was contended in this section that the only useful types were those for which signature checks could be confined to a single class. The types that resulted from this restriction were of three kinds: empty type, singleton types, and non-singleton sets of all subclasses of given classes.

In fact, the above interpretation of non-singleton types is not entirely useful, because it does not help to resolve the tension of subclass-based binding and covariant subclassing redefinition. An added criterion is proposed to discern useful types. This criterion demands that binding and parameter binding checks be the same. The types that result are also of three kinds:

- $\varnothing$ - the empty type. The signature check of an expression whose type is empty always succeeds, since an object of empty type should respond to any invocation, with the output of a string such as:

  "message sent to object of empty type" or
  "message sent to unconstructed object."

- C - the singleton type, $\{C\}$ (where C is a class). A signature check of an expression whose type is singleton, is confined to the class denoting the type of the expression.

- $\uparrow C$ - the non-singleton type, {all *equivariant* subclasses ($\lhd_e$) of C}. Equivariance is defined as the strict direction of signature redefinition which is both covariant and contravariant. A class is an equivariant subclass of another class if the signatures of the superclass' system are redefined in the subclass' system according to the following rules:

  1. neither the types of instance variables nor method parameters are redefined, and
  2. method return types, if redefined, are specialized in subclasses.

An equivariant subclass must also be covariant since PS subclasses allow for genericity. An equivariant subclass must also be contravariant so as to resolve the tension of covariance and dynamic binding. The result is a fairly strict dynamic binding, though arguably not much more strict then contravariance. (It is hard to think of an example class and subclass, where the generalization of a method parameter is desirable!)

A signature check of an expression of type $\uparrow C$ is confined to class C. Binding checks for variables and objects of any of the type categories are identical to equivalent parameter binding checks. The nature of these binding checks are summarized in **Table 6.1**. It is assumed for this table, that an object, o of type O is being bound to a variable or parameter v of type V (as in the assignment, v := o.) Each row in the table demonstrates a different case with regard to the types that are V and O. It should be assumed for these examples that A and B name classes.

V	O	*bind* (V,O)
Anything	$\varnothing$	TRUE
$\varnothing$	Anything but $\varnothing$	FALSE
A	B	$A \equiv B$
A	$\uparrow$B	FALSE
$\uparrow$A	B or $\uparrow$B	$A \lhd_e B$

Table 6.1

It should be pointed out here, that the binding checks above only succeed if the type of the object bound to a variable, is a subtype of the type of the variable, and is also a useful type. Strictly speaking, the set of all subtypes of $\uparrow A$ is the set of all subsets of,

$$\{x \mid A \lhd_e x\}$$

which is larger than the set of all useful subtypes,

$$\{\{x\} \mid A \lhd_e x\} \cup \{\{z \mid x \lhd_e z\} \mid A \lhd_e x\}.$$

6.1.2 Class Metamorphosis

The advantages of an OOP language's support for class metamorphosis were outlined in Section 5.7.3. It should be remembered from that discussion that support for class metamorphosis allows an object to change its class dynamically. The PS-based Mini-Dee parser, implemented as part of the preliminary work for this thesis and described in Section 5.7, would have benefited from support for class metamorphosis. In order for the parser to handle cyclic structures such as mutually supplying classes or classes lying on temporal cycles, it was desirable to introduce a notion of state to class representation objects. At any point during execution of the parse, class representation objects could be resolved or unresolved and could change their state from unresolved to resolved dynamically.

Support for class metamorphosis comes for free with the support of non-singleton classes, and definitions for type-safety over these classes. An object, through some syntactic mechanism would be able to change its class to any other class included in its type. As an example, given that B and C are subclasses of some class A, an object denoted by the expression, $e : \Uparrow A$ would be able to change its class with the execution of statements such as:

```
      e becomes (C),
      e becomes (A)
or    e becomes (B).
```

For any class X, the type-checker can statically guarantee the safety of the statement,

```
      e becomes (X)
```

by establishing that $A \lhd_e X$. This follows, since as far as type-safety is concerned, the changing of the class of e to X is equivalent to assigning an object whose type is $\{X\}$ to e. In either case, every message that is sent to e is guaranteed understood by the signature check of A.

Class metamorphosis would be meaningless in a language that supported only singleton classes since singleton types only include objects belonging to a single class. It appears that in expanding class definitions to include non-singleton types, support for class metamorphosis comes for free.

6.1.3 An Extension of Mini-Dee

Support for non-singleton classes demands that a new extension of Mini-Dee be defined, over which new definitions of nodes and type-safety can be based. The grammar for this extension can be found in **Appendix I-G**. Semantically, the new extension of Mini-Dee is the same as that defined for PS theory in 4.2. Syntactically, the rules for types below should replace the rules for the nonterminal, Type found in the extended PS-based grammar for Mini-Dee in **Appendix I-E**.

```
T    →    '∅'              -- the empty type
     |    CName            -- the singleton type, {CName}
     |    '↑' CName        -- the non-singleton type,
                              {all equivariant subclasses of CName}
```

Figure 6.1

Semantically, binding, parameter binding and signature checks should be as described in Section 6.1.1. Binding and parameter binding checks should be the same, and should both demand that the type, *O* of an object o be denoted by an equivariant subclass of the class denoting the type of the variable to which it is bound. Signature checks need only be confined to the class which denotes the type of the expression which receives a given message. Subclassing redefinition should still be covariant and consistent. Descriptions of inheritance, substitution and other subclassing mechanisms, as applied to non-singleton types, are provided in the next section.

6.2 Marrying Dynamic Binding to PS Subclassing

6.2.1 A Node Representation of Non-Singleton Classes

It is necessary to redefine nodes to reflect the fact that types can be non-singleton sets of classes. Accordingly, type and context fields of nodes (type_rep) are no longer represented by class identifiers, but labeled pairs of the form:

(name : Class Identifier, kind : {e, s, n}) where name identifies a class, and kind is e if the type is empty, s if the type is the singleton set of classes identified by name, and n if the type is the non-singleton set of all equivariant subclasses of name. Nodes are thus defined as in Section 5.3, but for the type and context components that are pairs of the above form. Thus, given the class definition for C in **Example 6.1** below,

```
class C
          var em : { }
          var a : A
          method m () : ↑X
                    begin ... end
end C
```

Example 6.1

we have

Rep (C) =

```
  {   (v,  em, (C, e), (C, s), «», ∅),
      (v, a, (A,s), (C,s), «», ∅),
      (m, m, (X,n), (C,s), «(v, self, (C,s), (C,s), «», ∅)», ∅)
  }.
```

As should be evident from the node representing the variable em, the empty type can be represented by a pair (z, e), where z is any class identifier. It should also be noted that the value of a node's context field is always of a singleton type. Strictly speaking, it is unnecessary to use the pair representation of types to represent the context of a node since the context is always a single class. However, the grammar defining type-safety of **Appendix VI**, relies on the equivalence of type and context representations, and therefore the pair representation is used for both node components.

6.2.2 Type-Safety Defined Over Non-Singleton Classes

The attribute grammar that defines type-safety in Mini-Dee for singleton classes must be revised in two additional ways to account for non-singleton classes also. Firstly, the actions that annotate the grammar rules for Type, CName, C_l, and C_c must be revised.

The new actions and rules are shown in **Figure 6.2** and should replace the rules and actions associated with the nonterminals, Type, CName, and C_I in the attribute grammar of **Appendix II**. Note that the attributes Type.value, CName.value, and C_I.id are assigned type representation pairs (as described in 6.2.1) to reflect the fact that the environment generated by the parse of a Mini-Dee program consists of a set of nodes that can be used to represent non-singleton classes. Additional rules and modified actions are emphasized with shading.

Type → $\varnothing$

`type.value := (type.context, e)`

| CName

`type.value := CName.value`

| ↑CName

`type.value := (CName.value.name, n)`

CName → IDEN

`CName.value := (IDEN.value, s)`

C_I → CLASS IDEN AList END IDEN

C_I`.env := AList.env`
C_I`.id := (IDEN.value, s)`
C_I`.unique_names := AList.unique_names`

`AList.context :=` C_I`.id`

Figure 6.2

The second revision necessary concerns the *bind* and *parm_bind* functions which perform binding checks. The behavior of the new binding check is summarized in Section 6.1.1. The entire attribute grammar defining type-safety on this version of Mini-Dee with non-singleton classes is given in **Appendix VI**.

6.2.3 Extending the Definition of Subclassing

It should be relatively straightforward to extend the definitions of subclassing, and subclass-related relations and functions to account for the new definition of nodes and the corresponding definition of type-safety over non-singleton classes. When defined over singleton classes in Section 5.5, the subclass relation ($\lhd$) was defined in terms of the *mapped system* relation, which in turn was defined in terms of the function, *map*. *Map* can be redefined in terms of the node representation presented in this chapter. The mapped system relation over such nodes has an equivalent definition to that of Section 5.5, but defined in terms of the new *map* definition. $\lhd_e$ is then defined as $\lhd$ was before, but in terms of the modified mapped system relation.

The new definition of *map* depends on the definition of an auxiliary function, $map_=$, which maps a function over an instance variable, local variable or parameter node. It is defined as:

$$map_=: ((\text{Class Identifier} \rightarrow \text{Class Identifier}) \times \text{Node}) \rightarrow \text{Node}$$

$$map_= (f, (\text{v}, n, (c_1, t), (c_2, s), \langle\!\langle\rangle\!\rangle, \varnothing)) = (\text{v}, n, (c_1, t), (f(c_2), s), \langle\!\langle\rangle\!\rangle, \varnothing)$$

where n is any name, c_1 and c_2 any class names and t is one of: e, s or n. Note that $map_=$ returns a node with identical type, though differing context.

As was the case with the original definition of *map* of Section 5.5, the definition of $map_=$ is extended over sets and sequences of nodes such that:

$$map_= (f, \{n_1, \ldots, n_k\}) = \{map_= (f, n_1), \ldots, map_= (f, n_k)\} \text{ and}$$

$$map_= (f, \langle\!\langle p_1, \ldots, p_k \rangle\!\rangle) = \langle\!\langle map_= (f, p_1), \ldots, map_= (f, p_k) \rangle\!\rangle$$

The node returned by the function *map*, depends on whether the input node is a variable or method, and on whether it has empty, singleton or non-singleton type. The definition is given by cases below:

i. method or variable nodes of singleton type:

Here, *map* is defined as *map* was defined over nodes, sets of nodes and sequences of nodes in Section 5.5.

$$map\ (f, (k, i, (c_1, t), (c_2, s), p, l)) = (k, i, (f(c_1), t), (f(c_2), s), map\ (f, p), map\ (f, l))$$

ii. instance variable, local variable and parameter nodes of non-singleton (or empty) type

$$map\ (f, (\mathrm{v}, i, (c_1, \mathrm{n}), (c_2, \mathrm{s}), «», \emptyset)) = map_{=}(f, (\mathrm{v}, i, (c_1, \mathrm{n}), (c_2, \mathrm{s}), «», \emptyset))$$

Though defined in terms of a node with non-singleton type, the same definition applies over variable nodes with empty type.

iii. method nodes of non-singleton (or empty) type

$$map\ (f, (\mathrm{m}, i, (c_1, \mathrm{n}), (c_2, \mathrm{s}), p, l)) = (\mathrm{m}, i, (f(c_1), \mathrm{n}), (f(c_2), \mathrm{s}), map_{=}(f, p), map_{=}(f, l))$$

Though defined in terms of a node with non-singleton type, the same definition applies over method nodes with empty type.

It should be noted that the definition for $\lhd_e$ only recognizes subclasses which can be defined in the explicit Mini-Dee grammar. For example, given a class A which contains a single method with interface,

```
method m : ↑C
```

then another class, A' is a subclass of A if it defines the same method with a return type which is a subset of the set $\{x \mid C \lhd_e x\}$. The only subclasses of A recognized by the definition of $\lhd_e$, are those with method definitions such as

```
method m : ↑C'     or     method m : C"
```

where $C \lhd_e C'$ and $C \lhd_e C''$. Note that if a class A', were to include a method definition for m such that its return type were the set of classes, {C', C"}, it would not be recognized as a subclass, despite the fact that it is one.

6.2.4 Extending the Definitions of Inheritance and Substitution

The definitions of $\lhd_e$ and mapped system over nodes of the kind introduced in this chapter, are identical to the definitions of $\lhd$ and mapped system given in Section 5.5, but defined over a modified *map* function. Similarly, inheritance and substitution should be defined over modified nodes as they are defined in Section 5.5, but in terms of a modified deep substitution operation ($\leftarrow_{\text{Deep}}$).

Palsberg and Schwartzbach claimed in [PaSc90d] that subclasses of non-singleton classes denoted by substitution did not preserve the type-safety of inherited code. However, two contradictory assumptions are made in [PaSc90d] concerning how class substitution is defined over non-singleton classes, and more specifically, non-singleton types. The first assumption can be inferred from a result that shows that subclassing preserves subtyping. From the proof of this result, it would appear that given any classes, $C_1, \ldots, C_n$, A, and B:

$$\{C_1, \ldots, C_n\}\,[A \leftarrow B] = \{C_1\,[A \leftarrow B], \ldots, C_n\,[A \leftarrow B]\}.$$

On the other hand, another section of [PaSc90d] describes how:

↑Object [Object ← Boolean] = ↑Boolean and that
↑Integer [Object ← Boolean] = ↑Integer.

The latter assumption is not stated explicitly, but can be inferred from the discussion of how PS subclassing does not preserve type-safety when applied to classes with infinite types. The text includes definitions for the two classes shown in **Example 6.2.**

```
class T
      var x : ↑Object
      var y : ↑Integer
      method m
            begin
                  x := y
            end
end T

class S
      var x : ↑Boolean
      var y : ↑Integer
      method m
            begin
                  x := y
            end
end S
```

Example 6.2

The claim is made that the second class is a subclass of the first, and discussions with one of the authors [Pa92] confirmed that this is because S is considered equivalent to T [Object ← Boolean]. The authors further conclude that the above example shows that class substitution cannot denote type-safe subclasses when applied to classes with infinite types, since the assignment of y to x satisfies the binding check in class T, but does not in class S.

The above two interpretations of class substitution are contradictory. If we assume the first interpretation, then

$$\uparrow\texttt{Object}\ [\texttt{Object} \leftarrow \texttt{Boolean}] \not\equiv \uparrow\texttt{Object}$$

and $\uparrow\texttt{Integer}\ [\texttt{Object} \leftarrow \texttt{Boolean}] \not\equiv \uparrow\texttt{Integer}$

In fact, applying the initial interpretation of class substitution to determine the meaning of `T [Object ← Boolean]` does yield a type-safe subclass. Consider the following reasoning:

1. What is `T [Object ← Boolean]`?

   By application of the algorithms presented in Chapter 5, since `Object` $\lhd_I$ `Boolean`, `T [Object ← Boolean]` = `T [Object` $\leftarrow_{\text{Deep}}$ `Boolean]` = `S`, where `S` is defined below:

   ```
   class S
           var x : ↑Object [Object ←Deep Boolean]
           var y : ↑Integer [Object ←Deep Boolean]
   end S
   ```

2. What is the type of x in `S`?

   The type of x is equivalent to the set of classes:

   { c ∈ ↑Object | Object ∉ *system*(c) }

   by the reasoning below:

   a. Partition ↑Object into sets *A* and *B* where:

   A = {c ∈ ↑Object | Object ∈ *system*(c)}
   B = {c ∈ ↑Object | Object ∉ *system*(c)}

   b. Clearly:

   - ↑Object [Object $\leftarrow_{\text{Deep}}$ Boolean] =
     A [Object $\leftarrow_{\text{Deep}}$ Boolean] ∪ *B* [Object $\leftarrow_{\text{Deep}}$ Boolean]

   - *B* [Object ← Boolean] = *B*

The latter point follows since `Object` $\lhd_1$ `Boolean` and for any $b, b \in B$, `Object` $\notin$ *system* (b)

c. To see what is A [`Object` $\leftarrow_{Deep}$ `Boolean`], consider any class $a \in A$:

- `Object` $\notin$ *system*(a [`Object` $\leftarrow_{Deep}$ `Boolean`])
- `Since` $a \in \uparrow$`Object, and` $a \lhd a$ [`Object` $\leftarrow_{Deep}$ `Boolean`], `then` a [`Object` $\leftarrow_{Deep}$ `Boolean`] $\in \uparrow$`Object`

Therefore, A [`Object` $\leftarrow_{Deep}$ `Boolean`] $\subseteq B$, and $\uparrow$`Object` [`Object` $\leftarrow_{Deep}$ `Boolean`] $= B$

3. What is the type of `y` in `S`?

The type of `y` is equivalent to the set of classes:

$\{$ `c` $\in \uparrow$`Integer` $|$ `Object` $\notin$ *system*(`c`) $\}$

by the reasoning below:

a. Partition $\uparrow$`Integer` into sets C and D where:

$C = \{$`c` $\in \uparrow$`Integer` $|$ `Object` $\in$ *system*(`c`)$\}$
$D = \{$`c` $\in \uparrow$`Integer` $|$ `Object` $\notin$ *system*(`c`)$\}$

b. By the same reasoning as was given in 2b:

- $\uparrow$`Integer` [`Object` $\leftarrow_{Deep}$ `Boolean`] =
  C [`Object` $\leftarrow_{Deep}$ `Boolean`] $\cup$ D [`Object` $\leftarrow_{Deep}$ `Boolean`]

- D [`Object` $\leftarrow_{Deep}$ `Boolean`] $= D$.

c. By the same reasoning as was given in 2c,

$$C \text{ [Object } \leftarrow_{Deep} \text{ Boolean] } \subseteq D \text{ and } \uparrow\text{Integer [Object } \leftarrow_{Deep} \text{ Boolean] } = D$$

Thus, the subclass `S = T [Object ← Boolean]` is:

```
class S
    var x : {c ∈ ↑Object | Object ∉ system(c)}
    var y : {c ∈ ↑Integer | Object ∉ system(c)}
    method assign ()
      begin
          x := y
      end
end S
```

and S preserves the binding checks satisfied in T.

There is no explicit description in [PaSc90d] of how substitution is applied to non-singleton sets of classes. However, two contradictory interpretations are assumed. Of these, the interpretation that defines the substitution of some classes, A ← B, over a set of classes to return another set of classes, where each class C in the initial set has as its image C [A ← B] in the resulting set, ensures that type-safety is preserved in subclasses of non-singleton classes.

The definition of the substitution operations, inherits, ←, and $\leftarrow_{Deep}$ of Chapter 5 can be extended over nodes that represent signatures with non-singleton type references to reflect this interpretation. But while this interpretation does achieve covariant signature redefinition (the types of x and y in S are subtypes of their corresponding types in T), it is not a very useful form of covariance. The resulting types of substitutions such as those of x and y above, are neither intuitively predictable, nor easy to express without greatly enriching the grammar defining type expressions (ie: they are not "useful" types). Further, the resulting subtypes are not the kind of subtypes usually desired from covariant redefinition. Considering the example above, no more behavior can be inferred of the types of x and y in S than could been have of their types in T. On the other hand, if the type of x could be specialized from *↑Object* to *↑Boolean*, then it could be statically inferred that any object bound to x would understand messages defined in the class Boolean. Finally, it should be noted that substitution over sets of classes does not yield a large range of subclasses. For example, the class `S` above would have resulted from the substitution of any class for `Object` and not just `Boolean`. When applied to a class `C` with non-singleton types, the substitution `C` $[A \leftarrow x]$ appears to give the same result no matter what the class denoted by x.

We have given thought to how to achieve the more powerful form of covariance that is usually desired and that no doubt motivated the second interpretation of substitution over sets of classes in [PaSc90d]. One proposal is presented along with its advantages and shortcomings, in the section that follows.

6.2.5 A Proposal for Useful, Intuitive Covariant Subclassing

A more desirable form of covariance would allow a type reference of $\uparrow$*Object* be specialized to be $\uparrow$*Integer* or $\uparrow$*Boolean* or some other set of classes for which more behavior can be statically inferred. In other words, it is desirable to be able to specialize the root class of the set of subclasses, since the variable and method names understood by the root class are also understood by their subclasses. We have seen that selective signature redefinition in subclasses demands contravariance. The PS approach which involves applying a redefinition operation to a class, demands consistency of redefinition but allows covariance. Therefore, the ideal subclassing mechanism should specialize type references intuitively, but via some kind of operation that can be applied to all type references consistently.

Since types are sets of classes, and subtyping is set inclusion, the most natural operation that results in specialized types is set intersection. '$\cap$' could be introduced to the language as an operation on types. The operation, $<_\cap$ ("specialize by intersection") could be introduced as an operation that when applied to a class, results in a subclass with type references specialized by intersection. An example follows that illustrates how $<_\cap$ could work. The class that is used as a superclass in this example is quite similar to class `T` of **Example 6.2**, but avoids the use of base classes in deference to the conclusions drawn in 5.6 establishing that base classes should have no subclasses. For this example, it should be assumed that there exists a class hierarchy as shown in **Figure 6.3**.

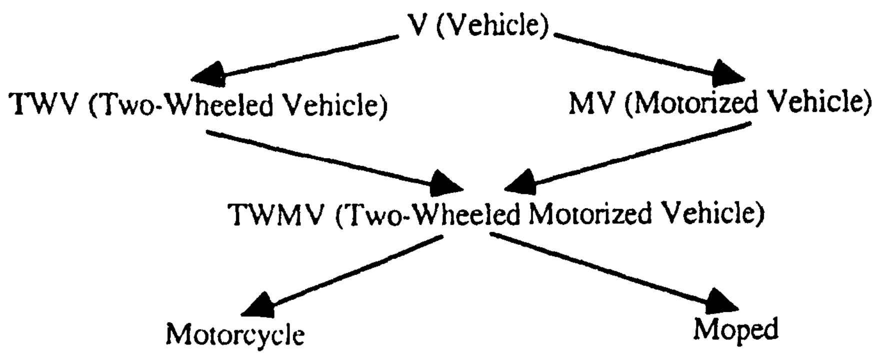

V (Vehicle)
TWV (Two-Wheeled Vehicle)
MV (Motorized Vehicle)
TWMV (Two-Wheeled Motorized Vehicle)
Motorcycle
Moped

Figure 6.3

where given classes A and B, $A \rightarrow B$ if $A \triangleleft_I B$.

Let class T be defined as in **Example 6.3**.

```
class T
        var x : ↑V
        var y : ↑MV
        method m
                begin
                        x := y
                end
end T
```

Example 6.3

Note that T is type-safe since the set of equivariant subclasses of `Vehicle` subsumes (is a supertype of) the set of equivariant subclasses of `Motorized_Vehicles`. If one wanted to denote a subclass of T where the type of x is specialized to be the set of equivariant subclasses of `Two_Wheeled_Vehicle`, then one would write:

```
class S = T <∩ ↑TWV
```

which denotes the class S defined in **Example 6.4**.

```
class S
        var x : ↑TWV
        var y : ↑TWMV
        method m
                begin
                        x := y
                end
end S
```

Example 6.4

↑Vehicle $\cap$ *↑Two_Wheeled_Vehicle* is *↑Two_Wheeled_Vehicle*, and *↑Two_Wheeled_Vehicle* $\cap$ *↑Motorized_Vehicle* is *↑Two_Wheeled_Motorized_Vehicle*. Class S is a type-safe subclass of T.

Besides achieving a more natural form of covariance over non-singleton classes than does class substitution, the addition of $<_\cap$ to the language machinery does not demand that the grammar for type expressions be enriched. $\cap$ can be defined over type expressions as shown in **Table 6.2**. It should be assumed while interpreting the table that A and B are class names and that T_1 and T_2 are type expressions. Each row of the table describes a different scenario depending on the forms that T_1 and T_2 can take.

T_1	T_2	$T_1 \cap T_2$
$\varnothing$	Anything	$\varnothing$
A	B	if A $\equiv$ B then A else $\varnothing$
A	↑B	if B $\lhd_e$ A then A else $\varnothing$
↑A	B	if A $\lhd_e$ B then B else $\varnothing$

↑A	↑B	if A $\lhd_e$ B then ↑B
		else if B $\lhd_e$ A then ↑A
		else ↑(*inherits* ({A,B}, $\varnothing$))*

Table 6.2

* This row shows how class S was derived from class T. *↑V* $\cap$ *↑TWV* is *↑TWV* because V $\lhd$ TWV. *↑MV* $\cap$ *↑TWV* is ↑(*inherits*({MV, TWV}, $\varnothing$) (or *↑TWMV*) because MV and TWV are not subclasses of one another.

The intersection of any two type expressions results in a type expression of the usual kind: $\varnothing$, A, or $\uparrow$A where A is a class identifier. Therefore, the addition of set intersection to the language machinery need not require enriching the grammar of type expressions.

Given the definition of $\cap$ over type expressions as described above, $<_\cap$ can be defined as a higher order operation on class identifiers. Therefore, we have $<_\cap$: Class Identifier $\times$ Type_Rep $\rightarrow$ Class Identifier defined for any class identifier C and type expression, T as:

$$
\begin{aligned}
&C <_\cap T = C' \text{ where} \\
&\quad Rep\,(system\,(C')) = \{\ (k, i, t \cap T, c <_\cap T, p', l') \mid \\
&\qquad\qquad \bullet\ (k, i, t, c, p, l) \in Rep\,(system\,(C)) \\
&\qquad\qquad \bullet\ p' = \{\ (x, (k', i', t' \cap T, c' <_\cap T, «», \varnothing)) \mid \\
&\qquad\qquad\qquad\qquad p\,(x) = (k', i', t', c', «», \varnothing))\ \} \\
&\qquad\qquad \bullet\ l' = \{\ (k', i', t' \cap T, c' <_\cap T, «», \varnothing) \mid \\
&\qquad\qquad\qquad\qquad (k', i', t', h', «», \varnothing) \in l\ \} \\
&\qquad \}
\end{aligned}
$$

As was the case with the definition of $\leftarrow_{\text{Deep}}$ given in Chapter 5, the above definition can be simplified by first extending $<_\cap$ over nodes, such that:

$$
\begin{aligned}
&(k, i, t, c, «p_1, \ldots, p_m», \{l_1, \ldots, l_n\}) <_\cap T = \\
&\qquad (k, i, t \cap T, c <_\cap T, «p_1 <_\cap T, \ldots, p_m <_\cap T», \{l_1 <_\cap T, \ldots, l_n <_\cap T\}).
\end{aligned}
$$

Then, $<_\cap$ can be redefined over classes as:

$$
\begin{aligned}
&C <_\cap T = C' \text{ where} \\
&\quad Rep\,(system\,(C')) = \{\ n <_\cap T \mid n \in Rep\,(system\,(C))\ \}
\end{aligned}
$$

The proof that $<_\cap$ preserves binding and parameter binding checks in subclasses is fairly straightforward. The binding of an expression of type O to a variable type V only holds if $V \supseteq O$. In any subclass formed by $<_\cap$, the types of the corresponding variable and expression will be $V \cap T$ and $O \cap T$, for some type T. Since $V \supseteq O$, then $V \cap T \supseteq O \cap T$, and the binding check holds in the subclass.

$<_\cap$ preserves signature checks in subclasses provided that the assumption is made that an expression whose type is the empty type, can be sent any message. We saw in Chapter 4, that such a property has to hold of any expression that can be bound to variables of any type, if guarantees of static type-safety are to be achieved. The problem arises again with the use of $<_\cap$ as a subclassing mechanism since resulting subclasses can contain type references that have been specialized to the empty type. The code of superclasses might include the sending of messages to expressions whose type is specialized to the empty type in subclasses.

Analysis of the table that summarizes how '$\cap$' is applied to type expressions reveals that empty types only arise in subclasses if the original type in the superclass was empty or a singleton type. It is only the latter case that is problematic, since only expressions of non-empty types would have been sent messages in the superclass code. Therefore, if a language is to avoid values whose types are empty, then $<_\cap$ should only be applicable to pure non-singleton classes (classes with no signatures with singleton types). Because of this, and because substitution appears only to be useful when applied to singleton classes, the ideal solution might be to not allow subclasses of "impure" non-singleton classes. Subclassing of singleton classes would be by inheritance and class substitution. Subclassing of pure non-singleton classes would be by specializing by intersection. It is not clear as yet what would be the benefits and consequences of such a segregation of classes. Among the questions that would have to be considered:

- Is there an intuitive division between objects whose classes should be defined as singleton classes and those whose classes should be non-singleton? Would it be obvious to a programmer when to use each?

- Should impure non-singleton classes be allowed with a restriction limiting the subclassing of such classes? Are such classes strictly necessary?

- Does the demand that classes strictly include or exclude non-singleton types provide any insight on the binary operator problem?

Questions such as these will provide the basis for further research.

6.3 A Summary of Short Term Goals

The previous section described two challenges to the extension of PS theory. The first challenge is to expand the rules for class composition to allow non-singleton classes, and hence, dynamic binding. It is fairly easy to extend the grammar for type expressions to allow for the infinite types of equivariant subclasses of some given class. As well, given such type expressions it is straightforward to define binding checks using superclass and class equivalence algorithms. With support for dynamic binding should also come support for class metamorphosis; the ability of an object to change its class dynamically. However, dynamic binding requirements are quite strict, requiring the class denoting the type of an object be an equivariant subclass of the class denting the type of the object to which it is bound. Equivariance requirements are strict, demanding that a subclass' interface be both contravariant and covariant versions of the interface of its superclass. It remains to be seen whether useful dynamic binding can coexist with covariant subclassing.

The challenge of defining subclassing mechanisms which generate type-safe subclasses of classes containing non-singleton types has been resolved, though not in an intuitively satisfying way. Class substitution and inheritance can be naturally extended as operations over non-singleton classes, and resulting subclasses are guaranteed to retain the type-safety of inherited code. However, the subtypes that are type references in subclasses are neither intuitive, nor easy to express without a rich grammar for type expressions. Further, no more behavior can be statically inferred of these types since the class which is a superclass of all other classes in the subtype, is the same class with that property in the superclass.

A more intuitive form of covariance was addressed as the follow-up extension of PS theory. Set intersection was proposed as an operation which could be applied to every type, and the "specialize by intersection" ($<_\cap$) operation was introduced as a subclassing mechanism that guarantees type-safe code reuse in subclasses, and which specializes type references in a manner usually desired by programmers. Much work has yet to be done with regard to this proposal. The practicality of the proposal is dependent on determining the usefulness of supporting a constant expression that can be bound to variables of all types, and which understands all messages. If such an expression is allowed, then the "specialize by intersection" operation might be practical in its present form. If such an expression is not allowed, then it is worth exploring whether classes should be segregated with singleton classes subclassed by inheritance and substitution, and non-singleton classes subclassed by "specialize by intersection."

6.4 Long Term Goals

Once the short term goals have been addressed, I would like to consider carrying extended PS theory into a working implementation. In particular, I would like to add the

PS type system and subclassing mechanisms to a typed object-oriented language such as Dee [Gr91]. Work in this area will bring up a number of related issues such as:

• Designing a browser that would present the user in some readable form, the interfaces from a system of classes at one time (and not just the interface from a single class). With such a browser, the user could preview the results of a particular substitution or inheritance.

• Designing a database that minimizes the space occupied by a class' definition. Ideally this database would manage two structures, as shown in **Figure 6.4.**

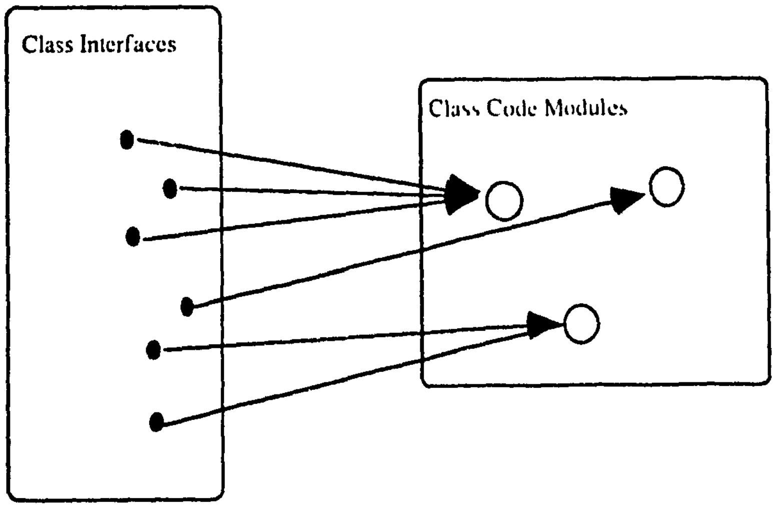


Figure 6.4

Given some function *f* that can map several class interfaces to a single class' code, an effective class would consist of some interface, *i* and its code image, *f(i)*. Compiled class code would be independent of its environments, and no duplication of code would be necessary. Subclassing mechanisms and browser operations would manipulate the class interface segment of the database only.

Naturally, long term concerns have been considered only to a limited degree since addressing long term goals is dependent on the resolution of short term goals.

Chapter 7.

Summary and Conclusion

7.1 Summary

The six chapters of this thesis can be partitioned according to the three objectives outlined in Chapter 1: analysis, survey and design.

7.1.1 Analysis

Chapter 2 contains a study of polymorphisms and classifies them while identifying the tensions associated with each classification. The Cardelli-Wegner taxonomy is used as a starting point for the classification. Each category in their hierarchy is distinguished according to the resulting nature of invocations to code. Their category of overloading is shown to result in a one-to-one mapping of invocation to code, and is renamed "nominal polymorphism" because of the sharing of operation name which is solely responsible for making operations polymorphic. The category of parametric polymorphism is shown to result in a many-to-one mapping of invocation to code and is termed "static binding" to include the equivalent polymorphism resulting from inheritance. Inclusion polymorphism is shown to result in a one-to-many mapping of invocation to code, and is referred to as "dynamic binding" because of the requirement that it be implemented using the mechanism of dynamic binding. The final Cardelli-Wegner category, coercion is not included in the final analysis as a form of polymorphism because of the spirit of its intent which demands that arguments be flexible, rather than operations.

In using the mappings of invocation to code as criteria determining natures of polymorphism, it becomes clear that inheritance typically achieves all forms of polymorphism, and not just inclusion. Nominal polymorphism results when a subclass redefines the code of inherited methods. Static binding results when a subclass does not redefine inherited methods or inherits variable definitions. Dynamic binding results when inheritance relations are used as the basis for determining what objects can be bound to a variable. Tensions are identified with each category of polymorphism, as each relates to the use of inheritance. These tensions are:

- Covariance and Selective Redefinition (for nominal polymorphism and static binding),
- Contravariance and Genericity (for static binding) and
- Covariance and Subclass-Based Type Interpretations

7.1.2 Survey

Chapters 3 and 4 contain detailed surveys of two language proposals, and examines how each proposal achieves the three forms of polymorphism and resolves the associated tensions. The Bruce language proposal is examined in Chapter 3. It is shown that nominal polymorphism and dynamic binding are achieved in this proposal. Dynamic binding is subclass-independent, since objects whose type is denoted by a subclass cannot necessarily be bound to variables declared of a type denoted by a superclass. Static binding is supported by way of prefixing, but not by way of genericity. Signature redefinition in subclasses must be accompanied by redefinition of code, and must be contravariant in its direction while selective in scope. (Except for recursive signatures which are covariant and consistently redefined).

The validity of a class is dependent not only on its type-safety, but on other more subtle criteria. Class code cannot bind recursively typed objects to variables whose type is not recursive, and therefore. code such as that which double dispatches is illegal. This restriction is a result of the coexistence of covariant signature redefinition (in the case of recursive types) and redefinition scope that is only consistent within a class. Classes also cannot have recursive instance variables because of limitations of the object model when applied to a language where objects have state. This precludes the creation of non-trivial data structures such as lists and trees. Classes are also invalid if they contain temporal cycles, and can only use single inheritance to borrow code from other classes.

In the final analysis, the Bruce approach successfully resolves the tension of dynamic binding, but offers very limited class composition and subclassing flexibility. The extent of the latter inflexibility is such as to lead Bruce to conclude that inheritance is a mechanism to be avoided in general.

In contrast to the Bruce proposal, the PS language allows for flexible subclassing redefinition, and somewhat more lenient class validity criteria, but only in the absence of dynamic binding. All forms of nominal polymorphism and static binding are supported, and all signature redefinition must be covariant in direction and consistent in scope. Consistent redefinition is achieved by inheritance and class substitution: redefinition operations on systems of classes that denote systems of subclasses. Class validity depends not only on type-safety, but also demands that classes not comprise temporal cycles and contain no constants, save for the constant that can be bound to variables of all types.

An intuitive proposal for dynamic binding is given, but is subclass-based and thus subject to tensions with covariant subclassing redefinition. As a result, PS subclassing has flexible redefinition, but is only well-defined over classes whose types preclude dynamic binding. Class validity criteria are less strict than in Bruce, but still disallow temporal cycles, constants, and only allow for the use of single inheritance.

7.1.3 Design

Chapters 5 and 6 constitute original design proposals. Chapter 5 contains an extension and reinterpretation of PS subclassing that we think offers more intuitive definitions of subclasses, inheritance and class substitution than the original proposal, and which is compatible with constants, multiple subclassing relations and inheritance and most temporal cycles. New definitions of subclassing and its associated operations are presented in terms of an alternative representation of classes, which are based on the interfaces rather than the code of a class. The alternative representation improves on the original representation, by making all class representations finite, and by defining subclassing relations such that any given class can have any number of superclasses. This chapter includes references to proofs found in the Appendices, which establish that inheritance and class substitution preserve type-safety in the subclasses they denote.

Chapter 6 contains a discussion of preliminary work being done, to add dynamic binding to the subclassing proposal of Chapter 5. Two issues are addressed in this chapter. A proposal for the addition of non-singleton types to PS classes, and the resulting definition for type-safety are presented first. Thereafter, it is shown that natural extensions of subclassing, inheritance and substitution definitions are such that inheritance and substitution result in type-safe subclasses of classes which include non-singleton types. Unfortunately, the types resulting from inheritance and substitution are neither intuitive nor very useful. An alternative subclassing mechanism is proposed, that is covariant and consistent and based on set intersection. Exploration of issues related to the use of this operation constitute the bulk of immediate demands for further study.

7.2 Conclusion

The important conclusions drawn from the work of this thesis can also be classified according to the three-part objectives outlined in Chapter 1. With regards to the analysis of polymorphism, we conclude that:

- **Inheritance is Typically Overloaded:**

Restrictive or unsafe inheritance found in most current typed object-oriented languages results from its overloaded use as a means of achieving all forms of polymorphism. Because each form of polymorphism has an associated tension, inheritance redefinition guidelines must either be made strict to resolve all tensions, or must ignore some tensions and be unsafe.

With regards to the survey of language proposals, we conclude that:

- **The Bruce Approach achieves Flexible Dynamic Binding:**

  The Bruce approach achieves flexible dynamic binding, but overly limited nominal polymorphism, static binding and flexibility of class composition.

- **The PS Approach Achieves Flexible Subclassing:**

  The PS approach achieves flexible nominal polymorphism and static binding, but as it stands in the original proposal, is incompatible with dynamic binding, is not intuitive and is overly strict in its criteria determining the validity of a class.

With regards to the design, we conclude that:

- **The PS Approach Can Be Made More Flexible:**

  The limitations of the PS approach with regard to how intuitive it is, and its strictness of class validity criteria, can be overcome by use of an alternate class representation, and by use of corresponding definitions for subclassing and subclassing operations.

- **The PS Approach Can Be Combined with Dynamic Binding:**

  The limitations of the PS approach with regard to dynamic binding reflect inattention to the tension of subclass-based dynamic binding and covariant subclassing, rather than to the inapplicability of inheritance and substitution operations to classes for which dynamic binding can take place. The tension can be resolved by employing a Bruce-like subclass independent interpretation of types. Subclassing operations do result in subclasses that while type-safe, are not particularly useful. Alternate subclass operations that are still covariant and consistent are worth exploration.

References

[Br92] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. Computer Science Department, Williams College. CS-92-01, 1992.

[CaWe85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17 (4), December 1985.

[Co89] William Cook. A proposal for making Eiffel type-safe. In *Proc. ECOOP'89, European Conference on Object-Oriented Programming*, 1989.

[CoHiCa89] William R. Cook, Walter L. Hill, and Peter S. Canning. F-bounded polymorphism for object-oriented programming. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, 1989.

[DaMyNy70] Ole-Johan Dahl, Bjørn Myrhaug and Kristen Nygaard. (Simula 67) Common base language. Publication N. S-22, Norsk Regnesentral (Norwegian Computing Center), Oslo, October 1970.

[DaTo88] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20 (1), March 1988.

[GoRo83] A. Goldberg and D, Robson. *Smalltalk-80-The Language and its Implementation*. Addison-Wesley, 1983.

[Gr91] Peter Grogono. Issues in the design of an object-oriented programming language. *Structured Programming*, 12 (1), January 1991.

[GrBe89] Peter Grogono and Anne Bennett. Polymorphism and type checking in object-oriented languages. *SIGPLAN Notices*, 24 (11), November 1989.

[HeJo90] Kurt J. Hebel and Ralph E. Johnson. Arithmetic and double dispatching in Smalltalk. *Journal of Object-Oriented Programming*, 2 (6), March/April 1990.

[Ho69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12 (10), October 1969.

[LiZi74] Barbara H. Liskov and Stephen N. Zilles. Programming with Abstract Data Types. Computation Structures Group, Memo no. 99, MIT, Project MAC, Cambridge (Mass.), 1974.

[Me88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[MiToHa90] B. Milner, M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Pa92] Jens Palsberg. Personal correspondence.

[PaSc90a] Jens Palsberg and Michael I. Schwartzbach. Type substitution for object-oriented programming. In *Proc. OOPSLA/ECOOP'90, ACM SIGPLAN Fifth Annual Conference on Object-Oriented Programming Systems, Languages and Applications; European Conference on Object-Oriented Programming*, 1990.

[PaSc91] Jens Palsberg and Michael I. Schwartzbach. What is type-safe code reuse? In *Proc. ECOOP'91, Fifth European Conference on Object-Oriented Programming*, 1991.

[PaSc90b] Jens Palsberg and Michael I. Schwartzbach. Genericity and Inheritance. Computer Science Department, Aarhus University. PB-318, 1990.

[PaSc90c] Jens Palsberg and Michael I. Schwartzbach. A unified type system for object-oriented programming. Computer Science Department, Aarhus Univers [illegible]. PB-341, 1990.

[PaSc90d] Jens Palsberg and Michael I. Schwartzbach. Static typing for object-oriented programming. Computer Science Department, Aarhus University. PB-355, 1990.

[Ho69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12 (10), October 1969.

[LiZi74] Barbara H. Liskov and Stephen N. Zilles. Programming with Abstract Data Types. Computation Structures Group, Memo no. 99, MIT, Project MAC. Cambridge (Mass.), 1974.

[Me88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[MiToHa90] B. Milner, M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Pa92] Jens Palsberg. Personal correspondence.

[PaSc90a] Jens Palsberg and Michael I. Schwartzbach. Type substitution for object-oriented programming. In *Proc. OOPSLA/ECOOP'90, ACM SIGPLAN Fifth Annual Conference on Object-Oriented Programming Systems, Languages and Applications; European Conference on Object-Oriented Programming*, 1990.

[PaSc91] Jens Palsberg and Michael I. Schwartzbach. What is type-safe code reuse? In *Proc. ECOOP'91, Fifth European Conference on Object-Oriented Programming*, 1991.

[PaSc90b] Jens Palsberg and Michael I. Schwartzbach. Genericity and Inheritance. Computer Science Department, Aarhus University. PB-318, 1990.

[PaSc90c] Jens Palsberg and Michael I. Schwartzbach. A unified type system for object-oriented programming. Computer Science Department, Aarhus University. PB-341, 1990.

[PaSc90d] Jens Palsberg and Michael I. Schwartzbach. Static typing for object-oriented programming. Computer Science Department, Aarhus University. PB-355, 1990.

[ScCoBuKiWi86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian and Carrie Wilpolt. An introduction to Trellis-Owl. In *Proc. OOPSLA'86, ACM SIGPLAN First Annual Conference on Object-Oriented Programming Systems, Languages and Applications; European Conference on Object-Oriented Programming,* 1986

[Ta91] A. Taivalsaari. Towards a taxonomy of inheritance mechanisms in object-oriented programming. Licentiate thesis, University of Jyväskylä (Finland), 1991.

[WeZd88] P. Wegner and S.B. Zdonik. Inheritance as an incremental modification mechanism or what like is and isn't like. In *Proc. ECOOP'88, European Conference on Object-Oriented Programming.* Springer-Verlag (*LNCS* 322), 1988.

Appendix I.

Grammars for the Language and Extensions of Mini-Dee

For the purposes of a theoretical discussion, the interface of a class is separated from the code implementing its methods. In fact, an implementation of Mini-Dee need not require the programmer to separate interface from code. Such an implementation though, will require a linker which while compiling a class A will allow for the use of the interfaces of supplier classes of A. The result is the same. The interfaces for all classes needed will be available to the compiler as it compiles each class.

The set of *n* classes which comprise a program in Mini-Dee are meant to correspond to a system of classes. It should be remembered that for any class A, *system* (A) corresponds to the set of all supplier classes of A as well as all suppliers of those classes, and so on.

The grammars below are in extended BNF form, and use the conventions of capitalizing nonterminals, boldfacing tokens which are keywords, and quoting tokens which are punctuation. All metasymbols are italicized. Braces, (*[* and *]*) are used to denote optional clauses within a rule. Ellipses (...) indicate 0 or more occurrences of the specified pattern. Numeric subscripts are used to distinguish between tokens or nonterminals which are found more than once in the same rule. Variations between extensions of Mini-Dee are emphasized by shading. Comments are prefixed with double dashes (--) as in Ada.

I-A. Core Mini-Dee (Chapter 1)

The core Mini-Dee language is informally described in Section 1.6, and is the skeleton language upon which all extensions are based.

(Program)

P $\rightarrow$ C_{I1} ... C_{In} C_{C1} ... C_{Cn}

(Class Interface)

C_I $\rightarrow$ **CLASS IDEN** *[* **INHERITS** IList *]* AList **END IDEN**

(Class Name)

CName $\rightarrow$ IDEN

(Inheritance List)

IList $\rightarrow$ CName -- rule for single inheritance

or

IList $\rightarrow$ $\text{CName}_1 \ldots \text{CName}_n$ -- rule for multiple inheritance

(Attribute List)

AList $\rightarrow$ $A_1 \ldots A_k$

(Attribute)

A $\rightarrow$ VAR IDEN ':' Type

| METHOD IDEN_0 ['(' IDEN_1 ':' $\text{Type}_1 \ldots \text{IDEN}_k$ ':' Type_k ')'] R
[VAR
IDEN_{k+1} ':' Type_{k+1}
...
IDEN_n ':' Type_n]

(Type)

Type $\rightarrow$ CName

(Result)

R $\rightarrow$ ':' Type
| ϵ

(Class Code)

C_C $\rightarrow$ $M_1 \ldots M_n$

(Methods)

M $\rightarrow$ METHOD IDEN B

(Method Body)

B $\rightarrow$ BEGIN S_1 ';' ... ';' S_n END

(Statement)

S → IDEN ':=' E
| RETURN E
| E '.' IDEN
| E_0 '.' IDEN '(' E_1 ... E_n ')'
| IF E THEN S_1 ELSE S_2

(Expression)

E → SELF
| NEW IDEN
| IDEN
| E '.' IDEN
| E_0 '.' IDEN *[* '(' E_1 ... E_n ')' *]*
| Con

(Constant)

Con → INTNUMBER
| FLOATNUMBER
| CHARACTER
| BOOLCONSTANT

I-B. Bruce-Based Mini-Dee (Chapter 3)

The following Mini-Dee extension is described informally in Section 3.2, and is used as the basis for a description of the Bruce language proposal ([Br92]) in Chapter 3. This extension supports single inheritance, and demands explicitly recursive type references with use of the special type name, MT.

(Program)

P → C_{I1} ... C_{In} C_{C1} ... C_{Cn}

(Class Interface)

C_I → CLASS IDEN *[* INHERITS IList *]* AList END IDEN

(Class Name)

CName → IDEN

(Inheritance List)

IList $\rightarrow$ CName -- rule for single inheritance

(Attribute List)

AList $\rightarrow$ A_1 ... A_k

(Attribute)

A $\rightarrow$ VAR IDEN ':' Type

| METHOD IDEN_0 *[* '(' IDEN_1 ':' Type_1 ... IDEN_k ':' Type_k ')' *]* R
[VAR
IDEN_{k+1} ':' Type_{k+1}
. . .
IDEN_n ':' Type_n *]*

(Type)

Type $\rightarrow$ CName
| MT

(Result)

R $\rightarrow$ ':' Type
| ϵ

(Class Code)

C_C $\rightarrow$ M_1 ... M_n

(Methods)

M $\rightarrow$ METHOD IDEN B

(Method Body)

B $\rightarrow$ BEGIN S_1 ';' ... ';' S_n END

(Statement)

S $\rightarrow$ IDEN ':=' E
| RETURN E
| E '.' IDEN
| E_0 '.' IDEN '(' E_1 ... E_n ')'
| IF E THEN S_1 ELSE S_2

(Expression)

E → SELF
| NEW IDEN
| IDEN
| E '.' IDEN
| E_0 '.' IDEN ['(' E_1 ... E_n ')']
| Con

(Constant)

Con → INTNUMBER
| FLOATNUMBER
| CHARACTER
| BOOLCONSTANT

I-C. PS-Based Mini-Dee (Chapter 4)

This extension of Mini-Dee is described informally in Section 4.2, and is used as the basis for a description of the PS type and subclassing proposals ([PaSc90a, PaSc90b, PaSc90c, PaSc90d, PaSc91]). The extension supports single inheritance, and class substitution. Constants expressions are disallowed so as to preserve the property that inheritance and class substitution denote type-safe subclasses.

(Program)

P → C_{I1} ... C_{In} C_{C1} ... C_{Cn}

(Class Interface)

C_I → CLASS IDEN [INHERITS IList] AList END IDEN

(Class Name)

CName → IDEN
$\boxed{\text{| CName}_1 \text{ '[' CName}_2 \text{ '}\leftarrow\text{' CName}_3 \text{ ']'}}$

(Inheritance List)

$\boxed{\text{IList} \rightarrow \text{CName} \quad \text{-- rule for single inheritance}}$

(Attribute List)

AList	$\rightarrow A_1 \ldots A_k$

(Attribute)

```
A       → VAR IDEN ':' Type
        | METHOD IDEN_0 [ '(' IDEN_1 ':' Type_1 ... IDEN_k ':' Type_k ')' ]  R
              [ VAR
                   IDEN_k+1 ':' Type_k+1
                   ...
                   IDEN_n ':' Type_n ]
```

(Type)

Type	$\rightarrow$ CName

(Result)

```
R       → ':' Type
        | ∈
```

(Class Code)

C_C	$\rightarrow M_1 \ldots M_n$

(Methods)

M	$\rightarrow$ METHOD IDEN B

(Method Body)

B	$\rightarrow$ BEGIN S_1 ';' ... ';' S_n END

(Statement)

```
S       → IDEN ':=' E
        | RETURN E
        | E '.' IDEN
        | E_0 '.' IDEN '(' E_1 ... E_n ')'
        | IF E THEN S_1 ELSE S_2
```

(Expression)

```
E        → SELF
         | NEW IDEN
         | IDEN
         | E '.' IDEN
         | E0 '.' IDEN [ '(' E1 ... En ')' ]
```

$\boxed{\text{Missing: Rules for Constants (Con), "E} \rightarrow \text{Con"}}$

I-D. Extended, Singleton Class, PS-Based Mini-Dee (Chapter 5)

This extension of Mini-Dee is informally described in Section 5.2 and resembles that of **I-C**, but supports the use of constants and multiple inheritance. Additions to this extension are made possible by the alternate interpretation and extension of PS subclassing presented in Chapter 5.

(Program)

P $\rightarrow C_{I1} \ldots C_{In} \;\; C_{C1} \ldots C_{Cn}$

(Class Interface)

C_I $\rightarrow$ CLASS IDEN [INHERITS IList] AList END IDEN

(Class Name)

CName $\rightarrow$ IDEN

$\boxed{\mid \; \text{CName}_1 \;\text{'['}\; \text{CName}_2 \;\text{'}\leftarrow\text{'}\; \text{CName}_3 \;\text{']'}}$

(Inheritance List)

$\boxed{\text{IList} \quad \rightarrow \text{CName}_1 \ldots \text{CName}_n \quad \text{-- rule for multiple inheritance}}$

(Attribute List)

AList $\rightarrow A_1 \ldots A_k$

(Attribute)

```
A          → VAR IDEN ':' Type
           | METHOD IDEN$_0$ [ '(' IDEN$_1$ ':' Type$_1$ ... IDEN$_k$ ':' Type$_k$ ')' ]  R
                 [ VAR
                       IDEN$_{k+1}$ ':' Type$_{k+1}$
                       ...
                       IDEN$_n$ ':' Type$_n$ ]
```

(Type)

```
Type       → CName
```

(Result)

```
R          → ':' Type
           | ∊
```

(Class Code)

```
C$_C$      → M$_1$ ... M$_n$
```

(Methods)

```
M          → METHOD IDEN  B
```

(Method Body)

```
B          → BEGIN  S$_1$ ';'  ...  ';' S$_n$  END
```

(Statement)

```
S          → IDEN ':=' E
           | RETURN E
           | E '.' IDEN
           | E$_0$ '.' IDEN  '(' E$_1$ ... E$_n$ ')'
           | IF E THEN S$_1$ ELSE S$_2$
```

(Expression)

```
E          → SELF
           | NEW IDEN
           | IDEN
           | E '.' IDEN
           | E$_0$ '.' IDEN  [  '(' E$_1$ ... E$_n$ ')' ]
           | Con
```

(Constant)

```
Con        → INTNUMBER
           | FLOATNUMBER
           | CHARACTER
           | BOOLCONSTANT
```

I-E. PS-Based Mini-Dee - Explicit Classes Only (Type-Safety Defined) (Chapter 5, Appendix II)

This extension resembles the PS-Based extension of **I-D**, but does not include inheritance and class substitution. **Appendix II** consists of an attribute grammar built from this grammar that formally defines type-safety. Inheritance and class substitution are reintroduced and formalized in Chapter 5, and shown to denote systems of classes which preserve the type-safety of their superclasses.

(Program)

P $\rightarrow$ C_{I1} ... C_{In} C_{C1} ... C_{Cn}

(Class Interface)

C_I $\rightarrow$ CLASS IDEN AList END IDEN

(Class Name)

CName $\rightarrow$ IDEN

(Attribute List)

AList $\rightarrow$ A_1 ... A_k

(Attribute)

```
A          → VAR IDEN ':' Type

           | METHOD IDEN_0 [ '(' IDEN_1 ':' Type_1 ... IDEN_k ':' Type_k ')' ]   R
                 [ VAR
                         IDEN_k+1 ':' Type_k+1
                         ...
                         IDEN_n ':' Type_n ]
```

(Type)

Type	$\rightarrow$ CName

(Result)

R	$\rightarrow$ ':' Type
	| ϵ

(Class Code)

C_C	$\rightarrow$ M_1 ... M_n

(Methods)

M	$\rightarrow$ METHOD IDEN B

(Method Body)

B	$\rightarrow$ BEGIN S_1 ';' ... ';' S_n END

(Statement)

S	$\rightarrow$ IDEN ':=' E
	| RETURN E
	| E '.' IDEN
	| E_0 '.' IDEN '(' E_1 ... E_n ')'
	| IF E THEN S_1 ELSE S_2

(Expression)

E	$\rightarrow$ SELF
	| NEW IDEN
	| IDEN
	| E '.' IDEN
	| E_0 '.' IDEN ['(' E_1 ... E_n ')']

Missing: Rules for Constants (Con), "E $\rightarrow$ Con", Rules for inheritance (IList), and Substitution (CName)

I-F. Extended, Non-Singleton Class, PS-Based Mini-Dee (Chapter 6)

This extension is PS-based and resembles the extension of **I-E** except that types can be non-singleton or empty, and not just singleton. This extension is informally described in Section 6.1.3, and is used to describe a preliminary proposal for including dynamic binding within PS classes.

(Program)

P → C_{I1} ... C_{In} C_{C1} ... C_{Cn}

(Class Interface)

C_I → CLASS IDEN *[* INHERITS IList *]* AList END IDEN

(Class Name)

CName → IDEN

$\boxed{\mid \ \text{CName}_1 \ \text{'['} \ \text{CName}_2 \ \text{'}\leftarrow\text{'} \ \text{CName}_3 \ \text{']'}}$

(Inheritance List)

$\boxed{\text{IList} \quad \rightarrow \text{CName}_1 \ldots \text{CName}_n \quad \text{-- rule for multiple inheritance}}$

(Attribute List)

AList → A_1 ... A_k

(Attribute)

A → VAR IDEN ':' Type

```
  | METHOD IDEN₀ [ '(' IDEN₁ ':' Type₁ ... IDENₖ ':' Typeₖ ')' ] R
      [ VAR
          IDENₖ₊₁ ':' Typeₖ₊₁
          ...
          IDENₙ ':' Typeₙ ]
```

(Type)

$\boxed{\text{Type} \quad \rightarrow \text{'}\varnothing\text{'}}$
| CName
$\boxed{\quad\quad\quad\quad | \ \uparrow\text{CName}}$

(Result)

R → ':' Type
| ϵ

(Class Code)

C_C → $M_1 \ldots M_n$

(Methods)

M → METHOD IDEN B

(Method Body)

B → BEGIN S_1 ';' ... ';' S_n END

(Statement)

S → IDEN ':=' E
| RETURN E
| E '.' IDEN
| E_0 '.' IDEN '(' $E_1 \ldots E_n$ ')'
| IF E THEN S_1 ELSE S_2

(Expression)

E → SELF
| NEW IDEN
| IDEN
| E '.' IDEN
| E_0 '.' IDEN ['(' $E_1 \ldots E_n$ ')']
| Con

(Constant)

Con → INTNUMBER
| FLOATNUMBER
| CHARACTER
| BOOLCONSTANT

I-G. PS-Based Extended Mini-Dee - Explicit Classes Only (Type-Safety Defined) (Chapter 6, Appendix VI)

This extension of Mini-Dee is based on the extension of **I-F** just as the extension of **I-E** was based on the extension of **I-D**. Inheritance and class substitution are removed from the grammar, allowing one only to define explicit classes. An attribute supplemented version of this grammar is used in **Appendix VI** to define type-safety formally, over classes that allow for non-singleton types and hence, dynamic binding.

(Program)

P $\rightarrow$ $C_{I1} \ldots C_{In}$ $C_{C1} \ldots C_{Cn}$

(Class Interface)

C_I $\rightarrow$ CLASS IDEN AList END IDEN

(Class Name)

CName $\rightarrow$ IDEN

(Attribute List)

AList $\rightarrow$ $A_1 \ldots A_k$

(Attribute)

A $\rightarrow$ VAR IDEN ':' Type

| METHOD IDEN_0 *[* '(' IDEN_1 ':' Type_1 ... IDEN_k ':' Type_k ')' *]* R
  [VAR
    IDEN_{k+1} ':' Type_{k+1}
    . . .
    IDEN_n ':' Type_n *]*

(Type)

$\boxed{\text{Type} \rightarrow \text{'}\varnothing\text{'}}$
| CName
$\boxed{\mid\ \uparrow\text{CName}}$

(Result)

```
R          → ':' Type
           | ε
```

(Class Code)

```
Cc         → M1 ... Mn
```

(Methods)

```
M          → METHOD IDEN B
```

(Method Body)

```
B          → BEGIN  S1 ';' ... ';' Sn  END
```

(Statement)

```
S          → IDEN ':=' E
           | RETURN E
           | E '.' IDEN
           | E0 '.' IDEN '(' E1 ... En ')'
           | IF E THEN S1 ELSE S2
```

(Expression)

```
E          → SELF
           | NEW IDEN
           | IDEN
           | E '.' IDEN
           | E0 '.' IDEN [ '(' E1 ... En ')' ]
           | Con
```

(Constant)

```
Con        → INTNUMBER
           | FLOATNUMBER
           | CHARACTER
           | BOOLCONSTANT
```

Appendix II.

Type-Safety: An Attribute Grammar of PS-Based, Singleton Classed, Mini-Dee

The Mini-Dee extension on which type-safety is defined is that of **Appendix I-D**. This grammar is of the PS-based extension which defines the structure of all explicitly defined classes (ie: without inheritance and substitution).

The attribute grammar has two components: environment generation and type checking. The environment generation is part of the parse of class interfaces, and for any class C consists of generating the environment *Rep* (C). The type checking uses the information contained within the environment to ensure that the code associated with each method declared satisfies binding and parameter binding checks (which in PS-Based Mini-Dee, are identical) and signature checks. In the context of the attribute supplemented grammar for Mini-Dee which follows, a program $P = \{C_{I1}, ..., C_{In}, C_{C1}, ..., C_{Cn}\}$ is type-safe if the parse of P results in the attribute, P.type_safe being TRUE.

II.A. Type Definitions:

```
Type_Rep = Class Identifier

node = (kind : {m,v},
        name : String,
        type : Type_Rep,
        context : Type_Rep,
        parms : Seq (node),
        locals : P (node))
```

II.B. Functions:

II.B.1. Environment Search Functions

Environment search operations are defined in terms of domain calculus, since they involve searches of symbol-table like data structures.

```
function locals_of (n : String, c : Type_Rep, e : P (node)) : P (node) | undefined

-- return set of local variable nodes associated with node with name, n and
-- defined in class, c in the environment, e

begin
    if ∃ p, l, t • signature ((m, n, t, c, p, l), e) then
        (l | ∃ p, t • signature ((m, n, t, c, p, l), e))
    else
        undefined
end
```

Note that the environment database has as its primary key, name + context. Actions of the parser that build the environment ensure that no two main nodes have the same name and context (although, nodes with the same node and context might be found in the parm or local fields of nodes).

```
function parms_of (n : String, c : Type_Rep, e : P (node)) : P (node) | undefined

-- return set of parameter nodes associated with node with name, n and
-- defined in class, c in the environment, e

begin
    if ∃ p, l, t • signature ((m, n, t, c, p, l), e) then
        (p (i) | ∃ l, t • signature ((m, n, t, c, p, l), e)) ∧  1 ≤ i ≤ |p|)
    else
        undefined
end
```

```
function type_of (n : String, c : Type_Rep, e : P (node)) : Type_Rep | undefined

-- return type associated with node with name, n and
-- defined in class, c in the environment, e

begin
    if ∃ p, l, t, k • signature ((k, n, t, c, p, l), e) then
        (t | ∃ l, p, k • signature ((k, n, t, c, p, l), e))
    else
        undefined
end
```

```
function mtype_of (n : String, c : Type_Rep, e : P (node)) : Type_Rep | undefined

-- return type associated with method node with name, n and
-- defined in class, c in the environment, e

begin
     if ∃ p, l, t • signature ((m, n, t, c, p, l), e) then
          (  | ∃ l, p • signature ((m, n, t, c, p, l), e))
     else
          undefined
end
```

```
function vtype_of (n : String, c : Type_Rep, e : P (node)) : Type_Rep | undefined

-- return type associated with variable node with name, n and
-- defined in class, c in the environment, e

begin
     if ∃ p, l, t • signature ((v, n, t, c, p, l), e) then
          (t | ∃ l, p • signature ((v, n, t, c, p, l), e))
     else
          undefined
end
```

```
function common ($e_1$, $e_2$ : P (node)) : Bool

-- returns TRUE if a node in $e_1$ shares its name with a node in $e_2$

begin
     ($\exists$ $k_1$, $k_2$, $t_1$, $t_2$, $c_1$, $c_2$, $p_1$, $p_2$, $l_1$, $l_2$, n $\bullet$
          signature (($k_1$, n, $t_1$, $c_1$, $p_1$, $l_1$), $e_1$)      $\wedge$
          signature (($k_1$, n, $t_1$, $c_1$, $p_1$, $l_1$), $e_2$))
end
```

```
function $\uplus$ ($e_1$, $e_2$ : P (node)) : P(node)

-- infix function: returns the set of nodes in $e_1 \cup e_2$, minus those
-- nodes in $e_1$ which share their names with nodes in $e_2$

begin
    if $e_1 = \emptyset$ then
        $e_2$
    else   -- $e_1 = \{(k, i, t, c, p, l)\} \cup R$
        if common ({(k, i, t, c, p, l)}, $e_2$) then
            $R \uplus e_2$
        else
            $\{(k, i, t, c, p, l)\} \cup (R \uplus e_2)$
end
```

II.B.2. Type-Safety Check Functions

Bind and Parameter Bind Checks:

```
function bind (v, o : Type_Rep) : Bool
    -- return TRUE if an object of type, o can be bound to a variable
    -- of type, v
begin
     ((v = o)  ∧   (v ≠ undefined))
end
```

Signature Check:

```
function signature ((k, n, t, c, a, l) : node, e : P (node)) : node | undefined
    -- return node in e, with kind, k; name, n; type, t; context, c and same
    -- numbers of parameters as a; or undefined


begin
    if ∃ p, l' • ((k, n, t, c, p, l') ∈ e  ∧  |a| = |p| )  then
         ((k, n, t, c, p, l') ∈ e  ∧  |a| = |p| )
    else
         undefined
end
```

Signature + Parameter Binding Check:

```
function sig_and_parmbind (n : node, e : P (node)) : Bool


    -- return TRUE n is in e, and binding of arguments is valid
var
    t : node
begin
    t := signature (n, e)
    if t ≠ undefined then
         ∀i • 1 ≤ i ≤ |n.parms| •
                bind (vtype_of (t.parms (i)), vtype_of (n,parms (i)))
    else
         false
end
```

II.C. The Attribute Grammar

The following is an attribute supplemented version of the Mini-Dee extension grammar of **Appendix I-D**.

(Program)

$$P \rightarrow C_{I1} \ldots C_{In} \quad C_{C1} \ldots C_{Cn}$$

$$\texttt{P.env} := \bigcup_{j=1}^{n} C_{Ij}\texttt{.env}$$

$$\forall j \bullet 1 \le j \le n \bullet C_{Cj}\texttt{.env} := \texttt{P.env}$$
$$\forall j \bullet 1 \le j \le n \bullet C_{Cj}\texttt{.id} := C_{Ij}\texttt{.id}$$

$$\texttt{P.type\_safe} := \bigwedge_{j=1}^{n} C_{Ij}\texttt{.unique\_names} \quad \wedge$$
$$\bigwedge_{j=1}^{n} C_{Cj}\texttt{.type\_safe}$$

Notes:

- The code for C_{Cj} corresponds to the interface for C_{Ij}
- Each block of code must be associated with its corresponding interface via the attribute, id.
- P is type safe if every feature in a given class interface is uniquely named and all class code blocks are type-safe with respect to the environment
- Note that given some class C with class interface C_I, C_I.env = *Rep* (C)

(Class Interface)

C_I → CLASS IDEN AList END IDEN

```
C_I.env := AList.env
C_I.id := IDEN.value
C_I.unique_names := AList.unique_names

AList.context := C_I.id
```

Note:

- IDEN.value contains the identifier name (in both cases here, the name of the class)

(Class Name)

CName → IDEN

```
CName.value := IDEN.value
```

(Attribute List)

AList → $A_1 \dots A_k$

```
∀j • 1 ≤ j ≤ k • A_j.context := AList.context
AList.env := {A_j.node | 1 ≤ j ≤ k}
AList.unique_names :=
  |{id | ∃ a, t, c, p, l, j • A_j.node = (a, id, t, c, p, l) }| = k
```

Note:

- Check on assignment of TRUE to AList.unique_names makes sure that no two attributes are declared with the same name

(Attribute)

A → VAR IDEN ':' Type

```
Type.context := A.context
A.node := (v, IDEN.value, Type.value, A.context, «», Ø)
```

| **METHOD** $\mathbf{IDEN}_0$ *[* '(' $\mathbf{IDEN}_1$ ':' Type_1 ... $\mathbf{IDEN}_k$ ':' Type_k ')' *]* R
[**VAR**
$\mathbf{IDEN}_{k+1}$ ':' Type_{k+1}
...
$\mathbf{IDEN}_n$ ':' Type_n *]*

```
∀j • 1 ≤ j ≤ n • Type_j.context := A.context
R.context := A.context
pnodes :=
 {(j, (v, IDEN_j.value, Type_j.value, A.context, «», ∅)) |
   1 ≤ j ≤ k}
lnodes :=
  {(v, IDEN_j.value, Type_j.value, A.context, «», ∅) |
   k+1 ≤ j ≤ n}

A.node := (m, IDEN_0.value, R.type, A.context, pnodes, lnodes)
```

(Type)

Type $\rightarrow$ CName

```
Type.value := CName.value
```

(Result)

R $\rightarrow$ ':' Type

```
Type.context := R.context
R.type := Type.value
```

| ϵ

```
R.type := R.context
```

Note:

– Methods without an explicitly declared return type are assumed to return `self`

(Class Code)

$C_C \rightarrow M_1 \ldots M_n$

$\forall j \bullet 1 \le j \le n \bullet M_j.\text{env} := C_C.\text{env}$
$\forall j \bullet 1 \le j \le n \bullet M_j.\text{context} := C_C.\text{id}$

$$C_C.\text{type\_safe} := \bigwedge_{j=1}^{n} M_j.\text{type\_safe}$$

M → METHOD IDEN B

```
B.env := M.env
B.lenv := locals_of (IDEN.value, M.context, M.env)
B.penv := parms_of (IDEN.value, M.context, M.env)
B.expected_type := mtype_of (IDEN.value, M.context, M.env)
B.context := M.context
```

```
M.type_safe :=
  ∃ r, p • signature ((m, IDEN.value, r, M.context, p, ∅), M.env)
     ∧ ¬Common (M.lenv, M.penv)
     ∧ B.type_safe
```

Note:

- B's environment consists of three parts – the global environment inherited from M, and the environments of parameters and local variables found in the entry in M.env corresponding to the method named IDEN.value. Parameters and local variables cannot share the same name, but can share the same name as variables or methods found in the global environment.

(Method Body)

B $\rightarrow$ **BEGIN** S_1 ';' ... ';' S_n **END**

```
∀ j • 1 ≤ j ≤ n • Sj.env := B.env
∀ j • 1 ≤ j ≤ n • Sj.lenv := B.lenv
∀ j • 1 ≤ j ≤ n • Sj.penv := B.penv
∀ j • 1 ≤ j ≤ n • Sj.context := B.context
∀ j • 1 ≤ j ≤ n • Sj.return_type := B.expected_type
```

$$B.type\_safe := \bigwedge_{j=1}^{n} S_j.type\_safe$$

Note:

- The body of a method is type-safe if each statement in the body is type-safe and if the expected return type is what is returned by the statements

(Statement)

S $\rightarrow$ **IDEN** ':=' E

```
E.env := S.env
E.lenv := S.lenv
E.penv := S.penv

E.context := S.context
S.type_safe :=
   bind (vtype_of (IDEN.value, S.context, S.env ⊎ S.lenv),
         E.type)
```

Note:

- Any variable receiving an assignment must be a parameter, local variable, or instance variable declared in the enclosing class, and must be declared of the same type as the entity it is assigned.

| **RETURN** E

```
E.env := S.env
E.lenv := S.lenv
E.penv := S.penv

E.context := S.context
S.type_safe := bind (S.return_type, E.type)
```

| E '.' IDEN

```
E.env := S.env
E.lenv := S.lenv
E.penv := S.penv

E.context := S.context

S.type_safe :=
   ∃ x • signature ((x, IDEN.value, E.type, E.type, «» ∅),
                    S.env ⊎  (S.lenv ∪ S.penv)
```

Note:

- Any reference to an instance variable or method call with no parameters (x can refer to m or v) must have a corresponding declaration node with context, E.type.

| E_0 '.' IDEN '(' E_1 ... E_n ')'

```
∀ j •  0 ≤ j ≤ n • Ej.env := S.env
∀ j •  0 ≤ j ≤ n • Ej.lenv := S.lenv
∀ j •  0 ≤ j ≤ n • Ej.penv := S.penv

∀ j •  0 ≤ j ≤ n • E  context := S.context

p := { (j, (v, "", Ej.type, S.context, «», ∅)) | 1 ≤ j ≤ n}

S.type_safe :=
  sig_and_parmbind ((m, IDEN.value, E0.type, E0.type, p, ∅),
S.env)
```

Note:

- Any method call with n parameters must have a corresponding declaration node with context, E_0.type (signature check), and with parameter types the same as the types of the corresponding arguments (parameter binding check).

| IF E THEN S_1 ELSE S_2

```
S.type_safe := E.type ≠ undefined
               ∧ S1.type_safe
               ∧ S2.type_safe
```

(Expression)

E → **SELF**

```
E.type := E.context
```

| **NEW IDEN**

```
E.type := vtype_of (IDEN.value, E.context, E.env ⊎ E.lenv)
```

Note:

- One can assign new objects to local and instance variables, but not to parameters

| **IDEN**

```
E.type :=
    vtype_of (IDEN.value, E.context, E.env ⊎ (E.lenv ∪ E.penv))
```

Note:

- Variables referred to as expressions can be instance variables found in E.context, or local variables or parameters declared in the current method

| E_1 '.' **IDEN**

E_1`.env := S.env`
E_1`.lenv := S.lenv`
E_1`.penv := S.penv`

E_1`.context := S.context`
`S.type := type_of (IDEN.value, `E_1`.type, E.env ⊎ E.lenv)`

Note:

- Any reference to an instance variable or method call with no parameters (x can refer to m or v) must have a corresponding declaration node with context, E.type.

| E_0 '.' IDEN '(' E_1 ... E_n ')'

```
∀ j •  0 ≤ j ≤ n • E_j.env := S.env
∀ j •  0 ≤ j ≤ n • E_j.lenv := S.lenv
∀ j •  0 ≤ j ≤ n • E_j.penv := S.penv

∀ j •  0 ≤ j ≤ n • E_j.context := S.context

p := { (j, (v, "", E_j.type, S.context, «», ∅)) | 1 ≤ j ≤ n}

E.type :=
  if ∃ t •
    sig_and_parmbind ((m, IDEN.value, t, E_0.type, p, ∅), E.env)
         then t
         else undefined
```

Note:

– Any method call with n parameters must have a corresponding declaration node with context, E_0.type (signature check), and with parameter types the same as the types of the corresponding arguments (parameter binding check).

Appendix III.

Proof: Mapped Systems Preserve Type-Safety

III.A. Lemma 3.1:

Given node n and mapping function f:

$$(f\,(type\_of\,(n)) = (type\_of\,(map\,(f, n)))$$

<u>**Proof:**</u>

Consider $n = (k, i, t, c, p, l)$.

We have:

$$type\_of\ (n) = t$$
$$map\ (f, n) = (k, i, f\,(t), f\,(c), map\ (f, p), map\ (f, l))$$

Thus:

$$f\,(type\_of\,(n)) = f\,(t) = type\_of\,(map\,(f, n))$$

III.B. Lemma 3.2:

Let *system* (A_1) = {$A_1, \ldots, A_r$} be a system of classes, and
f: Class Identifier $\rightarrow$ Class Identifier, a total function over the domain, *system* (A)

Let environment $\rho_1 = Rep\ (system\ (A_1))$, and $\rho_2 = map\ (f, \rho_1)$.

Let e be a well-formed expression used in the code for an arbitrary method q, found in the code for an arbitrary member of *system* (A_1), A_j with type t in environment ρ_1.

Then e has type $f(t)$ in environment ρ_2.

Proof:

Let T_A be the parse tree of the code for *system* (A_1), generated from the grammar of Mini-Dee and decorated with attribute values calculated from environment, ρ_1.

Let T_B be the corresponding parse tree of the code for *system* (A_1), but decorated with attribute values calculated from environment, ρ_2.

Let S_A and S_B be subtrees of T_A and T_B respectively, corresponding to the parse of the expression *e*. Note that S_A and S_B are identical trees (as are T_A and T_B), save for the values of the attributes of the nonterminal tree nodes. The nonterminal labels of nodes of S_B will be distinguished from those of S_A by the affixing of primes ('). Thus the root of S_A is labeled E while that of S_B is E'. Nonterminals in the grammar corresponding to node labels will be italicized. Thus, a possible rule determining the structure of the top-most levels of S_A and S_B might be $E \longrightarrow$ SELF.

The proof is an induction proof on the depth n, of S_A and S_B.

<u>Base Case:</u> $n = 1$

Given that the depth of S_A and S_B is 1, the grammar rule which determines the structure of the subtrees must be one of the following:

(1) $E \longrightarrow$ SELF

(2) $E \longrightarrow$ NEW IDEN

(3) $E \longrightarrow$ IDEN

Attribute values inherited by E and E' are easily determined, given the grammar for the entire tree and the information provided above:

E.context = A_3
E'.context = $f(A_3)$

E.env = ρ_1
E'.env = ρ_2

For some class, A_k, some sequence of nodes, p and some set of nodes, l:

- $(m, q, A_k, A_j, p, l) \in \rho_1$

Since $\rho_2 = map\ (f, \rho_1)$:

- $(m, q, f(A_k), f(A_j), map\ (f, p), map\ (f, l)) \in \rho_2$

Therefore:

E.penv $= p$
E.lenv $= l$

E'.penv $= map\ (f, p)$
E'.lenv $= map\ (f, l)$

Case (1) $E \rightarrow$ **SELF**

E.type = E.context $= A_j$
E'.type = E'.context $= f(A_j) = f$ (E.type)

Case (2) $E \rightarrow$ **NEW IDEN**

Since e is well-formed, then for some class, t:

- (v, **IDEN**.value, t, A_j, «», $\varnothing) \in \rho_1 \cup l$

and
- E.type $= t$

Since $\rho_2 = map\ (f, \rho_1)$:

- (v, **IDEN**.value, $f(t), f(A_j)$, «», $\varnothing) \in \rho_2 \cup map\ (f, l)$

Since E'.context $= f(A_j)$, it follows that E'.type $= f(t) = f$ (E.type)

Case (3) $E \rightarrow$ IDEN

Since e is well-formed, then for some class, t :

- (v, IDEN.value, t, A_j, «», $\varnothing$) $\in$ $\rho_1 \cup l \cup range\ (p)$

and

- E.type = t

Since $\rho_2 = map\ (f, \rho_1)$:

- (v, IDEN.value, f(t), f(A_j), «», $\varnothing$) $\in$
  $\rho_2 \cup map\ (f, l) \cup range\ (map\ (f, p))$

Since E'.context = f(A_j), it follows that E'.type = f(t) = f(E.type)

Inductive Hypothesis:

For some $k \geq 1$, **Lemma 3.2** is established for all values of $n \leq k$.

Inductive Step:

Consider subtrees S_A and S_B with depth, $k + 1$.

Since these subtrees have depth > 1, the rule which determines their structure must be one of:

(1) $E \rightarrow E_1$ '.' IDEN
(2) $E \rightarrow E_0$ '.' IDEN '('E_1 ... E_m ')'

A case analysis follows:

Case (1) $E \rightarrow E_1$ '.' IDEN

Since e is well-formed, then for some class, t:

- (x, IDEN.value, t, E_1.type, «», ∅) $\in \rho_1$

and
- E.type = t

Since $\rho_2 = map\ (f, \rho_1)$:

- (v, IDEN.value, f (t), f (E_1.type), «», ∅) $\in \rho_2$

By the inductive hypothesis:

- E'_1.type = f (E_1.type)

Therefore it follows that E'.type = f (t) = f (E.type)

Case (2) $E \rightarrow E_0$ '.' IDEN '(' E_1 ... E_z ')'

Since e is well-formed, then for some class, t, some sequence of nodes, p, some set of nodes, l and for x = 'm' or x = 'v':

- (x, IDEN.value, t, E_0.type, p, l) $\in \rho_1$
- $\forall j \bullet 1 \leq j \leq z \bullet E_j$.type = $type\_of\ (p\ (j))$

and
- E.type = t

Since $\rho_2 = map\ (f, \rho_1)$:

- (x, IDEN.value, f (t), f (E_0.type), $map\ (f,p)$, $map\ (f,l)$) $\in \rho_2$

Also, by the inductive hypothesis:

$\forall j \bullet 0 \leq j \leq z \bullet E'_j$.type = f (E_j.type)

By **Lemma 3.1**:

$$\forall j \bullet 1 \leq j \leq z \bullet f(type\_of(p(j))) = type\_of(map(f, p(j)))$$

Therefore it follows that:

$$\forall j \bullet 1 \leq j \leq z \bullet E'_j.\text{type} = type\_of(map(f, p(j)))$$

and that:

$$E'.\text{type} = f(\texttt{t}) = f(E.\text{type})$$

III.C. Theorem 3.1

For any class A, given that the code for *system* (A), *c* is type-safe with respect to the interface of *system* (A), *Rep* (*system* (A)), then *c* will be type-safe with respect to the interface of any mapped system of *system* (A).

Proof:

Let *system* (A_1) = {A_1, ..., A_k}

Let *c* = {C_1, ..., C_k} be the corresponding blocks of code for the classes listed in *system* (A_1). Assume *c* to be type-safe with respect to the environment (interface) ρ_1 = *Rep* (*system* (A_1)).

Let *system* (B) be a mapped system of *system* (A_1) by the mapping function, *f*.

Let $\rho_2 = map\ (f, \rho_1)$. ρ_2 is a subset of the environment produced by the interfaces of *system* (B), *Rep* (*system* (B)). Note that ρ_2 might be larger than *Rep* (*system* (B)) because:

(1) *system* (B) might contain more classes than the actual range of f

or (2) for any class $A_i \in$ *system* (A_1), it will be the case that $|Rep\ (A_i)| < |Rep\ (f\ (A_i))|$ if $f\ (A_i)$ has added methods or instance variables not found in A_i.

A proof that c is type-safe with respect to ρ_2 is sufficient to prove that c is type-safe with respect to *Rep* (*system* (B)), since the former is a subset of the latter.

From the grammar of Mini-Dee it is clear that a program is type-safe if each of its class code blocks is type-safe, and a class code block is type-safe if each of its methods is type-safe. Thus it will be sufficient to prove that an arbitrary method, m found in an arbitrary member of c, C_j with corresponding interface, *Rep* (*system* (A_j)), is type-safe with respect to ρ_2.

Given the parse tree of c, generated from the grammar of Mini-Dee and decorated with attribute values calculated from ρ_1, let S_A be the subtree of this parse tree corresponding to the parse of method m. Let S_B be the corresponding subtree showing the parse of m decorated with attribute values calculated from ρ_2. Note that S_A and S_B have nodes labeled by the nonterminals of the grammar and are identical in structure, differing only by the attribute values attached to their respective nodes. The labels of nodes of S_B will be distinguished from those of S_A by the affixing of primes ('). Thus the root of S_A is labeled, M while that of S_B is M'. Nonterminals in the grammar corresponding to node labels will be italicized. Thus, the rule determining the structure of the top-most levels of S_A and S_B is $M \rightarrow$ METHOD IDEN B .

Attribute values inherited by M and M' are easily determined, given the grammar of Mini-Dee and the information provided above.

$$M.env = \rho_1$$
$$M'.env = \rho_2$$

$$M.context = A_j$$
$$M'.context = f(A_j)$$

Level 1: Root nodes: M, and M'. Rule: $M \rightarrow$ **METHOD IDEN** B

Thus, the attribute values attached to B and B' are:

$$B.env = \rho_1$$
$$B'.env = \rho_2$$

$$B.context = A_j$$
$$B'.context = f(A_j)$$

Since M.type_safe holds, then for some class A_h ($1 \leq h \leq k$), some sequence of nodes, p and some set of nodes, l:

- (m, IDEN.value, A_h, A_j, p, l) $\in \rho_1$

Since $\rho_2 = map\ (f, \rho_1)$:

- (m, IDEN.value, $f(A_h)$, $f(A_j)$, $map\ (f,p)$, $map\ (f,l)$) $\in \rho_2$

Thus:

B.lenv = l
B'.lenv = *map* (f, l)

B.penv = p
B'.penv = *map* (f, p)

B.extype = r
B'.extype = $f(r)$

Since M.type_safe holds, then so too does common (M.lenv, M.penv). Since *map* introduces no new attribute names and changes no old names, it is also the case that common (M'.lenv, M'.penv) holds. Since (m, IDEN.value, $f(A_n)$, $f(A_j)$, *map* (f,p), *map* (f,l)) $\in \rho_2$, it need only be shown that B'.type_safe holds to show that M'.type_safe holds.

Level 2: Root nodes: B and B'. Rule: $B \rightarrow$ BEGIN S_1 ';' ... ';' S_n END

A method body is type-safe if each of the statements contained within the body is type-safe. Thus, in showing that an arbitrary statement, s found in m is type-safe with respect to ρ_2, it is also demonstrated that B'.type_safe holds. From the rule for B , we can determine the attribute values passed to the corresponding nodes for s:

S.env = ρ_1
S'.env = ρ_2

S.lenv = l
S'.lenv = *map* (f, l)

S.penv = p
S'.penv = *map* (f, p)

S.context = A_j
S'.context = $f(A_j)$

S.extype = r
S'.extype = $f(r)$

Level 3: Root nodes: S and S'.
Rules:

(1) $S \rightarrow$ IDEN ':=' E
(2) $S \rightarrow E$ '.' IDEN
(3) $S \rightarrow E_0$ '.' IDEN '(' E_1 ... E_n ')'
(4) $S \rightarrow$ RETURN E
(5) $S \rightarrow$ IF E THEN S_1 ELSE S_2

Case (1) - $S \rightarrow$ IDEN ':=' E

Since S.type_safe holds:

- (v, IDEN.value, E.type, A_j, «», $\varnothing$) $\in \rho_1 \cup l$

Since $\rho_2 = map\ (f, \rho_1)$:

- (v, IDEN.value, f(E.type), $f(A_j)$, «», $\varnothing$) $\in \rho_2 \cup map\ (f, l)$

By **Lemma 3.2**, we know:

- E'.type = f (E.type)

Therefore: S'.type_safe holds.

Case (2) - $S \rightarrow E$ '.' IDEN

Since S.type_safe holds, then for x = 'm' or x = 'v'

- (x, IDEN.value, E.type, E.type, «», $\varnothing$) $\in \rho_1$

Since $\rho_2 = map\ (f, \rho_1)$:

- (v, IDEN.value, f (E.type), f (E.type), «», $\emptyset$) $\in \rho_2$

By **Lemma 3.2**, we know:

- E'.type $= f$ (E.type)

Therefore: S'.type_safe holds.

Case (3) - $S \rightarrow E_0$ '.' IDEN '(' E_1 ... E_n ')'

Since S.type_safe holds, then for x = 'm' or x = 'v':

- (x, IDEN.value, E_0.type, E_0.type, p, l) $\in \rho_1$
- $\forall z \bullet 1 \leq z \leq n \bullet E_z$.type = $type\_of\ (p\ (z))$

Since $\rho_2 = map\ (f, \rho_1)$:

- (v, IDEN.value, f (E_0.type), f (E_0.type), $map\ (f, p)$, $map\ (f, l)$) $\in \rho_2$

By **Lemma 3.2**, we know:

- E'_0.type $= f$ (E_0.type)

By extending the result of **Lemma 3.1** over sequences of nodes, we know:

- $\forall z \bullet 1 \leq z \leq n \bullet type\_of\ (map\ (f, p\ (z)) = f\ (type\_of\ (p\ (z)))$

and thus that:

- $\forall z \bullet 1 \leq z \leq n \bullet E'_z$.type = $type\_of\ (map\ (f, p)\ (z))$

Therefore: S'.type_safe holds.

Case (4) - $S \rightarrow$ RETURN E

Since S.type_safe holds:

E.type = S.extype = r

By **Lemma 3.2**, we know that:

E'.type = f (E.type) = f (r) = S'.extype

Therefore: S'.type_safe holds.

Case (5) - $S \rightarrow$ IF E THEN S_1 ELSE S_2 FI

We know since S.type_safe holds, that E.type $\neq$ *undefined*, and that S_1.type_safe and S_2.type_safe both hold.

From **Lemma 3.2**, we know that since E.type = t for some class, t, then E'.type is not *undefined* but is instead, f(t).

That S'.type_safe then holds follows from an inductive proof on the number, n of nested **IF** statements that form the parse tree of S.

Base Case: $n = 0$

In this case, both S_1 and S_2 are statements of one of the forms discussed in Cases (1) to (4). We know from the preceding case analysis that if S_1.type_safe and S_2.type_safe both hold, then S'_1.type_safe and S'_2.type_safe also hold. Therefore, S'.type_safe holds.

Inductive Hypothesis:

For some $k \geq 0$, then $n \leq k$ is sufficient a condition to ensure that S.type_safe implies S'.type_safe.

Inductive Step:

Assume $n = k + 1$.

In this case, the number of nested **IF** statements forming S_1 and S_2 are at most n-1. By the inductive S_1.type_safe and S_2.type_safe imply S'_1.type_safe and S'_2.type_safe, and therefore S'.type_safe. Thus, if s is an **IF** statement, then it is type-safe with respect to ρ_2 provided that it is type-safe with respect to ρ_1.

All cases then considered, statement s is type-safe with respect to ρ_2 if type-safe with respect to ρ_1, and thus so too are m, C_j and c.

Appendix IV.

Proof: Class Substitution Denotes Mapped Systems

IV.A. Lemma 4.1: $A \lhd B \wedge B \lhd C \Rightarrow A \lhd C$

Proof:

By the definitions of $\lhd$ and *mapped system*, for some functions, *f* and *g*:

$$Rep\ (system\ (B)) \supseteq map\ (f, Rep\ (system\ (A)))$$
$$Rep\ (system\ (C)) \supseteq map\ (g, Rep\ (system\ (B)))$$

where *f* is a total function over domain, *system* (A) and with range, *system* (B), and *g* is a total function over domain, *system* (B) and with range, *system* (C).

By the transitivity of $\supseteq$:

$$Rep\ (system\ (C)) \supseteq map\ (g, map\ (f, Rep\ (system\ (A))))$$

and, $A \lhd C$ by $g \circ f$.

IV.B. Theorem 4.1: $B \lhd_I C \Rightarrow A \lhd A\,[B \leftarrow_{Deep} C]$

Proof:

This can be proven using a case analysis of the definition of $\leftarrow_{Deep}$:

Case 1: $A \cong B,\ A\,[B \leftarrow_{Deep} C] = C$

Since $B \lhd_I C$, trivially $B \lhd C$.
Since $A \cong B$, $A \lhd B$.

By the transitivity of $\lhd$ established in **Lemma 4.1**, $A \lhd C$ and thus $A \lhd A\,[B \leftarrow_{Deep} C]$

<u>Case 2:</u> $(B \equiv C \vee B \notin \mathit{system}(A))$, $A\,[B \leftarrow_{Deep} C] = A$

It is trivially the case that $A \lhd A$ by $\{(x,x) \mid x \in \mathit{system}(A)\}$.
Thus, $A \lhd A\,[B \leftarrow_{Deep} C]$.

<u>Case 3:</u> $A \not\equiv B, B \not\equiv C$, $B \in \mathit{system}(A)$,
$A\,[B \leftarrow_{Deep} C] = A'$
where $\mathit{Rep}(\mathit{system}(A')) = \{\, n\,[B \leftarrow_{Deep} C] \mid n \in \mathit{Rep}(\mathit{system}(A))\}$

Note that since $B \in \mathit{system}(A)$:

- $\mathit{system}(B) \subseteq \mathit{system}(A)$.
- $\mathit{Rep}(\mathit{system}(B)) \subseteq \mathit{Rep}(\mathit{system}(A))$

By the definition of $\leftarrow_{Deep}$, for every node in $\mathit{Rep}(\mathit{system}(A))$, there is a corresponding node in $\mathit{Rep}(\mathit{system}(A'))$.

Let n be any node in $\mathit{Rep}(\mathit{system}(A))$, and t any type reference found in n.
Then there is a corresponding node, $n' \in \mathit{Rep}(\mathit{system}(A'))$ with a corresponding type reference to t, t'. There are three possible ways that t' is determined from t.

<u>Case 3a:</u> $n \notin \mathit{Rep}(\mathit{system}(B))$,
$n \in (\mathit{Rep}(\mathit{system}(A)) - \mathit{Rep}(\mathit{system}(B)))$

In this case, n' is determined by the definition of $\leftarrow_{Deep}$, and $t' = t\,[B \leftarrow_{Deep} C]$

<u>Case 3b:</u> $n \in \mathit{Rep}(\mathit{system}(B))$,
$n \notin (\mathit{Rep}(\mathit{system}(A)) - \mathit{Rep}(\mathit{system}(B)))$

In this case, n' is determined by the relationship between B and C. Since $B \lhd_I C$, then $B \lhd C$ by $\{(x, x[B \leftarrow_{Deep} C]) \mid x \in \mathit{system}(B)\}$, and thus $t' = t\,[B \leftarrow_{Deep} C]$.

Case 3c: $n \in Rep\ (system\ (\text{B}))$,
$n \in (Rep\ (system\ (\text{A})) - Rep\ (system\ (\text{B})))$

In this case, it is possible for two nodes, n' and n'' to be found in $Rep\ (system\ (\text{A}))$ where n' is determined from n by the definition of $\leftarrow_{\text{Deep}}$, and n'' is determined by the relationship between B and C. However, because B $\lhd_I$ C, in fact n' and n'' will be the same node, and $t' = t\ [\text{B} \leftarrow_{\text{Deep}} \text{C}]$.

It is Case 3c that demonstrates that B $\lhd$ C by f is not a sufficient condition by itself to guarantee that A $\lhd$ A [B $\leftarrow_{\text{Deep}}$ C], since it could be for some type, $s \neq t\ [\text{B} \leftarrow_{\text{Deep}} \text{C}]$, that $f(t) = s$..

IV.C. Lemma 4.2: $\text{A} \lhd \text{B}$ by $f = \{\ldots, (\text{D,E}), \ldots\} \Rightarrow \text{D} \lhd \text{E}$ by $g,\ g \subseteq f$

Proof:

Since $(\text{D,E}) \in f$:

- $\text{D} \in system\ (\text{A})$
- $system\ (\text{D}) \subseteq system\ (\text{A})$
- $\text{E} \in system\ (\text{B})$
- $system\ (\text{E}) \subseteq system\ (\text{B})$

Let $n = (k, i, t, \text{D}, «p_1, \ldots, p_j», \{l_1, \ldots, l_m\}) \in Rep\ (system\ (\text{D}))$. Then:

(1) • $t \in system\ (\text{D})$
(2) • $\forall z \bullet 1 \leq z \leq j \bullet type\_of\ (p_z) \in system\ (\text{D})$
(3) • $\forall z \bullet 1 \leq z \leq m \bullet type\_of\ (l_z) \in system\ (\text{D})$

Since $n \in Rep\ (system\ (\text{A}))$ also:

$$n' = map\ (f, n)$$
$$= (k, i, f(t), \text{E}, map\ (f, «p_1, \ldots, p_j»), map\ (f, \{l_1, \ldots, l_m\})) \in Rep\ (system\ (\text{B}))$$

It is clear from the context field of n' that $n' \in Rep\ (system\ (\text{E}))$ also.

Since $n \in Rep\ (system\ (\text{D}))$ and $n' \in Rep\ (system\ (\text{E}))$:

- $t \in system\ (\text{D})$
- $f(t) \in system\ (\text{E})$

Since $t \in system\ (\text{D})$, we can choose another node, q:

$$q = (k', i', t', t, «p_1', \ldots, p_j'», \{l_1', \ldots, l_m'\}) \in Rep\ (system\ (\text{D}))$$

We only require that $type\_of\ (q) \in system\ (\text{D})$. Thus q generalizes <u>any</u> node in $Rep\ (system\ (\text{D}))$. By the same reasoning as before:

$$q' = (k', i', f\ (t'), f\ (t), map\ (f, «p_1', \ldots, p_j'»), map\ (f, \{l_1', \ldots, l_m'\})) \in Rep\ (system\ (\text{E}))$$

And thus for any node, $n \in Rep\ (system\ (\text{D}))$

$$map\ (f, n) \in Rep\ (system\ (\text{E}))$$

Thus, at the very least, $\text{D} \lhd \text{E}$ by f. Naturally, we can narrow f to include only pairs whose first member belongs to $system\ (\text{D})$. Thus we have:

$$\text{D} \lhd \text{E} \text{ by } g = \{(x,y) \in f \mid x \in system\ (\text{D})\ \}$$

IV.D. Lemma 4.3: $\text{A} \lhd_I \text{B} \quad \Rightarrow \quad DOS\ (\text{A,B}) = 0$

Let DOS: (Class Identifier $\times$ Class Identifier) $\rightarrow Z^+$ be a function which returns the "Degree Of Substitution" for any (superclass, subclass) pair. Intuitively, DOS (A,B) will be 0 if $\text{A} \lhd_I \text{B}$, and will be n if it takes n substitutions on A to denote a class which is a parent of B. More formally:

For any classes A and B, where A $\lhd$ B by *f*:

DOS (A,B) =

$$\sum_{(x,y) \in f} \text{if } y \cong x \; [\mathrm{A} \leftarrow_{\mathrm{Deep}} \mathrm{B}] \text{ then } 0 \text{ else } 1$$

Proof:

By the definition of $\lhd_1$ –

- A $\lhd$ B by $\{(x, x\,[\mathrm{A} \leftarrow_{\mathrm{Deep}} \mathrm{B}]) \mid x \in \mathit{system}\,(\mathrm{A})\}$

The result follows.

IV.E. Lemma 4.4: A $\cong$ B $\Rightarrow$ *DOS* (A,B) = 0

Proof:

By the definition of $\cong$:

- A $\lhd$ B by *f* and B $\lhd$ A by *g*

Therefore:

- $\mathit{system}\,(\mathrm{A}) \supseteq \mathit{map}\,(f, \mathit{map}\,(g, \mathit{system}\,(\mathrm{A})))$

and:

- $f = g^{-1}$

By **Lemma 4.2**, for any pair $(x,y) \in f, x \lhd y$. Since $(y,x) \in g$:

- $y \lhd x$

and thus:

- $x \cong y$

Since $\text{A} \cong \text{B}$:

- $x = x\ [\text{A} \leftarrow_{\text{Deep}} \text{B}]\ \cong y$
- $f = \{(x, x\ [\text{A} \leftarrow_{\text{Deep}} \text{B}]\ ) \mid x \in \textit{system}\ (\text{A})\}$
- $\text{A} \lhd_{\text{I}} \text{B}$

By **Lemma 4.3**:

- $\textit{DOS}\ (\text{A},\text{B}) = 0.$

IV.F. Theorem 4.2: $\text{B} \lhd \text{C} \quad \Rightarrow \quad \text{A} \lhd \text{A}\ [\text{B} \leftarrow \text{C}]$

Theorem 4.2 will be proved by induction, on the value of $n = \textit{DOS}\ (\text{B}, \text{C})$. Note that since $\text{B} \lhd \text{C}$, $\textit{DOS}(\text{B}, \text{C})$ is defined.

<u>Base Case:</u> $n = 0$

There are three possible cases for determining the value of $\text{A}\ [\text{B} \rightarrow \text{C}]$, given that $n = 0$.

Case (1) – $\text{A} \cong \text{B},\ \text{A}\ [\text{B} \leftarrow \text{C}] = \text{C}$

Since $\text{A} \cong \text{B}$:

- $\text{A} \lhd \text{B}$

By the transitivity of $\lhd$ established in **Lemma 4.1**, and since $B \lhd C$:

- $A \lhd C$

Case (2) – $B \cong C$, $A\,[B \leftarrow C] = A$

Trivially, $A \lhd A$ by $\{(x, x) \mid x \in \mathit{system}\ (A)\ \}$

Case (3) - $B \lhd_I C$

By **Theorem 4.1**, $A \lhd A\,[B \leftarrow_{\text{Deep}} C]$

Inductive Hypothesis:

For some $k \geq 0$, the theorem holds for all $n \leq k$.

Inductive Step: $n = k, k > 0$.

There are three possible cases for determining the value of $A\,[B \rightarrow C]$, given that $n > 0$. The first two cases are the same as Case (1) and Case (2) shown in the proof of the base case. The third case is:

Case (3) – Given $B \lhd C$ by $f = \{\ldots, (D,E), \ldots\}$, $E \ncong D\,[B \leftarrow_{\text{Deep}} C])$

$$A\,[B \leftarrow C] = A\,[D \leftarrow E]\,\big[B\,[D \leftarrow E] \longleftarrow C\big]$$

By **Lemma 4.2**, $D \lhd E$ by $g \subseteq f$. Therefore it follows:

- $B\,[D \leftarrow E] \lhd C$ by $f\text{–}g$

Since $(D,E) \in f$ and $(D,E) \in f\text{–}g$:

- $g \subset f$
- $f - g \subset f$

Since $\mathrm{E} \not\trianglelefteq \mathrm{D}\ [\mathrm{B} \leftarrow_{\mathrm{Deep}} \mathrm{C}]$):

- $DOS\ (\mathrm{B}\ [\mathrm{D} \leftarrow \mathrm{E}], \mathrm{C}) < DOS\ (\mathrm{B},\mathrm{C})$

By the inductive hypothesis:

- $\mathrm{A}\ [\mathrm{D} \leftarrow \mathrm{E}] \triangleleft \mathrm{A}\ [\mathrm{D} \leftarrow \mathrm{E}] \left[\mathrm{B}\ [\mathrm{D} \leftarrow \mathrm{E}] \longleftarrow \mathrm{C}\right]$

Since $\mathrm{D} \triangleleft \mathrm{E}$ by g (and by the inductive hypothesis):

- $\mathrm{A} \triangleleft \mathrm{A}\ [\mathrm{D} \leftarrow \mathrm{E}]$

By the transitivity of $\triangleleft$ established in **Lemma 4.1**,

- $\mathrm{A} \triangleleft \mathrm{A}\ [\mathrm{D} \leftarrow \mathrm{E}] \left[\mathrm{B}\ [\mathrm{D} \leftarrow \mathrm{E}] \longleftarrow \mathrm{C}\right]$

Appendix V.
Mini-Dee Parser Output

V.A Base Classes

```
Input File: Tests:1 - Base Classes:Example 1

Input

----------------------------------------------------

====================================================


Output

----------------------------------------------------
Index: 1
Class: Object
Aliases:
           "Object"
State: Resolved
Nodes:



Index: 2
Class: Bool
Aliases:
           "Bool"
           "Bool [Comparable <- Bool]"
           "Bool [Comparable <- Order]"
           "Bool [Order <- Int]"
           "Bool [Order <- Float]"
           "Bool [Order <- Char]"
State: Resolved
Nodes:

  (m, and, 2*, 2, « (v, self, 2*, 2, «», {}) (v, other, 2*, 2, «», {}) », { })
  (m, or, 2*, 2, « (v, self, 2*, 2, «», {}) (v, other, 2*, 2, «», {}) », { })
  (m, not, 2*, 2, « (v, self, 2*, 2, «», {}) », { })
  (m, plus, 2*, 2, « (v, self, 2*, 2, «», {}) (v, other, 2*, 2, «», {}) », { })
  (m, zero, 2*, 2, « (v, self, 2*, 2, «», {}) », { })
  (m, equals, 2*, 2, « (v, self, 2*, 2, «», {}) (v, other, 2*, 2, «», {}) », { })
```

```
Index: 3
Class: Ring
Aliases:
        "Ring"
State: Resolved
Nodes:

  (m, plus, 3*, 3, « (v, self, 3*, 3, «», ()) (v, other, 3*, 3, «», ()) », ( ))
  (m, zero, 3*, 3, « (v, self, 3*, 3, «», ()) », ( ))


Index: 4
Class: Comparable
Aliases:
        "Comparable"
State: Resolved
Nodes:

  (m, equals, 2, 4, « (v, self, 4*, 4, «», ()) (v, other, 4*, 4, «», ()) », ( ))


Index: 6
Class: Int
Aliases:
        "Int"
State: Resolved
Nodes:

  (m, times, 6*, 6, « (v, self, 6*, 6, «», ()) (v, other, 6*, 6, «», ()) », ( ))
  (m, div, 6*, 6, « (v, self, 6*, 6, «», ()) (v, other, 6*, 6, «», ()) », ( ))
  (m, mod, 6*, 6, « (v, self, 6*, 6, «», ()) (v, other, 6*, 6, «», ()) », ( ))
  (m, lt, 2, 6, « (v, self, 6*, 6, «», ()) (v, other, 6*, 6, «», ()) », ( ))
  (m, gt, 2, 6, « (v, self, 6*, 6, «», ()) (v, other, 6*, 6, «», ()) », ( ))
  (m, lte, 2, 6, « (v, self, 6*, 6, «», ()) (v, other, 6*, 6, «», ()) », ( ))
  (m, gte, 2, 6, « (v, self, 6*, 6, «», ()) (v, other, 6*, 6, «», ()) », ( ))
  (m, equals, 2, 6, « (v, self, 6*, 6, «», ()) (v, other, 6*, 6, «», ()) », ( ))
  (m, plus, 6*, 6, « (v, self, 6*, 6, «», ()) (v, other, 6*, 6, «», ()) », ( ))
  (m, zero, 6*, 6, « (v, self, 6*, 6, «», ()) », ( ))


Index: 7
Class: Order
Aliases:
        "Order"
State: Resolved
Nodes:

  (m, lt, 2, 7, « (v, self, 7*, 7, «», ()) (v, other, 7*, 7, «», ()) », ( ))
  (m, gt, 2, 7, « (v, self, 7*, 7, «», ()) (v, other, 7*, 7, «», ()) », ( ))
  (m, lte, 2, 7, « (v, self, 7*, 7, «», ()) (v, other, 7*, 7, «», ()) », ( ))
  (m, gte, 2, 7, « (v, self, 7*, 7, «», ()) (v, other, 7*, 7, «», ()) », ( ))
  (m, equals, 2, 7, « (v, self, 7*, 7, «», ()) (v, other, 7*, 7, «», ()) », ( ))
```

```
Index: 10
Class: Float
Aliases:
          "Float"
State: Resolved
Nodes:
  (m, times, 10*, 10, « (v, self, 10*, 10, «», {}) (v, other, 10*, 10, «», {}) », { })
  (m, div, 10*, 10, « (v, self, 10*, 10, «», {}) (v, other, 10*, 10, «», {}) », { })
  (m, sqrt, 10*, 10, « (v, self, 10*, 10, «», {}) », { })
  (m, sin, 10*, 10, « (v, self, 10*, 10, «», {}) », { })
  (m, cos, 10*, 10, « (v, self, 10*, 10, «», {}) », { })
  (m, lt, 2, 10, « (v, self, 10*, 10, «», {}) (v, other, 10*, 10, «», {}) », { })
  (m, gt, 2, 10, « (v, self, 10*, 10, «», {}) (v, other, 10*, 10, «», {}) », { })
  (m, lte, 2, 10, « (v, self, 10*, 10, «», {}) (v, other, 10*, 10, «», {}) », { })
  (m, gte, 2, 10, « (v, self, 10*, 10, «», {}) (v, other, 10*, 10, «», {}) », { })
  (m, equals, 2, 10, « (v, self, 10*, 10, «», {}) (v, other, 10*, 10, «», {}) », { })
  (m, plus, 10*, 10, « (v, self, 10*, 10, «», {}) (v, other, 10*, 10, «», {}) », { })
  (m, zero, 10*, 10, « (v, self, 10*, 10, «», {}) », { })


Index: 12
Class: Char
Aliases:
          "Char"
State: Resolved
Nodes:
  (m, asc, 6, 12, « (v, self, 12*, 12, «», {}) », { })
  (m, plus, 12*, 12, « (v, self, 12*, 12, «», {}) (v, other, 12*, 12, «», {}) », { })
  (m, zero, 12*, 12, « (v, self, 12*, 12, «», {}) », { })
  (m, lt, 2, 12, « (v, self, 12*, 12, «», {}) (v, other, 12*, 12, «», {}) », { })
  (m, gt, 2, 12, « (v, self, 12*, 12, «», {}) (v, other, 12*, 12, «», {}) », { })
  (m, lte, 2, 12, « (v, self, 12*, 12, «», {}) (v, other, 12*, 12, «», {}) », { })
  (m, gte, 2, 12, « (v, self, 12*, 12, «», {}) (v, other, 12*, 12, «», {}) », { })
  (m, equals, 2, 12, « (v, self, 12*, 12, «», {}) (v, other, 12*, 12, «», {}) », { })

---------------------------------------------------
Class: Object is a subclass of:
 Object (1) by: { }

Class: Bool is a subclass of:
 Object (1) by: { }
 Bool (2) by: { (Bool, Bool) } Ring (3) by: { (Ring, Bool) }
 Comparable (4) by: { (Bool, Bool) (Comparable, Bool) }

Class: Ring is a subclass of:
 Object (1) by: { }
 Ring (3) by: { (Ring, Ring) }

Class: Comparable is a subclass of:
 Object (1) by: { }
 Comparable (4) by: { (Bool, Bool) (Comparable, Comparable) }

Class: Int is a subclass of:
 Object (1) by: { }
 Ring (3) by: { (Ring, Int) }
 Comparable (4) by: { (Bool, Bool) (Comparable, Int) }
 Int (6) by: { (Int, Int) (Bool, Bool) }
 Order (7) by: { (Bool, Bool) (Order, Int) }

Class: Order is a subclass of:
 Object (1) by: { }
 Comparable (4) by: { (Bool, Bool) (Comparable, Order) }
 Order (7) by: { (Bool, Bool) (Order, Order) }
```

```
Class: Float is a subclass of:
  Object (1) by: { }
  Ring (3) by: { (Ring, Float) }
  Comparable (4) by: { (Bool, Bool) (Comparable, Float) }
  Order (7) by: { (Bool, Bool) (Order, Float) }
  Float (10) by: { (Float, Float) (Bool, Bool) }

Class: Char is a subclass of:
  Object (1) by: { }
  Ring (3) by: { (Ring, Char) }
  Comparable (4) by: { (Bool, Bool) (Comparable, Char) }
  Order (7) by: { (Bool, Bool) (Order, Char) }
  Char (12) by: { (Int, Int) (Char, Char) (Bool, Bool) }

---------------------------------------------------
Class: Object is a child (subclass by inheritance) of:
  Object (1) by: { }

Class: Bool is a child (subclass by inheritance) of:
  Object (1) by: { }
  Bool (2) by: { (Bool, Bool) }
  Ring (3) by: { (Ring, Bool) }
  Comparable (4) by: { (Bool, Bool) (Comparable Bool) }

Class: Ring is a child (subclass by inheritance) of:
  Object (1) by: { }
  Ring (3) by: { (Ring, Ring) }

Class: Comparable is a child (subclass by inheritance) of:
  Object (1) by: { }
  Comparable (4) by: { (Bool, Bool) (Comparable, Comparable) }

Class: Int is a child (subclass by inheritance) of:
  Object (1) by: { }
  Ring (3) by: { (Ring, Int) }
  Comparable (4) by: { (Bool, Bool) (Comparable, Int) }
  Int (6) by: { (Int, Int) (Bool, Bool) }
  Order (7) by: { (Bool, Bool) (Order, Int) }

Class: Order is a child (subclass by inheritance) of:
  Object (1) by: { }
  Comparable (4) by: { (Bool, Bool) (Comparable, Order) }
  Order (7) by: { (Bool, Bool) (Order, Order) }

Class: Float is a child (subclass by inheritance) of:
  Object (1) by: { }
  Ring (3) by: { (Ring, Float) }
  Comparable (4) by: { (Bool, Bool) (Comparable, Float) }
  Order (7) by: { (Bool, Bool) (Order, Float) }
  Float (10) by: { (Float, Float) (Bool, Bool) }

Class: Char is a child (subclass by inheritance) of:
  Object (1) by: { }
  Ring (3) by: { (Ring, Char) }
  Comparable (4) by: { (Bool, Bool) (Comparable, Char) }
  Order (7) by: { (Bool, Bool) (Order, Char) }
  Char (12) by: { (Int, Int) (Char, Char) (Bool, Bool) }
---------------------------------------------------
```

V.B Example 2.7

```
Input File: Tests:2 - Turtle:Example 2

Input
-------------------------------------------------------
class Turtle

           var x         : Int
           var y         : Int
           var direction : Float

           method move (m : Int)
                       begin ... end

           method distance (a : Int  b : Int) : Float
                       begin ... end

           method turn (amt : Float)
                       begin ... end

           method set_pos (a : Int  b : Int)
                       begin ... end

           method within_one (a : Int  b : Int)
                       begin ... end

end Turtle
-------------------------------------------------------
class ScreenRange inherits Int

           var k : Int

end ScreenRange


-------------------------------------------------------
class WrappingTurtle inherits Turtle [Int <- ScreenRange]

end WrappingTurtle

-------------------------------------------------------

=======================================================
Output
-----------------------------------------------------
Index: 1
Graph: Object
Aliases:
           "Object"

Index: 2
Graph: Bool
Aliases:
           "Bool"
           "Bool [Comparable <- Bool]"
           "Bool [Comparable <- Order]"
           "Bool [Order <- Int]"
           "Bool [Order <- Float]"
           "Bool [Order <- Char]"
           "Bool [Int <- ScreenRange]"

Index: 3
Graph: Ring
Aliases:
           "Ring"

Index: 4
Graph: Comparable
Aliases:
           "Comparable"

Index: 6
Graph: Int
Aliases:
           "Int"

Index: 7
Graph: Order
Aliases:
           "Order"

Index: 10
Graph: Float
Aliases:
           "Float"
           "Float [Int <- ScreenRange "
           "Float [Turtle [Int <- ScreenRange  <- WrappingTurtle]"

Index: 12
Graph: Char
Aliases:
           "Char"


---------
Index: 14
Class: Turtle
Aliases:
          "Turtle"
State: Resolved
Nodes:

  (v, x, 6, 14, «», {})
  (v, y, 6, 14, «», {})
  (v, direction, 10, 14, «», {})
  (m, move, 14*, 14, « (v, se [illegible], 14*, 14, «», {}) (v, r, 6, 14, «», {})  », {  })
  (m, distance, 10, 14,
       « (v, self, 14*, 14, «», {})  (v, a, 6, 14, «», {})  (v, b, 6, 14, «», {})  »,
       {  })
  (m, turn, 14*, 14, « (v, self, 14*, 14, «», {})
  (v, amt, 10, 14, «», {})  », {  })
  (m, set_pos, 14*, 14,
       « (v, self, 14*, 14, «», {})  (v, a, 6, 14, «», {})  (v, b, 6, 14, «», {})  »,
       {  })
  (m, within_one, 14*, 14,
       « (v, self, 14*, 14, «», {})  (v, a, 6, 14, «», {})  (v, b, 6, 14, «», {})  »,
       {  })


Index: 15
Class: ScreenRange
Aliases:
       "ScreenRange"
       "Int [Int <- ScreenRange]"
       "ScreenRange [Turtle [Int <- ScreenRange] <- WrappingTurtle]"
State: Resolved
Nodes:

  (v, k, 6, 15, «», {})
  (m, times, 15*, 15, « (v, self, 15*, 15, «», {})  (v, other, 15*, 15, «», {})  », {  })
  (m, div, 15*, 15, « (v, self, 15*, 15, «», {})  (v, other, 15*, 15, «», {})  », {  })
  (m, mod, 15*, 15, « (v, self, 15*, 15, «», {})  (v, other, 15*, 15, «», {})  », {  })
  (m, lt, 2, 15, « (v, self, 15*, 15, «», {})  (v, other, 15*, 15, «», {})  », {  })
  (m, gt, 2, 15, « (v, self, 15*, 15, «», {})  (v, other, 15*, 15, «», {})  », {  })
  (m, lte, 2, 15, « (v, self, 15*, 15, «», {})  (v, other, 15*, 15, «», {})  », {  })
  (m, gte, 2, 15, « (v, self, 15*, 15, «», {})  (v, other, 15*, 15, «», {})  », {  })
  (m, equals, 2, 15, « (v, self, 15*, 15, «», {})  (v, other, 15*, 15, «», {})  », {  })
  (m, plus, 15*, 15, « (v, self, 15*, 15, «», {})  (v, other, 15*, 15, «», {})  », {  })
  (m, zero, 15*, 15, « (v, self, 15*, 15, «», {})  », {  })


Index: 18
Class: WrappingTurtle
Aliases:
          "Turtle [Int <- ScreenRange]"
          "WrappingTurtle"
State: Resolved
Nodes:

  (v, x, 15, 18, «», {})
  (v, y, 15, 18, «», {})
  (v, direction, 10, 18, «», {})
  (m, move, 18*, 18, « (v, self, 18*, 18, «», {})  (v, n, 15, 18, «», {})  », {  })
  (m, distance, 10, 18,
       « (v, self, 18*, 18, «», {})  (v, a, 15, 18, «», {})  (v, b, 15, 18, «», {})  »,
       {  })
  (m, turn, 18*, 18, « (v, self, 18*, 18, «», {})  (v, amt, 10, 18, «», {})  », {  })
  (m, set_pos, 18*, 18,
       « (v, self, 18*, 18, «», {})  (v, a, 15, 18, «», {})  (v, b, 15, 18, «», {})  »,
       {  })
  (m, within_one, 18*, 18,
       « (v, self, 18*, 18, «», {})  (v, a, 15, 18, «», {})  (v, b, 15, 18, «», {})  »,
       {  })

----------------------------------------------------
Class: Turtle is a subclass of:
 Object (1) by: { }
 Turtle (14) by: { (Int, Int) (Float, Float) (Turtle, Turtle) (Bool, Bool) }
```

```
Class: ScreenRange is a subclass of:
 Object (1) by: { }
 Ring (3) by: { (Ring, ScreenRange) }
 Comparable (4) by: { (Bool, Bool) (Comparable, ScreenRange) }
 Int (6) by: { (Int, ScreenRange) (Bool, Bool) }
 Order (7) by: { (Bool, Bool) (Order, ScreenRange) }
 ScreenRange (15) by: { (Int, Int) (ScreenRange, ScreenRange) (Bool, Bool) }

Class: WrappingTurtle is a subclass of:
 Object (1) by: { }
 Turtle (14) by: { (Int, ScreenRange) (Float, Float) (Turtle, WrappingTurtle) (Bool, Bool) }
 WrappingTurtle (18) by: { (ScreenRange, ScreenRange) (Float, Float)
                           (WrappingTurtle, WrappingTurtle) (Int, Int) (Bool, Bool) }

-------------------------------------------------------
Class: Turtle is a child (subclass by inheritance) of:
 Object (1) by: { }
 Turtle (14) by: { (Int, Int) (Float, Float) (Turtle, Turtle) (Bool, Bool) }

Class: ScreenRange is a child (subclass by inheritance) of:
 Object (1) by: { }
 Ring (3) by: { (Ring, ScreenRange) }
 Comparable (4) by: { (Bool, Bool) (Comparable, ScreenRange) }
 Int (6) by: { (Int, ScreenRange) (Bool, Bool) }
 Order (7) by: { (Bool, Bool) (Order, ScreenRange) }
 ScreenRange (15) by: { (Int, Int) (ScreenRange, ScreenRange) (Bool, Bool) }

Class: WrappingTurtle is a child (subclass by inheritance) of:
 Object (1) by: { }
 Turtle (14) by: { (Int, ScreenRange) (Float, Float) (Turtle, WrappingTurtle) (Bool, Bool)
 }
 WrappingTurtle (18) by: { (ScreenRange, ScreenRange) (Float, Float)
                           (WrappingTurtle, WrappingTurtle) (Int, Int) (Bool, Bool) }
-------------------------------------------------------
```

V.C Example 4.6

```
Input File: Tests:3 - Inheritance:Example 3

Input
--------------------------------------------------
class U
          var a : U
end U


--------------------------------------------------
class V inherits U
          var b : W
end V


-------------------------------------------------
class W
          var c : V
end W

-------------------------------------------------
class R inherits V
          var d : W
end R

--------------------------------------------------

==================================================


Output
-------------------------------------------------
Index: 1
Graph: Object
Aliases:
          "Object"

Index: 2
Graph: Bool
Aliases:
          "Bool"
          "Bool [Comparable <- Bool]"
          "Bool [Comparable <- Order]"
          "Bool [Order <- Int]"
          "Bool [Order <- Float]"
          "Bool [Order <- Char]"

Index: 3
Graph: Ring
Aliases:
          "Ring"

Index: 4
Graph: Comparable
Aliases:
          "Comparable"
```

```
Index: 6
Graph: Int
Aliases:
           "Int"

Index: 7
Graph: Order
Aliases:
           "Order"

Index: 10
Graph: Float
Aliases:
           "Float"

Index: 12
Graph: Char
Aliases:
           "Char"


---------
Index: 14
Class: U
Aliases:
           "U"
State: Resolved
Nodes:

       (v, a, 14*, 14, «», {})


Index: 15
Class: V
Aliases:
           "V"
State: Resolved
Nodes:

  (v, b, 16, 15, «», {})
  (v, a, 15*, 15, «», {})


Index: 16
Class: W
Aliases:
           "W"
State: Resolved
Nodes:

  (v, c, 15, 16, «», {})


Index: 17
Class: R
Aliases:
           "R"
State: Resolved
Nodes:

  (v, d, 16, 17, «», {})
  (v, b, 18, 17, «», {})
  (v, a, 17*, 17, «», {})
```

```
Index: 18
Class: W [V <- R]
Aliases:
           "W [V <- R]"
State: Resolved
Nodes:

  (v, c, 17, 18, «», {})


------------------------------------------------------
Class: U is a subclass of:
 Object (1) by: { }
 U (14) by: { (U, U) }

Class: V is a subclass of:
 Object (1) by: { }
 U (14) by: { (U, V) }
 V (15) by: { (W, W) (V, V) }

Class: W is a subclass of:
 Object (1) by: { }
 W (16) by: { (V, V) (W, W) }

Class: R is a subclass of:
 Object (1) by: { }
 U (14) by: { (U, R) }
 V (15) by: { (W, W [V <- R]) (V, R) }
 R (17) by: { (W, W) (W [V <- R], W [V <- R]) (R, R) (V, V) }

Class: W [V <- R] is a subclass of:
 Object (1) by: { }
 W (16) by: { (V, R) (W, W [V <- R]) }
 W [V <- R] (18) by: { (R, R) (W, W) (W [V <- R], W [V <- R]) (V, V) }

------------------------------------------------------
Class: U is a child (subclass by inheritance) of:
 Object (1) by: { }
 U (14) by: { (U, U) }

Class: V is a child (subclass by inheritance) of:
 Object (1) by: { }
 U (14) by: { (U, V) }
 V (15) by: { (W, W) (V, V) }

Class: W is a child (subclass by inheritance) of:
 Object (1) by: { }
 W (16) by: { (V, V) (W, W) }

Class: R is a child (subclass by inheritance) of:
 Object (1) by: { }
 U (14) by: { (U, R) }
 V (15) by: { (W, W [V <- R]) (V, R) }
 R (17) by: { (W, W) (W [V <- R], W [V <- R]) (R, R) (V, V) }

Class: W [V <- R] is a child (subclass by inheritance) of:
 Object (1) by: { }
 W (16) by: { (V, R) (W, W [V <- R]) }
 W [V <- R] (18) by: { (R, R) (W, W) (W [V <- R], W [V <- R]) (V, V) }
------------------------------------------------------
```

V.D Examples 5.7a and 5.7b

```
Input File: Tests:3 - Inheritance:Example 4
Input
-------------------------------------------------------
class B
            var b : B

            method m (other : B) : Object
                        begin ... end

end B
-------------------------------------------------------
class B' inherits B

            var i : Int
end B'
------------------------------------------------------

========================================= - - - - - - -

Output
------------------------------------------------------
Index: 1
Graph: Object
Aliases:
            "Object"
            "Object [B <- B']"

Index: 2
Graph: Bool
Aliases:
            "Bool"
            "Bool [Comparable <- Bool]"
            "Bool [Comparable <- Order]"
            "Bool [Order <- Int]"
            "Bool [Order <- Float]"
            "Bool [Order <- Char]"
```

```
Index: 3
Graph: Ring
Aliases:
          "Ring"

Index: 4
Graph: Comparable
Aliases:
          "Comparable"

Index: 6
Graph: Int
Aliases:
          "Int"

Index: 7
Graph: Order
Aliases:
          "Order"

Index: 10
Graph: Float
Aliases:
          "Float"

Index: 12
Graph: Char
Aliases:
          "Char"



---------
Index: 14
Class: B
Aliases:
          "B"
State: Resolved
Nodes:

  (v, b, 14*, 14, «», ())
  (m, m, 1, 14, « (v, self, 14*, 14, «», ())  (v, other, 14*, 14, «», ())  », (  ))


Index: 15
Class: B'
Aliases:
          "B'"
State: Resolved
Nodes:

  (v, i, 6, 15, «», ())
  (v, b, 15*, 15, «», ())
  (m, m, 1, 15, « (v, self, 15*, 15, «», ())  (v, other, 15*, 15, «», ())  », (  ))

----------------------------------------------------
Class: B is a subclass of:
 Object (1) by: ( )
 B (14) by: ( (B, B) (Object, Object) )
```

```
Class: B' is a subclass of:
 Object (1) by: { }
 B (14) by: { (B, B') (Object, Object) }
 B' (15) by: { (Int, Int) (B', B') (Object, Object) (Bool, Bool) }

---------------------------------------------------
Class: B is a child (subclass by inheritance) of:
 Object (1) by: { }
 B (14) by: { (B, B) (Object, Object) }

Class: B' is a child (subclass by inheritance) of:
 Object (1) by: { }
 B (14) by: { (B, B') (Object, Object) }
 B' (15) by: { (Int, Int) (B', B') (Object, Object) (Bool, Bool) }
---------------------------------------------------
```

V.E Examples 4.10a, 4.10b

```
Input File: Tests:4 - CS:Examp [illegible]

Input
----------------------------------------------------
class B

          var o1 : Object
          var o2 : Object
          var o3 : C

          method m ()
                    begin ... end


----------------------------------------- ----------
class C
          method p () : Object
                    begin ... end

end C


---------------------------------------------------
class B' inherits B {Object <- Int


end B'

----------------------------------------------------

====================================================


Output
---------------------------------------------------
Index: 1
Graph: Object
Aliases:
          "Object"

Index: 2
Graph: Bool
Aliases:
          "Bool"
          "Bool {Comparable <- Bool}"
          "Bool {Comparable <- Order}"
          "Bool {Order <- Int}"
          "Bool {Order <- Float}"
          "Bool {Order <- Char}"
```

```
Index: 3
Graph: Ring
Aliases:
            "Ring"

Index: 4
Graph: Comparable
Aliases:
            "Comparable"

Index: 6
Graph: Int
Aliases:
            "Int"
            "Int [B [Object <- Int] <- B']"

Index: 7
Graph: Order
Aliases:
            "Oraer"

Index: 10
Graph: Float
Aliases:
            "Float"

Index: 12
Graph: Char
Aliases:
            "Char"



---------
Index: 14
Class: B
Aliases:
            "B"
State: Resolvea
Nodes:

  (v, o1, 1, 14, «», {})
  (v, o2, 1, 14, «», {})
  (v, o3, 15, 14, «», {})
  (m, m, 14*, 14, « (v, self, 14*, 14, «», ()) », ())


Index: 15
Class: C
Aliases:
            "C"
State: Resolvea
Nodes:

        (m, p, 1, 15, « (v, self, 15*, 15, «», ()) », ())
```

```
Index: 17
Class: B'
Aliases:
            "B [Object <- Int]"
            "B'"
State: Resolved
Nodes:

  (v, o1, 6, 17, «», ( ))
  (v, o2, 6, 17, «», ( ))
  (v, o3, 18, 17, «», ( ))
  (m, m, 17*, 17, « (v, self, 17*, 17, «», ( ))  », (  ))


Index: 18
Class: C [Object <- Int]
Aliases:
            "C [Object <- Int]"
            "C [Object <- Int, 'B [Object <- Int] <- B']"
State: Resolved
Nodes:

  (m, p, 6, 18, « (v, self, 18*, 18, «», ( ))  », (  ))

----------------------------------------------------
Class: B is a subclass of:
 Object (1) by: ( )
 B (14) by: ( (Object, Object) (C, C) (B, B) )

Class: C is a subclass of:
 Object (1) by: ( )
 C (15) by: ( (Object, Object) (C, C) )

Class: B' is a subclass of:
 Object (1) by: ( )
 B (14) by: ( (Object, Int) (C, C [Object <- Int]) (B, B') )
 B' (17) by: ( (Int, Int) (C [Object <- Int], C [Object <- Int]) (B', B') (Bool, Bool) )

Class: C [Object <- Int] is a subclass of:
 Object (1) by: ( )
 C (15) by: ( (Object, Int) (C, C [Object <- Int]) )
 C [Object <- Int] (18) by: ( (Int, Int) (C [Object <- Int], C [Object <- Int])
                              (Bool, Bool) )

----------------------------------------------------
Class: B is a child (subclass by inheritance) of:
 Object (1) by: ( )
 B (14) by: ( (Object, Object) (C, C) (B, B) )

Class: C is a child (subclass by inheritance) of:
 Object (1) by: ( )
 C (15) by: ( (Object, Object) (C, C) )

Class: B' is a child (subclass by inheritance) of:
 Object (1) by: ( )
 B (14) by: ( (Object, Int) (C, C [Object <- Int]) (B, B') )
 B' (17) by: ( (Int, Int) (C [Object <- Int], C [Object <- Int]) (B', B') (Bool, Bool) )

Class: C [Object <- Int] is a child (subclass by inheritance) of:
 Object (1) by: ( )
 C (15) by: ( (Object, Int) (C, C [Object <- Int]) )
 C [Object <- Int] (18) by: ( (Int, Int) (C [Object <- Int], C [Object <- Int])
                              (Bool, Bool) )
----------------------------------------------------
```

V.F Example 4.11

```
Input File: Tests:4 - CS:Example 6

Input
----------------------------------------------------
class B

           var o1 : Object
           var o2 : Object
           var o3 : C

           method m ()
                      begin ... end


----------------------------------------------------
class C
           method p () : Object
                      begin ... end

end C


----------------------------------------------------
class C' inherits C {Object <- Int}


end C'

----------------------------------------------------
class B" inherits B {C <- C'}


end B"

----------------------------------------------------

==========================
Output
---------------------------------------------------
Index: 1
Graph: Object
Aliases:
           "Object"
           "Object [C <- C']"

Index: 2
Graph: Bool
Aliases:
           "Bool"
           "Bool [Comparable <- Bool]"
           "Bool [Comparable <- Order]"
           "Bool [Order <- Int]"
           "Bool [Order <- Float]"
           "Bool [Order <- Char]"

Index: 3
Graph: Ring
Aliases:
           "Ring"

Index: 4
Graph: Comparable
Aliases:
           "Comparable"

Index: 6
Graph: Int
Aliases:
           "Int"
           "Int [C [Object <- Int] <- C' ]"
           "Int [B [Object <- Int] <- B"]"

Index: 7
Graph: Order
Aliases:
           "Order"

Index: 10
Graph: Float
Aliases:
           "Float"

Index: 12
Graph: Char
Aliases:
           "Char"

---------
Index: 14
Class: B
Aliases:
           "B"
State: Resolved
Nodes:

  (v, o1, 1, 14, «», {})
  (v, o2, 1, 14, «», {})
  (v, o3, 15, 14, «», {})
  (m, m, 14*, 14, « (v, se[illegible], [illegible]*, 14, «», {}) », [illegible] })
```

```
Index: 15
Class: C
Aliases:
               "C"
State: Resolved
Nodes:

  (m, p, 1, 15, « (v, self, 15*, 15, «», {}) », {  })


Index: 17
Class: C'
Aliases:
               "C [Object <- Int]"
               "C'"
               "C' [B [Object <- Int] <- B"]"
State: Resolved
Nodes:

  (m, p, 6, 17, « (v, self, 17*, 17, «», {}) », {  })

Index: 22
Class: B"
Aliases:
               "B [Object <- Int]"
               "B [C <- C' "
               "B [Object <- Int    C' <- C'']" "B""
State: Resolved
Nodes:

  (v, o1, 6, 22, «», {})
  (v, o2, 6, 22, «», {})
  (v, o3, 17, 22, «», {})
  (m, m, 22*, 22, « (v, self, 22*, 22, «», {}) », {  })


----------------------------------------------------
Class: B is a subclass of:
 Object (1) by: { }
 B (14) by: { (Object, Object) (C, C) (B, B) }

Class: C is a subclass of:
 Object (1) by: { }
 C (15) by: { (Object, Object) (C, C) }

Class: C' is a subclass of:
 Object (1) by: { }
 C (15) by: { (Object, Int) (C, C') }
 C' (17) by: { (Int, Int) (C', C') (Bool, Bool) }

Class: B" is a subclass of:
 Object (1) by: { }
 B (14) by: { (Object, Int) (C, C') (B, B") }
 B" (22) by: { (Int, Int) (C', C') (B", B") (Bool, Bool) }

----------------------------------------------------
Class: B is a child (subclass by inheritance) of:
 Object (1) by: { }
 B (14) by: { (Object, Object) (C, C) (B, B) }

Class: C is a child (subclass by inheritance) of:
 Object (1) by: { }
 C (15) by: { (Object, Object) (C, C) }
```

```
Class: C' is a child (subclass by inheritance) of:
 Object (1) by: ( )
 C (15) by: ( (Object, Int) (C, C') )
 C' (17) by: ( (Int, Int) (C', C') (Bool, Bool) )

Class: B" is a child (subclass by inheritance) of:
 Object (1) by: ( )
 B (14) by: ( (Object, Int) (C, C') (B, B") )
 B" (22) by: ( (Int, Int) (C', C') (B", B") (Bool, Bool) )
----------------------------------------------------------
```

V.G Examples 4.12a - 4.12f

Input File: Tests:4 - CS:Example 7

Input

```
-----------------------------------------------------
class Animal
            var mate : Animal
            var owns : Object

            method lends () : Object
                        begin ... end

end Animal

-----------------------------------------------------
class Human inherits Animal [Object <- Int]
            method marry (other : Human)
                        begin ... end

end Human


-----------------------------------------------------
class Family
            var leader : Animal
            var has : Object

            method borrow ()
                        begin ... end
end Family

-----------------------------------------------------
class Human_Family1 inherits Family [Animal <- Human]


end Human_Family1

-----------------------------------------------------
class Human_Family2 inherits Family [Object <- Int]


end Human_Family2

-------------------------------------------------------

====================-- --            ---   - --- --  --
```

```
Output
---------------------------------------------------
Index: 1
Graph: Object
Aliases:
            "Object"
            "Object [Animal <- Human]"

Index: 2
Graph: Bool
Aliases:
            "Bool"
            "Bool [Comparable <- Bool]"
            "Bool [Comparable <- Order]"
            "Bool [Order <- Int]"
            "Bool [Order <- Float]"
            "Bool [Order <- Char]"
            "Bool [Animal [Object <- Int] <- Human]"

Index: 3
Graph: Ring
Aliases:
            "Ring"

Index: 4
Graph: Comparable
Aliases:
            "Comparable"

Index: 6
Graph: Int
Aliases:
            "Int"
            "Int [Animal [Object <- Int] <- Human]"
            "Int [Family [Animal <- Human] <- Human_Family1]"
            "Int [Family [Object <- Int] <- Human_Family2]"

Index: 7
Graph: Order
Aliases:
            "Order"

Index: 10
Graph: Float
Aliases:
            "Float"

Index: 12
Graph: Char
Aliases:
            "Char"

---------
Index: 14
Class: Animal
Aliases:
            "Animal"
State: Resolved
Nodes:

  (v, mate, 14*, 14, «», {})
  (v, owns, 1, 14, «», {})
  (m, lends, 1, 14, « (v, self, 14*, 14, «», {})  », {  })
```

```
Index: 15
Class: Human
Aliases:
            "Human"
            "Animal [Animal <- Human]"
            "Animal [Object <- Int] [Animal [Object <- Int] <- Human]"
            "Human [Family [Animal <- Human] <- Human_Family1]"
State: Resolved
Nodes:

  (m, marry, 15*, 15, « (v, self, 15*, 15, «», ()) (v, other, 15*, 15, «», ()) », ( ))
  (v, mate, 15*, 15, «», ())
  (v, owns, 6, 15, «», ())
  (m, lends, 6, 15, « (v, self, 15*, 15, «», ()) », ( ))


Index: 16
Class: Animal [Object <- Int]
Aliases:
            "Animal [Object <- Int]"
            "Animal [Object <- Int] [Family [Object <- Int] <- Human_Family2]"
State: Resolved
Nodes:

  (v, mate, 16*, 16, «», ())
  (v, owns, 6, 16, «», ())
  (m, lends, 6, 16, « (v, self, 16*, 16, «», ()) », ( ))


Index: 18
Class: Family
Aliases:
            "Family"
State: Resolved
Nodes:

  (v, leader, 14, 18, «», ()
  (v, has, 1, 18, «», ())
  (m, borrow, 18*, 18, « (v, self, 18*, 18, «», ()) », ( ))


Index: 20
Class: Human_Family1
Aliases:
            "Family [Animal <- Human]"
            "Family [Object <- Int] [Animal [Object <- Int] <- Human]"
            "Human_Family1"
State: Resolved
Nodes:

  (v, leader, 15, 20, «», ())
  (v, has, 6, 20, «», ())
  (m, borrow, 20*, 20, « (v, self, 20*, 20, «», ()) », ( ))
```

```
Index: 23
Class: Human_Family2
Aliases:
            "Family [Object <- Int]"
            "Human_Family2"
State: Resolved
Nodes:

  (v, leader, 16, 23, «», {})
  (v, has, 6, 23, «», {})
  (m, borrow, 23*, 23, « (v, self, 23*, 23, «», {}) », {  })

------------------------------------------------------
Class: Animal is a subclass of:
 Object (1) by: { }
 Animal (14) by: { (Animal, Animal) (Object, Object) }

Class: Human is a subclass of:
 Object (1) by: { }
 Animal (14) by: { (Animal, Human) (Object, Int) }
 Human (15) by: { (Human, Human) (Int, Int) (Bool, Bool) }
 Animal [Object <- Int] (16) by: { (Animal [Object <- Int], Human) (Int, Int)
                                  (Bool, Bool) }

Class: Animal [Object <- Int] is a subclass of:
 Object (1) by: { }
 Animal (14) by: { (Animal, Animal [Object <- Int]) (Object, Int) }
 Animal [Object <- Int] (16) by: { (Animal [Object <- Int], Animal [Object <- Int])
                                  (Int, Int) (Bool, Bool) }

Class: Family is a subclass of:
 Object (1) by: { }
 Family (18) by: { (Animal, Animal) (Object, Object) (Family, Family) }

Class: Human Family1 is a subclass of:
 Object (1) by: { }
 Family (18) by: { (Animal, Human) (Object, Int) (Family, Human_Family1) }
 Human_Family1 (20) by: { (Human, Human) (Int, Int) (Human_Family1, Human_Family1)
                        (Bool, Bool) }
 Human_Family2 (23) by: { (Animal [Object <- Int], Human) (Int, Int)
                        (Human_Family2, Human_Family1) (Bool, Bool) }

Class: Human Family2 is a subclass of:
 Object (1) by: { }
 Family (18) by: { (Animal, Animal [Object <- Int]) (Object, Int)
                 (Family, Human_Family2) }
 Human_Family2 (23) by: { (Animal [Object <- Int], Animal [Object <- Int]) (Int, Int)
                 (Human_Family2, Human_Family2) (Bool, Bool) }

------------------------------------------------------
Class: Animal is a child (subclass by inheritance) of:
 Object (1) by: { }
 Animal (14) by: { (Animal, Animal) (Object, Object) }

Class: Human is a child (subclass by inheritance) of:
 Object (1) by: { }
 Animal (14) by: { (Animal, Human) (Object, Int) }
 Human (15) by: { (Human, Human) (Int, Int) (Bool, Bool) }
 Animal [Object <- Int] (16) by: { (Animal [Object <- Int], Human) (Int, Int)
                                  (Bool, Bool) }
```

```
Class: Animal [Object <- Int] is a child (subclass by inheritance) of:
 Object (1) by: { }
 Animal (14) by: { (Animal, Animal [Object <- Int]) (Object, Int) }
 Animal [Object <- Int] (16) by: { (Animal [Object <- Int], Animal [Object <- Int])
                                   (Int, Int) (Bool, Bool) }

Class: Family is a child (subclass by inheritance) of:
 Object (1) by: { }
 Family (18) by: { (Animal, Animal) (Object, Object) (Family, Family) }

Class: Human_Family1 is a child (subclass by inheritance) of:
 Object (1) by: { }
 Family (18) by: { (Animal, Human) (Object, Int) (Family, Human_Family1) }
 Human_Family1 (20) by: { (Human, Human) (Int, Int) (Human_Family1, Human_Family1)
                          (Bool, Bool) }
  Human_Family2 (23) by: { (Animal [Object <- Int], Human) (Int, Int)
                          (Human_Family2, Human_Family1) (Bool, Bool) }

Class: Human_Family2 is a child (subclass by inheritance) of:
 Object (1) by: { }
 Family (18) by: { (Animal, Animal [Object <- Int]) (Object, Int)
                   (Family, Human_Family2)
 Human_Family2 (23) by: { (Animal [Object <- Int], Animal [Object <- Int])
                   (Int, Int) (Human_Family2, Human_Family2) (Bool, Bool) }
-----------------------------------------------------
```

V.H Examples 5.3 and 5.4

```
Input File: Tests:5 - Subclasses:Example 8

Input
----------------------------------------------------
class A

          var a : A

          method m (other : A) : D
                    begin ... end

end A


----------------------------------------------------
class D
          var a : D

end D


----------------------------------------------------
class B1
          var a : A

          method m (other : A) : D
                    begin ... end

          var b : B1

end B1

----------------------------------------------------
class B2
          var a : B2

          method m (other : B2) : D
                    begin ... end
          var b : B2
end B2
----------------------------------------------------

[illegible]
```

```
Output
-----------------------------------------------------
Index: 1
Graph: Object
Aliases:
            "Object"

Index: 2
Graph: Bool
Aliases:
            "Bool"
            "Bool {Comparable <- Bool}"
            "Bool {Comparable <- Order}"
            "Bool {Order <- Int}"
            "Bool {Order <- Float}"
            "Bool {Order <- Char}"

Index: 3
Graph: Ring
Aliases:
            "Ring"

Index: 4
Graph: Comparable
Aliases:
            "Comparable"

Index: 6
Graph: Int
Aliases:
            "Int"

Index: 7
Graph: Order
Aliases:
            "Order"

Index: 10
Graph: Float
Aliases:
            "Float"

Index: 12
Graph: Char
Aliases:
            "Char"


---------
Index: 14
Class: A
Aliases:
            "A"
State: Resolved
Nodes:

  (v, a, 14*, 14, «», {})
  (m, m, 15, 14, « (v, self, 14*, 14, «», {})  (v, other, 14*, 14, «», {})  », {  })
```

```
Index: 15
Class: D
Aliases:
            "D"
State: Resolved
Nodes:

  (v, d, 15*, 15, «», ())


Index: 16
Class: B1
Aliases:
            "B1"
State: Resolved
Nodes:
  (v, a, 14, 16, «», ())
  (m, m, 15, 16, « (v, self, 16*, 16, «», ())  (v, other, 14, 16, «», ())  », (  ))
  (v, b, 16*, 16, «», ())


Index: 17
Class: B2
Aliases:
            "B2"
State: Resolved
Nodes:
  (v, a, 17*, 17, «», ())
  (m, m, 15, 17, « (v, self, 17*, 17, «», ())  (v, other, 17*, 17, «», ())  », (  ))
  (v, b, 17*, 17, «», ())

------------------------- ---------------------------
Class: A is a subclass of:
  Object (1) by: ( )
  A (14) by: ( (A, A) (D, D) )

Class: D is a subclass of:
  Object (1) by: ( )
  D (15) by: ( (D, D) )

Class: B1 is a subclass of:
  Object (1) by: ( )
  B1 (16) by: ( (A, A) (D, D) (B1, B1) )

Class: B2 is a subclass of:
  Object (1) by: ( )
  A (14) by: ( (A, B2) (D, D) )
  B1 (16) by: ( (A, B2) (D, D) (B1, B2) )
  B2 (17) by: ( (B2, B2) (D, D) )

------------------------------------------------------
Class: A is a child (subclass by inheritance) of: Object (1) by: ( )
 A (14) by: ( (A, A) (D, D) )

Class: D is a child (subclass by inheritance) of: Object (1) by: ( )
 D (15) by: ( (D, D) )

Class: B1 is a child (subclass by inheritance) of: Object (1) by: ( )
 B1 (16) by: ( (A, A) (D, D) (B1, B1) )
```

```
Class: B2 is a child (subclass by inheritance) of: Object (1) by: ( )
 A (14) by: ( (A, B2) (D, D) )
 B1 (16) by: ( (A, B2) (D, D) (B1, B2) )
 B2 (17) by: ( (B2, B2) (D, D) )
---------------------------------------------------
```

V.I Examples 5.3 and 5.5

```
Input File: Tests:5 - Subclasses:Example 9

Input
----------------------------------------------------
class A
           var a : A

           method m (other : A) : [illegible]
                      begin ... end

end A


----------------------------------------------------
class D
           var d : A

end D


----------------------------------------------------
class B2
           var a : B2

           method m (other : B2) : [illegible]
                      begin ... end
           var b : B2
end B2

----------------------------------------------------

[illegible]
```

```
Output
-----------------------------------------------------
Index: 1
Graph: Object
Aliases:
           "Object"

Index: 2
Graph: Bool
Aliases:
           "Bool"
           "Bool (Comparable <- Bool)"
           "Bool (Comparable <- Order)"
           "Bool (Order <- Int)"
           "Bool (Order <- Float)"
           "Bool (Order <- Char)"

Index: 3
Graph: Ring
Aliases:
           "Ring"

Index: 4
Graph: Comparable
Aliases:
           "Comparable"

Index: 6
Graph: Int
Aliases:
           "Int"

Index: 7
Graph: Order
Aliases:
           "Order"

Index: 10
Graph: Float
Aliases:
           "Float"

Index: 12
Graph: Char
Aliases:
           "Char"



---------
Index: 14
Class: A
Aliases:
           "A"
State: Resolved
Nodes:

  (v, a, 14*, 14, «», ())
  (m, m, 15, 14, « (v, self, 14*, 14, «», ())  (v, other, 14*, 14, «», ())  », (   ))
```

```
Index: 15
Class: D
Aliases:
          "D"
State: Resolved
Nodes:

  (v, d, 14, 15, «», {})


Index: 16
Class: B2
Aliases:
          "B2"
State: Resolved
Nodes:
  (v, a, 16*, 16, «», {})
  (m, m, 15, 16, « (v, self, 16*, 16, «», {})  (v, other, 16*, 16, «», {})  », {  })
  (v, b, 16*, 16, «», {})

----------------------------------------------------
Class: A is a subclass of:
  Object (1) by: { }
  A (14) by: { (A, A) (D, D) }

Class: D is a subclass of:
  Object (1) by: { }
  D (15) by: { (A, A) (D, D) }

Class: B2 is a subclass of:
  Object (1) by: { }
  B2 (16) by: { (B2, B2) (D, D) (A, A) }


----------------------------------------------------
Class: A is a child (subclass by inheritance) of: Object (1) by: { }
 A (14) by: { (A, A) (D, D) }

Class: D is a child (subclass by inheritance) of: Object (1) by: { }
 D (15) by: { (A, A) (D, D) }

Class: B2 is a child (subclass by inheritance) of: Object (1) by: { }
 B2 (16) by: { (B2, B2) (D, D) (A, A) }
----------------------------------------------------
```

V.J Subclassing is Structural (1)

```
Input File: Tests:5 - Subclasses:Example 10

Input
-----------------------------------------------------
class A
            var c : Char
            var f : Float

end A

-----------------------------------------------------
class B
            var b : Bool
            var i : Int

end B

-----------------------------------------------------
class C
            var c : Char
            var b : Bool

end C

-----------------------------------------------------
class D
            var f : Float
            var i : Int

end D

-----------------------------------------------------
class E inherits A B
            method any () : E
                        begin ... end

end E

-----------------------------------------------------
class F inherits C D
            method any () : F
                        begin ... end

end F

-----------------------------------------------------

=====================================================
Output
----------------------------------------------------
Index: 1
Graph: Object
Aliases:
           "Object"

Index: 2
Graph: Bool
Aliases:
           "Bool"
           "Bool [Comparable <- Bool]"
           "Bool [Comparable <- Order]"
           "Bool [Order <- Int]"
           "Bool [Order <- Float]"
           "Bool [Order <- Char]"
           "Bool [B <- E]"
           "Bool [C <- F]"

Index: 3
Graph: Ring
Aliases:
           "Ring"

Index: 4
Graph: Comparable
Aliases:
           "Comparable"

Index: 6
Graph: Int
Aliases:
           "Int"
           "Int [B <- F]"
           "Int [D <- F]"

Index: 7
Graph: Order
Aliases:
           "Order"

Index: 10
Graph: Float
Aliases:
           "Float"
           "Float [A <- E]"
           "Float [D <- F]"

Index: 12
Graph: Char
Aliases:
           "Char"
           "Char [A <- F]"
           "Char [C <- F]"


---------
Index: 14
Class: A
Aliases:
           "A"
State: Resolved
Nodes:
  (v, c, 12, 14, «», ()) 
  (v, f, 10, 14, «», ())


Index: 15
Class: B
Aliases:
           "B"
State: Resolved
Nodes:

  (v, b, 2, 15, «», ())
  (v, i, 6, 15, «», ())


Index: 16
Class: C
Aliases:
           "C"
State: Resolved
Nodes:

  (v, c, 12, 16, «», ())
  (v, b, 2, 16, «», ())


Index: 17
Class: D
Aliases:
           "D"
State: Resolved
Nodes:

  (v, f, 10, 17, «», ())
  (v, i, 6, 17, «», ())


Index: 18
Class: F
Aliases:
           "E"
           "F"
State: Resolved
Nodes:

  (m, any, 18*, 18, « (v, self, 18*, 18, «», ())  », (  )) (v, c, 12, 18, «», ())
  (v, f, 10, 18, «», ())
  (v, b, 2, 18, «», ())
  (v, i, 6, 18, «», ())

------------------------------------------------------
Class: A is a subclass of:
 Object (1) by: ( )
 A (14) by: ( (Char, Char) (Float, Float) (Int, Int) (Bool, Bool) )

Class: B is a subclass of:
 Object (1) by: ( )
 B (15) by: ( (Bool, Bool) (Int, Int) )

Class: C is a subclass of:
 Object (1) by: ( )
 C (16) by: ( (Char, Char) (Bool, Bool) (Int, Int) )
```

```
Class: D is a subclass of:
 Object (1) by: ( )
 D (17) by: ( (Float, Float) (Int, Int) (Bool, Bool) )

Class: F is a subclass of:
 Object (1) by: ( )
 A (14) by: ( (Char, Char) (Float, Float) (Int, Int) (Bool, Bool) )
 B (15) by: ( (Bool, Bool) (Int, Int) )
 C (16) by: ( (Char, Char) (Bool, Bool) (Int, Int) )
 D (17) by: ( (Float, Float) (Int, Int) (Bool, Bool) )
 F (18) by: ( (F, F) (Char, Char) (Float, Float) (Bool, Bool) (Int, Int) )

---------------------------------------------------
Class: A is a child (subclass by inheritance) of:
 Object (1) by: ( )
 A (14) by: ( (Char, Char) (Float, Float) (Int, Int) (Bool, Bool) )

Class: B is a child (subclass by inheritance) of:
 Object (1) by: ( )
 B (15) by: ( (Bool, Bool) (Int, Int) )

Class: C is a child (subclass by inheritance) of:
 Object (1) by: ( )
 C (16) by: ( (Char, Char) (Bool, Bool) (Int, Int) )

Class: D is a child (subclass by inheritance) of:
 Object (1) by: ( )
 D (17) by: ( (Float, Float) (Int, Int) (Bool, Bool) )

Class: F is a child (subclass by inheritance) of:
 Object (1) by: ( )
 A (14) by: ( (Char, Char) (Float, Float) (Int, Int) (Bool, Bool) )
 B (15) by: ( (Bool, Bool) (Int, Int) )
 C (16) by: ( (Char, Char) (Bool, Bool) (Int, Int) )
 D (17) by: ( (Float, Float) (Int, Int) (Bool, Bool) )
 F (18) by: ( (F, F) (Char, Char) (Float, Float) (Bool, Bool) (Int, Int) )
---------------------------------------------------
```

V.K Subclassing is Structural (2)

```
Input File: Tests:5 - Subclasses:Example 11

Input
-------------------------------------------------------
class A

            var a : A

            method p (a : A  i : Int) : B
                        begin ... end

end A



-------------------------------------------------------
class B

            var b : B

            method p (b : B  i : Int) : A
                        begin ... end

end B

-------------------------------------------------------
class C
            var a : C

            method p (a : C  i : Int) : B
                        begin ... end

end C


-------------------------------------------------------
class D

            var a : A

            method p (a : C  [illegible] : Int) : [illegible]
                        begin ... end

end D


-------------------------------------------------------
class E

            var a : D

            method p (a : A  [illegible] : Int) : B
                        begin ... end

end E
```

```
-----------------------------------------------------
class F

           var a : A

           method p (f : F  i : Int) : B
                      begin ... end

end F

-----------------------------------------------------
class H

           var a : H
           method p (a : H  i : Int) : B
                      begin ... end

end H


-----------------------------------------------------
class I inherits A

end I

------------------------------------------------------

========================================
Output
---------------------------------------------------
Index: 1
Graph: Object
Aliases:
          "Object"

Index: 2
Graph: Bool
Aliases:
          "Bool"
          "Bool {Comparable <- Bool}"
          "Bool {Comparable <- Order}"
          "Bool {Order <- Int}"
          "Bool {Order <- Float}"
          "Bool {Order <- Char}"

Index: 3
Graph: Ring
Aliases:
          "Ring"

Index: 4
Graph: Comparable
Aliases:
          "Comparable"

Index: 6
Graph: Int
Aliases:
          "Int"
          "Int {A <- I}"

Index: 7
Graph: Order
Aliases:
          "Order"

Index: 10
Graph: Float
Aliases:
          "Float"

Index: 12
Graph: Char
Aliases:
          "Char"



---------
Index: 14
Class: I
Aliases:
          "A"
          "I"
State: Resolved
Nodes:

  (v, a, 14*, 14, «», {})
  (m, p, 15, 14,
       « (v, self, 14*, 14, «», {}) (v, [illegible], 14*, 14, «», {}) (v, i, 6, 14, «», {}) »,
       { })
```

```
Index: 15
Class: B [A <- I]
Aliases:
            "B"
            "B [A <- I]"
State: Resolved
Nodes:

  (v, b, 15*, 15, «», ())
  (m, p, 14, 15,
        « (v, self, 15*, 15, «», ())  (v, b, 15*, 15, «», ())  (v, i, 6, 15, «», ())  »,
        (  ))


Index: 16
Class: H
Aliases:
            "C"
            "H"
State: Resolved
Nodes:

  (v, a, 16*, 16, «», ())
  (m, p, 15, 16,
        « (v, self, 16*, 16, «», ())  (v, a, 16*, 16, «», ())  (v, i, 6, 16, «», ())  »,
        (  ))


Index: 17
Class: D
Aliases:
            "D"
State: Resolved
Nodes:

  (v, a, 14, 17, «», ())
  (m, p, 15, 17,
        « (v, self, 17*, 17, «», ())  (v, a, 16, 17, «», ())  (v, i, 6, 17, «», ())  »,
        (  ))


Index: 18
Class: E
Aliases:
            "E"
State: Resolved
Nodes:

  (v, a, 17, 18, «», ())
  (m, p, 15, 18,
        « (v, self, 18*, 18, «», ())  (v, a, 14, 18, «», ())  (v, i, 6, 18, «», ())  »,
        (  ))
```

```
Index: 19
Class: F
Aliases:
          "F"
State: Resolved
Nodes:

  (v, a, 14, 19, «», {})
  (m, p, 15, 19,
        « (v, self, 19*, 19, «», {})  (v, f, 19*, 19, «», {})  (v, i, 6, 19, «», {})  »,
        {  })

-----------------------------------------------------
Class: I is a subclass of:
 Object (1) by: { }
 I (14) by: { (I, I) (B [A <- I], B [A <- I]) (Int, Int) (Bool, Bool) }
 H (16) by: { (H, I) (B [A <- I], B [A <- I]) (Int, Int) (I, I) (Bool, Bool) }
 D (17) by: { (I, I) (B [A <- I], B [A <- I]) (D, I) (H, I) (Int, Int) (Bool, Bool) }
 E (18) by: { (D, I) (B [A <- I], B [A <- I]) (E, I) (I, I) (Int, Int) (H, I) (Bool, Bool) }

Class: B [A <- I] is a subclass of:
 Object (1) by: { }
 B [A <- I] (15) by: { (B [A <- I], B [A <- I]) (I, I) (Int, Int) (Bool, Bool) }

Class: H is a subclass of:
 Object (1) by: { }
 H (16) by: { (H, H) (B [A <- I], B [A <- I]) (Int, Int) (I, I) (Bool, Bool) }

Class: D is a subclass of:
 Object (1) by: { }
 D (17) by: { (I, I) (B [A <- I], B [A <- I]) (D, D) (H, H) (Int, Int) (Bool, Bool) }

Class: E is a subclass of:
 Object (1) by: { }
 E (18) by: { (D, D) (B [A <- I], B [A <- I]) (E, E) (I, I) (Int, Int) (H, H) (Bool, Bool) }

Class: F is a subclass of:
 Object (1) by: { }
 F (19) by: { (I, I) (B [A <- I], B [A <- I]) (F, F) (Int, Int) (Bool, Bool) }

-----------------------------------------------------
Class: I is a child (subclass by inheritance) of:
 Object (1) by: { }
 I (14) by: { (I, I) (B [A <- I], B [A <- I]) (Int, Int) (Bool, Bool) }
 H (16) by: { (H, I) (B [A <- I], B [A <- I]) (Int, Int) (I, I) (Bool, Bool) }
 D (17) by: { (I, I) (B [A <- I], B [A <- I]) (D, I) (H, I) (Int, Int) (Bool, Bool) }
 E (18) by: { (D, I) (B [A <- I], B [A <- I]) (E, I) (I, I) (Int, Int) (H, I) (Bool, Bool) }

Class: B [A <- I] is a child (subclass by inheritance) of:
 Object (1) by: { }
 B [A <- I] (15) by: { (B [A <- I], B [A <- I]) (I, I) (Int, Int) (Bool, Bool) }

Class: H is a child (subclass by inheritance) of:
 Object (1) by: { }
 H (16) by: { (H, H) (B [A <- I], B [A <- I]) (Int, Int) (I, I) (Bool, Bool) }

Class: D is a child (subclass by inheritance) of:
 Object (1) by: { }
 D (17) by: { (I, I) (B [A <- I], B [A <- I]) (D, D) (H, H) (Int, Int) (Bool, Bool) }

Class: E is a child (subclass by inheritance) of:
 Object (1) by: { }
 E (18) by: { (D, D) (B [A <- [illegible]], [illegible] [A <- [illegible]]) ([illegible], [illegible]) ([illegible], [illegible]) (Int, Int) (H, H) (Bool, Bool) }
```

```
Class: F is a child (subclass by inheritance) of:
 Object (1) by: ( )
 F (19) by: ( (I, I) (B {A <- I , B {A <- I}) (F, F) (Int, Int) (Bool, Bool) )
-----------------------------------------------------------
```

V.L Bruce Case Analysis: Ex13a

```
Input File: Tests:6 - vs Bruce:Example 12

Input
----------------------------------------------------
class Ex13a

          method rec method (any : Ex13a)
                    begin ... end

          var c : C

          method m ()
                    begin ... end

end Ex13a


---------------------------------------------------
class C

          method mc (v : Ex13a)
                    begin ... end

end C


---------------------------------------------------
class Child of Ex13a [illegible] Ex13a
          var c : C[illegible]
end Child of Ex13a

---------------------------------------------------

[illegible]
```

```
Output
------------------------------------------------------
Index: 1
Graph: Object
Aliases:
            "Object"

Index: 2
Graph: Bool
Aliases:
            "Bool"
            "Bool [Comparable <- Bool]"
            "Bool [Comparable <- Order]"
            "Bool [Order <- Int]"
            "Bool [Order <- Float]"
            "Bool [Order <- Char]"

Index: 3
Graph: Ring
Aliases:
            "Ring"

Index: 4
Graph: Comparable
Aliases:
            "Comparable"

Index: 6
Graph: Int
Aliases:
            "Int"

Index: 7
Graph: Order
Aliases:
            "Order"

Index: 10
Graph: Float
Aliases:
            "Float"

Index: 12
Graph: Char
Aliases:
            "Char"



---------
Index: 14
Class: Ex13a
Aliases:
            "Ex13a"
State: Resolved
Nodes:

  (m, rec_method, 14*, 14,
        « (v, self, 14*, 14, «», [illegible])  (v, arg, 14*, 14, «», [illegible])  », [illegible] })
          (v, c, 15, 14, «»,
        [illegible] })
  (m, m, 14*, 14, « (v, [illegible], [illegible], 14, «», [illegible] », [illegible])
Index: 15
Class: C
Aliases:
          "C"
State: Resolved
Nodes:

  (m, mc, 15*, 15, « (v, self, 15*, 15, «», ()) (v, v, 14, 15, «», ()) », ( ))


Index: 16
Class: Child_of_Ex13a
Aliases:
          "Child_of_Ex13a"
State: Resolved
Nodes:

  (v, o, 1, 16, «», ())
  (m, rec method, 16*, 16,
       « (v, self, 16*, 16, «», ()) (v, any, 16*, 16, «», ()) »,
       ( ))
  (v, c, 17, 16, «», ())
  (m, m, 16*, 16, « (v, self, 16*, 16, «», ()) », ( ))


Index: 17
Class: C {Ex13a <- Child_of_Ex13a}
Aliases:
          "C {Ex13a <- Child_of_Ex13a}"
State: Resolved
Nodes:

  (m, mc, 17*, 17, « (v, self, 17*, 17, «», ()) (v, v, 16, 17, «», ()) ». ( ))

----------------------------------------------------
Class: Ex13a is a subclass of:
 Object (1) by: ( )
 Ex13a (14) by: ( (Ex13a, Ex13a) (C, C) )

Class: C is a subclass of:
 Object (1) by: ( )
 C (15) by: ( (C, C) (Ex13a, Ex13a) )

Class: Child_of_Ex13a is a subclass of:
 Object (1) by: ( )
 Ex13a (14) by: ( (Ex13a, Child_of_Ex13a) (C, C {Ex13a <- Child_of_Ex13a}) )
 Child_of_Ex13a (16) by: ( (Object, Object) (Child_of_Ex13a, Child_of_Ex13a)
                          (C {Ex13a <- Child_of_Ex13a}, C {Ex13a <- Child_of_Ex13a}) )

Class: C {Ex13a <- Child_of_Ex13a} is a subclass of:
 Object (1) by: ( )
 C (15) by: ( (C, C {Ex13a <- Child_of_Ex13a}) (Ex13a, Child_of_Ex13a) )
 C {Ex13a <- Child_of_Ex13a} (17) by:
      ( (C {Ex13a <- Child_of_Ex13a}, C {Ex13a <- Child_of_Ex13a})
        (Child_of_Ex13a, Child_of_Ex13a) (Object, Object) )

----------------------------------------------------
Class: Ex13a is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Ex13a (14) by: ( (Ex13a, Ex13a) (C, C) )
```

```
Class: C is a child (subclass by inheritance) of:
 Object (1) by: ( )
 C (15) by: ( (C, C) (Ex13a, Ex13a) )

Class: Child_of_Ex13a is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Ex13a (14) by: ( (Ex13a, Child_of_Ex13a) (C, C [Ex13a <- Child_of_Ex13a]) )
 Child_of_Ex13a (16) by: ( (Object, Object) (Child_of_Ex13a, Child_of_Ex13a)
        (C [Ex13a <- Child_of_Ex13a], C [Ex13a <- Child_of_Ex13a]) )

Class: C [Ex13a <- Child_of_Ex13a] is a child (subclass by inheritance) of:
 Object (1) by: ( )
 C (15) by: ( (C, C [Ex13a <- Child_of_Ex13a]) (Ex13a, Child_of_Ex13a) )
 C [Ex13a <- Child_of_Ex13a] (17) by:
        ( (C [Ex13a <- Child_of_Ex13a], C [Ex13a <- Child_of_Ex13a])
          (Child_of_Ex13a, Child_of_Ex13a) (Object, Object) )
---------------------------------------------------------
```

V.M Bruce Case Analysis: Ex14a

```
Input File: Tests:6 - vs Bruce:Example 13

Input
-----------------------------------------------------
class Ex14a
            method rec method (arg : Ex14a)
                        begin ... end

            method m (c : C)
                        begin ... end

end Ex14a


-----------------------------------------------------
class C

            method mc (v : Ex14a)
                        begin ... end

end C


----------------------------------------------------
class Child_of Ex14a inherits Ex14a

            var c : Object

end Child of Ex14a

-----------------------------------------------------

[illegible]
```

```
Output
--------------------------------------------------
Index: 1
Graph: Object
Aliases:
            "Object"

Index: 2
Graph: Bool
Aliases:
            "Bool"
            "Bool (Comparable <- Bool)"
            "Bool (Comparable <- Order)"
            "Bool (Order <- Int)"
            "Bool (Order <- Float)"
            "Bool (Order <- Char)"

Index: 3
Graph: Ring
Aliases:
            "Ring"

Index: 4
Graph: Comparable
Aliases:
            "Comparable"

Index: 6
Graph: Int
Aliases:
            "Int"

Index: 7
Graph: Order
Aliases:
            "Order"

Index: 10
Graph: Float
Aliases:
            "Float"

Index: 12
Graph: Char
Aliases:
            "Char"


---------
Index: 14
Class: Ex14a
Aliases:
            "Ex14a"
State: Resolved
Nodes:

  (m, rec_method, 14*, 14,
        « (v, self, 14*, 14, «», [illegible] (v, [illegible], [illegible]*, 14, «», ()) »,
        ( ))
  (m, m, 14*, 14, « (v, self, 14*, 14, «», [illegible] (v, [illegible], 15, 14, «», ()) », ( ))
Index: 15
Class: C
Aliases:
            "C"
State: Resolved
Nodes:

  (m, mc, 15*, 15, « (v, self, 15*, 15, «», {}) (v, v, 14, 15, «», {}) », { })


Index: 16
Class: Child_of_Ex14a
Aliases:
            "Child_of_Ex14a"
State: Resolved
Nodes:
  (v, o, 1, 16, «», {})
  (m, rec_method, 16*, 16,
       « (v, self, 16*, 16, «», {}) (v, any, 16*, 16, «», {}) »,
       { })
  (m, m, 16*, 16, « (v, self, 16*, 16, «», {}) (v, c, 17, 16, «», {}) », { })


Index: 17
Class: C [Ex14a <- Child_of_Ex14a
Aliases:
            "C [illegible]"
State: Resolved
Nodes:

  (m, mc, 17*, 17, « (v, self, [illegible]*, 1[illegible], «», {}) (v, v, 16, 17, «», {}) », { })


-----------------------------------------------------
Class: Ex14a is a subclass of:
  Object (1) by: { }
  Ex14a (14) by: { (Ex14a, Ex14a) (C, C) }

Class: C is a subclass of:
  Object (1) by: { }
  C (15) by: { (C, C) (Ex14a, Ex14a) }

Class: Child_of_Ex14a is a subclass of:
  Object (1) by: { }
  Ex14a (14) by: { (Ex14a, Child_of_Ex14a) (C, C [Ex14a <- Child_of_Ex14a]) }
  Child_of_Ex14a (16) by: { (Object, Object) (Child_of_Ex14a, Child_of_Ex14a)
        (C [Ex14a <- Child_of_Ex14a], C [Ex14a <- Child_of_Ex14a]) }

Class: C [Ex14a <- Child_of_Ex14a is a subclass of:
   Object (1) by: { }
   C (15) by: { (C, C [Ex14a <- Child_of_Ex14a]) (Ex14a, Child_of_Ex14a) }
   C [Ex14a <- Child_of_Ex14a (17) by:
        { (C [Ex14a <- Child_of_Ex14a], C [Ex14a <- Child_of_Ex14a]
          (Child_of_Ex14a, Child_of_Ex14a) (Object, Object) }

-----------------------------------------------------
Class: Ex14a is a child (subclass by inheritance) of:
  Object (1) by: { }
  Ex14a (14) by: { (Ex14a, Ex14a) (C, C) }

Class: C is a child (subclass by inheritance) of:
  Object (1) by: { }
  C (15) by: { (C, C) (Ex14a, Ex14a) }
```

```
Class: Child_of_Exl4a is a child (subclass by inheritance) of:
 Object (1) by: { }
 Exl4a (14) by: { (Exl4a, Child_of_Exl4a) (C, C [Exl4a <- Child_of_Exl4a]) }
 Child_of_Exl4a (16) by: { (Object, Object) (Child_of_Exl4a, Child_of_Exl4a)
        (C [Exl4a <- Child_of_Exl4a], C [Exl4a <- Child_of_Exl4a]) }

Class: C [Exl4a <- Child_of_Exl4a] is a child (subclass by inheritance) of:
 Object (1) by: { }
 C (15) by: { (C, C [Exl4a <- Child_of_Exl4a]) (Exl4a, Child_of_Exl4a) }
 C [Exl4a <- Child_of_Exl4a] (17) by:
        { (C [Exl4a <- Child_of_Exl4a], C [Exl4a <- Child_of_Exl4a])
          (Child_of_Exl4a, Child_of_Exl4a) (Object, Object) }
---------------------------------------------------------
```

V.N Bruce Case Analysis: Ex15a

```
Input File: Tests:6 - vs Bruce:Example 14

Input
-----------------------------------------------------
class Ex15a
            method rec_method (any : Ex15a)
                        begin ... end

            method c () : C
                        begin ... end

            method m ()
                        begin ... end

end Ex15a


-----------------------------------------------------
class C
            method mc (v : Ex15a)
                        begin ... end

end C


-----------------------------------------------------
class Child_of_Ex15a inherits Ex15a

            var o : Object

end Child_of_Ex15a

-----------------------------------------------------

=====. [illegible]

Output
-----------------------------------------------------
Index: 1
Graph: Object
Aliases:
            "Object"

Index: 2
Graph: Bool
Aliases:
            "Bool"
            "Bool (Comparable <- Bool)"
            "Bool (Comparable <- Order)"
            "Bool (Order <- [illegible])"
            "Bool (Order <- [illegible])"
            "Bool (Order <- [illegible])"
```

```
Index: 3
Graph: Ring
Aliases:
           "Ring"

Index: 4
Graph: Comparable
Aliases:
           "Comparable"

Index: 6
Graph: Int
Aliases:
           "Int"

Index: 7
Graph: Order
Aliases:
           "Order"

Index: 10
Graph: Float
Aliases:
           "Float"

Index: 12
Graph: Cnar
Aliases:
           "Char"


---------
Index: 14
Class: Ex15a
Aliases:
           "Ex15a"
State: Resolveo
Nodes:

  (m, rec_methoo, 14*, 14,
        « (v, self, 14*, 14, «», {}) (v, ary, 14*, 14, «», {}) »,
        { })
  (m, c, 15, 14, « (v, self, 14*, 14, «», {}) », { })
  (m, m, 14*, 14, « (v, self, 14*, 14, «», {}) », { })

Index: 15
Class: C
Aliases:
           "C"
State: Resolveo
Nodes:

  (m, mc, 15*, 15, « (v, self, 15*, 15, «», {}) (v, v, 14, 15, «», {}) », { })
```

```
Index: 16
Class: Child_of_Ex15a
Aliases:
            "Child_of_Ex15a"
State: Resolved
Nodes:

  (v, o, 1, 16, «», ())
  (m, rec_method, 16*, 16,
      « (v, self, 16*, 16, «», ())  (v, any, 16*, 16, «», ())  »,
      (  ))
  (m, c, 17, 16, « (v, self, 16*, 16, «», ())  », (  ))
  (m, m, 16*, 16, « (v, self, 16*, 16, «», ())  », (  ))


Index: 17
Class: C [Ex15a <- Child_of_Ex15a
Aliases:
            "C [Ex15a <- Child_of_Ex15a]"
State: Resolved
Nodes:

      (m, mc, 17*, 17, « (v, self, 17*, 17, «», ())  (v, v, 16, 17, «», ())  », (  ))


-----------------------------------------------------
Class: Ex15a is a subclass of:
 Object (1) by: ( )
 Ex15a (14) by: ( (Ex15a, Ex15a) (C, C) )

Class: C is a subclass of:
 Object (1) by: ( )
 C (15) by: ( (C, C) (Ex15a, Ex15a) )

Class: Child_of_Ex15a is a subclass of:
  Object (1) by: ( )
  Ex15a (14) by: ( (Ex15a, Child_of_Ex15a) (C, C [Ex15a <- Child_of_Ex15a]) )
  Child_of_Ex15a (16) by: ( (Object, Object) (Child_of_Ex15a, Child_of_Ex15a)
        (C [Ex15a <- Child_of_Ex15a], C [Ex15a <- Child_of_Ex15a]) )

Class: C [Ex15a <- Child_of_Ex15a] is a subclass of:
 Object (1) by: ( )
 C (15) by: ( (C, C [Ex15a <- Child_of_Ex15a]) (Ex15a, Child_of_Ex15a) )
 C [Ex15a <- Child_of_Ex15a] (17) by:
       ( (C [Ex15a <- Child_of_Ex15a], C [Ex15a <- Child_of_Ex15a])
         (Child_of_Ex15a, Child_of_Ex15a) (Object, Object) )

-----------------------------------------------------
Class: Ex15a is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Ex15a (14) by: ( (Ex15a, Ex15a) (C, C) )

Class: C is a child (subclass by inheritance) of:
 Object (1) by: ( )
 C (15) by: ( (C, C) (Ex15a, Ex15a) )

Class: Child_of_Ex15a is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Ex15a (14) by: ( (Ex15a, Child_of_Ex15a) (C, C [Ex15a <- Child_of_Ex15a]) )
 Child_of_Ex15a (16) by: ( (Object, Object) (Child_of_Ex15a, Child_of_Ex15a)
       (C [Ex15a <- Child_of_Ex15a], C [Ex15a <- Child_of_Ex15a]) )
```

```
Class: C [Ex15a <- Child_of_Ex15a  is a child (subclass by inheritance) of:
  Object (1) by: ( )
  C (15) by: ( (C, C [Ex15a <- Child_of_Ex15a]) (Ex15a, Child_of_Ex15a) )
  C [Ex15a <- Child_of_Ex15a  (17) by:
         ( (C [Ex15a <- Child_of_Ex15a], C [Ex15a <- Child_of_Ex15a])
           (Child_of_Ex15a, Child_of Ex15a) (Object, Object) )
----------------------------------------------------
```

V.O Temporal Cycles: Voter/Candidate

```
Input File: Tests:7 - vs P5:Example 15

Input
-----------------------------------------------------
class Voter

           var backs : Candidate

           var influenced_by : Voter

           method vote ()
              begin ... end

end Voter


class Candidate inherits Voter
           var votes received : int
end Candidate

-----------------------------------------------------

[illegible]
```

```
Output
-----------------------------------------------------
Index: 1
Graph: Object
Aliases:
            "Object"

Index: 2
Graph: Bool
Aliases:
            "Bool"
            "Bool [Comparable <- Bool]"
            "Bool [Comparable <- Order]"
            "Bool [Order <- Int]"
            "Bool [Order <- Float]"
            "Bool [Order <- Char]"

Index: 3
Graph: Ring
Aliases:
            "Ring"

Index: 4
Graph: Comparable
Aliases:
            "Comparable"

Index: 6
Graph: Int
Aliases:
            "Int"

Index: 7
Graph: Order
Aliases:
            "Order"

Index: 10
Graph: Float
Aliases:
            "Float"

Index: 12
Graph: Char
Aliases:
            "Char"



---------
Index: 14
Class: Voter
Aliases:
            "Voter"
State: Resolved
Nodes:

  (v, backs, 15, 14, «», [illegible])
  (v, influenced_by, 14*, 14, «», [illegible])
  (m, vote, 14*, 14, « [illegible] », [illegible])
```

```
Index: 15
Class: Candidate
Aliases:
            "Candidate"
            "Candidate  Voter <- Candidate."
State: Resolved
Nodes:

  (v, votes_received, 6, 15, «», {})
  (v, backs, 15*, 15, «», {})
  (v, influenced_by, 15*, 15, «», {})
  (m, vote, 15*, 15, « (v, self, 15*, 15, «», {}) », {  })

-----------------------------------------------------
Class: Voter is a subclass of:
 Object (1) by: ( )
 Voter (14) by: ( (Candidate, Candidate) (Voter, Voter) (Int, Int) (Bool, Bool) )

Class: Candidate is a subclass of:
 Object (1) by: ( )
 Voter (14) by: ( (Candidate, Candidate) (Voter, Candidate) (Int, Int) (Bool, Bool) )
 Candidate (15) by: ( (Int, Int) (Candidate, Candidate) (Bool, Bool) )

-----------------------------------------------------
Class: Voter is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Voter (14) by: ( (Candidate, Candidate) (Voter, Voter) (Int, Int) (Bool, Bool) )

Class: Candidate is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Voter (14) by: ( (Candidate, Candidate) (Voter, Candidate) (Int, Int) (Bool, Bool) )
 Candidate (15) by: ( (Int, Int) (Candidate, Candidate) (Bool, Bool) )
-----------------------------------------------------
```

V.P Compiler Dependencies I

```
Input File: Tests:7 - vs PS:Example 16

Input
------------------------------------------------------
class Voter

            var backs : Candidate
            var influenced_by : Voter

            method votes ()
                        begin ... end

end Voter

------------------------------------------------------
class Candidate inherits Voter

            var best_friend : Voter

end Candidate

------------------------------------------------------

========================================================
Output
--------------------------------------------------
Index: 1
Graph: Object
Aliases:
            "Object"

Index: 2
Graph: Bool
Aliases:
            "Bool"
            "Bool [Comparable <- Bool]"
            "Bool [Comparable <- Order]"
            "Bool [Order <- Int]"
            "Bool [Order <- Float]"
            "Bool [Order <- Char]"

Index: 3
Graph: Ring
Aliases:
            "Ring"

Index: 4
Graph: Comparable
Aliases:
            "Comparable"

Index: 6
Graph: Int
Aliases:
            "Int"

Index: 7
Graph: Order
Aliases:
            "Order"

Index: 10
Graph: Float
Aliases:
            "Float"

Index: 12
Graph: Char
Aliases:
            "Char"


---------
Index: 14
Class: Voter
Aliases:
            "Voter"
State: Resolved
Nodes:

  (v, backs, 15, 14, «», [illegible])
  (v, influenced by, 14*, 14, «», [illegible])
  (m, votes, 14*, 14, « (v, [illegible], [illegible]4*, 14, «», [illegible]) », [illegible])
```

```
Index: 15
Class: Candidate
Aliases:
          "Candidate"
State: Resolved
Nodes:

  (v, best_friend, 14, 15, «», ( ))
  (v, backs, 16, 15, «», ( ))
  (v, influenced_by, 15*, 15, «», ( ))
  (m, votes, 15*, 15, « (v, self, 15*, 15, «», ( )) », ( ))


Index: 16
Class: Candidate [Voter <- Candidate]
Aliases:
          "Candidate [Voter <- Candidate]"
State: Unresolved: Deep Derived with Base: 15 Repl: 14 With: 15

---------------------------------------------------
Class: Voter is a subclass of:
 Object (1) by: ( )

Class: Candidate is a subclass of:
 Object (1) by: ( )

Class: Candidate [Voter <- Candidate] is a subclass of:

---------------------------------------------------
Class: Voter is a child (subclass by inheritance) of:
 Object (1) by: ( )

Class: Candidate is a child (subclass by inheritance) of:
 Object (1) by: ( )

Class: Candidate [Voter <- Candidate] is a child (subclass by inheritance) of:
---------------------------------------------------
```

V.Q Compiler Dependencies (1)

```
Input File: Tests:7 - vs PS:Example 17

Input
----------------------------------------------------
class A inherits B

end A
----------------------------------------------------
class B inherits C

        var b : B

end B
----------------------------------------------------
class C inherits A

        var c : Char

end C

----------------------------------------------------

[illegible]
```

```
Output
-----------------------------------------------------
Index: 1
Graph: Object
Aliases:
            "Object"

Index: 2
Graph: Bool
Aliases:
            "Bool"
            "Bool [Comparable <- Bool]"
            "Bool [Comparable <- Order]"
            "Bool [Order <- Int]"
            "Bool [Order <- Float]"
            "Bool [Order <- Char]"

Index: 3
Graph: Ring
Aliases:
            "Ring"

Index: 4
Graph: Comparable
Aliases:
            "Comparable"

Index: 6
Graph: Int
Aliases:
            "Int"

Index: 7
Graph: Order
Aliases:
            "Order"

Index: 10
Graph: Float
Aliases:
            "Float"

Index: 12
Graph: Char
Aliases:
            "Char"


---------
```

```
Index: 14
Class: A
Aliases:
            "A"
State: Unresolved: Explicitly Defined

Index: 15
Class: B
Aliases:
            "B"
State: Unresolved: Explicitly Defined

Index: 16
Class: C
Aliases:
            "C"
State: Unresolved: Explicitly Defined

-------------------------------------------------------
Class: A is a subclass of:

Class: B is a subclass of:

Class: C is a subclass of:

-------------------------------------------------------
Class: A is a child (subclass by inheritance) of:

Class: B is a child (subclass by inheritance) of:

Class: C is a child (subclass by inheritance) of:
-------------------------------------------------------
```

V.R Compiler Dependencies (2)

Input File: Tests:7 - vs P5:Fxarpie 18

```
Input
-----------------------------------------------------
class A inherits B [C <- D]

        var x : Int

end A
-----------------------------------------------------
class B

        var c : C
        method q (d : D  e : C)
               begin ... end

end B
-----------------------------------------------------
class C

        var a : A

end C
-----------------------------------------------------
class D

        var d : A
        var e : B
end D

------------------------------------------------- ----

=====================-----
```

```
Output
-------------------------------------------------------
Index: 1
Graph: Object
Aliases:
            "Object"

Index: 2
Graph: Bool
Aliases:
            "Bool"
            "Bool [Comparable <- Bool]"
            "Bool [Comparable <- Order]"
            "Bool [Order <- Int]"
            "Bool [Order <- Float]"
            "Bool [Order <- Char]"

Index: 3
Graph: Ring
Aliases:
            "Ring"

Index: 4
Graph: Comparable
Aliases:
            "Comparable"

Index: 6
Graph: Int
Aliases:
            "Int"

Index: 7
Graph: Order
Aliases:
            "Order"

Index: 10
Graph: Float
Aliases:
            "Float"

Index: 12
Graph: Char
Aliases:
            "Char"



---------
Index: 14
Class: A
Aliases:
            "A"
State: Unresolved: Explicitly Defined

Index: 15
Class: B
Aliases:
            "B"
State: Resolved
Nodes:
  (v, c, 16, 15, [illegible])
  (m, q, 15*, 15,
        « (v, self, 15*, 15, «», ()) (v, a, 14, 15, «», ()) (v, e, 16, 15, «», ()) »,
        {  })

Index: 16
Class: C
Aliases:
                "C"
State: Resolved
Nodes:

  (v, d, 14, 16, «», ())


Index: 17
Class: D
Aliases:
                "D"
State: Resolved
Nodes:
  (v, d, 14, 17, «», ())
  (v, e, 15, 17, «», ())

Index: 18
Class: B [C <- D]
Aliases:
                "B [C <- D]"
State: Unresolved: Derived with Base: 15 Real: 16 With: 17

-----------------------------------------------------
Class: A is a subclass of:

Class: B is a subclass of:
 Object (1) by: { }

Class: C is a subclass of:
 Object (1) by: { }

Class: D is a subclass of:
 Object (1) by: { }

Class: B [C <- D] is a subclass of:

----------------------------------------------------
Class: A is a child (subclass by inheritance) of:

Class: B is a child (subclass by inheritance) of: Object (1) by: { }

Class: C is a child (subclass by inheritance) of: Object (1) by: { }

Class: D is a child (subclass by inheritance) of: Object (1) by: { }

Class: B [C <- D] is a child (subclass by inheritance) of:
----------------------------------------------------
```

V.S A Complex Example with Multiple Inheritance

```
Input File: Tests:8 - Demo:Example 19

Input
-----------------------------------------------------
class C

        var c : C
        var o : Object
end C
-----------------------------------------------------
class H

        var h : Char

end H
-----------------------------------------------------
class F

        var f : C
        var j : Object

end F
-----------------------------------------------------
class E

        var f : Object
        method r (other : F) : F
               begin ... end

end E
-----------------------------------------------------
class D inherits C [Object <- Int]

        method q () : Object
               begin ... end

end D
-----------------------------------------------------
class B

        var b : B
        var c : C
        method p (c : C  d : D) : Object
               var b : B
               begin ... end

end B
-----------------------------------------------------
class G inherits F [C <- D]

        method s ()
               begin ... end

end G
-----------------------------------------------------
class A inherits : [B <- [illegible]    [illegible] C <- D]

        var a : A

end A
```

```
-----------------------------------------------------

=====================================================


Output
----------------------------------------------------
Index: 1
Graph: Object
Aliases:
            "Object"
            "Object [C <- D]"
            "Object [F [C <- D] <- G]"

Index: 2
Graph: Bool
Aliases:
            "Bool"
            "Bool [Comparable <- Bool]"
            "Bool [Comparable <- Order]"
            "Bool [Order <- Int]"
            "Bool [Order <- Float]"
            "Bool [Order <- Char]"
            "Bool [C [Object <- Int] <- D]"
            "Bool [F [C <- D] <- G]"

Index: 3
Graph: Ring
Aliases:
            "Ring"

Index: 4
Graph: Comparable
Aliases:
            "Comparable"

Index: 6
Graph: Int
Aliases:
            "Int"
            "Int [C [Object <- Int] <- D]" "Int [F [C <- D] <- G]"
            "Int [Object <- Int]"
            "Int [E [F <- G] <- A]"
            "Int [B [C <- D] <- A]"
```

```
Index: 7
Graph: Order
Aliases:
            "Order"

Index: 10
Graph: Float
Aliases:
            "Float"

Index: 12
Graph: Char
Aliases:
            "Char"
            "Char [H <- A]"


---------
Index: 14
Class: C
Aliases:
            "C"
            "C [H <- G]"
State: Resolved
Nodes:

  (v, c, 14*, 14, «», {})
  (v, o, 1, 14, «», {})


Index: 15
Class: H
Aliases:
            "H"
State: Resolved
Nodes:

  (v, h, 12, 15, «», {})


Index: 16
Class: F
Aliases:
            "F"
State: Resolved
Nodes:

  (v, f, 14, 16, «», {})
  (v, i, 1, 16, «», {})


Index: 17
Class: E
Aliases:
            "E"
State: Resolved
Nodes:

  (v, f, 1, 17, «», {})
  (m, r, 16, 17, « (v, self, 17*, 17, «», {}) (v, other, 17*, 17, «», {}) », { })
```

```
Index: 18
Class: D
Aliases:
            "D"
            "C [C <- D]"
            "C [Object <- Int] [C [Object <- Int] <- D]"
            "D [F [C <- D] <- G]"
            "D [B [C <- D] <- A]"
State: Resolved
Nodes:

  (m, q, 1, 18, « (v, self, 18*, 18, «», {}) », { })
  (v, c, 18*, 18, «», {})
  (v, o, 6, 18, «», {})


Index: 19
Class: C [Object <- Int]
Aliases:
            "C [Object <- Int]"
State: Resolved
Nodes:

  (v, c, 19*, 19, «», {})
  (v, o, 6, 19, «», {})


Index: 21
Class: B
Aliases:
            "B"
State: Resolved
Nodes:
  (v, b, 21*, 21, «», {})
  (v, c, 14, 21, «», {})
  (m, p, 1, 21,
        « (v, self, 21*, 21, «», {})  (v, c, 14, 21, «», {})  (v, d, 18, 21, «», {})  »,
        { (v, b, 21*, 21, «», {}) })

Index: 22
Class: G
Aliases:
            "G"
            "G [E [F <- G] <- A]"
State: Resolved
Nodes:

  (m, s, 22*, 22, « (v, self, 22*, 22, «», {})  », { })
  (v, f, 18, 22, «», {})
  (v, i, 6, 22, «», {})


Index: 23
Class: F [C <- D]
Aliases:
            "F [C <- D]"
            "F [Object <- Int] [C [Object <- Int] <- D]"
State: Resolved
Nodes:

  (v, f, 18, 23, «», {})
  (v, i, 6, 23, «», {})
```

```
Index: 26
Class: F [Object <- Int]
Aliases:
            "F [Object <- Int]"
State: Resolved
Nodes:

  (v, f, 19, 26, «», {})
  (v, i, 6, 26, «», {})


Index: 31
Class: A
Aliases:
            "A"
State: Resolved
Nodes:
  (v, a, 31*, 31, «», {})
  (v, f, 6, 31, «», {})
  (m, r, 22, 31, « (v, self, 31*, 31, «», {})  (v, other, 31*, 31, «», {})  », {  })
  (v, h, 12, 31, «», {})
  (v, b, 31*, 31, «», {})
  (v, c, 18, 31, «», {})
  (m, p, 6, 31,
        « (v, self, 31*, 31, «», {})  (v, c, 18, 31, «», {})  (v, d, 40, 31, «», {})  »,
        { (v, b, 31*, 31, «», {}) })

Index: 32
Class: E [F <- G]
Aliases:
            "E [F <- G]"
            "E [C <- D] [F [C <- D] <- G]"
State: Resolved
Nodes:

  (v, f, 6, 32, «», {})
  (m, r, 22, 32, « (v, self, 32*, 32, «», {})  (v, other, 32*, 32, «», {})  », {  })


Index: 34
Class: E [C <- D]
Aliases:
            "E [C <- D]"
            "E [Object <- Int] [C [Object <- Int] <- D]"
State: Resolved
Nodes:

  (v, f, 6, 34, «», {})
  (m, r, 23, 34, « (v, self, 34*, 34, «», {})  (v, other, 34*, 34, «», {})  », {  })


Index: 35
Class: E [Object <- Int]
Aliases:
            "E [Object <- Int]"
State: Resolved
Nodes:

  (v, f, 6, 35, «», {})
  (m, r, 26, 35, « (v, self, 35*, 35, «», {})  (v, other, 35*, 35, «», {})  », {  })
```

```
Index: 38
Class: B [C <- D]
Aliases:
            "B [C <- D]"
            "B [Object <- Int] [C [Object <- Int] <- D]"
State: Resolved
Nodes:

  (v, b, 38*, 38, «», ())
  (v, c, 18, 38, «», ())
  (m, p, 6, 38,
        « (v, self, 38*, 38, «», ())  (v, c, 18, 38, «», ())  (v, d, 40, 38, «», ())  »,
        ( (v, b, 38*, 38, «», ()) ))


Index: 39
Class: B [Object <- Int]
Aliases:
            "B [Object <- Int]"
State: Resolved
Nodes:

  (v, b, 39*, 39, «», ())
  (v, c, 19, 39, «», ())
  (m, p, 6, 39,
        « (v, self, 39*, 39, «», ())  (v, c, 19, 39, «», ())  (v, d, 40, 39, «», ())  »,
        ( (v, b, 39*, 39, «», ()) ))


Index: 40
Class: D [Object <- Int]
Aliases:
            "D [Object <- Int]"
            "D [Object <- Int] [C [Object <- Int] <- D]"
            "D [Object <- Int] [B [C <- D] <- A]"
State: Resolved
Nodes:
  (m, q, 6, 40, « (v, self, 40*, 40, «», ()) », ( ))
  (v, c, 40*, 40, «», ())
  (v, o, 6, 40, «», ())

-------------------------------------------------------
Class: C is a subclass of:
 Object (1) by: ( )
 C (14) by: ( (C, C) (Object, Object) )

Class: H is a subclass of:
 Object (1) by: ( )
 H (15) by: ( (Char, Char) (Int, Int) (Bool, Bool) )

Class: F is a subclass of:
 Object (1) by: ( )
 F (16) by: ( (C, C) (Object, Object) )

Class: E is a subclass of:
 Object (1) by: ( )
 E (17) by: ( (Object, Object) (F, F) (E, E) (C, C) )

Class: D is a subclass of:
 Object (1) by: ( )
 C (14) by: ( (C, D) (Object, Int) )
 D (18) by: ( (Object, Object) (D, D) (Int, Int) (Bool, Bool) )
 C [Object <- Int] (19) by: ( (C [Object <- Int], D) (Int, Int) (Bool, Bool) )
```

```
Class: C [Object <- Int] is a subclass of:
 Object (1) by: { }
 C (14) by: { (C, C [Object <- Int]) (Object, Int) }
 C [Object <- Int] (19) by: { (C [Object <- Int], C [Object <- Int]) (Int, Int)
        (Bool, Bool) }

Class: B is a subclass of:
 Object (1) by: { }
 B (21) by: { (B, B) (C, C) (Object, Object) (D, D) (Int, Int) (Bool, Bool) }

Class: G is a subclass of:
 Object (1) by: { }
 F (16) by: { (C, D) (Object, Int) }
 G (22) by: { (G, G) (D, D) (Int, Int) (Object, Object) (Bool, Bool) }
 F [C <- D] (23) by: { (D, D) (Int, Int) (Object, Object) (Bool, Bool) }
 F [Object <- Int] (26) by: { (C [Object <- Int], D) (Int, Int) (Bool, Bool) }

Class: F [C <- D] is a subclass of:
 Object (1) by: { }
 F (16) by: { (C, D) (Object, Int) }
 F [C <- D] (23) by: { (D, D) (Int, Int) (Object, Object) (Bool, Bool) }
 F [Object <- Int] (26) by: { (C [Object <- Int], D) (Int, Int) (Bool, Bool) }

Class: F [Object <- Int] is a subclass of:
 Object (1) by: { }
 F (16) by: { (C, C [Object <- Int]) (Object, Int) }
 F [Object <- Int] (26) by: { (C [Object <- Int], C [Object <- Int]) (Int, Int)
        (Bool, Bool) }

Class: A is a subclass of:
 Object (1) by: { }
 H (15) by: { (Char, Char) (Int, Int) (Bool, Bool) }
 E (17) by: { (Object, Int) (F, G) (E, A) (C, D) }
 B (21) by: { (B, A) (C, D) (Object, Int) (D, D [Object <- Int]) (Int, Int) (Bool, Bool) }
 A (31) by: { (A, A) (Int, Int) (G, G) (Char, Char) (D, D)
        (D [Object <- Int], D [Object <- Int]) (Bool, Bool) (Object, Object) }
 E [F <- G] (32) by: { (Int, Int) (G, G) (E [F <- G], A) (Bool, Bool) (D, D)
        (Object, Object) }
 E [C <- D] (34) by: { (Int, Int) (F [C <- D], G) (E [C <- D], A) (Bool, Bool) (D, D)
        (Object, Object) }
 E [Object <- Int] (35) by: { (Int, Int) (F [Object <- Int], G) (E [Object <- Int], A)
        (Bool, Bool) (C [Object <- Int], D) }
 B [C <- D] (38) by: { (B [C <- D], A) (D, D) (Int, Int)
        (D [Object <- Int], D [Object <- Int]) (Object, Object) (Bool, Bool) }
 B [Object <- Int] (39) by: { (B [Object <- Int], A) (C [Object <- Int], D) (Int, Int)
        (D [Object <- Int], D [Object <- Int]) (Bool, Bool) }

Class: E [F <- G] is a subclass of:
 Object (1) by: { }
 E (17) by: { (Object, Int) (F, G) (E, E [F <- G]) (C, D) }
 E [F <- G] (32) by: { (Int, Int) (G, G) (E [F <- G], E [F <- G]) (Bool, Bool) (D, D)
        (Object, Object) }
 E [C <- D] (34) by: { (Int, Int) (F [C <- D], G) (E [C <- D], E [F <- G]) (Bool, Bool)
        (D, D) (Object, Object) }
 E [Object <- Int] (35) by: { (Int, Int) (F [Object <- Int], G)
        (F [Object <- Int], E [F <- G]) (Bool, Bool) (C [Object <- Int], D) }
```

```
Class: E [C <- D] is a subclass of:
 Object (1) by: ( )
 E (17) by: ( (Object, Int) (F, F [C <- D]) (E, E [C <- D]) (C, D) )
 E [C <- D] (34) by: ( (Int, Int) (F [C <- D], F [C <- D]) (E [C <- D], E [C <- D])
       (Bool, Bool) (D, D) (Object, Object) )
 E [Object <- Int] (35) by: ( (Int, Int) (F [Object <- Int], F [C <- D])
       (E [Object <- Int], E [C <- D]) (Bool, Bool) (C [Object <- Int], D) )

Class: E [Object <- Int] is a subclass of:
 Object (1) by: ( )
 E (17) by: ( (Object, Int) (F, F [Object <- Int]) (E, E [Object <- Int])
       (C, C [Object <- Int]) )
 E [Object <- Int] (35) by: ( (Int, Int) (F [Object <- Int], F [Object <- Int])
       (E [Object <- Int], E [Object <- Int]) (Bool, Bool)
       (C [Object <- Int], C [Object <- Int]) )

Class: B [C <- D] is a subclass of:
 Object (1) by: ( )
 B (21) by: ( (B, B [C <- D]) (C, D) (Object, Int) (D, D [Object <- Int]) (Int, Int)
       (Bool, Bool) )
 B [C <- D] (38) by: ( (B [C <- D], B [C <- D]) (D, D) (Int, Int)
       (D [Object <- Int], D [Object <- Int]) (Object, Object) (Bool, Bool) )
 B [Object <- Int] (39) by: ( (B [Object <- Int], B [C <- D]) (C [Object <- Int], D)
       (Int, Int) (D [Object <- Int], D [Object <- Int]) (Bool, Bool) )

Class: B [Object <- Int] is a subclass of:
 Object (1) by: ( )
 B (21) by: ( (B, B [Object <- Int]) (C, C [Object <- Int]) (Object, Int)
       (D, D [Object <- Int]) (Int, Int) (Bool, Bool) )
 B [Object <- Int] (39) by: ( (B [Object <- Int], B [Object <- Int])
       (C [Object <- Int], C [Object <- Int]) (Int, Int)
       (D [Object <- Int], D [Object <- Int]) (Bool, Bool) )

Class: D [Object <- Int] is a subclass of:
 Object (1) by: ( )
 C (14) by: ( (C, D [Object <- Int]) (Object, Int) )
 D (18) by: ( (Object, Int) (D, D [Object <- Int]) (Int, Int) (Bool, Bool) )
 C [Object <- Int] (19) by: ( (C [Object <- Int], D [Object <- Int]) (Int, Int)
       (Bool, Bool) )
 D [Object <- Int] (40) by: ( (Int, Int) (D [Object <- Int], D [Object <- Int])
       (Bool, Bool) )

--------------------------------------------------------
Class: C is a child (subclass by inheritance) of:
 Object (1) by: ( )
 C (14) by: ( (C, C) (Object, Object) )

Class: H is a child (subclass by inheritance) of:
 Object (1) by: ( )
 H (15) by: ( (Char, Char) (Int, Int) (Bool, Bool) )

Class: F is a child (subclass by inheritance) of:
 Object (1) by: ( )
 F (16) by: ( (C, C) (Object, Object) )

Class: E is a child (subclass by inheritance) of:
 Object (1) by: ( )
 E (17) by: ( (Object, Object) (F, F) (E, E) (C, C) )
```

```
Class: D is a child (subclass by inheritance) of:
 Object (1) by: { }
 C (14) by: { (C, D) (Object, Int) }
 D (18) by: { (Object, Object) (D, D) (Int, Int) (Bool, Bool) }
 C [Object <- Int] (19) by: { (C [Object <- Int], D) (Int, Int) (Bool, Bool) }

Class: C [Object <- Int] is a child (subclass by inheritance) of:
 Object (1) by: { }
 C (14) by: { (C, C [Object <- Int]) (Object, Int) }
 C [Object <- Int] (19) by: { (C [Object <- Int], C [Object <- Int]) (Int, Int)
       (Bool, Bool) }

Class: B is a child (subclass by inheritance) of:
 Object (1) by: { }
 B (21) by: { (B, B) (C, C) (Object, Object) (D, D) (Int, Int) (Bool, Bool) }

Class: G is a child (subclass by inheritance) of:
 Object (1) by: { }
 F (16) by: { (C, D) (Object, Int) }
 G (22) by: { (G, G) (D, D) (Int, Int) (Object, Object) (Bool, Bool) }
 F [C <- D] (23) by: { (D, D) (Int, Int) (Object, Object) (Bool, Bool) }
 F [Object <- Int] (26) by: { (C [Object <- Int], D) (Int, Int) (Bool, Bool) }

Class: F [C <- D] is a child (subclass by inheritance) of:
 Object (1) by: { }
 F (16) by: { (C, D) (Object, Int) }
 F [C <- D] (23) by: { (D, D) (Int, Int) (Object, Object) (Bool, Bool) }
 F [Object <- Int] (26) by: { (C [Object <- Int], D) (Int, Int) (Bool, Bool) }

Class: F [Object <- Int] is a child (subclass by inheritance) of:
 Object (1) by: { }
 F (16) by: { (C, C [Object <- Int]) (Object, Int) }
 F [Object <- Int] (26) by: { (C [Object <- Int], C [Object <- Int]) (Int, Int)
       (Bool, Bool) }

Class: A is a child (subclass by inheritance) of:
 Object (1) by: { }
 H (15) by: { (Char, Char) (Int, Int) (Bool, Bool) }
 E (17) by: { (Object, Int) (F, G) (E, A) (C, D) }
 B (21) by: { (B, A) (C, D) (Object, Int) (D, D [Object <- Int]) (Int, Int) (Bool, Bool) }
 A (31) by: { (A, A) (Int, Int) (G, G) (Char, Char) (D, D)
       (D [Object <- Int], D [Object <- Int]) (Bool, Bool) (Object, Object) }
 E [F <- G] (32) by: { (Int, Int) (G, G) (E [F <- G], A) (Bool, Bool) (D, D)
       (Object, Object) }
 E [C <- D] (34) by: { (Int, Int) (F [C <- D], D) (E [C <- D], A) (Bool, Bool) (D, D)
       (Object, Object) }
 E [Object <- Int] (35) by: { (Int, Int) (F [Object <- Int], G) (E [Object <- Int], A)
       (Bool, Bool) (C [Object <- Int], D) }
 B [C <- D] (38) by: { (B [C <- D], A) (D, D) (Int, Int)
       (D [Object <- Int], D [Object <- Int]) (Object, Object) (Bool, Bool) }
 B [Object <- Int] (39) by: { (B [Object <- Int], A) (C [Object <- Int], D) (Int, Int)
       (D [Object <- Int], D [Object <- Int]) (Bool, Bool) }

Class: F [F <- G] is a child (subclass by inheritance) of:
 Object (1) by: { }
 E (17) by: { (Object, Int) (F, G) (E, E [F <- G]) (C, D) }
 E [F <- G] (32) by: { (Int, Int) (G, G) (F [F <- G], E [F <- G]) (Bool, Bool) (D, D)
       (Object, Object) }
 E [C <- D] (34) by: { (Int, Int) (F [C <- D], C) (E [C <- D], E [F <- G]) (Bool, Bool)
       (D, D) (Object, Object) }
 E [Object <- Int] (35) by: { (Int, Int) (F [Object <- Int], G)
       (F [Object <- Int], F [F <- G]) (Bool, Bool) (C [Object <- Int], D) }
```

311

```
Class: E [C <- D] is a child (subclass by inheritance) of:
 Object (1) by: { }
 E (17) by: { (Object, Int) (F, F [C <- D]) (E, E [C <- D]) (C, D) }
 E [C <- D] (34) by: { (Int, Int) (F [C <- D], F [C <- D]) (E [C <- D], E [C <- D])
        (Bool, Bool) (D, D) (Object, Object) }
 E [Object <- Int] (35) by: { (Int, Int) (F [Object <- Int], F [C <- D])
        (E [Object <- Int], E [C <- D]) (Bool, Bool) (C [Object <- Int], D) }

Class: E [Object <- Int] is a child (subclass by inheritance) of:
 Object (1) by: { }
 E (17) by: { (Object, Int) (F, F [Object <- Int]) (E, E [Object <- Int])
        (C, C [Object <- Int]) }
 E [Object <- Int] (35) by: { (Int, Int) (F [Object <- Int], F [Object <- Int])
        (E [Object <- Int], E [Object <- Int]) (Bool, Bool)
        (C [Object <- Int], C [Object <- Int]) }

Class: B [C <- D] is a child (subclass by inheritance) of:
 Object (1) by: { }
 B (21) by: { (B, B [C <- D]) (C, D) (Object, Int) (D, D [Object <- Int]) (Int, Int)
        (Bool, Bool) }
 B [C <- D] (38) by: { (B [C <- D], B [C <- D]) (D, D) (Int, Int)
        (D [Object <- Int], D [Object <- Int]) (Object, Object) (Bool, Bool) }
 B [Object <- Int] (39) by: { (B [Object <- Int], B [C <- D]) (C [Object <- Int], D)
        (Int, Int) (D [Object <- Int], D [Object <- Int]) (Bool, Bool) }

Class: B [Object <- Int] is a child (subclass by inheritance) of:
 Object (1) by: { }
 B (21) by: { (B, B [Object <- Int]) (C, C [Object <- Int]) (Object, Int)
        (D, D [Object <- Int]) (Int, Int) (Bool, Bool) }
 B [Object <- Int] (39) by: { (B [Object <- Int], B [Object <- Int])
        (C [Object <- Int], C [Object <- Int]) (Int, Int)
        (D [Object <- Int], D [Object <- Int]) (Bool, Bool) }

Class: D [Object <- Int] is a child (subclass by inheritance) of:
 Object (1) by: { }
 C (14) by: { (C, D [Object <- Int]) (Object, Int) }
 D (18) by: { (Object, Int) (D, D [Object <- Int]) (Int, Int) (Bool, Bool) }
 C [Object <- Int] (39) by: { (C [Object <- Int], D [Object <- Int]) (Int, Int)
        (Bool, Bool) }
 D [Object <- Int] (40) by: { (Int, Int) (D [Object <- Int], D [Object <- Int])
        (Bool, Bool) }
-----------------------------------------------------
```

V.T PS Examples: Stacks and Arrays

```
Input File: Tests:8 - Demo:Example 20

Input

------------------------------------------------------
class Array
            method at (i : Int) : Object
                        begin ... end

            method atput (i : Int  x : Object) : Array
                        begin ... end

            method initialize (size : Int) : Array
                        begin ... end

end Array
------------------------------------------------------
class BoolStack inherits Stack [Object <- Bool]

end BoolStack
------------------------------------------------------
class IntStack inherits Stack [Object <- Int]

end IntStack
------------------------------------------------------
class Stack

        var space : Array
        var index : Int
        method push (x : Object) : Stack
               begin ... end
        method top () : Object
               begin ... end
        method pop () : Stack
               begin ... end
        method initialize (size : Int) : Stack
               begin ... end
end Stack

------------------------------------------------------

[illegible]
```

```
Output
-----------------------------------------------------
Index: 1
Graph: Object
Aliases:
            "Object"

Index: 2
Graph: Bool
Aliases:
            "Bool"
            "Bool [Comparable <- Bool]"
            "Bool [Comparable <- Order]"
            "Bool [Order <- Int]"
            "Bool [Order <- Float]"
            "Bool [Order <- Char]"
            "Bool [Stack [Object <- Bool] <- BoolStack]"

Index: 3
Graph: Ring
Aliases:
            "Ring"

Index: 4
Graph: Comparable
Aliases:
            "Comparable"

Index: 6
Graph: Int
Aliases:
            "Int"
            "Int [Object <- Bool]"
            "Int [Stack [Object <- Bool] <- BoolStack]"
            "Int [Object <- Int]"
            "Int [Stack [Object <- Int] <- IntStack]"

Index: 7
Graph: Order
Aliases:
            "Order"

Index: 10
Graph: Float
Aliases:
            "Float"

Index: 12
Graph: Char
Aliases:
            "Char"



---------
Index: 14
Class: Array
Aliases:
            "Array"
State: Resolved
Nodes:

  (m, at, 1, 14, « (v, se [illegible], [illegible], [illegible], «», [illegible] [illegible], [illegible], 6, 4, «», [illegible]) », [illegible] ))
  (m, atput, 14*, 14, « (v, se[illegible], 4*, .4, «», [illegible]
  (v, i, 6, 14, «», {})
  (v, x, 1, 14, «», {})  », {  })
  (m, initialize, 14*, 14, « (v, self. 14*, 14, «», {})  (v, size, 6, 14, «», {})  », {  })


Index: 16
Class: Stack
Aliases:
        "Stack"
State: Resolved
Nodes:

  (v, space, 14, 16, «», {})
  (v, index, 6, 16, «», {})
  (m, push, 16*, 16, « (v, self, 16*, 16, «», {})  (v, x, 1, 16, «», {})  », {  })
  (m, top, 1, 16, « (v, self, 16*, 16, «», {})  », {  })
  (m, pop, 16*, 16, « (v, self, 16*, 16, «», {})  », {  })
  (m, initialize, 16*, 16, « (v, self, 16*, 16, «», {})  (v, size, 6, 16, «», {})  », {  })


Index: 17
Class: BoolStack
Aliases:
            "Stack [Object <- Bool]"
            "BoolStack"
State: Resolved
Nodes:

  (v, space, 18, 17, «», {})
  (v, index, 6, 17, «», {})
  (m, push, 17*, 17, « (v, self, 17*, 17, «», {})  (v, x, 2, 17, «», {})  », {  })
  (m, top, 2, 17, « (v, self, 17*, 17, «», {})  », {  })
  (m, pop, 17*, 17, « (v, self, 17*, 17, «», {})  », {  })
  (m, initialize, 17*, 17, « (v, self, 17*, 17, «», {})  (v, size, 6, 17, «», {})  », {  })


Index: 18
Class: Array [Object <- Bo
Aliases:
            "Array [Object <- Bool "
            "Array [Object <- Bool] [Stack [Object <- Bool] <- BoolStack]"
State: Resolved
Nodes:

  (m, at, 2, 18, « (v, self, 18*, 18, «», {})  (v, i, 6, 18, «», {})  », {  })
  (m, atput, 18*, 18,
       « (v, self, 18*, 18, «», {})  (v, i, 6, 18, «», {})  (v, x, 2, 18, «», {})  »,
       {  })
  (m, initialize, 18*, 18, « (v, self, 18*, 18, «», {})  (v, size, 6, 18, «», {})  », {  })
```

```
Index: 24
Class: IntStack
Aliases:
        "Stack [Object <- Int]"
        "IntStack"
State: Resolved
Nodes:

  (v, space, 25, 24, «», ())
  (v, index, 6, 24, «», ())
  (m, push, 24*, 24, « (v, self, 24*, 24, «», ())  (v, x, 6, 24, «», ())  », (  ))
  (m, top, 6, 24, « (v, self, 24*, 24, «», ())  », (  ))
  (m, pop, 24*, 24, « (v, self, 24*, 24, «», ())  », (  ))
  (m, initialize, 24*, 24, « (v, self, 24*, 24, «», ())  (v, size, 6, 24, «», ())  », (  ))


Index: 25
Class: Array [Object <- Int]
Aliases:
           "Array [Object <- Int]"
           "Array [Object <- Int  [Stack [Object <- Int] <- IntStack]"
State: Resolved
Nodes:

  (m, at, 6, 25, « (v, self, 25*, 25, «», ())  (v, i, 6, 25, «», ())  », (  ))
  (m, atput, 25*, 25,
        « (v, self, 25*, 25, «», ())  (v, i, 6, 25, «», ())  (v, x, 6, 25, «», ())  »,
        (  ))
  (m, initialize, 25*, 25, « (v, self, 25*, 25, «», ())  (v, size, 6, 25, «», ())  », (  ))

-------------------------------------------------
Class: Array is a subclass of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Object) (Array, Array) (Int, Int) (Bool, Bool) )

Class: Stack is a subclass of:
 Object (1) by: ( )
 Stack (16) by: ( (Array, Array) (Int, Int) (Stack, Stack) (Object, Object) (Bool, Bool) )

Class: BoolStack is a subclass of:
 Object (1) by: ( )
 Stack (16) by: ( (Array, Array [Object <- Bool]) (Int, Int) (Stack, BoolStack)
       (Object, Bool) (Bool, Bool) )
 BoolStack (17) by: ( (Array [Object <- Bool], Array [Object <- Bool]) (Int, Int)
       (BoolStack, BoolStack) (Bool, Bool) )

Class: Array [Object <- Bool] is a subclass of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Bool) (Array, Array [Object <- Bool]) (Int, Int) (Bool, Bool) )
 Array [Object <- Bool] (18) by: ( (Bool, Bool) (Array [Object <- Bool],
       Array [Object <- Bool]) (Int, Int) )

Class: IntStack is a subclass of:
 Object (1) by: ( )
 Stack (16) by: ( (Array, Array [Object <- Int]) (Int, Int) (Stack, IntStack)
       (Object, Int) (Bool, Bool) )
 IntStack (24) by: ( (Array [Object <- Int], Array [Object <- Int]) (Int, Int)
       (IntStack, IntStack) (Bool, Bool) )
```

```
Class: Array [Object <- Int] is a subclass of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Int) (Array, Array [Object <- Int]) (Int, Int) (Bool, Bool) )
 Array [Object <- Int] (25) by: ( (Int, Int) (Array [Object <- Int],
       Array [Object <- Int]) (Bool, Bool) )

-----------------------------------------------------
Class: Array is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Object) (Array, Array) (Int, Int) (Bool, Bool) )

Class: Stack is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Stack (16) by: ( (Array, Array) (Int, Int) (Stack, Stack) (Object, Object) (Bool, Bool) )

Class: BoolStack is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Stack (16) by: ( (Array, Array [Object <- Bool]) (Int, Int) (Stack, BoolStack)
       (Object, Bool) (Bool, Bool) )
 BoolStack (17) by: ( (Array [Object <- Bool], Array [Object <- Bool]) (Int, Int)
       (BoolStack, BoolStack) (Bool, Bool) )

Class: Array [Object <- Bool] is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Bool) (Array, Array [Object <- Bool]) (Int, Int) (Bool, Bool) )
 Array [Object <- Bool] (18) by: ( (Bool, Bool)
       (Array [Object <- Bool], Array [Object <- Bool]) (Int, Int) )

Class: IntStack is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Stack (16) by: ( (Array, Array [Object <- Int]) (Int, Int) (Stack, IntStack)
       (Object, Int) (Bool, Bool) )
 IntStack (24) by: ( (Array [Object <- Int], Array [Object <- Int]) (Int, Int)
       (IntStack, IntStack) (Bool, Bool) )

Class: Array [Object <- Int] is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Int) (Array, Array [Object <- Int]) (Int, Int) (Bool, Bool) )
 Array [Object <- Int] (25) by: ( (Int, Int)
       (Array [Object <- Int], Array [Object <- Int]) (Bool, Bool) )
-----------------------------------------------------
```

V.U PS Examples: Matrices and Arrays

```
Input File: Tests:8 - Demo:Example 21

Input
-------------------------------------------------------
class Array

           method at (i : Int) : Object
                      begin ... end

           method atput (i : Int  x : Object) : Array
                      begin ... end

           method initialize (size : Int) : Array
                      begin ... end

end Array
-------------------------------------------------------
class BoolMatrix inherits Matrix [Ring <- Bool]

end BoolMatrix
-------------------------------------------------------
class DoubleArray inherits Array [Object <- Array]

end DoubleArray
-------------------------------------------------------
class DoubleRingArray inherits DoubleArray [Object <- Ring]

end DoubleRingArray
-------------------------------------------------------
class Matrix inherits Ring [Object <- DoubleRingArray]

           var i : Int
           var j : Int
           var r : Array [Object <- Ring]

end Matrix
-------------------------------------------------------
class MatrixMatrix inherits Matrix [Ring <- Matrix]

end MatrixMatrix
-------------------------------------------------------

=====================
Output
-----------------------------------------------------
Index: 1
Graph: Object
Aliases:
            "Object"

Index: 2
Graph: Bool
Aliases:
            "Bool"
            "Bool [Comparable <- Bool]"
            "Bool [Comparable <- Order]"
            "Bool [Order <- Int]"
            "Bool [Order <- Float]"
            "Bool [Order <- Char]"
            "Ring [Ring <- Bool]"

Index: 3
Graph: Ring
Aliases:
            "Ring"
            "Ring [Object <- DoubleRingArray]"

Index: 4
Graph: Comparable
Aliases:
            "Comparable"

Index: 6
Graph: Int
Aliases:
            "Int"
            "Int [Object <- Array]"
            "Int [Array [Object <- Array] <- DoubleArray]"
            "Int [Object <- Ring]"
            "Int [DoubleArray [Object <- Ring] <- DoubleRingArray]"
            "Int [Ring <- Bool]"
            "Int [Matrix [Ring <- Bool] <- BoolMatrix]"
            "Int [Ring <- Matrix]"
            "Int [Matrix [Ring <- Matrix] <- MatrixMatrix]"
```

```
Index: 7
Graph: Order
Aliases:
          "Order"

Index: 10
Graph: Float
Aliases:
          "Float"

Index: 12
Graph: Char
Aliases:
          "Char"


---------
Index: 14
Class: Array
Aliases:
          "Array"
          "Array [Array [Object <- Array] <- DoubleArray]"
State: Resolved
Nodes:

  (m, at, 1, 14, « (v, self, 14*, 14, «», {})  (v, i, 6, 14, «», {})  », {  })
  (m, atput, 14*, 14,
       « (v, self, 14*, 14, «», {})  (v, i, 6, 14, «», {})  (v, x, 1, 14, «», {})  »,
       {  })
  (m, initialize, 14*, 14, « (v, self, 14*, 14, «», {})  (v, size, 6, 14, «», {})  », {  })


Index: 16
Class: Matrix
Aliases:
       "Matrix"
       "Ring [Ring <- Matrix] "
State: Resolved
Nodes:

  (v, i, 6, 16, «», {})
  (v, j, 6, 16, «», {})
  (v, r, 25, 16, «», {})
  (m, plus, 16*, 16, « (v, self, 16*, 16, «», {})  (v, other, 16*, 16, «», {})  », {  })
  (m, zero, 16*, 16, « (v, self, 16*, 16, «», {})  », {  })


Index: 19
Class: DoubleArray
Aliases:
          "Array [Object <- Array]"
          "DoubleArray"
State: Resolved
Nodes:

  (m, at, 14, 19, « (v, self, 19*, 19, «», {})  (v, i, 6, 19, «», {})  », {  })
  (m, atput, 19*, 19,
       « (v, self, 19*, 19, «», {})  (v, i, 6, 19, «», {})  (v, x, 14, 19, «», {})  »,
       {  })
  (m, initialize, 19*, 19, « (v, self, 19*, 19, «», {})  (v, size, 6, 19, «», {})  », {  })
```

```
Index: 23
Class: DoubleRingArray
Aliases:
        "DoubleArray [Object <- Ring]"
        "DoubleRingArray"
State: Resolved
Nodes:

  (m, at, 25, 23, « (v, self, 23*, 23, «», {})  (v, i, 6, 23, «», {})  », {  })
  (m, atput, 23*, 23,
        « (v, self, 23*, 23, «», {})  (v, i, 6, 23, «», {})  (v, x, 25, 23, «», {})  »,
        {  })
  (m, initialize, 23*, 23, « (v, self, 23*, 23, «», {})  (v, size, 6, 23, «», {})  », {  })


Index: 25
Class: Array [Object <- Ring]
Aliases:
        "Array [Object <- Ring]"
        "Array [Object <- Ring] [DoubleArray [Object <- Ring] <- DoubleRingArray]"
State: Resolved
Nodes:

  (m, at, 3, 25, « (v, self, 25*, 25, «», {})  (v, i, 6, 25, «», {})  », {  })
  (m, atput, 25*, 25,
        « (v, self, 25*, 25, «», {})  (v, i, 6, 25, «», {})  (v, x, 3, 25, «», {})  »,
        {  })
  (m, initialize, 25*, 25, « (v, self, 25*, 25, «», {})  (v, size, 6, 25, «», {})  », {  })


Index: 29
Class: BoolMatrix
Aliases:
        "Matrix [Ring <- Bool]"
        "BoolMatrix"
State: Resolved
Nodes:

  (v, i, 6, 29, «», {})
  (v, j, 6, 29, «», {})
  (v, r, 32, 29, «», {})
  (m, plus, 29*, 29, « (v, self, 29*, 29, «», {})  (v, other, 29*, 29, «», {})  », {  })
  (m, zero, 29*, 29, « (v, self, 29*, 29, «», {})  », {  })


Index: 32
Class: Array [Object <- Ring] [Ring <- Bool]
Aliases:
           "Array [Object <- Ring] [Ring <- Bool]"
           "Array [Object <- Ring] [Ring <- Bool] [Matrix [Ring <- Bool] <- BoolMatrix]"
State: Resolved
Nodes:

  (m, at, 2, 32, « (v, self, 32*, 32, «», {})  (v, i, 6, 32, «», {})  », {  })
  (m, atput, 32*, 32,
        « (v, self, 32*, 32, «», {})  (v, i, 6, 32, «», {})  (v, x, 2, 32, «», {})  »,
        {  })
  (m, initialize, 32*, 32, « (v, self, 32*, 32, «», {})  (v, size, 6, 32, «», {})  », {  })
```

```
Index: 36
Class: MatrixMatrix
Aliases:
        "Matrix [Ring <- Matrix]"
        "MatrixMatrix"
State: Resolved
Nodes:

  (v, i, 6, 36, «», ())
  (v, j, 6, 36, «», ())
  (v, r, 39, 36, «», ())
  (m, plus, 36*, 36, « (v, self, 36*, 36, «», ())  (v, other, 36*, 36, «», ())  », (  ))
  (m, zero, 36*, 36, « (v, self, 36*, 36, «», ())  », (  ))


Index: 39
Class: Array [Object <- Ring] [Ring <- Matrix]
Aliases:
            "Array [Object <- Ring] [Ring <- Matrix]"
            "Array [Object <- Ring] [Ring <- Matrix]
                      [Matrix [Ring <- Matrix] <- MatrixMatrix]"
State: Resolved
Nodes:
  (m, at, 16, 39, « (v, self, 39*, 39, «», ())  (v, i, 6, 39, «», ())  », (  ))
  (m, atput, 39*, 39,
       « (v, self, 39*, 39, «», ())  (v, i, 6, 39, «», ())  (v, x, 16, 39, «», ())  »,
       (  ))
  (m, initialize, 39*, 39, « (v, self, 39*, 39, «», ())  (v, size, 6, 39, «», ())  », (  ))

------------------------------------------------------
Class: Array is a subclass of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Object) (Array, Array) (Int, Int) (Bool, Bool) )

Class: Matrix is a subclass of:
 Object (1) by: ( )
 Ring (3) by: ( (Ring, Matrix) )
 Matrix (16) by: ( (Int, Int) (Array [Object <- Ring], Array [Object <- Ring])
       (Matrix, Matrix) (Bool, Bool) (Ring, Ring) )

Class: DoubleArray is a subclass of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Array) (Array, DoubleArray) (Int, Int) (Bool, Bool) )
 DoubleArray (19) by: ( (Array, Array) (DoubleArray, DoubleArray) (Int, Int)
       (Object, Object) (Bool, Bool) )

Class: DoubleRingArray is a subclass of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Array [Object <- Ring]) (Array, DoubleRingArray) (Int, Int)
       (Bool, Bool) )
 DoubleArray (19) by: ( (Array, Array [Object <- Ring]) (DoubleArray, DoubleRingArray)
       (Int, Int) (Object, Ring) (Bool, Bool) )
 DoubleRingArray (23) by: ( (Array [Object <- Ring], Array [Object <- Ring])
       (DoubleRingArray, DoubleRingArray) (Int, Int) (Ring, Ring) (Bool, Bool) )

Class: Array [Object <- Ring] is a subclass of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Ring) (Array, Array [Object <- Ring]) (Int, Int) (Bool, Bool) )
 Array [Object <- Ring] (25) by: ( (Ring, Ring)
       (Array [Object <- Ring], Array [Object <- Ring]) (Int, Int) (Bool, Bool) )
```

```
Class: BoolMatrix is a subclass of:
 Object (1) by: ( )
 Ring (3) by: ( (Ring, BoolMatrix) )
 Matrix (16) by: ( (Int, Int)
       (Array [Object <- Ring], Array [Object <- Ring] [Ring <- Bool])
       (Matrix, BoolMatrix) (Bool, Bool) (Ring, Bool) )
 BoolMatrix (29) by: ( (Int, Int)
       (Array [Object <- Ring] [Ring <- Bool], Array [Object <- Ring] [Ring <- Bool])
       (BoolMatrix, BoolMatrix) (Bool, Bool) )


Class: Array [Object <- Ring] [Ring <- Bool] is a subclass of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Bool) (Array, Array [Object <- Ring] [Ring <- Bool])
       (Int, Int) (Bool, Bool) )
 Array [Object <- Ring] (25) by: ( (Ring, Bool)
       (Array [Object <- Ring], Array [Object <- Ring] [Ring <- Bool]) (Int, Int)
       (Bool, Bool) )
 Array [Object <- Ring] [Ring <- Bool] (32) by: ( (Bool, Bool)
       (Array [Object <- Ring] [Ring <- Bool], Array [Object <- Ring] [Ring <- Bool])
       (Int, Int) )

Class: MatrixMatrix is a subclass of:
 Object (1) by: ( )
 Ring (3) by: ( (Ring, MatrixMatrix) )
 Matrix (16) by: ( (Int, Int)
       (Array [Object <- Ring], Array [Object <- Ring] [Ring <- Matrix])
       (Matrix, MatrixMatrix) (Bool, Bool) (Ring, Matrix) )
 MatrixMatrix (36) by: ( (Int, Int)
       (Array [Object <- Ring] [Ring <- Matrix], Array [Object <- Ring] [Ring <- Matrix])
       (MatrixMatrix, MatrixMatrix) (Bool, Bool) (Matrix, Matrix)
       (Array [Object <- Ring], Array [Object <- Ring]) (Ring, Ring) )

Class: Array [Object <- Ring] [Ring <- Matrix] is a subclass of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Matrix) (Array, Array [Object <- Ring] [Ring <- Matrix])
       (Int, Int) (Bool, Bool) )
 Array [Object <- Ring] (25) by: ( (Ring, Matrix)
       (Array [Object <- Ring], Array [Object <- Ring] [Ring <- Matrix]) (Int, Int)
       (Bool, Bool) )
 Array [Object <- Ring] [Ring <- Matrix] (39) by: ( (Matrix, Matrix)
       (Array [Object <- Ring] [Ring <- Matrix], Array [Object <- Ring] [Ring <- Matrix])
       (Int, Int) (Array [Object <- Ring], Array [Object <- Ring]) (Bool, Bool)
       (Ring, Ring) )

-----------------------------------------------------
Class: Array is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Object) (Array, Array) (Int, Int) (Bool, Bool) )

Class: Matrix is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Ring (3) by: ( (Ring, Matrix) )
 Matrix (16) by: ( (Int, Int) (Array [Object <- Ring], Array [Object <- Ring])
       (Matrix, Matrix) (Bool, Bool) (Ring, Ring) )

Class: DoubleArray is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Array) (Array, DoubleArray) (Int, Int) (Bool, Bool) )
 DoubleArray (19) by: ( (Array, Array) (DoubleArray, DoubleArray) (Int, Int)
       (Object, Object) (Bool, Bool) )
```

```
Class: DoubleRingArray is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Array [Object <- Ring]) (Array, DoubleRingArray) (Int, Int)
       (Bool, Bool) )
 DoubleArray (19) by: ( (Array, Array [Object <- Ring]) (DoubleArray, DoubleRingArray)
       (Int, Int) (Object, Ring) (Bool, Bool) )
 DoubleRingArray (23) by: ( (Array [Object <- Ring], Array [Object <- Ring])
       (DoubleRingArray, DoubleRingArray) (Int, Int) (Ring, Ring) (Bool, Bool) )

Class: Array [Object <- Ring] is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Ring) (Array, Array [Object <- Ring]) (Int, Int) (Bool, Bool) )
 Array [Object <- Ring] (25) by: ( (Ring, Ring)
       (Array [Object <- Ring], Array [Object <- Ring]) (Int, Int) (Bool, Bool) )

Class: BoolMatrix is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Ring (3) by: ( (Ring, BoolMatrix) )
 Matrix (16) by: ( (Int, Int)
       (Array [Object <- Ring], Array [Object <- Ring] [Ring <- Bool])
       (Matrix, BoolMatrix) (Bool, Bool) (Ring, Bool) )
 BoolMatrix (29) by: ( (Int, Int)
       (Array [Object <- Ring] [Ring <- Bool], Array [Object <- Ring] [Ring <- Bool])
       (BoolMatrix, BoolMatrix) (Bool, Bool) )

Class: Array [Object <- Ring] [Ring <- Bool] is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Bool) (Array, Array [Object <- Ring] [Ring <- Bool])
       (Int, Int) (Bool, Bool) )
 Array [Object <- Ring] (25) by: ( (Ring, Bool)
       (Array [Object <- Ring], Array [Object <- Ring] [Ring <- Bool]) (Int, Int)
       (Bool, Bool) )
 Array [Object <- Ring] [Ring <- Bool] (32) by: ( (Bool, Bool)
       (Array [Object <- Ring] [Ring <- Bool], Array [Object <- Ring] [Ring <- Bool])
       (Int, Int) )

Class: MatrixMatrix is a child (subclass by inheritance) of:
 Object (1) by: ( )
 Ring (3) by: ( (Ring, MatrixMatrix) )
 Matrix (16) by: ( (Int, Int)
       (Array [Object <- Ring], Array [Object <- Ring] [Ring <- Matrix])
       (Matrix, MatrixMatrix) (Bool, Bool) (Ring, Matrix) )
 MatrixMatrix (36) by: ( (Int, Int)
       (Array [Object <- Ring] [Ring <- Matrix], Array [Object <- Ring] [Ring <- Matrix])
       (MatrixMatrix, MatrixMatrix) (Bool, Bool) (Matrix, Matrix)
       (Array [Object <- Ring], Array [Object <- Ring]) (Ring, Ring) )

Class: Array [Object <- Ring] [Ring <- Matrix] is a child (subclass by inheritance)
of:
 Object (1) by: ( )
 Array (14) by: ( (Object, Matrix) (Array, Array [Object <- Ring] [Ring <- Matrix])
       (Int, Int) (Bool, Bool) )
 Array [Object <- Ring] (25) by: ( (Ring, Matrix)
       (Array [Object <- Ring], Array [Object <- Ring] [Ring <- Matrix]) (Int, Int)
       (Bool, Bool) )
 Array [Object <- Ring] [Ring <- Matrix] (39) by: ( (Matrix, Matrix)
       (Array [Object <- Ring] [Ring <- Matrix], Array [Object <- Ring] [Ring <- Matrix])
       (Int, Int) (Array [Object <- Ring], Array [Object <- Ring]) (Bool, Bool)
       (Ring, Ring) )
----------------------------------------------------
```

Appendix VI.

Type-Safety: An Attribute Grammar of PS-Based, Non-Singleton Classed, Mini-Dee

The Mini-Dee extension on which type-safety is defined is that of **Appendix I-F.** This grammar is of the PS-based extension which defines the structure of all explicitly defined classes (ie: without inheritance and substitution), and which includes non-singleton classes.

As in **Appendix II**, the attribute grammar has two components: environment generation and type checking, and a program P = $\{C_{I1}, ..., C_{In}, C_{C1}, ..., C_{Cn}\}$ is type-safe if the parse of P results in the attribute, P.type_safe being TRUE. All changes from **Appendix II** are emphasized with shading.

VI.A. Type Definitions:

```
Type_Rep = (name: Class Identifier, kind : {e, s, n})
```

```
node = (kind : {m,v},
        name : String,
        type : Type_Rep,
        context : Type_Rep,
        parms : Seq (node),
        locals : P (node))
```

VI.B. Functions:

VI.B.1. Environment Search Functions

Environment search operations are as defined in **Appendix II**, Section II.B.1.

VI.B.2. Type-Safety Check Functions

Bind and Parameter Bind Checks:

```
function bind (v, o : Type Rep) : Bool
    -- return TRUE if an object of type, o can be bound to a variable
    -- of type, v

begin
    if o.kind = e
        then TRUE
    else if v.kind = o.kind or (v.kind = s and o.kind = n)
        then FALSE
    else if v.kind = s and o.kind = s
        then v.name $\cong$ o.name
    else v.name $\lhd_e$ o.name
end
```

Signature Check:

As defined in Appendix II, Section II.B.2.

Signature + Parameter Binding Check:

As defined in Appendix II, Section II.B.2.

VI.C. The Attribute Grammar

The following is an attribute supplemented version of the Mini-Dee extension grammar of **Appendix I-F**.

(Program)

$P \quad \rightarrow C_{I1} \ldots C_{In} \;\; C_{C1} \ldots C_{Cn}$

$$\texttt{P.env} := \bigcup_{j=1}^{n} C_{Ij}\texttt{.env}$$

$\forall j \bullet 1 \leq j \leq n \bullet C_{Cj}\texttt{.env} := \texttt{P.env}$
$\forall j \bullet 1 \leq j \leq n \bullet C_{Cj}\texttt{.id} := C_{Ij}\texttt{.id}$

$$\texttt{P.type\_safe} := \bigwedge_{j=1}^{n} C_{Ij}\texttt{.unique\_names} \wedge \bigwedge_{j=1}^{n} C_{Cj}\texttt{.type\_safe}$$

Notes:

- The code for C_{C_j} corresponds to the interface for C_{I_j}
- Each block of code must be associated with its corresponding interface via the attribute, id.
- P is type safe if every feature in a given class interface is uniquely named and all class code blocks are type-safe with respect to the environment
- Note that given some class C with class interface C_I, C_I.env = *Rep* (C)

(Class Interface)

$C_I \quad \rightarrow$ CLASS IDEN AList END IDEN

$C_I\texttt{.env := AList.env}$
$\boxed{C_I\texttt{.id := (IDEN.value, s)}}$

```
C1.unique_names := AList.unique_names

AList.context := C1.id
```

Note:

- **IDEN**.value contains the identifier name (in both cases here, the name of the class)

(Class Name)

CName → IDEN

$\boxed{\texttt{CName.value := (IDEN.value, s)}}$

(Attribute List)

AList → $A_1 \dots A_k$

```
∀j • 1 ≤ j ≤ k • Aj.context := AList.context
AList.env := {Aj.node | 1 ≤ j ≤ k}
AList.unique_names :=
  |{id | ∃ a, t, c, p, l, j • Aj.node = (a, id, t, c, p, l) }| = k
```

Note:

- Check on assignment of TRUE to AList.unique_names makes sure that no two attributes are declared with the same name

(Attribute)

A → VAR IDEN ':' Type

```
Type.context := A.context
A.node := (v, IDEN.value, Type.value, A.context, «», Ø)
```

| METHOD $IDEN_0$ *[* '(' $IDEN_1$ ':' $Type_1$ ... $IDEN_k$ ':' $Type_k$ ')' *]* R
[VAR
$IDEN_{k+1}$ ':' $Type_{k+1}$
...
$IDEN_n$ ':' $Type_n$ *]*

```
∀j • 1 ≤ j ≤ n • Type_j.context := A.context
R.context := A.context
pnodes :=
 {(j, (v, IDEN_j.value, Type_j.value, A.context, «», ∅)) |
   1 ≤ j ≤ k}
lnodes :=
  {(v, IDEN_j.value, Type_j.value, A.context, «», ∅) |
   k+1 ≤ j ≤ n}

A.node := (m, IDEN_0.value, R.type, A.context, pnodes, lnodes)
```

(Type)

Type → ∅

```
Type.value := (Type.context, e)
```

| CName

```
Type.value := CName.value
```

| ↑CName

```
Type.value := (CName.value.name, n)
```

(Result)

R → ':' Type

```
Type.context := R.context
R.type := Type.value
```

| ε

```
R.type := R.context
```

Note:

– Methods without an explicitly declared return type are assumed to return `self`

(Class Code)

$C_C \rightarrow M_1 \dots M_n$

$\forall j \bullet 1 \le j \le n \bullet M_j\texttt{.env := } C_C\texttt{.env}$
$\forall j \bullet 1 \le j \le n \bullet M_j\texttt{.context := } C_C\texttt{.id}$

$$C_C\texttt{.type\_safe :=} \bigwedge_{j=1}^{n} M_j\texttt{.type\_safe}$$

M → **METHOD IDEN** B

```
B.env := M.env
B.lenv := locals_of (IDEN.value, M.context, M.env)
B.penv := parms_of (IDEN.value, M.context, M.env)
B.expected_type := mtype_of (IDEN.value, M.context, M.env)
B.context := M.context
```

```
M.type_safe :=
  ∃ r, p • signature ((m, IDEN.value, r, M.context, p, ∅), M.env)
      ∧ ¬Common (M.lenv, M.penv)
      ∧ B.type_safe
```

Note:

– B's environment consists of three parts – the global environment inherited from M, and the environments of parameters and local variables found in the entry in M.env corresponding to the method named IDEN.value. Parameters and local variables cannot share the same name, but can share the same name as variables or methods found in the global environment.

(Method Body)

B → BEGIN S_1 ';' ... ';' S_n END

$\forall\ j \bullet 1 \le j \le n \bullet S_j$.env := B.env
$\forall\ j \bullet 1 \le j \le n \bullet S_j$.lenv := B.lenv
$\forall\ j \bullet 1 \le j \le n \bullet S_j$.penv := B.penv
$\forall\ j \bullet 1 \le j \le n \bullet S_j$.context := B.context
$\forall\ j \bullet 1 \le j \le n \bullet S_j$.return_type := B.expected_type

$$\text{B.type\_safe} := \bigwedge_{j=1}^{n} S_j\text{.type\_safe}$$

Note:

- The body of a method is type-safe if each statement in the body is type-safe and if the expected return type is what is returned by the statements

(Statement)

S → IDEN ':=' E

```
E.env := S.env
E.lenv := S.lenv
E.penv := S.penv

E.context := S.context
S.type_safe :=
   bind (vtype_of (IDEN.value, S.context, S.env ⊎ S.lenv),
        E.type)
```

Note:

- Any variable receiving an assignment must be a parameter, local variable, or instance variable declared in the enclosing class, and must be declared of the same type as the entity it is assigned.

| RETURN E

```
E.env := S.env
E.lenv := S.lenv
E.penv := S.penv

E.context := S.context
S.type_safe := bind (S.return_type, E.type)
```

| E '.' IDEN

```
E.env := S.env
E.lenv := S.lenv
E.penv := S.penv

E.context := S.context

S.type_safe :=
   ∃ x • signature ((x, IDEN.value, E.type, E.type, «» ∅),
                          S.env ⊎  (S.lenv ∪ S.penv)
```

Note:

– Any reference to an instance variable or method call with no parameters (x can refer to m or v) must have a corresponding declaration node with context, E.type.

| E_0 '.' IDEN '(' E_1 ... E_n ')'

```
∀ j •  0 ≤ j ≤ n • E_j.env := S.env
∀ j •  0 ≤ j ≤ n • E_j.lenv := S.lenv
∀ j •  0 ≤ j ≤ n • E_j.penv := S.penv

∀ j •  0 ≤ j ≤ n • E_j.context := S.context

p := { (j, (v, "", E_j.type, S.context, «», ∅)) | 1 ≤ j ≤ n}

S.type_safe :=
  sig_and_parmbind ((m, IDEN.value, E_0.type, E_C.type, p, ∅),
S.env)
```

Note:

– Any method call with n parameters must have a corresponding declaration node with context, E_0.type (signature check), and with parameter types the same as the types of the corresponding arguments (parameter binding check).

| IF E THEN S_1 ELSE S_2

```
S.type_safe := E.type ≠ undefined
               ∧ S_1.type_safe
               ∧ S_2.type_safe
```

(Expression)

E → **SELF**

```
E.type := E.context
```

| **NEW IDEN**

```
E.type := vtype_of (IDEN.value, E.context, E.env ⊎ E.lenv)
```

Note:

- One can assign new objects to local and instance variables, but not to parameters

| **IDEN**

```
E.type :=
    vtype_of (IDEN.value, E.context, E.env ⊎ (E.lenv ∪ E.penv))
```

Note:

- Variables referred to as expressions can be instance variables found in E.context, or local variables or parameters declared in the current method

| E_1 '.' **IDEN**

```
E1.env := S.env
E1.lenv := S.lenv
E1.penv := S.penv

E1.context := S.context
S.type := type_of (IDEN.value, E1.type, E.env  ⊎  E.lenv)
```

Note:

- Any reference to an instance variable or method call with no parameters (x can refer to m or v) must have a corresponding declaration node with context, E.type.

| E_0 '.' IDEN '(' E_1 ... E_n ')'

```
∀ j •  0 ≤ j ≤ n • Ej.env := S.env
∀ j •  0 ≤ j ≤ n • Ej.lenv := S.lenv
∀ j •  0 ≤ j ≤ n • Ej.penv := S.penv

∀ j •  0 ≤ j ≤ n • Ej.context := S.context

p := { (j, (v, "", Ej.type, S.context, «», ∅)) | 1 ≤ j ≤ n}

E.type :=
  if ∃ t •
     sig_and_parmbind ((m, IDEN.value, t, E0.type, p, ∅), E.env)
         then t
         else undefined
```

Note:

– Any method call with n parameters must have a corresponding declaration node with context, E_0.type (signature check), and with parameter types the same as the types of the corresponding arguments (parameter binding check).

| Con

```
E.type := Con.type
```

(Constant)

Con → **INTNUMBER**

```
Con.type := (Int, s)
```

| **FLOATNUMBER**

```
Con.type := (Float, s)
```

| **CHARACTER**

```
Con.type := (Char, s)
```

| **BOOLCONSTANT**

```
Con.type := (Bool, s)
```