A Type-Passing Implementation of Value Types and Genericity

Tim Owen

Submitted for the degree of D. Phil.

University of Sussex

April, 2002

Declaration

I hereby declare that this thesis has not been submitted, either in the same or different form, to this or any other university for a degree.

Signature:

A Type-Passing Implementation of Value Types and Genericity

Tim Owen

Summary

Most programming languages make a distinction between value types and reference types: in Java, primitive types such as integers and floating point numbers are value types whereas classes are reference types. Some languages also provide support for genericity, which enables programmers to write reusable code that is independent of any specific data type — for example container data structures and algorithms. For maximum flexibility, generic code should be able to operate on all data, whether value type or reference type. However, the implementation of programming languages that support both genericity and value types can pose a challenge to efficiency. One implementation approach is to use type information passed at run-time. The work presented in this dissertation supports the thesis that an object-oriented language with value types and genericity can be implemented efficiently using type-passing.

After surveying various approaches to the implementation of languages with value types and genericity, a representative object-oriented language called Pluto is defined. To demonstrate the efficient implementation of Pluto using a type-passing approach, a complete compiler for the language was written. The translation scheme used by this compiler is presented and discussed in detail, showing how type information is passed at run-time to support genericity. The results of empirical measurements for a suite of test programs are given, showing the efficient handling of value types in both generic and non-generic contexts. One of the novel optimisations made by the Pluto compiler is a lazy strategy for boxing objects: pointers to stack-allocated objects are used where possible. The results demonstrate the success of this approach, with a 30% speedup achieved for the programs tested.

The conclusion of this work is that integrated support for both value types and genericity in an object-oriented language can be provided efficiently, using type-passing as an implementation technique.

Submitted for the degree of D. Phil.

University of Sussex

April, 2002

Contents

1 Introduction **1**

1.1 Overview . . . 1

1.2 Motivation . . . 2

1.3 Approach . . . 3

1.4 Terminology . . . 4

1.5 Outline of the dissertation . . . 4

2 Background **6**

2.1 Object-oriented languages . . . 6

2.1.1 Class-based versus object-based languages . . . 6

2.1.2 Static versus dynamic type systems . . . 7

2.2 Implementations of object-oriented languages . . . 7

2.2.1 Memory allocation . . . 7

2.2.2 Object models . . . 8

2.2.3 Vtables . . . 10

2.2.4 Inheritance . . . 10

2.2.5 Multiple inheritance . . . 11

2.2.6 Interfaces . . . 11

2.2.7 Run-time type information . . . 12

2.3 Value-based types . . . 12

2.3.1 Traditional procedural languages . . . 13

2.3.2 Built-in value types . . . 13

2.3.3 Distinct value classes . . . 14

2.3.4 Manual annotation of object storage . . . 14

2.3.5 Automatic value type detection . . . 15

2.4 Genericity . . . 16

2.4.1 Parametric polymorphism . . . 16

2.4.2 Constraint specification . . . 17

2.4.3 Virtual types . . . 18

2.5 Implementations of genericity . . . 19

2.5.1 Heterogeneous expansion . . . 19

2.5.2 Homogeneous translation . . . 20

2.5.3 Run-time type support . . . 21

2.6 Implementations of mixed genericity and value-based types . . . 22

2.6.1 Expansion approach . . . 22

2.6.2 Fully-boxed and tagged values . . . 22

2.6.3 Coercion-based boxing . . . 23
2.6.4 Type-passing . . . 24
2.6.5 Dictionaries and dope vectors . . . 25
2.6.6 Variable size data . . . 26
2.7 Variance support . . . 26
2.7.1 Method signatures . . . 26
2.7.2 Type parameter annotations . . . 28
2.8 Summary . . . 29

3 The basic Pluto language and compiler 31
3.1 Design introduction . . . 31
3.2 Language description . . . 31
3.3 Compiler structure . . . 33
3.3.1 Implementation language . . . 33
3.3.2 Front end . . . 35
3.3.3 Middle phases . . . 37
3.3.4 Back end . . . 37
3.3.5 External tools . . . 38
3.4 Choice of target language . . . 38
3.4.1 Using C . . . 40
3.4.2 GNU C extensions . . . 40
3.5 Object model . . . 41
3.5.1 Storage determination . . . 41
3.5.2 Instance layout . . . 42
3.5.3 Vtable/type descriptor layout . . . 42
3.5.4 Vtable/type descriptor instances . . . 43
3.5.5 Method kinds . . . 44
3.5.6 Method dispatch . . . 45
3.5.7 The type of `this` . . . 45
3.5.8 Class constructors . . . 46
3.6 Interfaces . . . 46
3.6.1 Interface vtables . . . 47
3.6.2 Interface instance layout . . . 47
3.6.3 Dynamic vtable lookup . . . 48
3.7 Code generation . . . 49
3.7.1 Representation . . . 49
3.7.2 Naming system . . . 52
3.7.3 Example generated code . . . 53
3.7.4 Type coercions . . . 56
3.7.5 Function inlining . . . 58
3.7.6 Initialisation of static fields . . . 58
3.7.7 Separate compilation . . . 59
3.7.8 Program entry point . . . 60

3.7.9 Run-time error handling . . . 60
3.8 Pluto library . . . 61
3.8.1 Predefined classes . . . 61
3.9 Summary . . . 63

4 First addition: value types **64**
4.1 Distinguishing value-based types . . . 64
4.1.1 The benefits of value types . . . 66
4.1.2 Immutable state . . . 67
4.1.3 Examples . . . 68
4.2 Organising the class hierarchy . . . 68
4.3 The `final` modifier . . . 69
4.4 Types of literal values . . . 71
4.5 Value type restrictions . . . 71
4.5.1 Subclass size . . . 71
4.5.2 Recursive containment . . . 72
4.6 Implementation details . . . 73
4.6.1 Instance handling . . . 73
4.6.2 Headless layout and full value information . . . 73
4.6.3 Coercions to supertypes . . . 74
4.6.4 Passing the receiver . . . 75
4.6.5 Constructor optimisation . . . 75
4.6.6 Example generated code . . . 76
4.7 Summary . . . 80

5 Second addition: genericity **81**
5.1 Benefits of generic programming . . . 81
5.2 Type parameters . . . 81
5.2.1 Bounded type parameters . . . 83
5.2.2 Method type parameters . . . 84
5.2.3 Limitations of use . . . 84
5.3 Implementation overview . . . 86
5.4 Coercing to and from generic values . . . 87
5.5 Type descriptors . . . 88
5.5.1 Meta-information . . . 88
5.5.2 Type descriptor instances . . . 89
5.6 Type arguments . . . 89
5.6.1 The type of type descriptors . . . 89
5.6.2 Type argument passing . . . 90
5.6.3 Class type argument storage . . . 91
5.6.4 Type argument aliasing . . . 92
5.6.5 Eliminating unused type arguments . . . 93
5.7 Use of run-time type information . . . 93

5.7.1 Dynamic coercions . . . 94
5.7.2 Method calls on generic receivers . . . 95
5.7.3 Object copying utilities . . . 95
5.8 Lazy versus eager boxing . . . 96
5.8.1 Stack pointer safety . . . 97
5.8.2 Boxing operations . . . 97
5.8.3 Analysis for return value boxing . . . 98
5.9 Example generated code . . . 98
5.10 Summary . . . 102

6 Third addition: variance 103
6.1 Overview . . . 103
6.2 Type parameter variance . . . 104
6.3 Restrictions to maintain type safety . . . 105
6.3.1 Variant types in method signatures . . . 106
6.3.2 Variant types in fields . . . 106
6.3.3 Variance subtyping rules . . . 106
6.3.4 Method type parameter variance . . . 107
6.4 Method signature variance . . . 107
6.4.1 Generic types in signatures . . . 107
6.4.2 Interface method signatures . . . 109
6.5 Implications for code generation . . . 109
6.5.1 Object coercions to and from variant generic types . . . 109
6.5.2 Wrapper functions . . . 111
6.5.3 New vtable entries . . . 112
6.6 Summary . . . 113

7 Evaluation and results 114
7.1 Test objectives . . . 114
7.2 Choice of test programs . . . 115
7.3 Methodology . . . 116
7.3.1 Test environment . . . 116
7.3.2 Building Pluto programs . . . 117
7.3.3 Building non-Pluto programs . . . 117
7.3.4 Measurement techniques . . . 118
7.4 The tests . . . 119
7.4.1 Reference, value and final value objects . . . 119
7.4.2 Use of garbage collection . . . 122
7.4.3 Cross-language comparison . . . 124
7.4.4 Eager versus lazy boxing . . . 127
7.5 Summary of results . . . 129

8 Conclusions **131**
8.1 Summary of contributions . . . 131
8.2 Design variations . . . 132
8.2.1 Final value objects in a generic context . . . 132
8.2.2 Non-generic arrays . . . 133
8.3 Possible extensions . . . 133
8.3.1 Reflection . . . 134
8.3.2 Cooperative garbage collection . . . 135
8.3.3 Object serialisation and persistence . . . 135
8.4 Conclusion . . . 135

A Grammar for Pluto core language **137**

B Test programs **140**
B.1 Program gen_compare: generic comparator method . . . 140
B.1.1 Pluto source . . . 140
B.2 Program gen_tree: generic binary search tree . . . 141
B.2.1 Pluto source . . . 141
B.3 Program word_tree: fixed-type binary search tree . . . 145
B.3.1 Pluto source . . . 145
B.4 Program matrix_mult: matrix multiplication with complex numbers . . . 148
B.4.1 Pluto source . . . 148
B.4.2 C++ source . . . 150
B.4.3 Eiffel source . . . 151
B.4.4 Java source . . . 153
B.5 Program ackermann: the Ackermann function . . . 154
B.5.1 Pluto source . . . 154
B.5.2 C++ source . . . 155
B.5.3 Eiffel source . . . 155
B.5.4 Java source . . . 156
B.6 Program prime_sieve: the Sieve of Eratosthenes . . . 156
B.6.1 Pluto source . . . 156
B.6.2 C++ source . . . 157
B.6.3 Eiffel source . . . 158
B.6.4 Java source . . . 159
B.7 Program gen_stack: generic stack . . . 159
B.7.1 Pluto source . . . 160
B.7.2 C++ source . . . 162
B.7.3 Eiffel source . . . 163
B.7.4 Java source . . . 165
B.8 Program gen_comptr_tree: generic binary search tree with comparator . . . 167
B.8.1 Pluto source . . . 167
B.8.2 C++ source . . . 169

B.8.3 Java source 173
B.9 Program word_comptr_tree: fixed-type binary search tree with comparator . 176
B.9.1 Pluto source 176

Bibliography **179**

List of Figures

2.1 Basic object model 9
2.2 Object model with inheritance 10
2.3 Explicit object storage in C++ 14
2.4 Data structure with fixed element type 16
2.5 Data structure with generic element type 16
2.6 Bounded type parameters 18
2.7 Covariant return types 27
2.8 Overriding binary methods 28
2.9 Subtyping of generic types 29

3.1 Arrangement of class kinds in the hierarchy 32
3.2 Example Pluto program 34
3.3 Example build process and output 34
3.4 Pluto compiler overview 36
3.5 Interface instance layout 48
3.6 Pluto naming scheme prefixes 52

4.1 Pluto program using reference and value classes 65
4.2 Memory layout for reference and value objects 65
4.3 Pluto program using a final value class 70
4.4 Memory layout with final value sub-objects 70
4.5 Algorithm for detecting recursive value class containment 72

5.1 Pluto program using type parameters 82
5.2 Generic methods, using type parameters 85
5.3 Memory layout of generic class instance 91
5.4 Pluto program needing run-time type information 94

6.1 Pluto types using variant type parameter 104
6.2 List and Collection subtyping relationships 104
6.3 Pluto classes using method signature variance 108

A.1 Pluto core abstract syntax (type-level and member-level) 138
A.2 Pluto core abstract syntax (low-level) 139

List of Tables

7.1 Summary of test programs 115
7.2 Test environment details 116
7.3 Comparing reference, value and final value types 120
7.4 Comparing types, using garbage collection 123
7.5 Cross-language comparison 125
7.6 Comparing lazy and eager boxing 128

Chapter 1

Introduction

This chapter gives a broad overview of the research topic that is addressed by the work described in this dissertation, explains why this area of study was chosen and describes the approach used to undertake the research. The last section of this chapter provides an outline of the remainder of the dissertation.

1.1 Overview

The work presented in this dissertation supports the thesis that "An object-oriented language with value types and genericity can be implemented efficiently using type-passing". This section briefly explains the key terms of interest: value types, genericity and type-passing.

Most programming languages make a distinction between value types and reference types. Value type objects typically include integers and floating point numbers that are handled directly, copied by value and often stack-allocated. In contrast, reference type objects such as those used for linked data structures are handled indirectly through a pointer, can be aliased and are usually heap-allocated. In Java for example, the primitive types including `int`, `float`, etc. are value types, whereas classes, interfaces and arrays are all reference types[49]. In the background chapter, Section 2.3 considers value types in more detail.

Some languages provide support for genericity, which enables programmers to write reusable code that is independent of any specific data type — for example container data structures such as lists and trees, and algorithms such as sorting and searching routines. The most common example of a genericity mechanism is the C++ template feature[55]. Section 2.4 in the background chapter discusses genericity and examines its support in various programming languages.

A useful extension to genericity in a programming language is the support of variance. This is a feature of the type system that enables more flexibility and expressiveness in the subtyping relationships between different types. Section 2.7 in the background chapter explains the benefits of providing variance together with genericity.

For maximum flexibility, generic code should be able to operate on all data, whether value type or reference type. However, the implementation of programming languages that support both genericity and value types can pose a challenge to efficiency — a topic explored more fully in Section 2.6. One implementation approach is to use type information passed at run-time, to assist in the handling of generic data. This technique, known as type-passing, enables generic code to deal with data of different types (and hence potentially different sizes) by relying on the type information to describe the data.

1.2 Motivation

The original motivation for this choice of research topic came from involvement in the SafetyNet[90] research project. In this work, an object-oriented programming language for Active Networks was defined and implemented, with the aim of using language design techniques to ensure safe and secure application programming in a distributed environment.

One of the most important design goals for SafetyNet is that type safety should be checked at compile-time whenever possible. This allows many program errors to be caught by the compiler before the program is deployed and run, and has the additional benefit of minimising potentially costly safety checks at run-time. In an Active Network environment, programs may be deployed at remote network locations which may be suspicious of executing foreign code. The use of a language that affords static type-safety checks increases the ability of remote sites to perform more extensive verification of incoming code before it is executed.

To enable a high level of static type safety, the SafetyNet language has an expressive type system which includes genericity. This means, for example, that use of general purpose container data structures can be checked at compile-time, in contrast to non-generic languages such as Java where use of such containers involves type casts (known as downcasts) which must be checked at run-time to ensure type safety. Section 2.4 in the background chapter explains how genericity compares with the alternative approach typically used in languages that do not have generic types. The use of genericity therefore contributes towards increasing static type safety and reducing run-time checks.

Another important aspect of the SafetyNet language design is that types are classified as either value-based or reference-based. The main reason for this distinction is to help make programmers aware of the different semantics of objects that are copied by value from those that are heap-allocated and handled by reference. In a distributed programming environment, programs executing in different network locations may need to exchange data, and will not usually be able to access the heap at a remote site. An advantage of value-based objects is that they are typically small and immutable, so can be copied from one location to another without concern about aliasing issues or problems with consistency of state. In contrast, transferring a serialised copy of a mutable reference-based data structure (a large hash table, for example) from one location's heap to another is in most cases inefficient and could cause inconsistencies between the copies. The SafetyNet language semantics ensure that only value-based objects can be copied

between locations. This application shows that the distinction between value-based and reference-based objects is an important aspect of programming language design.

As stated above, the SafetyNet language supports genericity and distinguishes between value and reference types. Consequently, implementation of the language required some way of handling the combination of these two features efficiently — a problem that led to the research presented in this dissertation. Although the SafetyNet project provided the original motivation for this work, the benefits of genericity, variance and value types are applicable in a much wider context.

The importance of variance was highlighted by practical experience with use of generic types in SafetyNet, as some potentially useful subtype relationships could not be expressed within the type system. Variance was suggested as a future addition to the language that could solve this problem. Therefore, this research work provided an opportunity to examine the issues involved in the design and implementation of variance in a language with genericity and value types.

1.3 Approach

A survey of the literature related to implementation of genericity and value types shows that the type-passing approach has been used for functional programming languages, but not object-oriented languages. Those languages such as C++, Eiffel and Sather that support both value types and genericity are implemented by code expansion, whereby type-specific code is generated for different uses of a generic type. Most proposals for extending Java with genericity do not involve code expansion but commonly make the restriction that value types cannot be used with generic types. This simplifies the implementation at the expense of uniformity: value types are effectively second class types, and genericity cannot be applied for all types.

Given this situation, the direction of the research is clear: to demonstrate the application of type-passing in the context of an object-oriented language, by implementing uniform support for value types and genericity. To achieve this aim three main tasks contributed to the work presented in this dissertation, reflecting the activities undertaken during the course of the research:

1. Define an object-oriented language supporting value types and genericity.
2. Implement this language, by writing a compiler for it that uses type-passing.
3. Test and evaluate the compiler to determine how efficient the implementation is.

Using this approach makes it possible to examine experimentally how much language support for value types and genericity (including variance) can be provided without causing implementation difficulties. Writing a complete compiler enables the effectiveness of type-passing to be ascertained, and gives the opportunity to explore optimisations to the type-passing scheme.

1.4 Terminology

In order to prevent confusion about different terminology, this section defines a number of technical terms used in the dissertation. Different researchers use a variety of terms to describe similar concepts. The field of programming languages is no exception, so the following definitions are those that will be used consistently throughout this work.

Genericity The ability of a construct to be abstracted with respect to a type. This is independent of the way it is implemented: terms such as *template*, *parameterised type* and *parametric polymorphism* are specific mechanisms for providing genericity. Section 2.4 explains further.

Value type A type whose instances are manipulated directly, rather than through an indirection such as a handle, a pointer or a reference. Value types are variously described as *immediate*, *lightweight*, *stack-based*, *expanded*, *inline* or *unboxed*.

Dynamic dispatch The run-time selection of a function to be executed, in contrast to a fixed function name set at compile-time. This mechanism is also known as *dynamic binding*, *late binding*, *message dispatch* or a *virtual function call*.

Use (of a generic type) An application of a generic type, by supplying type arguments for the type parameters. Some sources use the term *instantiation* to describe this operation. However, that term is not used in this work because it might imply that a generic type is duplicated, specialised or somehow expanded each time it is used, which may not be the case.

Receiver The object upon which an instance method call is made. Within the method body, it is known as *this*, *self* or *Current*. At the method call site, this object is sometimes known as the *target* of the call.

Vtable A list of function pointers and other type-specific information, used to implement dynamic dispatch and support run-time operations. Other object-oriented implementations describe these as *dispatch tables*, *dispatch vectors*, *method tables*, *method suites* or *virtual function tables*. Since the vtable is often used to store information about a type for a variety of run-time mechanisms, it can also be described more generally as a *type descriptor*.

Function and method A *method* is an object-oriented construct: a member of a class that is invoked with respect to an object of that class. The term *function* is used to refer to the lower-level construct found in procedural and assembly languages.

1.5 Outline of the dissertation

Chapter 2 covers relevant background material and surveys existing approaches to the implementation of value types and/or genericity, ending with a summary of the research topic addressed by the present work. In Chapter 3, the language defined for use in this research (Pluto) is presented. That chapter explains the basic implementation approach used, and shows the translation process down to the production of C code. Specific details on the design and implementation of value types in Pluto are discussed in Chapter 4, which builds on the foundation given in Chapter 3. Similarly, Chapter 5 and Chapter 6 explain how the Pluto language supports genericity and variance respectively, and details how these features are implemented by the compiler. These three feature-specific chapters

focus on the language support and describe how type-passing implementation techniques are used to enable efficient generated code.

Evaluation of the Pluto compiler implementation is presented in Chapter 7, showing the results of testing Pluto program performance in terms of program size, memory consumption and running time. The testing process examines the effects of using value types and compares Pluto programs against similar code written in other languages with comparable features. The results also show the effect of optimisations such as the lazy boxing strategy used in the type-passing implementation. The dissertation concludes with Chapter 8, which summarises the contributions of this research, examines some potential improvements to the work and considers various language feature extensions that could be enabled by building on the type-passing infrastructure.

Chapter 2

Background

This chapter covers essential background material, and provides a survey of existing work in the research area. The topics covered in this chapter include the implementation of object-oriented languages and the design and implementation of the features of interest: value types, genericity and variance.

2.1 Object-oriented languages

While there are many different object-oriented language designs, some broad classifications can be used to help place the current research within the field of study.

2.1.1 Class-based versus object-based languages

Using the terminology from [1] a distinction can be made between those object-oriented languages that feature classes, and those that do not. Most mainstream object-oriented languages such as Java[49], C++[105], Eiffel[72] and Smalltalk[45] are class-based. A class definition specifies a new data type, which can be instantiated to create objects.

In object-based languages, for example Self[32] and Cecil[30], there is no class construct: objects are specified and created individually, or cloned from a prototype. Furthermore, object-based languages typically do not support the notion of objects having a particular type, often because there is no language-level construct to define types. However, many object-based languages have some mechanism for creating families of objects with a common trait. Object-based languages are often used for rapid development of programs where the discipline of defining classes is considered unnecessary.

The focus of the work presented in this dissertation is the support of language features such as generic types, variance and the distinction between value-based and reference-based types. These constructs would have little meaning in a language that does not even support the definition of types. Therefore, the Pluto language is class-based not object-based: it allows the definition of types using the class and interface constructs.

2.1.2 Static versus dynamic type systems

Another common distinction among object-oriented languages is whether they have a static type system. Those that do not are called dynamically-typed. The primary purpose of a static type system in an object-oriented language is to enable a compiler to guarantee that a method call is only invoked on objects that implement that method. Dynamically-typed languages delay these checks until run-time, and hence have to accept the possibility of failure at this late stage. In general, class-based languages are statically-typed, and object-based languages are dynamically-typed. An important exception is Smalltalk, which is class-based but dynamically-typed.

These two approaches have different design goals, and their own implementation consequences. A major benefit of statically-typed languages is that potentially more programming errors can be detected by the compiler, rather than left until run-time. The features of interest to the present work — value types, genericity and variance — can be incorporated into a statically-checkable type system, further increasing the ability of a compiler to detect type errors. In addition, one aim of this work is to demonstrate the use of type-passing (described in Section 2.6.4) as an implementation technique. This approach exploits the presence of a static type system. For these reasons, the Pluto language is statically-typed.

Because Pluto is a statically-typed class-based language, the following sections concentrate on the issues relating to that category of language.

2.2 Implementations of object-oriented languages

Since the purpose of the research is to show the feasibility of compiling advanced features of object-oriented languages, it is important to examine some aspects of how these languages are typically implemented. The work described in subsequent chapters is based on extensions to the basic techniques described here.

2.2.1 Memory allocation

When describing the data storage mechanisms used by programming language implementations, some common distinctions are made between different kinds of memory use. This division is important to bear in mind when discussing the storage of objects at run-time, as it helps to explain the behaviour of value-based and reference-based types (described in Section 2.3).

For most language designs, a distinction can be drawn between statically-determined memory, and dynamically allocated memory. The latter can be further divided into allocations whose lifetime can be predicted by the compiler, and those that cannot. For the implementation of an object model (discussed in Section 2.2.2), three kinds of storage are important:

Static Data whose size is known at compile-time, and whose storage duration is the lifetime of the program, can be allocated statically; in C++ terminology, this is external data. In practice, this allows storage for the data can be specified in the compiler's generated code — e.g. a `.o` object file or a Java `.class` file. When the

program is loaded into memory to be run, space can be reserved for the static data. In addition, some static data is constant, i.e. read-only — an example is the vtable discussed in Section 2.2.3. In this case, the compiler's generated code can contain the values to be stored as well as specifying the storage requirements.

Heap Most modern languages allow data to be allocated explicitly on demand. Since the compiler is unable to determine the size and lifetime of these allocations, they are taken from a *heap* (also known as a free store). This is a pool of memory reserved at run-time, typically supported by library functions that manage requests for allocation. Some language implementations automatically determine the lifetime of heap-allocated memory and return expired portions to the available pool, while others require the programmer to indicate when the memory can be reclaimed.

Stack For most object-oriented languages, the storage lifetime of method parameters and local variables follows a first-in-first-out nested pattern. It is natural therefore to store this data in a stack structure at run-time. Each method invocation results in a new frame of storage in the stack, which can be automatically removed when the method ends.

It should be noted that these are conceptual ways of partitioning memory — language specifications rarely insist that actual implementations must organise storage in this way. However, most compilers for standard architectures (and virtual machines) are likely to use the distinction shown above.

Variations on these memory layouts have been investigated, in attempts to automate the memory management task. For example, the concept of *regions*[115] has been proposed and used in the ML Kit compiler[113]. In this scheme, memory is managed as a stack of regions and each region is itself a stack of data. Compiler analysis is performed to infer which region should be used for any given item of data[114]. Regions are allocated and deallocated in whole units, in a stack-like manner. The purpose of this design is to avoid the need for a run-time garbage collector by determining the safe deallocation points at compile-time instead. The effectiveness of this approach depends on the accuracy of the region inference process.

A manual version of the region scheme is the *obstack* facility in the GNU C library[71], which provides auxiliary stacks for general programmer use. Obstacks are a hybrid between heaps and stacks, in that they are less general than a heap, but have the advantage of grouping together lots of small allocations and deallocations into larger units. This simplifies the task of managing memory deallocation, and reduces the number of individual heap allocations that must be managed by a run-time garbage collector.

2.2.2 Object models

An object model describes the way that classes and objects are conceptually represented in memory at run-time. The model is useful to programmers who wish to understand and optimise the behaviour of their programs, and to compiler writers who need to ensure the correct semantics in their generated code.

For languages like C++ and Eiffel, the object model is most commonly realised by tools such as compilers, assemblers, linkers and loaders, that eventually produce an executable

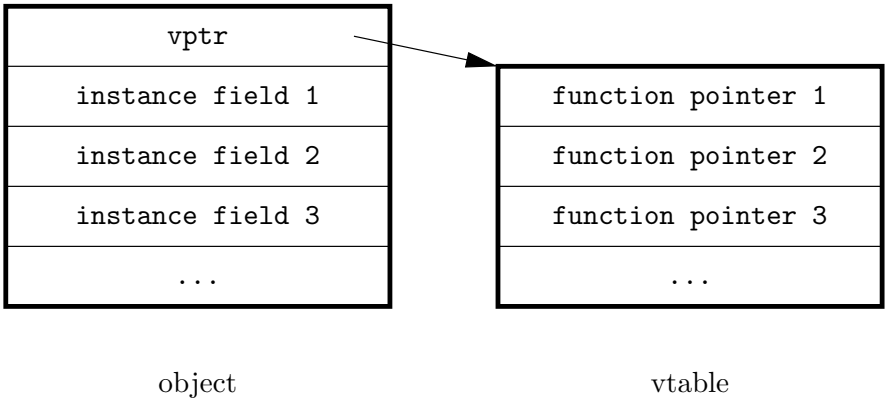

Figure 2.1: Basic object model

program for the target platform. An alternative is used for many Smalltalk and Java implementations, in the form of a virtual machine (VM) environment. Here, the storage of objects and the implementation of method calls is handled at run-time by the virtual machine rather than a compiler. Some VM-based designs prove to be a useful way of implementing sophisticated language features. For example, Sections 2.5.3 and 2.6.4 review some existing schemes for implementing genericity and value types that make good use of a VM environment.

While an object model generally suggests an implementation strategy, compiler writers are free to design the internal representation in any manner they choose, provided the language semantics are not violated. For example, the Java Virtual Machine specification[67] is careful not to impose any requirements on the representation of objects, leaving implementors free to explore different optimisations and design trade-offs[7, 9]. Likewise, the C++ language specification[55] does not mandate any implementation technique for dynamic dispatch. Furthermore, as many compilers and virtual machine implementations are commercial products, their detailed design is usually not available in published form.

Figure 2.1 shows the classical model for statically-typed object-oriented languages: it represents objects using an aggregate of the instance fields together with a pointer to information about the object's type. The type information is itself an aggregate, and is shared by all instances of its type. Each object has a vptr field that points to the type information data. An important property of this object model is that the position of any field in the object is known to the compiler: this is a benefit of statically-typed languages.

This is the model used to describe C++ implementations[50, 68, 104] and in compiler textbooks such as Appel[10] and Grune et al[51]. The description of Eiffel[72] uses this model to explain the language, but does not guarantee that actual implementations employ it. An alternative technique is used in the SmallEiffel compiler[123], where a binary search is performed at run-time to implement dynamic dispatch.

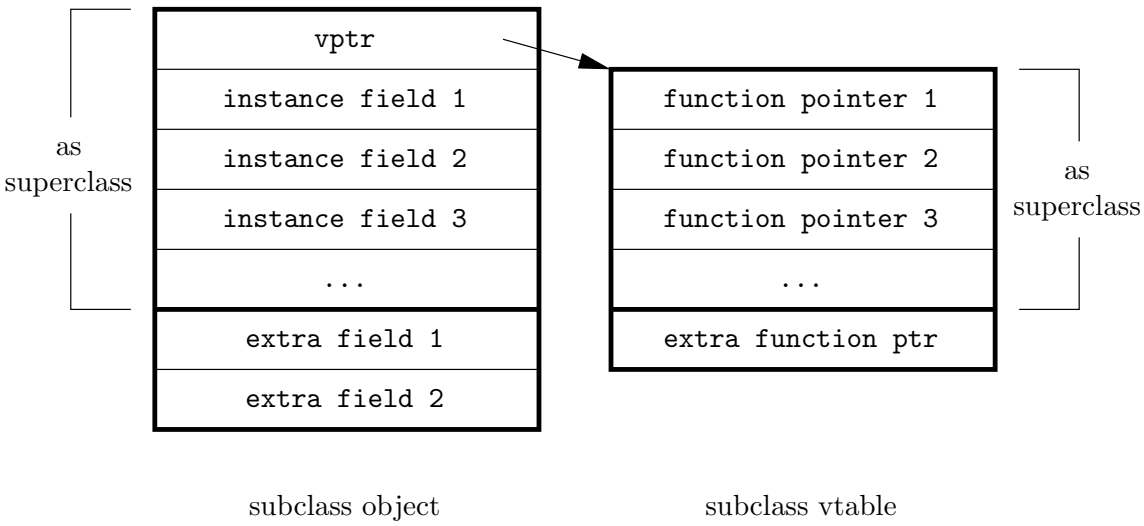

Figure 2.2: Object model with inheritance

2.2.3 Vtables

The type information shown in the object model is known as a *vtable*, since its primary purpose is to support dynamic dispatch of methods using a table of function pointers. Other type-specific information may be stored in the vtable, to enable language features or implementation techniques that need this information at run-time — see Section 2.2.7 for examples.

Like the layout of fields in an object, the layout of the vtable is also known at compile-time, allowing for an efficient implementation of dynamic dispatch:

1. Given an object, select the vptr field at its fixed offset within the object.
2. Select the required function pointer, at a fixed offset from the target of the vptr.
3. Perform an indirect function call using the function pointer.

2.2.4 Inheritance

The object model described above can be easily extended to support single inheritance for classes. A subclass may add new instance fields, and it may add or override instance methods. To implement subclasses, the layout of the superclass object and vtable is maintained, with the additional fields and methods appended as shown in Figure 2.2. This compatibility ensures that code expecting a superclass object can safely operate on subclass objects without being affected by the additional members.

When a subclass provides an overriding implementation of an instance method, no new entry is required in the vtable. All that is necessary is to store the correct function pointer in the entry corresponding to that method.

2.2.5 Multiple inheritance

Generalising the object model to support multiple inheritance is more complicated than for single inheritance. The positions of fields and vtable entries cannot always be known at compile-time: some must be calculated using offsets determined at run-time. Like other complex language features, a key design aim in languages such as C++ is to ensure that the cost of multiple inheritance is only paid by programs that use it.

The C++ object model supports multiple inheritance of fields and methods, and the traditional implementation is described in great detail by [50, 68, 104]. An interesting alternative is the bidirectional model shown in [80], which the author compares with the traditional C++ design. This model optimises the common cases where only method signatures are multiply inherited.

Eiffel also supports general multiple inheritance, although most implementations assume a closed-world system where the entire class hierarchy is known to the compiler. This assumption allows offsets to be calculated by the compiler, on a global basis. Details of Eiffel implementations are hard to come by, as most are commercial. A notable exception is SmallEiffel, the GNU Eiffel compiler project.

Implementing multiple method inheritance also requires more complex strategies than the single inheritance case. For example, if a subclass inherits methods from two superclasses, the layout of its vtable cannot match the layout of both its superclass vtables — the offsets may clash. Solutions involve run-time lookups, or storing more than one vptr in objects together with offset adjustments to object pointers.

2.2.6 Interfaces

A less general approach to multiple inheritance is taken by Java and Theta[69], which only allow multiple inheritance of method signatures. Instance fields and method implementations can only be singly inherited. This design reduces the complexity of the language, and simplifies implementation. It also eliminates the need for disambiguation of method calls that would occur if more than one implementation was inherited.

To ensure only method signatures are multiply inherited, Java provides *interfaces*. These are type specifications similar to classes except that they contain only declarations of method signatures: no method implementations or instance fields. Classes can inherit from only one superclass, but may also implement multiple interfaces.

Since an interface type can be implemented by many different classes, each with potentially different vtable layouts, calling a method through an interface is not as simple as the process shown in Section 2.2.3. As the vtable entry cannot be determined at compile-time, some form of run-time lookup is needed.

One approach is to use a run-time dictionary such as a hash table, and lookup the method using its name as the key. In this design, each class's vtable would provide a lookup function to determine the correct entry. The Marmot Java compiler[42] solves this problem using a lookup based on the interface name, which provides a handle to a special vtable called an itable. A dynamic dispatch to the required method can then be made from this table.

A related approach[64] is to perform a single lookup based on the interface name whenever an object of class type is assigned to a variable of an interface type. This run-time lookup provides a pointer to a vtable that is laid out in a known order for that interface, containing pointers to that class's implementation of the methods. The description of the CACAO compiler for Java[62] includes two alternative vtable layouts, one with interface method pointers directly in the vtable, and another with pointers to separate interface vtables. Offsets for interface vtables or method entries are allocated by the compiler on a global basis, with compaction techniques to avoid sparse distribution.

2.2.7 Run-time type information

Besides supporting dynamic dispatch, the vtable can also be used to store other type-specific information useful at run-time. Of particular relevance to the work in this dissertation is the use of information about object sizes stored in the vtable. Details of this can be found in Sections 3.5.3 and 5.7.

The following facilities can also be supported using run-time type information:

Type tests Determining if an object is an instance of a particular type. For example, Java's `instanceof` operator or Eiffel's `conforms_to` feature.

Downcasting An attempt to coerce a variable of some class to a variable of one of its subclasses — an operation that cannot be proven safe at compile-time, and hence could fail at run-time. Examples are the Java type cast operator, the Eiffel `?=` operator and the C++ `dynamic_cast` operator.

Reflection Obtaining type information about an object, in the form of a *meta object* that can be manipulated in the source language. This facility is provided by the Java Reflection API `java.lang.Object.getClass()` method and the C++ `typeid` operator.

Garbage collection Non-conservative garbage collectors[122] need information about object layouts in order to distinguish pointer fields from non-pointers.

Serialisation Object layout information is needed when objects are converted into a flat data stream, for transmission over networks or for persistent storage on disk or in a database.

The purpose and scope of the work in this dissertation is to use run-time type information to support the implementation of genericity and value types. Therefore, the features listed above have not been implemented in Pluto as it is described in subsequent chapters. Nevertheless, Section 8.3 discusses how they could be added using the existing object model infrastructure.

2.3 Value-based types

Among the data types available in a language, some are naturally value-based in that they are manipulated directly and have pass-by-value semantics. In contrast, reference-based types are manipulated indirectly through some kind of handle. Indirection allows for aliasing: one object can be referenced via more than one handle. For value-based objects there are no handles, so they are copied when passed around.

The following sections describe the most common techniques that programming language designs use to handle value-based types.

2.3.1 Traditional procedural languages

The C type system[60] is typical of a procedural language approach to value-based and reference-based types. The basic numeric types such as `int` and `float` are value-based, as is the `struct` mechanism for defining aggregates (sometimes called records) of data. Reference-based types can be built by applying the C pointer mechanism. Given any type `T`, a derived type pointer-to-`T` can be declared, which adds one level of indirection to values of type `T`. Since pointers are themselves values and can be copied, aliasing of objects is possible.

2.3.2 Built-in value types

Returning to object-oriented languages, there are two main approaches to handling simple numeric types within an object-oriented type system. The uniform way, pioneered by Smalltalk and continued in languages such as Eiffel, Sather[109] and Transframe[93], is to define classes for integers, characters etc. in the standard library. Eiffel, Sather and Smalltalk present numeric value types as classes, but allow implementations to treat them as special cases to retain efficiency.

The second approach, used by C++ and Java, is to define a fixed set of basic types and their operations as part of the language itself. These basic types are not part of the class hierarchy at all. Furthermore, Java treats these types as the only value-based types — all types defined by classes and interfaces are reference-based.

For an object-oriented language design, the choice of approach is typically made for reasons of efficiency. A simplistic handling of integers as objects could result in arithmetic operations being intolerably slow, especially when compared to C or Fortran. A related concern is that the syntax used for object operations might be too cumbersome compared with the conventional mathematical notation for addition, comparison etc. The operator overloading facilities in C++ help to alleviate this issue, however.

A number of research projects have attempted to reduce the performance gap between pure object-oriented languages and procedural languages. In particular, Self[29, 31, 119] takes a pure and uniform approach to the language design, relying on extensive analysis and aggressive optimisation in the compiler to reduce the so-called abstraction penalty. Results in [119] show that the Self compilation techniques give a four-fold speedup compared with a state-of-the-art Smalltalk implementation (also a pure object-oriented language).

The scientific programming community, which makes heavy use of specialist numeric data types such as complex numbers and intervals, has shown interest in improving support for these types in object-oriented languages. For example, the Java Grande Forum[57] has proposed extensions to Java to support value-based types[58]. They recognise that simply adding extra basic types to the language is an ad hoc solution: providing a general mechanism to add new value-based types is more useful in the long term. The general approach was propounded by Guy Steele in a keynote speech[102] about Java extensions.

```
class Point {
public:
  Point(int x, int y);
};

Point p(2,4);

Point* q = new Point(6,7);
```

Figure 2.3: Explicit object storage in C++

James Gosling also speculates on extending the language to support numerics in [48], proposing lightweight classes.

2.3.3 Distinct value classes

Although in languages such as Smalltalk and Java the only value types are those supported by the specification and compilers, a more general approach is to enable programmers to define their own. To a programmer, the principle advantage of value types is the potential efficiency gain. This subject is investigated more fully in Section 4.1.1.

Some class-based languages allow the programmer to specify which classes should be considered value-based, and which are reference-based. Eiffel is perhaps the most widespread example of this, with its `expanded` modifier for class definitions. Similarly, Sather supports immutable value classes. The Transframe language[93] allows programmers to specify value-based types, and to indicate the type's size. The specification details a variety of mechanisms for controlling the handling of objects via different *referential protocols* such as value-based, reference-based and others. Unfortunately, there appears to be no available information about how this language is implemented, or whether there are any existing compilers for it.

Most language designs that distinguish value classes make certain restrictions on how these classes fit into the type system. For example, Java extension proposals for value classes specify that they cannot have subclasses, or even that they are separate from the rest of the class hierarchy entirely[58]. Likewise, the classes for Sather's basic types are segregated from the rest of the class hierarchy, and cannot be extended. Sather also keeps the value-based hierarchy and reference-based hierarchy separated by abstract classes.

2.3.4 Manual annotation of object storage

Eiffel's `expanded` modifier can be applied to variable declarations as well as to classes. For example, a variable declared with a type of the form `expanded Point` means that it directly holds an object of class `Point` — not a reference to an object. In this way, any class can be used with value-based or reference-based semantics on a per-variable basis. In fact, its use on classes is really a shorthand way of specifying the default for variables whose declared type is that class.

Similarly in C++, classes can be instantiated as either value-based objects, or as

reference-based using the pointer or reference derived types. Referring to the code in Figure 2.3, variable `p` is a value-based object, most likely stored on the stack, whereas variable `q` is reference-based: it is a pointer to a heap-allocated object.

When dealing with value-based objects, most languages make restrictions to ensure the compiler can determine the amount of storage needed for objects stored on the stack. In the C++ example code, the compiler knows the size of a `Point` object and can therefore reserve space for variable `p`.

A subclass of `Point` could be defined with extra instance fields and hence require more storage space. However, almost all object-oriented languages support *subsumption*: the ability to use a subclass object wherever a superclass object is expected. So a potential problem occurs in situations where a subclass object is assigned to a variable such as `p` that is declared with a superclass type. The compiler has reserved enough space for a `Point` object, and cannot fit larger subclass objects in that space.

In Eiffel, this problem is resolved by disallowing subsumption for value-based objects — the object type must match the variable type exactly. A different approach is taken in C++, where the larger object is *sliced* to fit the available space, and its vptr field is not copied. A variable of value type will only ever contain an object of that exact type, and its vptr will always refer to the vtable for that type, so polymorphism is not present. This approach is explained and illustrated in [68, page 27]. A third solution, exemplified by Transframe's `fixed` modifier, is to prohibit subclasses of value types from adding new instance fields. This ensures that subclass objects will still fit in the space available, while keeping the subsumption property and hence enabling polymorphism.

Note that these restrictions do not apply to reference-based objects, as they are manipulated indirectly. Such objects are allocated on the heap, and the compiler need only reserve space on the stack for a pointer.

2.3.5 Automatic value type detection

Instead of placing the onus on programmers to annotate type definitions as value-based or reference-based, compiler analysis can be applied to determine which objects should be handled directly and which indirectly. Traditional Java implementations place all objects on the heap. However, a research project to automatically perform stack allocation of Java objects[44] found that typically 10–20% of objects could be stored on the stack instead of the heap. A similar result was obtained by an empirical study[70] that instrumented a Java VM to show which objects could have been stack-allocated.

Originating in the functional language research area, *escape analysis* has been applied to object-oriented languages, in particular Java[23, 34]. This analysis can be used to support automated stack-allocation of objects, as well as removing redundant synchronisation in multi-threaded programs. The results concur with the studies mentioned above.

A closely related area of research is *object inlining*[41] which uses compile-time analysis to determine when an object's instance fields can be stored directly inside the object rather than as pointers to separate objects on the heap. This optimisation can be made when the child object pointed to is not shared by any other parent object, i.e. the child object is not

```
class Stack {

  method push(element:int)

  method pop() : int
}
```

Figure 2.4: Fixed element type

```
class Stack [T] {

  method push(element:T)

  method pop() : T
}
```

Figure 2.5: Generic element type

aliased, and the pointer to it is redundant. While the aim of object inlining is to reduce pointer dereferencing, the result is to convert a reference-based object into a value-based one.

2.4 Genericity

A powerful addition to a programming language type system is a mechanism for *genericity* — the ability to write code that is not specific to just one type of data. This is analogous to the way an ordinary function can be generalised by giving it a parameter to stand in for any supplied value. The function is then far more useful than one in which a constant value is hardcoded.

The GJ project, to extend Java with genericity, has produced a tutorial[15] that gives good examples of programming with genericity, contrasted with non-generic code.

2.4.1 Parametric polymorphism

Just as functions may be parameterised by values, parametric polymorphism enables parameterisation by types. Rather than write code that refers to one particular type, a type parameter can be used to represent "some type" of data. Parametric polymorphism is a mechanism for genericity: it allows code to be more generally applicable than for a single fixed type. For object-oriented languages, parametric polymorphism can be applied to individual methods, and to entire types such as classes and interfaces. A good introduction to different forms of polymorphism, including parametric polymorphism, can be found in [26].

The classic example application of parametric polymorphism is data structures, where the code to implement the structure would work the same regardless of the type of element stored. Most genericity designs allow the programmer to associate one or more named type parameters with a method or type definition, then use those parameter names in the contained code as though they were the names of actual defined types.

The pseudo-code in Figure 2.4 shows a data structure class whose element type is fixed, so the class can be used to store `int` values only. However, the class in Figure 2.5 is generic: it has a type parameter `T` which represents any type. Users of this class must supply a type argument for `T` when creating instances of it, and declare that type whenever the class `Stack` is itself used as a type. For example, a variable declared and initialised with a `Stack` of `Point`s would look something like:

```
Stack[Point] sp = new Stack[Point]()
```

Where `Point` is part of the type specification, in a similar way to C type declarations involving pointers and arrays where the constituent type forms part of the full type specification.

In languages without parametric polymorphism, such as Java, general purpose generic data structures can be simulated using subtype polymorphism instead. This form takes advantage of the subsumption property mentioned in Section 2.3.4, where any object can be treated as an instance of one of its superclasses. In Java, the `java.lang.Object` class exists at the topmost position in the inheritance hierarchy, and so is the ultimate supertype of all objects. Java data structures can therefore be used to store any type of object, by treating its elements as `Object` instances.

However, this solution is less expressive than using parametric polymorphism, because a loss of type information occurs. Any type of object can be stored together with any other type, so when objects are retrieved from the structure their known compile-time type can only be `java.lang.Object`. Programmers must manually insert type casts to restore the lost type information. In Java at least, these casts are checked at run-time so that an exception can be thrown if the cast fails. A key benefit of parametric polymorphism is that its extra expressive power helps keep programs within the type system, rather than resorting to type casts, and reduces the need for run-time checks.

Parametric polymorphism is independent of object-oriented programming, as it can be seen in functional languages and in multi-paradigm languages such as C++. The Standard Template Library[87] for C++ is an example of the power of generic programming; no inheritance is used.

2.4.2 Constraint specification

In the description of parametric polymorphism above, the type parameters were *unbounded* in the sense that they could stand for any type. A limitation of this is that no assumptions can be made about the actual types that users of a generic class might supply. Within the generic class, objects of a generic type (e.g. `element` in Figure 2.5) cannot be used as the receiver of a method call, because their actual run-time type is unknown. Some generic classes might need to ensure that the type arguments used in place of a type parameter support some minimum functionality. For example, a hash table data structure needs to be able to find a hash code for objects stored within it, and be able to compare objects for equality.

With bounded parametric polymorphism, type parameters can be annotated with a type bound. In the example code of Figure 2.6, the type parameter `Key` in the `Dictionary` class has a bound whose type is `Comparable[Key]`. Only subtypes of this bound can be supplied as arguments to that type parameter. With this knowledge, code in the generic class can safely assume that generic objects will implement the method signatures present in the bound type. For languages with a topmost class such as `Object`, unbounded type parameters can be seen as implicitly bounded by that class. Continuing the function analogy, bounds are the equivalent of the types given to function parameters, which restrict argument values to only those of that type.

```
interface Comparable [T] {

  method equals(other:T) : boolean
}


class Square implements Comparable[Square] {

  method equals(other:Square) : boolean { ... }
}


class Dictionary [Key <: Comparable[Key] , Element ] { ... }
```

Figure 2.6: Bounded type parameters

A further extension of this bounding scheme is known as *F-bounded* parametric polymorphism, which allows the type parameter to appear recursively in its own bound[25]. This turns out to be useful in many practical examples of parameter bound specifications, especially with *binary methods* which are discussed in Section 2.7. The `Key` type parameter in Figure 2.6 is F-bounded, as it is used in its own bound. The intent is that `Key` represents any type that defines a comparison operation with itself — one such type is the `Square` class. The GJ proposal for Java[17] even allows type parameters to be mutually recursive: they can appear in each others bounds. Again, there are non-trivial examples where this ability enables static type safety instead of requiring type casts.

Some designs for parametric polymorphism have rejected bounding, on the grounds that it can require classes to explicitly state each and every supertype, which is tedious and cannot anticipate future needs. The PolyJ Java extension proposal[81] provides *where clauses*[39] that allow programmers to specify a list of method signatures for each type parameter. Any type that implements all these signatures is a valid type argument for that parameter. From a type checking point of view, this is more flexible than using bounds. However, implementation involves extra indirection costs when calling methods on generic objects, similar to the needs of interface method calls as covered in Section 2.2.6. This construct could be seen as an anonymous interface, with classes considered to implement it based on structural subtyping rather than named subtyping.

2.4.3 Virtual types

An alternative to parametric polymorphism for supporting genericity is the *virtual types* mechanism[22, 54], which originated in the Beta language and has appeared more recently in a proposal to extend Java[111]. Instead of type parameters, classes can contain types as members, alongside methods and fields. The bounds on these type members can be specialised by subclasses.

A variety of papers have compared the expressive power of parametric polymorphism and virtual types[20, 89, 112], and discussed whether virtual types require run-time type

checks to ensure type safety[20, 117]. The outcome of these studies is inconclusive, as both mechanisms appear to be equally expressive despite their different approaches to supporting genericity. Interestingly, the idea of virtual types as type fields in objects reappears in the context of implementing parametric polymorphism — see Section 2.6.4.

Most of the motivating examples of virtual types have been related to parallel class hierarchies, and finding ways of retaining type safety when subclasses attempt to specialise types. Using parametric polymorphism for these applications is often awkward, or risks static type safety. These issues are related to variance, which is discussed more fully in Section 2.7.

2.5 Implementations of genericity

The previous section described genericity mechanisms at the level of language design and type systems. It is important to distinguish the design features from the implementation technique, as there are many conceivable ways of compiling generic code, each with specific trade-offs. For example, the proposals for extending Java with support for genericity showed a variety of implementation techniques, described in the following sections.

Note that the terminology of some descriptions of genericity can imply an implementation approach, for example the use of terms like *generic instantiation*. Whether an implementation actually creates separate copies of a generic class for different type arguments is not always specified by the design, however.

2.5.1 Heterogeneous expansion

The C++ template feature is probably the most widely known genericity mechanism. A template is a specification for a family of classes or functions, and is often described as a code macro facility due to its implementation as a compile-time expansion process. By design, templates are completely handled at compile-time and link-time, and have no run-time representation[104, page 379]. In this case, the term *generic instantiation* really does apply, since only the instantiations exist at run-time. The possibility of several distinct instantiations of the same template gives rise to the term *heterogeneous* for this translation scheme.

The advantage of this approach is that there is no abstraction penalty, as each instantiation of a template is equivalent to the type-specific code that would otherwise have been written. This is particularly important when generic code must work with both value-based and reference-based types — a topic discussed further in Section 2.6. However, there are several potential disadvantages to template expansion:

- A program's code size could be significantly increased, due to many separate instantiations of a template with different types. This drawback, often known as *code bloat*, can be alleviated by careful class design and sophisticated compilers and linkers. Similar problems can occur with the generation of separate vtables for each expansion of a class template, in cases where the vtable contains a large number of function pointers.

- The source code for a template must be available to the compiler when it is used, which could cause difficulties for proprietary libraries, separate compilation and dynamic code loading.
- Templates are not type checked themselves, but each instantiation is. This can result in type error messages that are harder to relate to the programmer's original source code.

A limited comparison can be made by using the function analogy from previous sections: expansion could be viewed as the genericity equivalent of function inlining. In both cases, the body of the abstraction is required at the point of expansion, resulting in increased code size but with opportunities for local optimisation of the expanded code.

Expansion need not occur at compile-time. One approach for genericity in Java[8] was to extend the Java VM class loading mechanism to generate type-specific instantiations of a generic class on request. An advantage is that only a single Java class file need be distributed, with the VM using it as a template for expansion.

2.5.2 Homogeneous translation

In contrast to the expansion of generic code, an *homogeneous* translation results in a single run-time representation of generic code. The major advantage of this approach is that it does not give rise to code bloat. However, since the same code must handle all types of data, it may not be as efficient as code written specifically for one type. Compilers are better able to optimise code containing explicit types such as integers than generic values of unknown type. Hence, the trade-off is between code size and run-time speed.

Typical examples of the homogeneous translation are the Pizza[82] and GJ[17] compilers, which support genericity extensions to Java. The extensions can be translated into plain Java by replacing type parameters with their bounds, and inserting type casts as necessary. In effect, these translations are an automated version of the technique shown in Section 2.4 for simulating genericity using subtype polymorphism. While this approach provides a remarkably simple way of supporting parametric polymorphism, there are two significant drawbacks:

- Only reference-based types can be used as type arguments, not value-based types such as `int` and `double`. These types are not in the inheritance hierarchy, so cannot be treated as instances of the `java.lang.Object` class or the type of the parameter bound.
- Type information is erased by the compiler, and not kept at run-time. While many applications do not require this, the full expressive power of genericity cannot be supported in the language without it. Operations such as casts and reflection require run-time information about the type arguments used in generic code.

These two issues are related: the GJ compiler is only able to discard type information because it does not have to support value-based types. Otherwise, the type information could be needed at run-time — Section 2.6 explains this problem in more detail.

A minor issue related to the choice of expansion or homogeneous translation is how `static` fields are represented. With generic types, there are two design possibilities:

either there is one instance of the data for the generic class as a whole, or there is one for each possible parameterisation. This is discussed in the NextGen proposal[27] for Java extensions, which provides both forms depending on whether static fields and methods depend on any type parameters or not. To achieve this, it employs run-time type support as discussed in the following section.

2.5.3 Run-time type support

The potential restrictions of the homogeneous approach described above can be alleviated by maintaining type information at run-time (beyond that needed just for implementing dynamic dispatch). The NextGen proposal for Java genericity builds on the design and implementation of GJ, by adding extra wrapper classes and interfaces to hold the extra type information that GJ would have discarded. In this way, the bulk of the code for a generic class is kept in an unexpanded class form. Objects are instances of the simple wrapper classes that extend the unexpanded classes with information about the specific type arguments. NextGen supports use of type parameters in contexts that GJ cannot, such as casts and `instanceof` expressions.

Another Java genericity proposal[98] compares the different approaches to implementation, and highlights their lack of support for reflection. The paper describes extensions to the Java VM and class files that support the distinction between generic classes, type parameters and individual uses of a generic class. Additionally, Java's reflection capabilities are extended to enable introspection on generic code, which cannot be achieved with the purely homogeneous translations.

A further proposal for retaining run-time type information is shown in [120, 121] which uses *type descriptor* objects instead of wrapper subclasses. The design employs a type descriptor manager utility to create these objects at class load-time, and safely handles the potential runaway creation of type descriptors that could result from recursively defined generic types. The work highlights the design space for run-time type handling in the homogeneous translation of Java genericity, showing how the various approaches compare. However, neither this proposal, or the others previously described in this section, support the use of value types as type arguments.

Aside from Java, a proposal for extending the Microsoft Common Language Runtime (CLR)[74] with support for generics[59] also uses a load-time mechanism for managing type descriptors. The CLR class loader handles the construction of type information specific to each use of a generic class, sharing it whenever the representation is compatible with one already built. Furthermore, their design includes lazily-generated method code, specialised for the run-time type arguments. Like the type descriptors, compatible code is not duplicated. A key benefit of this scheme over the Java proposals is the ability to support value types as type arguments, as well as reference-based types. The authors acknowledge the benefit of using a VM environment such as the CLR to support this amount of design flexibility.

The support for genericity and value types in the Microsoft CLR is perhaps the most closely related work to that presented in this dissertation. However, their design was

first described publicly in [59] at which time the design and implementation of the Pluto language and compiler had already been completed. Therefore the work detailed in this dissertation was developed independently, without knowledge of that described in [59].

2.6 Implementations of mixed genericity and value-based types

As briefly mentioned above, the implementation of genericity is complicated by the presence of value types. This is because objects of value types can be a variety of sizes. In contrast, all reference-based types are manipulated indirectly via a pointer, whose size is the same regardless of the object pointed to. In generic classes, a type parameter may be used as the type of a variable, a field or a method parameter. To represent these generic variables at run-time, the compiler must reserve storage space for them. However, if the type parameter can abstract over value types as well as reference types then the required size of a generic variable cannot be determined at compile-time. Indeed, the size needs could be different in each use of a generic class, as different type arguments are used in place of a type parameter.

Besides data layout, the generated code must also be aware of the size of the objects it is manipulating. Variable assignments and method calls and returns require data to be copied from place to place, which needs to be explicitly programmed into the code executed at run-time. The following sections describe a variety of strategies for dealing with these problems, some of which are reviewed in [10, page 373].

2.6.1 Expansion approach

Implementations of genericity that use heterogeneous expansion schemes as described in Section 2.5.1 solve this problem by avoiding it. Since generic classes do not exist at run-time, there are no generic variables to be represented. After expansion, the compiler can determine the size of all the variables in each instantiation. Likewise, the method code is specialised for each instantiation and will therefore be able to handle known-size data.

This is the approach taken for C++ templates, which can be instantiated for potentially any type. Good quality C++ implementations can minimise the duplication of identical code by using a common instantiation for all pointer-based types, as the size of a pointer is the same regardless of what it points to. The SmallEiffel compiler also produces separate code for different parameterisations of a generic class[35]. Sather's parameterised classes are handled in the same way: the language specification[103] describes this as akin to a structured macro facility.

As described above, a proposal for extending the Microsoft CLR with generics is able to handle value types in generic code by generating specialised method code at class load-time. Code and type descriptors are shared for compatible type arguments (those whose representation is identical), avoiding unnecessary duplication.

2.6.2 Fully-boxed and tagged values

Another way of avoiding the problem of different value type sizes is to add a level of indirection, and treat all values like reference-based types. Variables of all types now hold

a pointer, with the object stored elsewhere (typically on the heap). Naturally, this defeats the purpose of distinguishing value types. In the functional language community, this technique is known as the *fully-boxed* approach.

In practice, the overhead of manipulating basic types such as integers using indirection is just too great. For example, test results in [66] show the disparity between arithmetic operations in C and various functional language implementations. A common improvement to the boxed representation is some form of *tagging*, where one or two bits of each value are used to distinguish pointers from certain types such as integers or floating-point values. This allows integer arithmetic to avoid the costs of full boxing, but does not generalise to handling all value types directly. In addition, tagged values cannot be used directly as integers: the tag must be masked off first, and the result re-tagged. Implementations of Smalltalk[45, 63, 118] use variations on the tagging scheme to handle small integer values more efficiently than general objects, although this is an optimisation rather than a programmer-visible feature.

The tag is a simple form of self-describing data. When values are passed into a generic context, the tag allows generic code to determine whether the value is a pointer to a boxed object or a simple integer value.

2.6.3 Coercion-based boxing

Ideally, only generic code should pay the cost of fully-boxed or tagged values. In non-generic code (integer arithmetic for example) the compiler can deduce a value's type and so the tag is unnecessary. An improvement on fully-boxed objects is to box value types only when they are used in generic situations. This can be achieved by *coercing* values to and from a boxed representation whenever they are passed to or returned from a generic context.

Several object-oriented languages perform this kind of automatic boxing and unboxing, and not just in the context of genericity. In Eiffel, `expanded` values are boxed when re-attached to a non-`expanded` variable. Microsoft's C# language[53] supports value type objects, and the specification states that they are boxed as necessary to support a uniform handling of types. Similarly, Sather's immutable value classes can extend abstract classes, so boxing occurs when value objects are assigned to abstract-typed variables. Although Java does not allow its primitive types to be used in situations requiring an object, programmers can manually box and unbox primitive values using wrapper classes.

Many functional languages support genericity, although a function's type parameters are usually implicit, and the types of variables are inferred by the compiler rather than explicitly written. However, the problem of implementing genericity while efficiently handling values remains — in this setting, it is known as *unboxed polymorphism*. A coercion-based solution was outlined in [65], and implemented[66] in the Gallium compiler for ML. Benchmark results showed a significant improvement over existing ML compilers that used a fully-boxed representation, particularly for non-generic code such as arithmetic. The SML/NJ compiler[96] was also modified to use coercions instead of a fully-boxed representation.

The coercion technique does not require type information to be kept for use at run-time. However, a limitation of this approach is that recursive (and mutable) data structures such as lists and trees must keep their elements in a boxed representation. To avoid recursively boxing an entire data structure when it is passed into a generic context, these data types are implemented using boxed elements at all times[66]. This restriction is potentially expensive: for example, an array of integers would be implemented as an array of pointers to boxed integers. Assuming pointers and integers are the same size, this can require at least twice as much space as an array of unboxed integers.

2.6.4 Type-passing

A number of research efforts have attempted to alleviate the limitations of the simple coercion-based solution in functional languages by using run-time type information. These solutions are generally described as *type-passing* since they involve passing type information to generic functions at run-time, to enable them to manipulate generic values. In effect, type parameters become ordinary function parameters, with the type arguments that are passed in used to describe the actual type of generic values. The function body can then examine these type descriptors and handle the values accordingly.

A framework and potential applications of inspecting run-time type information are described in [38, 52] under the term *intensional type analysis*. In essence, the compiler uses a second-order intermediate language, with explicit type parameters and type switch constructs inserted to dispatch on the type arguments. A similar technique was used in [116] to implement garbage collection without tagging data, by representing type parameters and type arguments as ordinary first-order constructs.

Type-passing has been implemented in ML compilers such as FLINT[95] and TIL[110], showing improvements over the coercion-based compilers Gallium and SML/NJ. The FLINT compiler employs a trade-off between run-time intensional type analysis and the full boxing performed by simple coercion-based implementations. Choosing the trade-off point is done using *flexible representation analysis*[94], which can mitigate the cost of boxing by limiting it to only one or two levels deep.

A consequence of using type information at run-time is that this information must be propagated throughout the compiler itself. Indeed, the TIL compiler[110] is named after the term Typed Intermediate Language. Many of the type-passing implementations described above have made use of typed intermediate representations[77], which enable features such as those presented in Section 2.2.7 and have the added benefit of making the compiler easier to debug. The Java bytecode format[67] can also be considered a typed intermediate language, as it is designed to be type checked by the verification process in a Java VM.

Concerns about the efficiency of using type-passing to implement languages like ML that contain nested functions have been addressed by attempts to compute the type descriptors at compile-time. This avoids the potentially expensive creation of complex descriptors at run-time, by *lifting*[76, 91] the applications of type arguments out of functions. Similar concerns about the overhead of type descriptor construction are addressed in the

work on extending SML/NJ to use the FLINT intermediate representation[97].

Compared to functional languages, there are comparatively few descriptions of type-passing in object-oriented language implementations. While functions are the constructs using type parameters in a functional language, it is classes that are parameterised in object-oriented languages. Some genericity designs allow methods to be parameterised as well as classes, although this is a lesser used feature. Therefore, type arguments are not passed explicitly at each method call, as they would be in a functional language implementation, but are associated with the individual objects that are instances of a generic class.

The proposed design for genericity in Microsoft's CLR[59] neatly describes various approaches to storing this type argument data, either in the object as hidden fields, or factored into a separate *type environment* shared by all objects that use those same type arguments. In this design, each different use of a generic class gives rise to a separate vtable, with the type descriptors stored alongside the method pointers. The paper also acknowledges the importance of lifting type descriptor construction to avoid excessive run-time overhead, based on information that can be determined at compile-time. One of the designs for genericity in Java[121] details a similar scheme, where generic classes are augmented with an additional field that points to a type descriptor. These descriptors are constructed at class load-time, and are shared to avoid duplication. The Theta language for the Thor object-oriented database system is implemented using *p-blocks*[79], which combine vtables and run-time type information. Again, careful design features are included to avoid infinite type descriptor structures.

2.6.5 Dictionaries and dope vectors

The reason for maintaining type information at run-time is to enable code to deal correctly with data whose type cannot be known at compile-time. This is similar to the needs of dynamic dispatch, where the precise code to be executed will not be known until run-time.

Another approach to solving the problem of mixed genericity and value types is to make the type descriptors used as type arguments behave like vtables. Instead of producing generic code that uses information from the type descriptors to manipulate generic values, each type descriptor has pointers to functions such as `copy`, `compare` etc. that are specific to that type. Generic code can then dispatch to these functions at run-time, to manipulate the generic values. In Grune et al[51], this special vtable is termed a *dope vector*. It is also known as a *dictionary*, as used in the implementation of Haskell's type classes[83] and a version of the SML/NJ compiler[33].

The simple coercion-based implementation of functional languages[66] suggests a similar idea, enabling unboxed elements in data structures such as arrays, using type-specific access functions to read and write elements. A possible disadvantage is the cost of dynamic dispatch for every basic operation on generic values, as the bodies of these functions will follow a similar pattern for each type, only varying in the specific values for size, layout etc.

2.6.6 Variable size data

The problems of dealing with data of unknown size has also been investigated in the context of databases and persistent object stores. These require type information in order to store and retrieve objects in a database, using some form of serialisation process. Run-time type information is used to implement genericity in the Napier88 language[78], which supports object and procedure persistence. The design is similar to the coercion-based techniques described above for functional languages, except that type parameters are encoding in an *envelope procedure* — essentially a curried function. The implementation uses closures for these nested functions, which capture the type arguments for each call of the generic procedure. Generic code can manipulate values of unknown type by using this captured information at run-time to determine how to convert the value to and from a uniform representation.

The same researchers showed a related use[37] of run-time type information to implement object field addressing in languages with structural subtyping, or where multiple inheritance of data prevents compile-time determination of field locations. This can be viewed as a comparable solution to the way object models cope with multiple inheritance, as described in Section 2.2.5. By using *address maps* at run-time, that are shared wherever possible, data of unknown size and layout can be handled. This approach is similar to the storage of type descriptors described in [59].

A simple way of handling variable sized data is to first pass a value that indicates the size of data to follow. This is used in a variety of application areas: Ethernet frames[36], Java's serialisation protocol[107] and the variable length arrays supported by the GNU C compiler[101]. Such a technique could be applied to the implementation of generic code, by storing generic values directly on the stack as variable sized data with the self-describing size field at the beginning. A disadvantage is that the location of data stored after the generic value is unknown at compile-time. For example, a sequence of variable sized values passed to a generic method would be accessed by calculating their locations using offsets determined at run-time. The same situation would occur with object fields, if a class contained more than one instance field of a generic type.

2.7 Variance support

A further extension of object-oriented type systems is to allow *variance* in method signatures and in the type parameters used for parametric polymorphism. The purpose of this is to enhance the expressiveness of the type system, by increasing the flexibility of its rules. A central concern with type systems is the issue of type safety: ensuring that programs cannot "go wrong" at run-time, by catching type errors at compile-time. The following sections describe how variance can be supported in these two language constructs, while maintaining type safety.

2.7.1 Method signatures

In Java, the type system rule for method overriding[49, Section 8.4.6] states that the signatures of the two methods must be identical. The C++ type system is more flexible

```
class Shape {
  method clone() : Shape
}

class Triangle extends Shape {
  method clone() : Triangle
}

class Circle extends Shape {
  method clone() : Circle
}
```

Figure 2.7: Covariant return types

in this respect[55], as it allows the return type of an overriding method to be a subtype of the original. This is known as *covariance* on the return type, and is type safe[1] since the more specialised return type can always be treated as the original due to subsumption. A common example application of this rule is a `clone` method, as shown in the pseudo-code of Figure 2.7. In Java, the type checker would disallow the attempts in classes `Triangle` and `Circle` to override the `clone` method with a narrower return type. However, this would be accepted by the C++ type system, and in other object-oriented languages such as Eiffel and the GJ extension to Java.

For method parameter types, the situation is reversed: overriding methods can use wider types, while retaining type safety[1]. This rule is known as *contravariance* on method parameters. Most object-oriented languages, including Java and C++, do not even allow contravariance of method parameter types. They require *invariance* — the types cannot change at all when overriding. However, allowing contravariance of method parameters is considered less important in comparison with allowing covariance of return types, since there are fewer cases in which that flexibility can be used.

In many practical cases, the intuitive requirement is for method parameter types to be specialised on overriding. The most common examples are known as *binary methods*[19] — those with one parameter whose type is the same as the class that defines the method. For example, binary relations such as *equals* or *less-than*, and operators such as *add* or *multiply* often appear in object-oriented programs as methods within classes. In Figure 2.8, the override of the `equals` method in class `Triangle` attempts to specialise the parameter type — a violation of the contravariant rule. This is unfortunate, as the natural desire is for a `Triangle` to be compared with another `Triangle`, and likewise for other subclasses of `Shape`. Type safe overriding is used in the `Circle` subclass, but the parameter type is only `Shape` rather than the desired type.

Controversially, Eiffel allows specialisation of method parameter types, which enables binary methods to be written at the expense of type safety. A long-running debate in the literature[19, 28, 92] has been concerned with supporting binary methods within a type system that is still safe. A variety of proposals have been put forward, resulting in interesting new type system concepts such as F-bounded type parameters[25], MyType[18],

```
class Shape {

  // Original method signature
  method equals(other:Shape) : boolean
}

class Triangle extends Shape {

  // Desired signature for overriding, but not type safe
  method equals(other:Triangle ) : boolean
}

class Circle extends Shape {

  // Type-safe overriding, but not desired parameter type
  method equals(other:Shape) : boolean
}
```

Figure 2.8: Overriding binary methods

multi-methods[75] and the Match relation[21]. The trade-off between these techniques is extensively reviewed in [1] and [19].

2.7.2 Type parameter annotations

The other part of the type system where variance can usefully be applied is the type parameters used in parametric polymorphism. The aim is to increase the expressiveness of the type system by allowing more subtyping between different uses of a generic class. In languages such as C++ and GJ, type parameters are invariant, so the assignment attempt shown in Figure 2.9 would be disallowed by the type system. Even though the type `Triangle` is a subtype of `Shape`, this does not extend into type arguments, so the type `List[Triangle]` is not considered a subtype of `List[Shape]`.

Similar to its treatment of method parameter variance, Eiffel has implicit covariance of type parameters. This means that a supplied type argument (such as `Triangle`) can be a subtype of the expected type (in this case `Shape`) and hence Eiffel would allow the example assignment attempt. In the NextGen[27] extension for Java, type parameters are invariant by default, but the programmer can annotate them to indicate that covariance is required. To ensure type safety, NextGen makes restrictions on the use of a covariant type parameter, disallowing its appearance in method parameter types and publicly-visible fields. As noted in [27], contravariant type parameters also could be allowed, with corresponding restrictions, but were not supported due to the limitations of translating NextGen into the Java language.

Although Java does not currently support genericity, it has an inbuilt array type which is parameterised by the type of element stored. Interestingly, Java permits covariance in the subtyping of arrays, and allows assignments such as:

```
Shape[] shapeArray = new Triangle[10];  // Assignment permitted
```

```
class Shape { ... }

class Triangle extends Shape { ... }


class List [T] {

  method get(index:int) : T

  method set(index:int, element:T)
}


List [Shape] shapeList = new List[Triangle]()
```

Figure 2.9: Subtyping of generic types

```
shapeArray[3] = new Square();          // Permitted at compile-time
```

However, the operation of storing an element into the array can be seen as equivalent to the `set` method in the `List` class of Figure 2.9. Since the element type appears in the method parameter list, allowing covariance leads to a potential loophole in the type system. In Java, this is solved by a run-time check[49, Section 10.10] on the actual type of element being stored into the array, against the expected element type of the array itself. In the code above, an exception will be thrown at run-time by the attempt to store a `Square` object into an array of `Triangle`s. To support this, Java array objects maintain run-time type information about their element type.

2.8 Summary

In this chapter, three type system extensions are described that enhance the expressive power of a programming language:

Value Types Enables the definition of data types whose instances are manipulated directly. By providing both value types and reference types in the language, programmers are able to choose the most appropriate kind to suit the intended use of their data types.

Genericity Allows language constructs such as types and methods to be parameterised with respect to type. Program code can be generalised to work with many different types of data, rather than one specific type. This promotes code reuse, and is particularly important for the design of general purpose libraries.

Variance Increases the flexibility of the type system, allowing it to recognise relationships among types that match the actual concepts being modelled by programs. Extra expressive power reduces the need for programmers to circumvent the type system in their designs, and avoids the introduction of run-time checks.

In addition, this chapter surveys many techniques used to implement these features in compilers for actual programming languages, including functional and object-oriented. In almost all designs, a trade-off must be made between two or more conflicting goals, e.g. size against speed, or expressive power against efficient implementation. With this in mind, and based on the background work reviewed in this chapter, some observations can be made:

- There are few object-oriented languages that support both genericity and user-defined value types together. C++ allows both, but implements genericity using the expansion technique, which can result in code and data duplication. Eiffel implementation details are relatively scarce, although the SmallEiffel compiler appears to use expansion for uses of genericity with value types. The C# extension with genericity employs load-time expansion of generic code to support value types, and is based on a VM run-time instead of being fully-compiled.
- Support for variance in object-oriented languages is rare, despite its benefits and relative ease of implementation.
- Advanced techniques for implementing genericity and value types, such as the type-passing approach, are mostly used for functional languages. Type-passing for object-oriented languages is less well developed, although some of the later proposals for genericity in Java and C# have employed it.
- A common theme running through the review has been the use of type information at run-time. Besides enabling efficient implementations, this approach can be essential to support language facilities, such as reflection, that rely on it.

Consequently, the aim of the work presented in this dissertation is to demonstrate that an object-oriented language with the three key features listed above can be efficiently implemented using type-passing, supported by run-time type information.

Chapter 3

The basic Pluto language and compiler

The Pluto language, and the compiler written to implement it, is the focus of the work presented in this dissertation. This chapter gives a rationale for the language and describes the design of the compiler. In addition, the object model implementation and support of object-oriented features in the target language is covered. Subsequent chapters build on this chapter by covering more specific details about the implementation of value types, genericity and variance.

3.1 Design introduction

The purpose of the Pluto language design is to be a proof of concept for the implementation of the features summarised at the end of the previous chapter. Using an existing language such as Eiffel or C++ would have required much extra effort to support all the language features, beyond those of interest. An alternative approach would be to devise a cut down version of one of these languages. However, this would in effect be a new language anyway, so Pluto was designed to contain just the features required. Although Pluto is not intended as a production language, for commercial software development, it contains enough functionality so as to be realistic.

3.2 Language description

The basic Pluto language is a straightforward class-based object-orientated language, with classes, fields and methods. Its syntax should appear familiar to those programmers with experience of Java or C++. The salient features of Pluto are:

- Like Java, types are defined by classes and interfaces, where classes can extend one superclass, but implement multiple interfaces. Interfaces contain only abstract method signatures, and can extend multiple superinterfaces.
- Class definitions allow for a list of constructor value parameters, that can be used in field initialisation expressions to construct instances. Subclasses must provide a matching list of values in their `extends` superclass clause. The example Pluto program in Figure 3.2 below shows how these parameters are written and used.

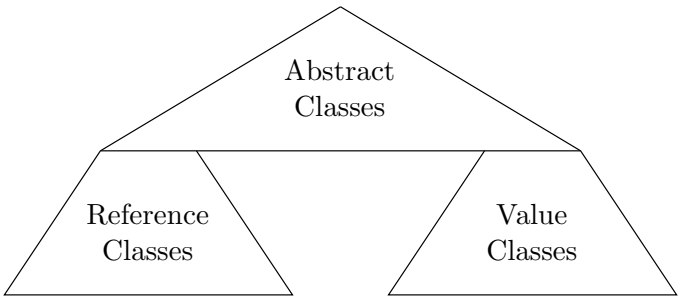

Figure 3.1: Arrangement of class kinds in the hierarchy

- Class members include instance fields and methods, which are associated with specific objects of the class, and static fields and methods which are independent of objects. An instance method may be marked `final` to indicate that it cannot be overridden, or `abstract` to indicate that it has no implementation body. For simplicity, field names and method names cannot be overloaded or hidden.

- Method names can be symbolic as well as alphabetic, allowing common operator names such as `+` and `>=` to be used. Calls to an instance method with a symbolic name can be expressed in infix form if it has one parameter, or prefix form if it has no parameters. The example program described below shows how this syntax is used. The Boolean logical infix operators (`&&` and `||`) are the only operators that are built into the language syntax. This means there is a shallow precedence hierarchy: the dot notation `x.y` binds most tightly, prefix expressions bind more tightly than infix expressions, and the Boolean logical operators bind least tightly.

- There are three kinds of class:

  Abstract classes cannot be instantiated, but can be extended by any kind of class. They can extend only other abstract classes. There is a distinguished abstract class called `Object` which is the ultimate superclass of any other class. A class containing abstract methods, either declared or inherited, must itself be declared abstract.

  Reference classes are similar to traditional Java classes: their instances are allocated on the heap, and are manipulated indirectly via pointers. Objects of a reference class can be aliased, as there may be several pointers to the same object. The superclass of a reference class must be either another reference class or an abstract class.

  Value class instances are manipulated directly and cannot be aliased. A value class can extend only abstract classes or other value classes. Value classes are the only kind of value types in Pluto — Chapter 4 gives more detail on these, including some restrictions on subclasses of value classes. An object whose type is a value class is *immutable*: its instance fields cannot be updated once created. Section 4.1.2 explains the rationale for this decision.

  Reference and value classes may be declared `final`, to indicate that they cannot have subclasses. The arrangement of the different kinds in the class hierarchy is shown in Figure 3.1. The restriction on abstract classes extending only other abstract classes keeps the type system simpler, but it could be loosened to allow abstract classes to extend value and reference classes. However, the type system would then need to ensure that a reference class does not extend a value class indirectly via an

intermediate abstract class. Otherwise there could be confusing behaviour caused by the different semantics of value-based and reference-based types. See Chapter 4 for details about the support for value-based types in Pluto.

- Fields, value parameters and local variables always refer to a distinct object for reference types — there is no `null` or missing value. Local variables can be declared mutable, but cannot be used until they have been assigned a value at least once.
- Classes, interfaces and methods support multiple type parameters, which can be bounded. Chapter 5 covers the design and implementation of genericity in more detail.
- In the class hierarchy, method signatures can be varied when overridden: return types can be specialised and parameters can be generalised, in accordance with the variance rules covered in Section 2.7. This ability is not supported for multiple inheritance of method signatures through interface types. Whenever a method of some name is inherited through more than one supertype, the signatures must match exactly. The reason for this restriction is covered in Section 6.4.2.

Figure 3.2 lists a simple but complete Pluto program. The `point` class shows the use of symbolic method names. It defines a method called `==` with one parameter, and a method called `$` with no parameters. These methods can be called using infix and prefix expressions, respectively. Given a `point` object, say `p`, the prefix expression `$p` invokes the `$` method of the `p` object. Similarly, an infix expression such as `p == p` invokes the `==` method of the `p` object, with itself as the single argument. The code within the methods of the `point` class also shows use of this syntax, as it invokes the `==` and `$` methods that are defined by the `int` class.

The transcript in Figure 3.3 shows a typical build session, where the Pluto source in file `example.pto` is compiled into C code by the first command, then the C compiler and linker is invoked to build an executable.

3.3 Compiler structure

The overall design of the Pluto compiler is shown in Figure 3.4, which lists the different data structures used at each stage of compilation, together with the transformations applied at each pass.

3.3.1 Implementation language

The Pluto compiler is written in Java[49], which proved useful in the development of a complex piece of software such as a compiler. For the purpose of this experimental work, speed of execution was not especially important compared to correctness and ease of development. The addition of genericity to Java (in particular for collection data structures) would have improved the clarity of the compiler code, and reduced development time. It would be straightforward to rewrite the compiler in the Pluto language itself, provided some form of file reading and writing capability were added to the standard library.

The Java 2 Platform standard libraries contain a rich set of classes for building a graphical user interface, such as the AWT and Swing packages. In particular, the Swing JTree

```
final value class point(x:int, y:int) {
  field x : int = x;
  field y : int = y;
  method ==(other:point) : boolean {
    return this.x == other.x && this.y == other.y;
  }
  method $() : String {
    return "[" + $this.x + "," + $this.y + "]";
  }
  static field origin : point = new point(0, 0);
}

abstract class Shape(pos:point) {
  field position : point = pos;
  abstract method draw();
}

class Circle(centre:point, rad:int) extends Shape(centre) {
  field radius : int = rad;
  method draw() {
    String.println("A circle at " + $this.position + " with radius: " + $this.radius);
  }
}

abstract class Main() {
  static method run() : int {
    let s : Shape = new Circle(point.origin, 4);
    s.draw();
    return 0;
  }
}
```

Figure 3.2: Example Pluto program

```
% java pluto.Main example.pto

% gcc pto_src*.c -lpluto -o example

% ./example
A circle at [0,0] with radius: 4
```

Figure 3.3: Example build process and output

component allows a tree or graph structure to be examined using an intuitive graphical browser. It proved simple to add the necessary code to the Pluto compiler so that the abstract syntax tree, core tree and C level intermediate representation could be viewed using JTrees. This facility was of great use in locating errors in the compiler itself, by visual inspection of the intermediate representation of each stage.

3.3.2 Front end

The parser was constructed using the JavaCC tool[108], which takes a combined lexical and grammatical specification of a language and generates an LL(1) parser in Java code. Semantic action code within the grammar production rules is used to build an abstract syntax tree. Tree nodes are annotated with line and column number information, enabling accurate error messages to be produced from subsequent semantic checks. The next pass translates the abstract syntax into the core language, by a process of syntax desugaring. A formal grammar for the core language is given in Appendix A.

The following syntactic transformations are made at this stage:

- Symbolic method names are translated into alphabetic equivalents, and prefixed with an underscore to avoid clashes with defined names. Each symbolic character is translated, and the resulting words concatenated. For example, `+` becomes `_plus`, and `>=` becomes `_moreequals` etc. Note that the translated names for symbols do not imply anything about the meaning of that method: programs are free to use symbolic names for any method definition.

- Infix and prefix expressions are converted to the method call notation. E.g. `!x` becomes `x._not()`, and `this equals that` becomes `this.equals(that)`. With the symbolic name translation described above, code such as `x+y` would become `x._plus(y)`.

- Expressions involving the short-circuit Boolean logical operators `&&` and `||` are converted into equivalent control flow code that respects the lazy evaluation of the right-hand argument. For example, code such as `exp1 && exp2` would be translated into a sequence such as:

  ```
  var tmp1 : boolean
  if (exp1) { var tmp1 = exp2 } else { var tmp1 = false }
  tmp1
  ```

- Complex expressions are separated into single operations, with temporaries introduced for the intermediate results. As an example, `x+y+z` is translated into the code sequence `let tmp1 = x._plus(y) ; tmp1._plus(z)`. The temporary variable's type is inferred at a later stage.

- Default clauses are inserted for omitted syntax, such as the `else` block of an `if` declaration. The default class kind, if not specified, is `reference`. Classes that do not declare a superclass will extend class `Object`, which is also the default bound for type parameters. A missing `implements` clause for a class, or an `extends` clause for an interface, is assumed to be an empty list. The insertion of empty lists in place of missing explicit lists is also applied to the type arguments of all type names, and the type parameter lists of classes, interfaces and methods.

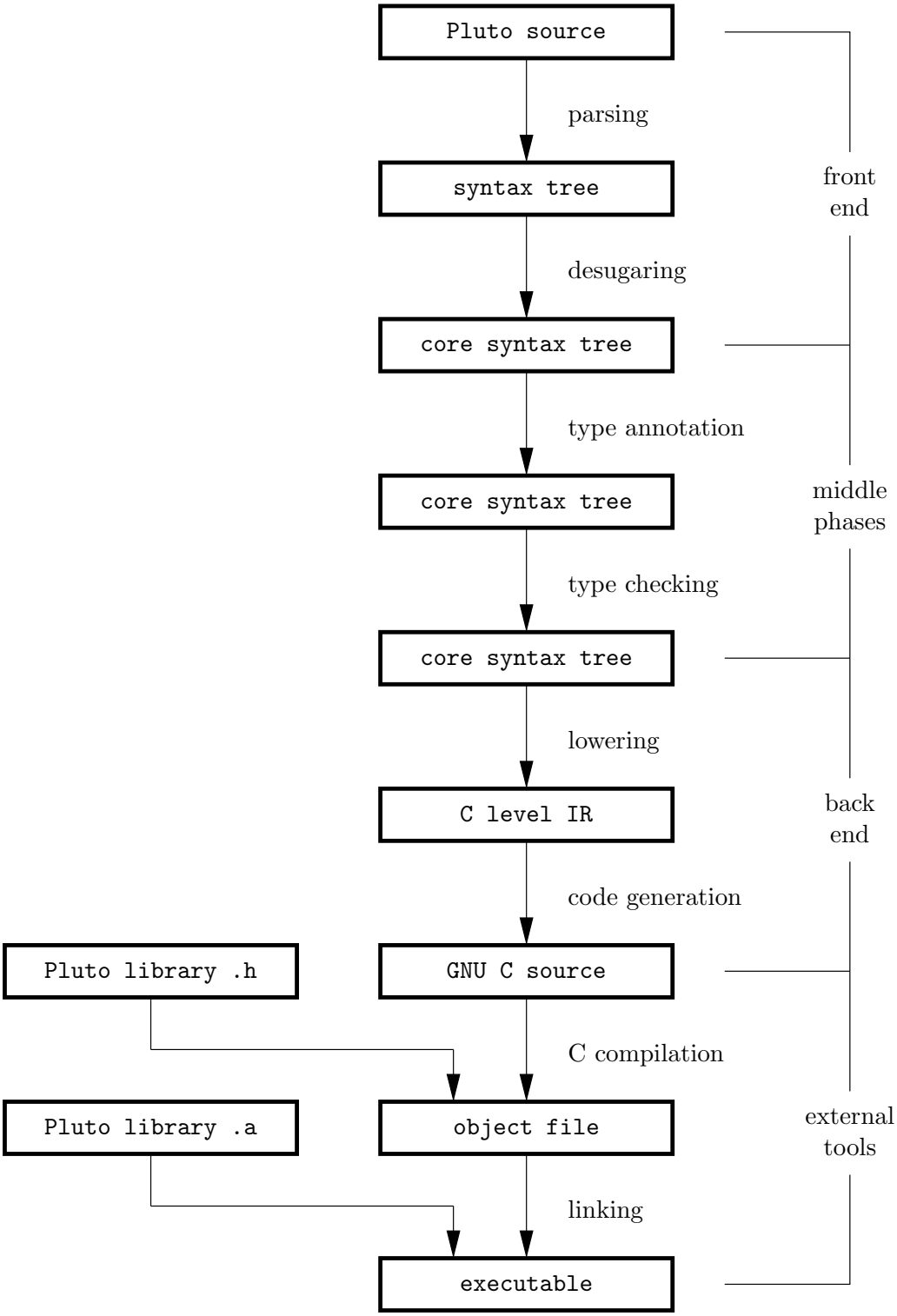

Figure 3.4: Pluto compiler overview

The only semantic checks made during the front end desugaring pass are to detect duplicates, such as two type definitions with the same name, or two fields or methods with the same name. As mentioned above, methods cannot be overloaded, so each class has at most one method with any given name. Field names and method names are in separate namespaces, allowing a field and method within some class to have the same name.

3.3.3 Middle phases

Once the core abstract syntax tree has been built, the next task is to do a complete pass over the entire tree, annotating nodes with type information. At this stage, the names of all class and interface definitions are known. The annotation phase works by linking the nodes that represent type names and variable names to their corresponding definition nodes, by looking up the definitions in a supplied context. The correct type context and value context for each node is usually constructed by its parent, and passed in to the node for annotation.

During type annotation simple context errors are detected, such as uses of variables or type parameter names that are not in scope. In addition, this pass ensures that classes and interfaces form a proper hierarchy, i.e. that there are no cycles in the graph of subtype relationships. Basic control flow analysis is done to ensure that variables are not used until they have definitely been assigned a value, and that reassignments are only made to mutable local variables.

After type annotation, another complete pass over the core tree is done to check the type safety rules of the language. During this process, class and interface definitions build a full set of member information including inherited fields and methods. Type checking proceeds by inferring the types of expressions such as method calls, field accesses and values, using the member information accessible through the type definitions. Then, expression types are checked against the expected type, i.e. that declared by the corresponding variable, field or parameter.

The type checking phase also ensures that class and interface inheritance is well-formed: for example, that overriding method signatures are compatible with the overridden one, and that `final` methods and classes are not overridden. Type names, and their list of type arguments, are checked in a similar manner to expressions by ensuring the number and suitability of the type arguments correctly matches the type definition.

A number of other type checks are made that are specific to the topics of value types, genericity and variance: these are discussed in Sections 4.5, 5.2.3 and 6.3 respectively.

3.3.4 Back end

When type checking is complete, the compiled program is deemed to have been accepted by the type system so no further type-errors can be generated beyond this point. The remaining work of the compiler is to convert the intermediate representation into target code. For the Pluto compiler the target language is C, a relatively low-level procedural language that has virtually no support for polymorphism. Therefore, the main task of the compiler back end is to lower the object-oriented core language to the level of C.

Similar to the translation from abstract syntax tree to core tree, the core tree is converted to a C level intermediate representation, `clir`, that makes explicit all data type conversions. Object-oriented constructs are replaced with the lower level operations that implement them, and run-time data structure layouts are determined. Once the program being compiled has been converted to the clir form, actual C code can easily be generated.

Later sections of this chapter describe in more detail the run-time representation (Section 3.5) and the code that is generated by the compiler back end (Section 3.7).

3.3.5 External tools

The code generated by the Pluto compiler is placed in C header and source files, which can be directly compiled by the GNU compiler, gcc[101], to produce ordinary object code files. Since most Pluto programs will use standard library types such as `int` and `Object`, the header files for these types must also be available when the generated code is compiled.

The object code produced by gcc can now be linked with the Pluto run-time environment, which is a small static library `libpluto.a`. The types defined by this library are described in Section 3.8. Both the generated code and the run-time library make use of a few standard C library utility functions such as `malloc`, `fputs` and `strlen`. In addition, an optional plug-in conservative garbage collector, such as Boehm's static library[12, 13], can be linked in to provide automatic memory management of the heap.

After linking the object files and libraries, for example with the GNU linker (ld) or via gcc, a platform-specific executable file is generated. Depending on the platform and linker options, the executable may have a dynamic dependency on the shared C library and support libraries. However, there are no dynamically linked libraries for the Pluto run-time itself.

3.4 Choice of target language

An important decision in the design of a compiler is the choice of target language. Traditionally, compilers generate assembly code or corresponding object modules, but for the Pluto compiler described in this dissertation the target is C[60]. There are a number of motivations for choosing C:

- C is often characterised as "portable assembly language", which means Pluto programs can be built and run on any platform that supports a C compiler. For example, the GNU C compiler suite is available on many platforms, and is modular enough to allow relatively easy porting to new target architectures. Having the Pluto compiler generate C code means that it is decoupled from the platforms supporting C compilers.

- There is a wealth of mature optimisation technology in many C compilers, obviating the need for the Pluto compiler to concern itself with common optimisations such as constant folding, function inlining and efficient register allocation. Additionally, future improvements in C compiler optimisation can be exploited without modifying the Pluto compiler at all: the generated C code need only be recompiled.

- The generation of C code rather than assembly code makes it easier to read the output of the Pluto compiler. While this is of most use to the compiler writer in checking for errors, readable code is also useful to explain how Pluto code is translated. Most C compilers can also output assembly code if required, enabling inspection of the quality of its output.

- Pluto generated code can interoperate with external C code, or object code that uses the same calling conventions. This is similar to the use of inline assembly code in C, or Java's native methods. Once C code has been generated, the original Pluto source could be discarded and the C code modified by hand. The Pluto-generated C code is sufficiently readable to allow easy editing. This ability is essential to the implementation of Pluto's standard library classes, as described in Section 3.8.

- Since the type system of C is more strict than raw assembly code, the C compiler can be used to help find errors in the Pluto code generator. Code generated by Pluto must itself compile cleanly with the C compiler. In a sense, this provides a very limited form of the typed intermediate languages used in functional language compilers[77, 110]. Unfortunately, the extent to which the C code can be considered a typed intermediate language is limited by the use of type casts: these bypass the normal type checking.

Other projects have used[24, 35, 88] or considered[85, 86] C as a code generation target. A typical observation is that there are a few areas in which C is potentially restrictive compared to using assembly or other low-level languages:

- Some languages, especially functional ones, rely heavily on *tail-call* optimisation, that C compilers may not handle well. A tail-call is a a function call that appears as the last operation in a function — a common occurrence in the functional programming style. Instead of adding a new stack frame for the function call, the existing frame can be reused instead. This prevents deeply recursive function calls from overflowing the program stack, and significantly improves run-time efficiency.

- Being tied to the C calling convention and stack frame layout can inhibit the implementation of language features such as nested functions, exceptions and precise garbage collection.

- A C compiler must sometimes make pessimistic assumptions in its analysis of the code being compiled, resulting in missed opportunities for optimisation. This is unfortunate for systems that generate C code, since they typically have higher-level knowledge about the properties and invariants of the generated code. The problem is that there is often no way of telling the C compiler about this knowledge, or controlling its behaviour.

For Pluto, the last point is the most significant. Using C did not cause any significant obstructions to code generation: the Pluto language constructs could be represented effectively, without resorting to special layouts or features not available in C.

An interesting possible alternative to C or assembly language is the `C--` portable assembly language[84, 85], which is designed to provide the portability of C without the limitations noted above. For Pluto as described here, the extra benefits of using `C--` would not outweigh the loss of optimisations gained from using gcc. This is because Pluto does not contain, or require support for, features that cannot easily be translated into

C. However, should Pluto be extended with exceptions or support for precise garbage collection, `C--` could well be a more attractive target for code generation.

3.4.1 Using C

A key aim of code generation is to produce C code that is amenable to optimisation by the C compiler. Some improvements in the C code produced by the Pluto code generator were made during development, based on examination of the assembly code output of the C compiler. For example, redundant temporary copies of objects were found in a number of situations. Elimination of these is particularly important for instances of value types, since the cost of unnecessary copying is magnified by the size of the value class instance.

Where possible, the generated C code uses the `const` keyword for the types of most local variables and for function parameters and return types. This is possible because the corresponding constructs in Pluto are immutable values: only those local variables declared with `var` in Pluto can be reassigned new values. The intended benefit of using `const` annotations in the generated code is to increase opportunities for optimisation by the C compiler. Naturally, the actual effect of `const` declarations is compiler-specific, but generating them requires little extra complexity in the Pluto code generation process.

The use of C `typedef` declarations and various features of the C preprocessor proved to be very useful for code generation, just as they are for manual C coding. Using type abbreviations cut the amount of text to be generated, made the code easier to read and reduced the likelihood of generating inconsistent code. Preprocessor macros were employed to simplify code generation for inherited class members, and used extensively in the type coercion operations described in Section 3.7.4.

An unfortunate limitation of the C syntax is that variable declarations must precede statements within a braced block of code. In Pluto code however, new variables may be introduced and initialised at any point within a code block. A simple solution used in the Pluto C code generation is to prefix each new statement or declaration with an opening brace, then finish the block with a matching number of closing braces. This makes for visually less appealing C code, but does not affect the meaning of the program or the quality of the generated code.

3.4.2 GNU C extensions

A number of GNU C extensions[101] are employed by the Pluto compiler to ease the translation process. The following constructs are used in the Pluto-generated code:

- Braced expressions, allowing a block of code to be used in situations expecting an expression. For example:

  ```
  newObj->fld1 = ({ int* temp=malloc(sizeof(int)); *temp=43; temp; });
  ```

  Naturally, code using this construct could be rewritten without it. However, the code generation process was simplified by the ability to use general code blocks in places that would otherwise require expressions only.

- Constructor expressions, that create new anonymous instances of a `struct` with initial values for its members. This construct is useful to avoid the creation of redundant temporary values that the C compiler may not be able to eliminate. An example use is the construction of a return value of a value class instance:

  ```
  return ((struct point) { 3.5, 4.0 });
  ```

  compared to the creation of a temporary:

  ```
  struct point temp = { 3.5, 4.0 };
  return temp;
  ```

  These constructions are also used in the type coercion macros (described in Section 3.7.4) since they make for more concise, readable code.

- The `inline` declaration for functions. This keyword indicates to gcc that it may replace calls to that function with the function's body. The Pluto code generator also declares inline functions as `static`, so that the function code will not be generated as a distinct function in the assembly code output. Section 3.7.5 describes the use of inline functions in Pluto code generation.

- Structures with no members: with gcc, empty structures can be declared and instantiated. This facility is used by Pluto for interfaces that contain no method signatures, and final value classes that have no instance fields.

One consequence of using these extensions is that the generated code must be compiled with gcc (or a C compiler that supports the same extensions). This trade-off is no great loss, as gcc is freely available and widely ported. While the extensions are currently gcc-specific, the new ISO C language specification[56] has incorporated inline functions and constructor expressions (known as *compound literals*) into the new standard. Support for these features by other C compilers would mean that code generated by Pluto is closer to being acceptable to C implementations other than gcc: only braced expressions and empty structures would need to be replaced.

The new ISO C specification also allows mixed statements and declarations within a code block, which would remove the need for generating extra braces as described above. Unfortunately, this new C standard is not currently supported by many C compilers, including gcc, so its new features are not used in the Pluto compiler's generated code.

3.5 Object model

This section gives an overview of the way Pluto constructs are represented at run-time, avoiding the details of the generated C code. It documents some of the important design decisions made, with reference to the design issues covered by the background material in Chapter 2.

3.5.1 Storage determination

A fundamental property of the Pluto object model is that the size of all immediate data — local variables, method parameters and fields — is known at compile-time. The fixed size

of these entities allows the generated code to determine their location without requiring run-time offset calculations. For example, the size and layout of a class instance is known at compile-time due to the fixed size of each of its instance fields. Thus, each field is located at a compile-time constant offset within an object. Likewise, the fixed size of local variables and method parameters means that the C compiler can determine the stack frame size and layout. Access to local variables does not require any offset calculations at run-time, since there is no variable-sized data on the stack. A consequence of requiring known-size immediate data is that subclasses of value classes cannot add new instance fields: this topic was highlighted in Section 2.3.4 and is discussed further in Section 4.5.

One benefit of known-size stack frames is that the C compiler does not need to generate code to maintain separate stack and frame pointers, although this is compiler- and target-specific. In the case of gcc, its `-fomit-frame-pointer` flag can be used, resulting in more compact code for functions. On some targets, including the Intel x86 family, the frame pointer register is then available for general purpose use.

3.5.2 Instance layout

Pluto objects follow the basic layout model described in Section 2.2.2: all the instance fields are arranged in an aggregate, headed by the vptr field which points to the object's vtable. Instances of reference classes are allocated on the heap, with a pointer to the object being the immediate value stored in variables and fields. Pointers always refer to the start of an object: there are no *interior* pointers, such as the pointer-to-member mechanism in C++, and no pointer arithmetic.

Since Pluto has single inheritance of instance fields, subclass objects are arranged using the simple technique of placing fields declared in the subclass immediately after the superclass fields. This property greatly simplifies the implementation of subsumption, as the same field offsets used for objects of a superclass are also valid for objects of any of its subclasses.

The Pluto code generator specifies the order of field layout as being that in which they were declared in the source. Besides the vptr link to type information, the compiler may add other fields to the instance structure for internal use. For example, the support of genericity (detailed in Chapter 5) relies on type descriptor pointers stored within generic objects. The precise run-time representation and alignment of fields in the instance structure is determined by the C compiler. At the level of Pluto source however, these decisions are not visible: this means the programmer is insulated from implementation-specific details.

3.5.3 Vtable/type descriptor layout

For each type (class or interface) declared in the Pluto source, a *vtable* structure layout is defined in the generated C code. This structure plays two main roles: it specifies the layout of function pointers used for the implementation of dynamic dispatch, and it acts as a run-time repository for information about that type. When discussing the latter role, the term *type descriptor* may also be used, emphasizing the use of a vtable instance to

uniquely identify a type. This topic is discussed further in Section 3.5.4.

Due to the dual purpose nature of the vtable in Pluto, there are two main kinds of information stored in it. The first is meta-information about the type, which occupies a fixed-sized portion at the beginning of the vtable. There are three fields of information about each type:

Immediate size indicates the amount of immediate storage required. Specifically, it contains the result of applying the C `sizeof` compile-time operator to the instance type. For abstract and reference classes this is the size of a pointer, but for interfaces and value types it will be the actual size of an object of that type.

Type kind gives more information about the kind of type: whether it is an `abstract` or `reference` class, a `value` class, a `final value` class or an interface.

Full value instance is used for `final value` classes, where it points to a second vtable containing information about the class in its full form. Details of this are given in Section 4.6. For other kinds of class, this field simply points back to the same vtable.

The main use of this information is supporting the implementation of genericity, which is covered in Section 5.7. In principle, other fields could easily be added in support of features such as run-time type tests, reflection and precise garbage collection. Such facilities are beyond the scope of this work, as they are not essential to the aims of the Pluto design. However, Section 8.3 suggests how the object model could be extended with further run-time type information.

The second kind of data in the vtable is the set of function pointers used to implement the dynamic dispatch mechanism. Since the number of entries varies with each type, they are placed after the meta-information. This ensures the meta-information can always be accessed using known offsets: for example, by code that must work with any type descriptor. The required offset to locate a function pointer in the vtable is known at compile-time, in the same way that instance fields can be found within an object.

Similar to the layout requirements of subclass instances described in the previous section, the layout of subclass vtables must be compatible with their superclass vtable. As Pluto has only single class inheritance, subclass vtable layout is simply a matter of appending new function pointers to the superclass layout. Section 3.6 details how multiple interface inheritance is supported in the object model.

3.5.4 Vtable/type descriptor instances

For each non-`abstract` class in Pluto, one instance of its vtable structure is created. It is this vtable instance which is pointed to by the vptr links in objects of that class.

For an `abstract` class there is also one instance of a vtable created, but it is not an instance of the vtable structure defined by the class. The reason for this is that no instances of an `abstract` class can exist, so no object will ever have a vptr field that points to this vtable. Furthermore, creating an instance of the vtable would require pointers to functions that may not be implemented by that class — because it is `abstract`. Consequently, the only use for the vtable instance of an `abstract` class is to serve as a type descriptor, storing the meta-information listed in the previous section. Therefore, the vtable instance

for an `abstract` class is an instance of a generic vtable structure that includes only the meta-information and no function pointer entries.

Since the data in the vtable instances (for all types) is constant, it can be stored in the read-only section of the generated object code. Also, because there is only one vtable instance for a given type, the address of the vtable can be used as a unique identifier for that type — hence the alternative name, type descriptor. This property is used in the implementation of interfaces, covered in Section 3.6.3, and is fundamental to the support of genericity: see Section 5.5.

3.5.5 Method kinds

Methods defined in Pluto are translated into C functions by the code generator. Since methods exist as members of classes or interfaces, some form of naming system must be used when they are placed in the global namespace of functions in C. Because the Pluto language does not allow overloading of methods, the compiler uses the straightforward technique of combining the method's name with the name of the type in which it is defined. Section 3.7.2 gives more details on the naming conventions used by the Pluto compiler.

The representation of methods in C code must take account of the different kinds of method that may appear in Pluto source. The simplest case to consider is `static` methods, as they are implemented directly as functions with the same parameter list. However, instance methods are invoked with respect to a particular object: the receiver. For these, the Pluto compiler uses the traditional approach of adding an extra parameter called `this` to the generated function, to pass the receiver.

Besides the compilation of methods themselves, support for dynamic dispatch must be provided by building the set of function pointers in the vtable. No entries are needed for `static` methods since, as their name indicates, they are never dynamically dispatched. In addition, the Pluto compiler omits vtable entries for instance methods that will never require dynamic dispatch — in C++ terminology these would be called non-virtual member functions. There are two criteria to be met in determining when an instance method never needs dynamic dispatch:

1. The method must be *final*, either by a direct `final` annotation in its definition or implicitly by appearing in a `final` class.

2. It must be an *original* method, in the sense that it does not override an inherited method of the same name.

In the example code of Figure 3.2, the two instance methods `==` and `$` in class `point` fulfil both criteria. They are defined in a `final` class, and so are themselves implicitly `final`. The class does not inherit any methods of those names, so they are also original methods. The `draw` method defined in class `Circle` does not fulfil either of these requirements, so the `Circle` vtable will have an entry for it.

The translation of an `abstract` method declaration simply involves adding an entry to the vtable layout, causing all of its subclass vtables to have the entry too. The vtable

structure for the `Shape` class in the example will have an entry for the `draw` method, as will the `Circle` subclass.

3.5.6 Method dispatch

With the vtable infrastructure in place, the translation of method calls can use one of two forms: static or dynamic dispatch. Static dispatch is implemented as a direct function call in C. Naturally, Pluto methods that do not have vtable entries cannot be dynamically dispatched: this includes `static` methods and original `final` methods. In addition to these methods, static dispatch can be used even for some calls to instance methods that have vtable entries. The Pluto compiler avoids generating code for dynamic dispatch in either of two cases:

1. If the static (compile-time) type of the receiver is a `final` class.
2. The signature of the method being called, based on the static type of the receiver, is a `final` method (either implicitly or directly).

A further use of statically dispatched method calls is for `super` calls, which enable code in some class to call a method that is overridden by that class. The actual method definition invoked is the one in the immediate superclass of the class in which the `super` call appears. The receiver of these calls is the `this` object, which means `super` can only be used in contexts where `this` is valid.

For method calls that do not match any of the above criteria, dynamic dispatch is used. This involves an indirect function call, using a function pointer selected from the receiving object's vtable. As noted in previous sections, both the offset of the vptr field within the receiver object, and the offset of the function pointer within the vtable are known at compile-time. This helps to mitigate the relative cost of dynamic dispatch compared to the direct function call used for static dispatch. From a programmer's point of view, liberal use of the `final` keyword can help to avoid unnecessary dynamic dispatch. This is strictly an optimisation, as it does not affect the semantics of the program.

3.5.7 The type of `this`

Within a Pluto source code method, the type of `this` is the same as the class in which the method definition appears. As methods can be overridden in subclasses, this means the type of `this` changes — it is implicitly covariant.

The translation of instance methods adds a `this` parameter to the generated C function, corresponding to the receiver of the method. Calls to these methods are translated into function calls (whether dynamically or statically dispatched) with the receiver object passed as an extra argument, to match the `this` parameter. For original `final` instance methods (that do not have entries in the vtable) the type of the `this` parameter in the generated C code is simply that of the class in which the method is defined. Related to this, Section 4.6.4 describes an optimisation used when the method is defined in a value class.

For those instance methods that do have entries in the vtable, the type of the `this` parameter in the generated C code is always `Object`: the topmost class in the inheritance hierarchy, and hence the ultimate superclass of all other classes. The reason for using this type is that it avoids function signature mismatches between superclass and subclass vtables. For example, a method defined in a subclass that overrides an inherited one would have a different type for `this` than the one in the superclass. Attempting to store a pointer to the subclass function in a vtable entry that has a different declared type for the function signature would elicit a warning from the C compiler. When value classes are involved, an error could be produced due to a mix of values and pointer types. Using `Object` as the type of `this` ensures no type mismatches in the generated C code.

When the type of `this` is `Object`, calls to these methods must coerce the receiver from its compile-time type to `Object`. In the case of `reference` classes, this is simply a cast between pointer types: an operation that has no run-time consequence. Section 4.6.4 discusses the implications for value types, while Section 3.7.4 covers the subject of type coercions in detail.

In generated code, uses of the `this` variable are translated into coercions from `Object` back to the expected compile-time type. Due to the Pluto language type system, checked by the compiler, these type conversions are guaranteed to be safe in the sense that a value is never given a type that overstates its capabilities.

3.5.8 Class constructors

For each Pluto class, a special *constructor* function is generated in the C code. In addition, a *new* function is created for non-`abstract` classes, which allocates memory for an instance of the class, sets the vptr field, then calls the constructor function to initialise the object, before returning it. A direct call to this function is made to implement the `new` operation in Pluto. The vptr entry is always set first, in case any of the instance field initialisation code uses constructs that need it, such as dynamic dispatch.

The constructor is much like the functions generated for an instance method, except that it is always statically dispatched and does not have an entry in the vtable. The function's parameters are those appearing in the header of its surrounding class definition, together with the `this` parameter which refers to the object under construction. The primary purpose of the constructor is to initialise the instance fields of the object. Constructor functions start with a call to the superclass's constructor, using the arguments specified in the class header. The remainder of the constructor initialises each instance field declared in that class by assigning values to the relevant entries of the object being built.

Section 4.6.5 details an optimisation that is made in the construction of some `final value` classes, to minimise the creation of temporaries by gcc.

3.6 Interfaces

While the previous sections have described how classes and objects are represented in the Pluto object model, this section details how interfaces are implemented. Because classes

can implement multiple interfaces, and an interface may be implemented by many different classes, the dispatch of methods through an interface variable is more complicated than for class-based method dispatch — a topic introduced in Section 2.2.6.

3.6.1 Interface vtables

For each interface defined in Pluto source, an interface vtable structure is specified in the generated C code, much the same as for classes. These vtables contain only function pointer entries: one per method signature that makes up the interface. There are no fields for meta-information, since interface instances (described in the next section) always refer to an underlying object that points to the appropriate meta-information. Therefore, the interface vtable structure is purely used to specify the layout of function pointers for dynamic dispatch.

Although interfaces can themselves extend multiple superinterfaces, there can only be at most one method signature with a given name in a vtable. As Section 6.4.2 explains further, the type system ensures that if more than one method with a given name is inherited by a class or interface then their signatures must match exactly.

Where an interface extends superinterfaces, its vtable starts with all the method signature entries from the first declared superinterface. Following those are any entries from subsequent superinterfaces whose method names do not already appear in the vtable. Finally the interface's own method signatures are added at the end, again except for duplicates. A consequence of this design is that an interface vtable is only layout-compatible with the vtable of its first superinterface.

In the code generated for a non-`abstract` class, an interface vtable instance is created for each interface that the class implements — these can be seen as a *view* of the class through a particular interface. Each entry in the vtable instance points to the generated C function that implements the method in that class. Since an interface vtable is layout-compatible with its first superinterface, further vtable instances need only be created for any subsequent superinterfaces. In practice, the number of distinct interfaces that a class implements is likely to be small[1]. Also, interface vtables are typically small — containing just one function pointer entry per method in the interface.

As explained in Section 3.5.4, each type has a vtable instance and this can be used as a type descriptor — its address uniquely identifies that type. Like `abstract` classes, the single vtable instance for an interface is not created as an instance of its own vtable structure; the generic vtable structure that contains only the fields of meta-information is used. This is because an interface provides no implementation for any methods, and so cannot provide any function addresses to store in a vtable instance.

3.6.2 Interface instance layout

A variable whose declared type is an interface is represented in the generated C code as a pair of pointer fields, as shown in Figure 3.5. One field points to a vtable instance that

[1]For example, the Java 2 SDK 1.3 from IBM contains 1700 classes in its API library. Only 9 of these implement more than 7 interfaces (either directly or indirectly), the most being 15. Over 87% of the classes implement at most 3 interfaces.

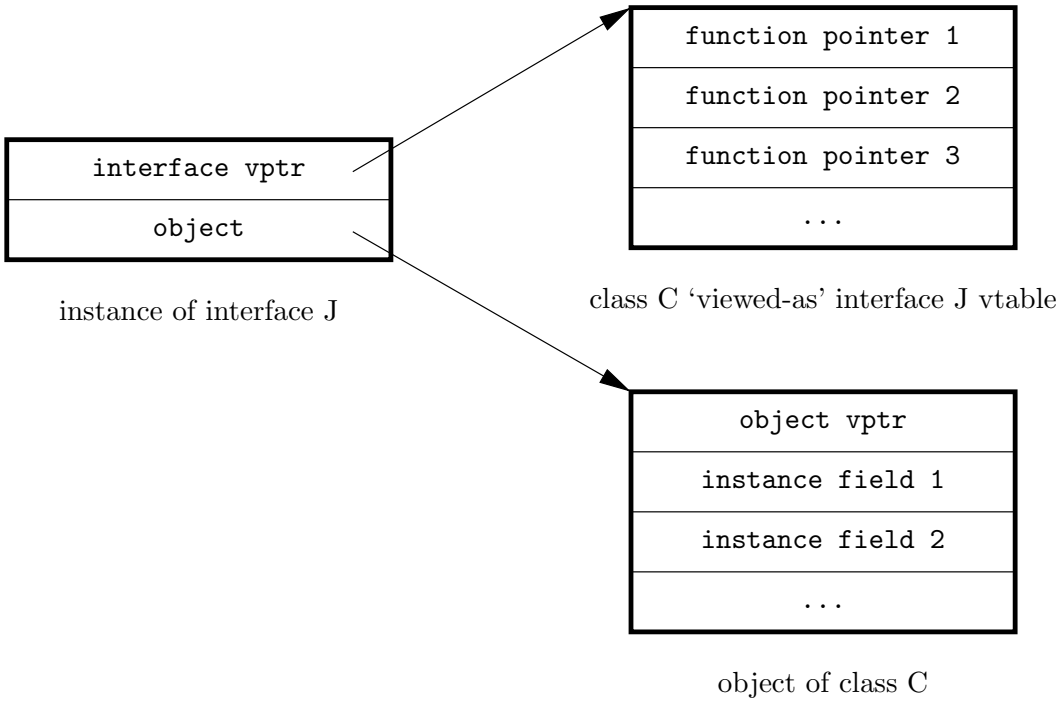

Figure 3.5: Interface instance layout

contains function pointers laid out in the order specified by the interface vtable structure. The second field points to the underlying object itself. The type system ensures that whatever the run-time class of the object, it will be one that implements that interface. When an object of class type is converted to interface type, a pointer to the object is paired with the address of the interface vtable instance defined by the object's run-time class. In the case of `reference` class objects, they are already manipulated via a pointer so no further indirection is necessary.

When a method call on a receiver of interface type is translated, dynamic dispatch is always used in the generated code. The technique is almost identical to that used for dynamic dispatch on a receiver of class type. The only difference is that the vtable pointer is immediately available as the first component in the interface instance, and the receiver object (to be passed as the argument to the `this` parameter) is the second component. Again, the offset of the function pointer in the vtable is still known at compile-time, just as it is for class-based dynamic dispatch.

3.6.3 Dynamic vtable lookup

The remaining part of the interface implementation is the technique used to actually find the address of the interface vtable instance given an object of some class. If the object's compile-time type is a `final` class, then the vtable address can be determined immediately by the compiler during code generation. Otherwise, a dynamic lookup call must be made at run-time that returns the correct vtable address for a given interface, based on the object's actual run-time class. In each class's vtable, the first function pointer entry is

reserved for the address of this interface lookup function.

A specific lookup function is generated for each class that implements interfaces, alongside the class-as-interface views described above. The function's address is stored in the vtable instance for that class, where it can be accessed at run-time to perform the lookup. The function has a single parameter that identifies the interface being looked up, and returns the address of a vtable instance for that interface. Calls to the lookup function use the address of an interface's type descriptor to specify the interface of interest. As highlighted in Section 3.5.4 the uniqueness of type descriptor addresses can be exploited by using them to identify a type. The body of the lookup function simply compares the parameter value against all the type descriptors for interfaces that the class implements, returning the corresponding vtable address as soon as a match is found. This linear search is deemed acceptable due to the small expected number of distinct interfaces a class would implement, but could easily be replaced with a more efficient algorithm if necessary. The Pluto type system ensures that a match will always be found: no lookup will fail at run-time. Should the end of the lookup function ever be reached, the program is aborted with a diagnostic message.

It is this lookup operation that can make interface method dispatch potentially slower than the dispatch used for class instances, although the cost is incurred only when an object is coerced from class to interface type. Once converted, each method call is no more complicated than via class types. Use of `final` classes reduces the overhead of working with interfaces, because the dynamic vtable lookup can be avoided.

3.7 Code generation

This section details how the C code generated by the Pluto compiler reflects the object model described in the previous sections.

3.7.1 Representation

At the top level, a Pluto program consists of a set of type definitions: classes and interfaces. For each type definition, the code generator produces three files:

1. A C header file containing pure declarations of types, structures, global data and preprocessor macros.
2. A second C header file containing definitions of `inline` functions, including class constructors and `new` functions. Section 3.7.5 discusses the use of inlining.
3. A C source file containing the actual definitions of non-`inline` functions and global data, including vtable instances.

Both header files use the common C preprocessor idiom of wrapping their contents inside an `#ifndef` check, to ensure that multiple inclusions of the file do not cause problems with duplicate declarations. For classes, the first header file has the following general form:

- Forward declarations of the type names for instances of this class. These need to be placed at the beginning so that they are available for use by other header files in their own declarations.

- Inclusion of first header files for all other types that are used in the definition of this type. This process operates transitively, so that all supporting declarations are available for use in the remainder of the header file.
- A forward declaration of the type used for the vptr field — a pointer to the vtable structure instance.
- Declaration of the instance structure for that class. This contains the vptr and all the instance fields, whether declared or inherited.
- The vtable structure declaration, containing meta-information fields and the set of function pointers. The first function pointer entry is for the interface lookup function. Each function pointer occupies a named member of the vtable structure: this contrasts with some discussions of vtables[50, 68, 104] which describe them as arrays of function pointers. Hence, Pluto's vtables are indexed by name rather than an integer index, with each member having a C declaration type that reflects the original method's signature.
- A declaration of the type descriptor instance, as a read-only global variable. The address of this instance is used as the value of the vptr field in objects of the class.
- A (possibly empty) list of declarations of interface vtable instances, for each interface that the class implements. Again, these are read-only global variables. Where possible, a single instance covers more than one interface: in this case aliases are defined, using the C preprocessor.
- Several definitions of coercion macros, which are used to convert objects between this class and other types. Section 3.7.4 gives details on these.
- Alias macros for members that are inherited unchanged in this class — this includes functions and global variables.
- Forward declarations of function prototypes, for both `inline` and ordinary functions. These represent methods that are actually defined in this class. Section 3.7.5 explains how the Pluto compiler decides which functions are declared `inline`.
- A set of macros and declarations for vtable entries, one per entry in the class's vtable layout. In most cases, the entry will be an alias for either a function declared above or a vtable entry in the superclass. However, this mechanism is also used to support the wrapper functions described in Section 6.5 for the implementation of variance.
- Forward declarations of function prototypes for three compiler-generated functions: the constructor, the `new` function and a static initialiser for the class (the role of this last function is discussed in Section 3.7.6).
- Declarations of external data (global variables), each of which represents a `static` field that was declared in the Pluto class. Since these variables are initialised at run-time, they are not declared read-only.

The format of the first header file for interface types is broadly similar, but lacks the items relating to function and data declarations. Note also that some extra declarations are present in the generated code for value classes — these are detailed in Section 4.6.

The second header file consists of `inline` function definitions only. Like the first header file, it begins by including the second header files for other types that it uses. This works

transitively, ensuring that all needed definitions are available for use when including this file. Besides the `inline` functions that represent methods defined in the original Pluto source program, one or two compiler-generated functions may be present. The constructor function simply initialises the instance fields of a new object, after calling the constructor for the superclass. The `new` function is present for non-`abstract` classes, and is responsible for creating a new object of the class. These functions are described in Section 3.5.8. For interface types, the second header file contains no function definitions: it simply contains the preprocessor directives for file inclusion.

The generated C source file provides the definitions of data and functions that were declared in the first header file. For classes, the source file has the following general form:

- Inclusion of the first and second header files for this class.
- A list of interface vtable instances, corresponding to the interfaces that this class implements (if any).
- An interface lookup function definition that, given an interface type descriptor tag, returns the address of one of the above vtable instances. This function is declared `static` in C, since it is only referenced by the type descriptor initialisation.
- The definition of the type descriptor data itself, for this class.
- Definitions of external data, used to represent Pluto `static` fields.
- A *static initialiser* function definition, and a flag variable used to indicate whether the initialiser has been executed. The purpose of this function is to initialise the external variables defined above, and to call the initialiser functions for all other types that are used by this one. Section 3.7.6 explains some of the issues relating to field initialisation.
- Definitions of ordinary (non-`inline`) functions, that represent methods defined in this Pluto class.
- If this is the `Main` bootstrap class (described below, in Section 3.7.8) a `main` function is generated.

For interface types, the source file contains just the definition of the type descriptor for that interface — no other data or functions are defined.

Ordering of declarations is necessary to ensure the correct dependencies in a language such as C that does not allow implicit forward references. In addition, the use of two header files per type enables the source of `inline` function definitions to be available to gcc when it is compiling code that calls those functions — a condition that must be met if gcc is to inline function calls.

Each C source file can be compiled by itself to produce an object file, or all the source files for a program can be compiled and linked in one step. Although the C compiler is invoked on just one or more C source files, the inclusion of supporting declarations can result in many different header files being combined during preprocessing. The careful ordering of declarations ensures that, after the inclusion of header files, the C compiler sees one stream of code in the following order:

pto_hd1_	First header filename
pto_hd2_	Second header filename
pto_src_	Source filename
PTO_DEF_	Preprocessor flag — has first header been included already?
PTO_IDEF_	Preprocessor flag — has second header been included already?
pto_ins_	Structure tag name for instance of a class or interface
pto_fvi_	Structure tag name for full instance of a final value class
pto_hvi_	Structure tag name for headless instance of a non-final value class
pto_hvt_	Type name for pointer to headless value instance
pto_typ_	Type name for an instance of a class or interface
pto_cty_	Type name for a constant instance of a class or interface
pto_vit_	Type name for a full instance of a final value class
pto_cls_	Structure tag name for a class vtable
pto_ifc_	Structure tag name for an interface vtable
pto_tti_	Type name for pointer to vtable instance
pto_fnp_	Member name for a function pointer in a vtable structure
pto_vte_	Preprocessor macro name for a vtable entry
pto_inf_	Global variable name for a vtable instance — a type descriptor
pto_fif_	Global variable name for a vtable instance used for full values
pto_fld_	Member name for a field in class instance structure
pto_ivu_	Global variable name for a class-as-interface vtable instance
pto_ilu_	Function name for interface lookup function
pto_cvt_	Preprocessor macro name for type coercion operation
pto_fun_	Function name for a translated method
pto_gbl_	Global variable name for translated static field
pto_ini_	Function name for static initialiser
pto_var_	Local variable or function parameter name

Figure 3.6: Pluto naming scheme prefixes

1. Forward declarations of all type names.
2. Remaining declarations.
3. Inline function definitions.
4. Remaining function and data definitions.

3.7.2 Naming system

To avoid potential clashes with reserved identifiers in C, and to encode the correct scoping rules for Pluto programs, the Pluto compiler uses a system of naming for identifiers in the generated code. The prefix `pto_` is used for nearly all names, providing a namespace for Pluto code that does not conflict with any C keywords or library-defined names. Following this prefix is another mnemonic that describes the identifier's purpose: for example `pto_typ_Point` is a type name in C, referring to a type called `Point` in the original Pluto source code. Figure 3.6 lists the naming prefixes used throughout the generated code. Some of these are specific to the implementation of value types, genericity and variance; they are discussed in the relevant chapters.

Some C names are constructed by combining two names from the Pluto source code. As briefly mentioned in Section 3.5.5, methods are translated into C functions that com-

bine the method's name with the name of the class it was defined in, separated with an underscore. Since a class can only have at most one definition of a method with a given name, the combination of class and method name prevents clashes. Similarly for Pluto's `static` fields, that are translated into global variables whose name is a combination of the class name and the field name.

During the Pluto front end desugaring, covered in Section 3.3.2, symbolic names are converted to an alphabetic form prefixed with an underscore. A leading underscore is also used for synthesised names made during the Pluto code generation: e.g. for temporary variables. The desugaring process creates uniquely-named temporaries using an ever-increasing counter. This results in temporary variables with names such as `pto_var__tmp3`.

3.7.3 Example generated code

To best illustrate the general format of the C code produced by the Pluto code generator, an example is included here. Given this simple Pluto class definition:

```
class Point(x:int, y:int) {
  field x : int = x;
  field y : int = y;
  method ==(other:Point) : boolean {
    return this.x == other.x && this.y == other.y;
  }
}
```

the Pluto compiler produces three files of C code, listed below. For presentation purposes, the divisions between files are shown with a line. Note that the generated comment at the beginning of the first file shows the signature of the class and its members, as used in the core syntax tree.

```
/* Header file 'pto_hd1_Point.h' – generated code */
/*

class Point[](x : int[], y : int[]) extends Object[] implements []
 field x : int[]
 field y : int[]
 original_dynamic method _equalsequals[](other : Point[]) : boolean[]

*/
#ifndef PTO_DEF_Point
#define PTO_DEF_Point

#include "pluto.h"

/* Typedefs for this class */
typedef struct pto_ins_Point* pto_typ_Point;
typedef struct pto_ins_Point*const pto_cty_Point;

/* Other types used by this one */
```

```
#include "pto_hd1_int.h"
#include "pto_hd1_Object.h"

/* Class info type */
typedef const struct pto_cls_Point* pto_tti_Point;

/* Class instance (object) structure */
struct pto_ins_Point {
  pto_tti_Point vptr;
  pto_typ_int pto_fld_x;
  pto_typ_int pto_fld_y;
};

/* Class info structure/vtable */
struct pto_cls_Point {
  const unsigned short imm_size;
  const short type_kind;
  pto_tti_Point full_info;
  itable (*lookup_itable)(pto_cty__gentype itable_tag);
  pto_cty_int (*pto_fnp__equalsequals)(pto_cty_Object pto_var_this, pto_cty_Point
  pto_var_other);
};

/* Declaration of type info instance(s) */
extern const struct pto_cls_Point pto_inf_Point;

/* Declaration of interface views */

/* Coercion macros to and from other types */
#define pto_cvt_Point_fromThis ((pto_cty_Point)pto_var_this)
#define pto_cvt_Point_toReference(CLASS,VALUE) ((pto_cty_ ## CLASS)VALUE)
#define pto_cvt_Point_toAbstract(CLASS,VALUE) ((pto_cty_ ## CLASS)VALUE)
#define pto_cvt_Point_toAbstractHeap(CLASS,VALUE) ((pto_cty_ ## CLASS)VALUE)

#define pto_cvt_Point_toInvGeneric(VALUE) ((pto_cty__genvalue)VALUE)
#define pto_cvt_Point_toVarGeneric(VALUE) ((pto_cty__genvalue)VALUE)
#define pto_cvt_Point_toInvGenericHeap(VALUE) ((pto_cty__genvalue)VALUE)
#define pto_cvt_Point_toVarGenericHeap(VALUE) ((pto_cty__genvalue)VALUE)
#define pto_cvt_Point_fromInvGeneric(VALUE) ((pto_cty_Point)VALUE)
#define pto_cvt_Point_fromVarGeneric(VALUE) ((pto_cty_Point)VALUE)
#define pto_cvt_Point_dynFromInvGeneric(VALUE,TYPEARG) ((pto_cty_Point)VALUE)
#define pto_cvt_Point_dynFromVarGeneric(VALUE,TYPEARG)
pto_cvt_Point_fromVarGeneric(VALUE)
#define pto_cvt_Point_toIfc(INTERFACE,OBJ) ((pto_cty_ ## INTERFACE) { (pto_tti_ ##
INTERFACE)OBJ->vptr->lookup_itable((pto_cty__gentype)&pto_inf_ ## INTERFACE),
OBJ })
#define pto_cvt_Point_toInterface(INTERFACE,VALUE)
pto_cvt_Point_toIfc(INTERFACE,pto_cvt_Point_toAbstract(Object,VALUE))
#define pto_cvt_Point_toInterfaceHeap(INTERFACE,VALUE)
pto_cvt_Point_toIfc(INTERFACE,pto_cvt_Point_toAbstractHeap(Object,VALUE))

/* Aliases for functions + globals */
```

```
/* Function prototypes */
extern pto_cty_int pto_fun_Point__equalsequals(pto_cty_Object pto_var_this, pto_cty_Point
pto_var_other);

/* Inline function prototypes */

/* Vtable entries */
#define pto_vte_Point__equalsequals pto_fun_Point__equalsequals

/* Static initialiser function prototype */
extern void pto_ini_Point(void);

/* Constructor function prototype */
static inline void pto_fun_Point__con(pto_cty_Object pto_var_this, pto_cty_int pto_var_x,
pto_cty_int pto_var_y);

/* New instance function prototype */
static inline pto_cty_Point pto_fun_Point__new(pto_cty_int pto_var_x, pto_cty_int pto_var_y);

/* Global variable declarations */

#endif
```

```
/* Inline header file 'pto_hd2_Point.h' - generated code */
#ifndef PTO_IDEF_Point
#define PTO_IDEF_Point

/* Other inline code used by this type */
#include "pto_hd2_int.h"
#include "pto_hd2_Object.h"

/* The class constructor function */
static inline void pto_fun_Point__con(pto_cty_Object pto_var_this, pto_cty_int pto_var_x,
pto_cty_int pto_var_y) {
  { }
  pto_cvt_Point_fromThis->pto_fld_x = pto_var_x;
  pto_cvt_Point_fromThis->pto_fld_y = pto_var_y;
}

/* New instance creation function */
static inline pto_cty_Point pto_fun_Point__new(pto_cty_int pto_var_x, pto_cty_int pto_var_y) {
  pto_cty_Point nu = malloc(sizeof(struct pto_ins_Point));

  nu->vptr = &pto_inf_Point;
  pto_fun_Point__con((pto_cty_Object)nu, pto_var_x, pto_var_y);
  return nu;
}

/* Inline function definitions */
```

```
#endif
```

```
/* Source file 'pto_src_Point.c' – generated code */
#include "pto_hd1_Point.h"
#include "pto_hd2_Point.h"

#define pto_ilu_Point default_lookup_itable

/* Definition of type info instance(s) */
const struct pto_cls_Point pto_inf_Point = {
  sizeof(pto_cty_Point),
  TYPEKIND_ABSTRACT_OR_REF,
  &pto_inf_Point,
  pto_ilu_Point,
  pto_vte_Point__equalsequals
};

/* Global variable definitions */

/* Static initialiser flag and function */
static int need_init = 1;
void pto_ini_Point(void) {
  if (need_init) {
    need_init = 0;
    pto_ini_int();
    pto_ini_Object();
  }
}

/* Function definitions */
pto_cty_int pto_fun_Point__equalsequals(pto_cty_Object pto_var_this, pto_cty_Point
pto_var_other) {
  { pto_cty_int pto_var__tmp1 = pto_cvt_Point_fromThis->pto_fld_x;
  { pto_cty_int pto_var__tmp2 = pto_var_other->pto_fld_x;
  { pto_cty_int pto_var__tmp3 = pto_fun_int__equalsequals(pto_var__tmp1, pto_var__tmp2);
  { pto_typ_int pto_var__tmp7;
  { if (pto_var__tmp3)
    {
      { pto_cty_int pto_var__tmp4 = pto_cvt_Point_fromThis->pto_fld_y;
      { pto_cty_int pto_var__tmp5 = pto_var_other->pto_fld_y;
      { pto_cty_int pto_var__tmp6 = pto_fun_int__equalsequals(pto_var__tmp4,
      pto_var__tmp5);
      { pto_var__tmp7 = pto_var__tmp6; }}}}
    } else {
      pto_var__tmp7 = false;
    }
  { return pto_var__tmp7; }}}}}}
}
```

3.7.4 Type coercions

Like most object-oriented languages, Pluto is polymorphic: objects can have several types. However, the Pluto compiler's target language is C, which is essentially monomorphic in that values have just one type. Therefore, the code generation phase inserts type coercions as necessary to convert between C types. For example, all Pluto classes have the `Object` class as their ultimate superclass. This means that an object of any class can be used in a context where an `Object` is expected:

```
let ob:Object = new Point(3,4);
```

In the Pluto source, this polymorphism is implicit: no special action is required to assign a `Point` object to an `Object` variable. In the generated code, an explicit coercion is required because the C type system does not consider these two types as compatible. In the case of `abstract` and `reference` classes in Pluto, they are manipulated indirectly through a pointer. So, assuming the `Point` class is reference-based, the coercion inserted for the code example above would be a simple C type cast between the two pointer types. This operation is safe due to the compatible layout of superclass and subclass objects, discussed in Section 3.5.2 regarding the object model.

The first header file of the generated C code for a Pluto type contains several preprocessor macro definitions (all prefixed with `pto_cvt_` and the type name) that specify the coercions for that type. The remaining portion of the macro name identifies the kind of coercion, e.g. `toAbstract` or `toInterface`. In most cases, the macro has parameters for the object being coerced and the name of the target type. An exception is the first macro `fromThis`, which is used to convert the `this` variable to the correct type in situations where it is referenced by method body code. As Section 3.5.7 explained, the type of `this` is standardised to `Object` for methods in the vtable. In the generated code listed above for the `Point` class example, the constructor function and the `pto_fun_Point__equalsequals` function definition show how this macro is used.

For reference-based classes, the coercions to their superclasses are simple type casts. The preprocessor concatenation operator `##` is used to build the C type name for the cast, allowing these macros to be used for any superclass instead of requiring one for each. Taking the example assignment shown above, the generated C code would be:

```
pto_cty_Object pto_var_ob =
    pto_cvt_Point_toAbstract(Object, pto_fun_Point__new(3,4));
```

After the preprocessor has expanded macros, the code seen by the C compiler is:

```
pto_cty_Object pto_var_ob = ((pto_cty_Object)pto_fun_Point__new(3,4));
```

The coercion macros for converting a class to an interface are more complicated than just a type cast: a new interface instance must be created. As Section 3.6.2 details, an interface instance consists of a pair of pointers, one to an interface vtable instance and the other to the underlying object. In the generated code for the `Point` example, listed in the previous section, the coercion to an interface consists of a C constructor expression that creates a new anonymous instance of the interface instance. The constructor provides

the two required values: the vtable pointer obtained with a dynamic lookup call, and the object itself. Since the `Point` class is reference-based, a simple cast is used to convert the underlying object to the expected type, `Object`. A dynamic call is used to find the vtable pointer, because `Point` is not a `final` class — the object may be an instance of a subclass.

Other type coercions are specific to the topics of value types and genericity: these are discussed in Sections 4.6.3 and 5.4 respectively. Some of the generated coercions have a normal definition and a `Heap` variation: these are used in the implementation of genericity with value types, to prevent *dangling pointers* (a topic covered in Section 5.8).

3.7.5 Function inlining

Programs written in object-oriented languages often contain many methods that are small, in the sense of having little code in their body. Typical examples are simple *get* methods that return an object's field, or methods that contain just a forwarding call to another method. This situation potentially introduces an abstraction cost due to the overhead of a function call. An optimisation provided by gcc is the ability to replace a call to a small function with the function's body — the call is *inlined*. To achieve this, the function must be marked with the `inline` keyword, and its source definition must be available at the point of the call. The nature of inlining means that only statically-dispatched function calls can be inlined (Section 3.5.6 discussed the dispatch mechanisms).

As stated in Section 3.7.1, the Pluto compiler declares the generated constructor function and the new-instance function with the `inline` keyword. In addition, some of the methods defined in the Pluto source are translated into inline functions. The Pluto compiler uses a simple scoring system for methods, that gives a rating for their complexity. The score is an approximate measure of the amount of code to be expected in the generated function. If the score is below a certain threshold (chosen by trial-and-error) then the generated function is marked as inline.

When the Pluto compiler generates an inline function, it also marks the function as `static`. The effect of this is to allow gcc to omit the generation of separate code for the function itself in cases where all calls to the function were inlined[101]. Functions whose address is taken to be stored in a vtable instance cannot be omitted, so they appear in the object files produced by gcc. However, this does not prevent statically-dispatched calls to these functions from being inlined where possible.

Naturally, the effectiveness of function inlining is highly dependent on the C compiler used — gcc in the case of the Pluto generated code. The gcc documentation[101] states that inlining is considered an optimisation: the `inline` annotation on functions is a hint and may be ignored. Also, the decision on which functions to inline is guided by gcc's own measure of the function's complexity. In practice, examination of the generated assembly code shows that gcc makes good use of inlining when compiling the C code produced by the Pluto code generator. For example, simple methods that retrieve or set an instance field are inlined, as are calls to the constructor for many final value classes (see also Section 4.6.5).

3.7.6 Initialisation of static fields

In Pluto source, a static field can be initialised with the result of an arbitrary expression. In the generated C code, these fields are represented by global variables, which can only be initialised with a C constant expression (i.e. one whose value is known to the C compiler). Because of this restriction, Pluto-generated global variables are initialised at run-time by the static initialisation function, first mentioned in Section 3.7.1. Each translated Pluto class defines an initialisation function, and a static flag variable that tells whether the function has already been called. The function first calls the initialisation functions of other types used by the class, then initialises all the global variables that represent static fields defined in that class.

Among the C code for a complete translated Pluto program, there will be a single definition of a `main` function: the entry point for a compiled C program (Section 3.7.8 explains how Pluto programs are started). One of the tasks of the `main` function is to call the static initialiser function for the class that defines the Pluto program entry point. This function in turn calls the initialisers for other classes it uses, and so on, ensuring that all used classes are initialised before the actual program proper is run. The flag variables are used as guards to ensure that mutually-dependent classes do not cause infinitely recursive calls between their initialisation functions.

This initialisation scheme performs a similar role to that required by C++, which also allows global variables to be initialised with arbitrary expressions. Techniques for C++ implementation are discussed in [68, 104].

3.7.7 Separate compilation

As stated in Section 3.7.1, the C source files generated by the Pluto compiler can be compiled individually. However, the Pluto compiler does not currently support separate compilation: the definitions of all referenced types must be presented to the compiler (apart from the standard library classes listed in Section 3.8.1). One benefit of this limitation is that the generated code is consistent among itself: subclass and superclass layouts are guaranteed to be compatible. The issue of separate compilation brings the possibility of incompatible changes between the code generated for several types. In the Java language specification[49] for example, an entire chapter is devoted to the topic of binary compatibility. It explains the changes that can be made to a type definition that still preserve compatibility with other types that depend on it.

The description of Pluto's object model (Section 3.5) explains that the layout of the class instance and vtable structures is required to be compatible with those of the superclass. If the Pluto compiler allowed separate compilation, changes to a superclass such as adding new instance fields or methods would mean that the existing generated C code for all its subclasses (which are necessarily dependent on their superclass) would then be incompatible. For simplicity, the Pluto compiler ignores this problem by disallowing separate compilation.

The Pluto compiler could be enhanced to allow separate compilation, by reading type signatures from any C header files present in the working directory, and from other specified

locations. In the example generated code listed above, a comment in the first header file contains the internal type signature for that type definition. This was generated as a first step towards allowing separate compilation: subsequent work to the Pluto compiler would involve reading and parsing these signatures into the internal representation used by the type checking phase of the compiler.

Extending the compiler for separate compilation, although interesting, was not done as it is considered non-essential to the work described in this dissertation. Nevertheless, the design of the Pluto language and the compiler do not preclude such additions.

3.7.8 Program entry point

Although a Pluto program may consist of many type definitions, there is exactly one entry point for the program. The convention used in the Pluto compiler is that the entry point is a static method called `run` in a class called `Main`. This method must have no parameters, and must have a return type of `int`. Since there can only be one class called `Main` in the C code that comprises the program at link-time, and methods cannot be overloaded, this distinguished method defines a unique entry point. If there is more than one `Main` class in the object files being linked, the linker will give an error due to the duplicate function and data names being defined. If there is no `Main` class, then the linker will be unable to create an executable because of an unresolved reference to the missing `main` function.

The Pluto type checker ensures that if a class called `Main` is defined, then it must have a static `run` method declared as above. The Pluto compiler generates a `main` function in the generated C code for the `Main` class. This function, which is the entry point for normal C programs compiled to executables, first calls the static initialisation function (described in Section 3.7.6) then calls the function that represents the translated `Main.run` method. The integer result of that call is used as the value returned from the `main` function — often called the exit code.

A possible extension to this scheme would be to allow command-line arguments to be passed to a Pluto program, by having the `Main.run` method take an array of `String` values as parameters. This would provide a similar facility to the `argv/argc` convention used in C and C++, and the String array parameters used in Java's `main` startup method. As this was not considered essential to the work presented in this dissertation, the Pluto compiler does not currently support passing of command-line arguments.

3.7.9 Run-time error handling

There are certain error conditions that the Pluto compiler cannot detect, and hence some programs can fail at run-time. Examples are integer division-by-zero, out-of-bounds array indexing and dynamic memory allocation failure (also known as heap exhaustion). By default, the code generated by the Pluto compiler does not check for these conditions: being C code, there are no implicit run-time checks. Optionally, a limited number of run-time checks can be activated by setting the `PTO_RT_CHECK` preprocessor flag (using the gcc -D switch, for example) before compiling the generated C code. This inserts run-time checks for the three conditions listed above. In the event of failure, an error message is

printed to the console and the program is terminated. A natural extension to the Pluto language would be support for exceptions: these are not provided in the language as described in this work, as they are not of central concern.

A related problem is the possibility of a Pluto field (instance or static) being accessed before it is initialised with a value. This can occur during object or class construction, if the expression used to initialise a field makes reference to as-yet uninitialised fields. The code generated by the Pluto compiler performs field initialisation in the order of declaration within a class. Therefore, any field initialisation expression that refers to fields that are defined later within the class will access invalid values. Currently, the Pluto compiler ignores this problem as it is not of central concern in this work. The Java language[49] specifies that fields are given default values before any initialisation expressions are evaluated. However, Java classes are all reference-based and so can use a `null` reference as a default value. Unfortunately, it is not always clear what default value should be used for `value` classes in Pluto, which are not reference-based.

3.8 Pluto library

As shown in the compiler overview diagram on page 36, the Pluto-generated C code is compiled and linked with the Pluto library. This takes the traditional C form of header files to declare the types, structures and functions provided by the library, and a static library archive containing the object files that provide the library functions and global data. The small size of the library (only 9 Kilobytes) means that a dynamically-linked library is unnecessary.

The main purpose of the library is to support the predefined classes, which are detailed below. In addition, all Pluto-generated programs include the `pluto.h` header file, which gives some general declarations of types and structures used in the implementation of interfaces and genericity. The library also defines some functions used for copying objects: these are discussed in Section 5.7.3, concerning the implementation of genericity.

3.8.1 Predefined classes

Besides those classes and interfaces that a Pluto program defines itself, a small number of classes are available as part of the support library. There are 7 classes in the library:

Object This is an `abstract` class that is the ultimate superclass of all other classes, and is the default superclass and type parameter bound. This class has no fields, defines no methods and has a do-nothing constructor.

int, float, boolean, String These classes are fundamental to the language, as they represent the types of literal values that can be written directly with the syntax. All of these are declared as `final value` classes and extend Object directly. They are implemented using appropriate C data types: for example, `pto_typ_int` is just a synonym for `int`. For Strings, the value is a pointer to the actual characters. Each class declares a handful of methods for common operations such as addition, comparison etc. The String class contains methods for output to the console, providing a simple printing mechanism.

Array, Mutable Both of these are declared as `final reference` classes that extend Object directly, and both are generic: they have a single type parameter to represent the type of data they contain. The Array class contains a fixed number of elements, set when it is constructed, and has operations for storing and retrieving elements by index, and a `size` method. Elements are stored directly within the Array: for example, in an Array of `value` class objects they are all packed together. The Mutable class is constructed with a single value of the supplied type, and provides the ability to update and retrieve the contained value — in essence, it is a one-element Array. Since in Pluto a value object's fields are immutable, this class can be used to provide mutable state in those situations that require it.

These provide a minimal foundation on which to build other types, by represented concepts that are either fundamental to the language design (Object, int, float, boolean and String) or support facilities that could not easily be expressed in the language itself (Array and Mutable). Although these types are predefined, they are still fully part of the class hierarchy and have no greater or lesser status than user-written types.

The Pluto compiler is aware of the type signatures for these predefined classes, which are used by the type annotation and checking phases. If the compiler was extended to support separate compilation, the signatures could even be read in from the header files that implement each class. This would eliminate the need to have the signatures coded into the compiler itself.

Having provided the type signatures for the predefined classes in the Pluto compiler, their actual implementation must be available to the C compiler and linker, so that complete programs can be built. One simplification made during the lowering process (see the compiler diagram on page 36) is that the `boolean` type is systematically replaced with the `int` type, and Boolean values are replaced by equivalent integer values. This conversion is done to reduce the number of separate classes that must be written manually in the library implementation source code. However, if the Pluto language was extended to support reflective capabilities, such a conversion would become visible and hence incorrect.

Each predefined class (apart from `boolean`) is implemented by two C header files and a C source file, using the same format as those generated for user-written types — detailed in Section 3.7.1. These files use the same naming conventions and contain the same pattern of declarations and definitions of types, structures, data, functions and preprocessor macros as do the Pluto-generated ones. Since the source code is hand-written, it can contain C code and C library calls that do not appear in the code generated by the Pluto compiler. For example, the implementation of the addition operation for integers is not a function definition but a preprocessor macro with the expected name. Pluto source such as:

```
let result:int = 3 + 4;
```

is converted into the following C code:

```
pto_cty_int pto_var_result = pto_fun_int__plus(3, 4);
```

where the definition of `pto_fun_int__plus` in the library source code is a macro that expands to a C addition operator between the two macro operands. So the expanded code seen by the C compiler is just:

```
pto_cty_int pto_var_result = 3 + 4;
```

Having types such as `int` appear as classes like any other allows for consistent and uniform code generation. Even so, the hand-implemented library source enables basic operations such as integer addition without any overhead compared to the equivalent C code.

In most cases, the implementation of methods in predefined classes is done using macros, because they are simple operations. This means that the first header files contain nearly all the implementation: the second header files contain no code, and the source files contain just the definition of the vtable instances (type descriptors) for each class. The small size of the Pluto library is due to the fact it only contains the object files resulting from compiling these source files.

3.9 Summary

This chapter introduces the Pluto language, describes the object model used to implement the language and details how the compiler written as part of this work implements the object model. The code generation details given in this chapter show how the compiler translates Pluto programs into C code.

Issues relating to the design and implementation of value types, genericity and variance are not discussed in this chapter, as they are examined in Chapters 4, 5 and 6 respectively.

Chapter 4

First addition: value types

This chapter is concerned with the issues relating to value types, how they are supported in the design of the Pluto language and how they are implemented by the Pluto compiler. The details described here build upon the implementation infrastructure covered in Chapter 3.

4.1 Distinguishing value-based types

In the review of background material, Section 2.3 explains the distinction between value-based and reference-based types, and shows how various object-oriented languages support value types. In Pluto, all types are described by class and interface definitions. Value types can be defined by annotating a class definition with the `value` keyword. Instances of these value classes are handled directly, rather than through a pointer indirection.

The Pluto code in Figure 4.1 defines two `Point` classes that are identical except that `RefPoint` is a reference class and `ValPoint` is a value class. The diagram in Figure 4.2 shows a typical memory layout for the objects created in the `Main.run` method. The different handling of value class objects as opposed to reference class objects is clear: the latter are allocated as separate objects on the heap, and are handled via pointers. The value class objects are allocated directly in the stack frame for that method — no heap space is allocated, and the objects are automatically and implicitly destroyed when execution of the method terminates. Copying these objects shows another distinction, as reference class instances can be aliased: variables `rp` and `rpcopy` both contain a pointer to the same object. In contrast, variables `vp` and `vpcopy` represent distinct objects with no indirection.

Value class objects need not exist only on the stack: a reference class may contain an instance field whose type is a value class. An example is shown in Figure 4.4 on page 70, where an instance of such a reference class is allocated on the heap and contains an embedded sub-object that is an instance of a value class. This shows that the contained instance field is an integral part of its parent object, rather than a distinct heap-allocated object in its own right.

The key issues are those of storage allocation and lifetime. Declaring a variable whose

```
class RefPoint(x:int, y:int) {
  field x : int = x;
  field y : int = y;
}

value class ValPoint(x:int, y:int) {
  field x : int = x;
  field y : int = y;
}

abstract class Main() {
  static method run() : int {
    let rp : RefPoint = new RefPoint(3, 4);
    let vp : ValPoint = new ValPoint(7, 8);
    let rpcopy : RefPoint = rp;
    let vpcopy : ValPoint = vp;
    return 0;
  }
}
```

Figure 4.1: Pluto program using reference and value classes

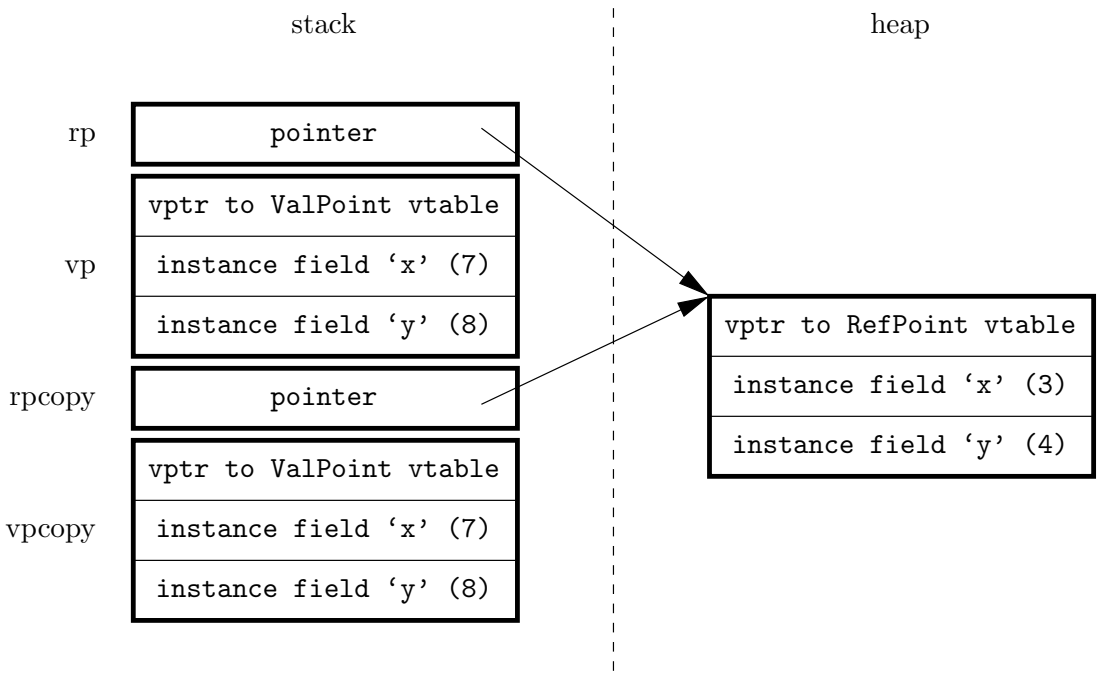

Figure 4.2: Memory layout for reference and value objects

type is a value class will result in the compiler arranging to reserve enough space for an instance of that class. When the variable is out of scope, the reserved storage (and hence the object stored there) is released and the object no longer exists. Conversely, variables whose declared type is reference-based (i.e. an abstract or reference class) only require the allocation of space for a pointer. The actual object whose address is stored in the variable has a lifetime that is not dictated by the pointer variable's lifetime. As Section 2.2.1 discussed, the flexibility of heap-allocated storage involves more complicated memory management, e.g. garbage collection or explicit deallocation of objects.

4.1.1 The benefits of value types

The main motivation for using value classes is one of potential efficiency gains. If all types were reference-based, simple operations such as arithmetic would suffer unnecessary space and time overhead from the extra indirection. As explained in Section 2.6.2, the *fully-boxed* approach of storing all objects on a heap and manipulating them via pointers is costly.

Several schemes using value-based objects[23, 41, 44, 70] highlight the preference for stack allocation of objects rather than use of the heap, where possible. There are a number of reasons for attempting to avoid heap allocation:

- Access to objects and their fields involves *pointer-chasing* — the dereferencing of indirections. Value class objects have no such indirection, and are accessible without following a pointer.
- The amount of heap space actually allocated may be more than that requested, in order to support the implementation of the storage allocator. For example, `malloc` may add extra book-keeping information to the allocated storage block[1]. When allocating large numbers of small objects, this per-block overhead could be a significant factor in a program's memory consumption.
- The management of a heap is likely to require more complex run-time expense than for a stack, due to the unpredictable lifetime of heap-allocated objects. In contrast, stack allocation and deallocation requires only the movement of a stack pointer. The lifetime of stack-allocated data is easily determined, enabling the compiler to arrange stack management automatically — no explicit deallocation or garbage collection is needed.
- Although implementation-specific, possible fragmentation of object storage on the heap can be detrimental to the use of a memory cache. The predictable pattern of stack storage gives better locality of reference, which is important for effective caching.
- Boxing of small objects on the heap can result in a doubling (or worse) of memory requirements. For example, a stack-based integer value typically consumes just one machine word. A heap-based object would require a pointer value on the stack (also likely to be a machine word) together with heap storage for an integer, and perhaps a vptr field. Again, this extra cost is magnified for large numbers of small objects.

[1] The implementation of malloc in the GNU C library[71] on a 32-bit system imposes a 4 byte overhead for each allocation. In addition, the minimum size of actual allocations is 16 bytes.

Even where heap allocation is used, space and time savings can be made by reducing the number of separate allocations: this is shown in the results of *object inlining* studies[40, 41]. Use of value class objects embedded within other objects as instance fields (shown in Figure 4.4) achieves a similar effect to object inlining.

Use of value classes is not always beneficial, due to the increasing cost of object copying for larger value class instances. For larger objects, heap allocation and copying-by-reference is more desirable; for mutable objects, heap allocation is essential. Therefore, the choice between value or reference classes is a matter of using the right tool for the job. Having both options available in the language allows programmers to use whichever best matches the use of that class. Exactly what constitutes a small object or a large one is program-specific, however. The results presented in Chapter 7 show the effects of using the `value` and `final` modifiers for classes, in a series of test programs.

4.1.2 Immutable state

One important property of value classes in Pluto is that their instances are immutable. Once an object of value class is created, its internal state (represented by instance fields) does not change. This is because value classes, as their name suggests, have pure value-semantics: instances are copied by value and are never aliased. The memory layout in Figure 4.2 shows an example of copying by value.

Other languages also feature types with immutable state: Java's `java.lang.String` class represents a character string whose state never changes. Values of type `int` in Java, C and C++ do not have mutable state — arithmetic operations result in a new value rather than modifying the existing one. Note that these languages do provide mutable variables, even if the values held in those variables are not themselves mutable. For example, in a fragment of C or C++ code such as:

```
int count = 42;
count++;
```

the value `42` itself is not modified, even though the variable `count` is updated. Similarly, Pluto allows mutable local variables (declared with the `var` keyword) that can be updated to hold different values.

Value type semantics do not interact well with mutable state[46], as the nature of copying objects by value means that modifications to one value would not be reflected in any of its copies. In contrast, reference-based types (such as Pluto's abstract and reference classes) can be aliased: modifications to the object are visible through all pointers to that object. This explains why the `Mutable` class in Pluto, listed in Section 3.8.1, is reference-based, as it ensures that mutable state is heap-allocated and thus not copied by value.

An advantage of immutable state for value classes is the property of *referential transparency*, in which references to separate copies of a value cannot be distinguished from references to a single instance of that value (i.e. aliases). This allows the Pluto compiler to choose copying of values by reference if desired, e.g. to avoid costly duplication of large values. For example, Section 4.6.4 describes how referential transparency is exploited to pass the receiver for value class method calls.

4.1.3 Examples

The use of value classes is suited to the representation of concepts that require only a small amount of state, and are naturally immutable. A running example used in this dissertation is the `point` class, which represents a simple co-ordinate composed of two integers. Similarly, data types such as complex numbers, rational numbers and intervals that are used in scientific and numerical programming are good candidates for implementation with value classes. As discussed in the background material of Section 2.3.2, support for numeric data types is one of the motivations behind various proposals for value type support in Java.

Some object-oriented programming patterns[43] make use of indirection to provide more flexibility in a design, and employ extra objects such as iterators, adaptors, proxies and surrogates. These could potentially be implemented with value classes, as they often contain just a single instance field that refers to another object. Similar designs can be seen in the various forms of smart-pointer or indirection handle that are described in C++ programming guides[61, 73].

The description of the Eiffel language[72] highlights another useful application of value classes: modelling of a one-to-one relationship between an object and one of its contained fields. In situations where an object forms part of only one parent object (i.e. not shared by several) it need not be allocated as a distinct object. Since there would only be one reference to this object, the pointer to it is redundant and can be eliminated. The contained object can then be embedded within its parent, as a unique sub-object — Figure 4.4 shows some examples. A value class can be used to specify types such as these for which no sharing is needed. The example given in [72, page 205] is a `Wheel` class, in relation to a parent `Car` class. An individual `Wheel` object belongs to just one `Car` object, and is not shared. Therefore, the `Wheel` class is declared value-based and its instances are embedded directly within parent objects such as `Car`s.

The layout of interface instances in the Pluto object model, detailed in Section 3.6.2, shows a similarity to value class instances. A variable of interface type is represented by a pair of pointer fields, one to a vtable and the other to an object that implements the interface. These pointer-pairs are manipulated directly, and could be seen as a special kind of value class object. Moreover, the single pointer used to refer to a heap-allocated reference class object could also be viewed as an instance of a special `Pointer` value class — in effect, a value class that represents a handle.

4.2 Organising the class hierarchy

There are three kinds of classes in Pluto: abstract, value (both specified by annotating the class definition) and reference (the default, if no annotation is present). As briefly listed in Section 3.2, the Pluto type system makes some restrictions on how different kinds of class may be related in the class hierarchy. In summary, abstract classes exist at the top of the hierarchy, with reference and value classes below — as shown in Figure 3.1 on page 32. Value classes are segregated from reference classes: both kinds may extend abstract classes, but cannot extend each other (in this respect, Pluto follows the type

system of Sather[109]). This division reflects the different semantics of value-based and reference-based types.

Although abstract classes cannot be instantiated, a Pluto program may contain variables, value parameters or fields whose declared type is an abstract class. These variables are represented in the generated code in the same way as those whose type is a reference class: a pointer to the actual object. This means that abstract classes are also considered reference-based types. However, an abstract class can be extended by both value and reference classes. When a value class instance is stored in a variable of abstract class type, the value is *boxed* (copied to the heap) and then treated as a reference-based object. Because value class objects are immutable, their storage on the heap and possible aliasing is transparent to programs. The inverse operation (unboxing) is never needed, as Pluto has no facility for downcasting objects.

4.3 The `final` modifier

As stated in the Pluto language description (Section 3.2) any non-abstract class can be annotated with the `final` keyword, to indicate that the class cannot be extended — it will have no subclasses. One benefit of final classes is that they enable the Pluto compiler to make some optimisations to the generated code. In particular, instance method calls where the receiver type is a final value class can be statically dispatched (see Section 3.5.6). This means that the method dispatch code does not need to use the vptr or the vtable: the call is translated into a direct function call in the generated C code.

As Section 4.6 details further, the Pluto compiler uses a more compact representation for instances of final value classes, by omitting the vptr field. Since value class instances are manipulated directly, i.e. copied by value when assigned and passed to and from functions, eliminating the vptr field gives an immediate saving by reducing the amount of copying required. For those final value classes that have just one or two instance fields, the overhead of the vptr field can be a significant part of the object size. If many objects of these classes are stored in a data structure such as an array, eliminating the vptr can result in a substantial space-saving.

The Pluto program shown in Figure 4.3 defines a final value class called `point`, and some reference-based classes, `Desktop` and `Window`, that contain instance fields of type `point`. In Figure 4.4, the conceptual memory layout of some `Desktop` and `Window` objects shows how the `point` sub-objects are stored within. Note that these sub-objects do not contain their own vptr fields — in contrast with the reference and non-final value class objects shown in Figures 4.1 and 4.2 on page 65.

In practice, many examples of the use of value classes are monomorphic: their instances are manipulated using variables whose declared type is that value class, rather than a superclass. It is therefore important to optimise for this common case, by removing the vptr overhead for monomorphic uses of final value class objects. This technique allows the predefined value classes in the Pluto library, such as `int` and `float`, to be implemented directly using the corresponding C data types, rather than incur the overhead of being paired with a vptr field.

```
final value class point(x:int, y:int) {
  field x : int = x;
  field y : int = y;
}

class Desktop(o:point, n:int) {
  field origin : point = o;
  field number : int = n;
}

class Window(tl:point, br:point, dt:Desktop, i:int) {
  field topleft : point = tl;
  field bottomright : point = br;
  field desktop : Desktop = dt;
  field id : int = i;
}

abstract class Main() {
  static method run() : int {
    let dtop : Desktop = new Desktop(new point(0, 0), 1);
    let firstcorner : point = new point(10, 20);
    let secondcorner : point = new point(300, 250);
    let win : Window = new Window(firstcorner, secondcorner, dtop, 4);
    return 0;
  }
}
```

Figure 4.3: Pluto program using a final value class

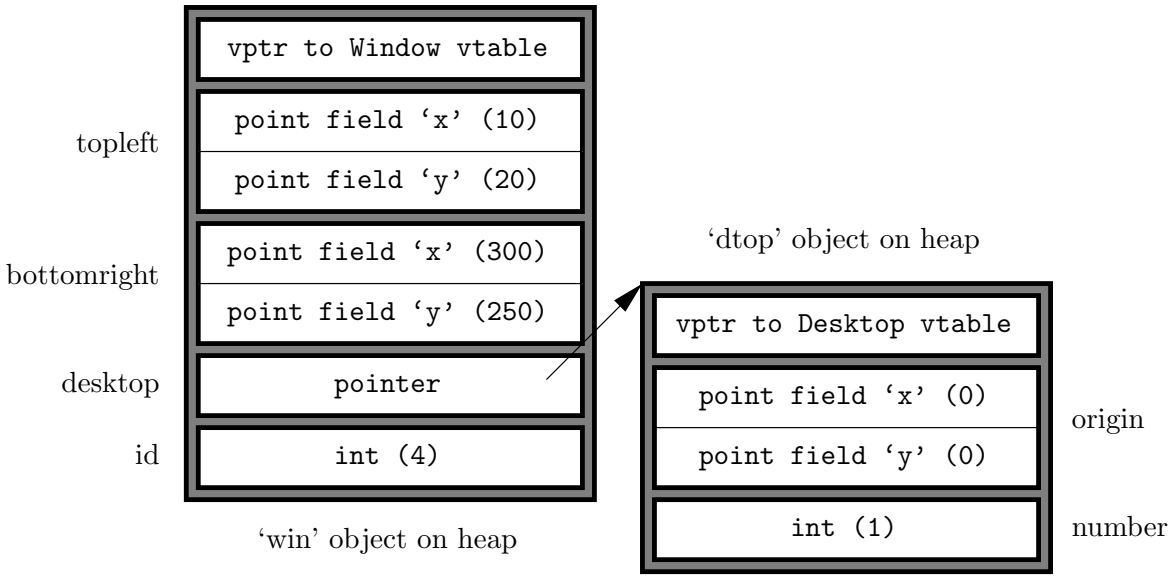

Figure 4.4: Memory layout with final value sub-objects

In situations where a final value class object is used in a polymorphic way, such as being coerced to one of its supertypes, the vptr field must be reinstated to match the expected layout. Nevertheless, monomorphic code does not pay the extra cost associated with having a vptr field. The topic of value class coercions is discussed in Section 4.6.3.

4.4 Types of literal values

Pluto programs can contain literal values such as `43` and `true` that denote objects directly with the language syntax. Because these literal values are integral to the language, their types must be predefined by the language implementation. Languages such as C++ and Java have a small number of primitive types such as `int` and `float` built into the syntax of the language. As covered by Section 3.8.1, Pluto supports such types by defining them as value classes instead. The general mechanism for defining value classes enables Pluto to support these fundamental types without making them a special part of the language.

The predefined Pluto types `int`, `float` and `boolean` are ideal candidates for representation by value classes, as they correspond to small, immutable data with value-based semantics. In addition, these types can be declared `final` and thus benefit from the optimisation described in Section 4.3. The use of final value classes means that the vptr field can be avoided, and hence these types can be implemented directly with corresponding C data types.

The Pluto `String` type represents an immutable sequence of characters, with particular instances being manipulated indirectly to prevent unnecessary copying of a potentially long String instances. This would suggest that the Pluto library should implement the String type as a reference class. However, it is defined as a final value class like those above, where the value to be manipulated is actually a pointer to the characters. This representation means that the Pluto library can directly use the C data type `char*` to implement the String type.

4.5 Value type restrictions

Due to the nature of value type objects, and to enable efficient implementation, some special restrictions on them are made by the Pluto language type system. These are listed below, with explanations of why the restriction is made.

4.5.1 Subclass size

A value class that extends another value class cannot add new instance fields. This is due to the compiler's need to be able to reserve space for a variable whose type is a value class. As first highlighted in Section 2.3.4, there is a potential implementation difficulty with value subclasses when subsumption is allowed: a subclass object might not fit into the space reserved for a variable of its superclass. The solution chosen in Pluto is to preserve subsumption, but prevent subclasses of value classes from introducing any new instance fields. The size of all subclasses will then be exactly the same as the size of the superclass.

The significance of this restriction naturally depends on how extensive the use of value subclassing might be. In many cases, a value class represents an isolated concept, so it

```
/* This procedure checks whether any value class contains itself */
procedure CheckAll():
    for each value class C:
        call CheckContainment(C, C)


/* This procedure checks whether class A contains a field of class B */
procedure CheckContainment(value class A, value class B):
    for each instance field F (inherited or defined) in A:
        if type of F is a value class VC:
            if VC is B or VC extends B (directly or indirectly):
                error "Value class B contains itself"
            else
                call CheckContainment(VC, B)
```

Figure 4.5: Algorithm for detecting recursive value class containment

can be declared `final` and extend `Object` (an abstract class). This is the case for the value classes in the Pluto standard library (int, float, boolean and String). As reviewed in Section 2.3.3, some language designs do not allow subclassing for value classes at all. By making the restriction on field addition, the Pluto language enables subclassing of value classes. This allows all the flexibility of method overriding and dynamically-dispatched method calls even for value class objects.

A related issue is that value subclasses cannot add new type parameters, as these are implemented using hidden instance fields. See Section 5.6 for a discussion of the implementation of genericity, and the storage of type parameters.

4.5.2 Recursive containment

A value class cannot contain an instance field whose type is that same value class, or one of its subclasses. In essence, a value class cannot contain itself recursively — directly or otherwise. This rule also affects indirect (mutual) containment: for example, given two value classes VA and VB, if VA contains an instance field of type VB and, in turn, VB contains an instance field of type VA then the type system would reject this combination of definitions. Using the Pluto compiler, whichever definition came later in the source would be rejected as incompatible with the former definition. Figure 4.5 shows the algorithm used in the Pluto compiler to detect recursive containment of value classes.

The reason for this restriction is simply to enable the compiler to determine the storage requirements of a value class instance. The instance size is just the sum of the sizes of all instance fields and compiler-generated fields. For reference-based types the field size is that of a pointer, but for value classes the size is the actual instance size. In order for the compiler to determine the instance size S, there must be no recursive mention of S in the size-calculation sum.

This restriction is similar to that in Eiffel, whose language definition contains a rule for *expanded clients*[72, page 93] that prohibits any cycles in the graph of containment

relationships among expanded classes. Likewise in C[56] and C++[55], where a structure or class definition cannot have members of incomplete type (i.e. one whose size is unknown to the compiler). Since the Pluto compiler generates C code, performing the check described above also ensures that the generated code does not contain any invalid structure definitions.

4.6 Implementation details

This section describes implementation issues that are specific to value classes in the Pluto compiler, building on the design and code generation outlined in Sections 3.5 and 3.7.

4.6.1 Instance handling

The most fundamental change in the code generated for value classes, as compared to reference-based classes, is the definition of the type names, i.e. those beginning with `pto_typ_` and `pto_cty_`. For value classes, the `typedef` for these type names defines them as an actual instance of the `pto_ins_` structure. This contrasts with abstract and reference classes, where the `typedef` specifies a pointer to the instance structure. The effect of this is that wherever the type name is used (in variables, structure members, function parameters and return types) then objects of the instance structure are passed by value.

Use of a structure type as a function return type in C is implemented by gcc using a hidden extra pointer parameter, rather than the straightforward approach of copying a local return value back to the function's caller. Instead, space for an instance of the structure is allocated in the caller's stack frame, and a pointer to it is passed to the called function as an extra parameter. The function's return value is placed directly within this space, eliminating the extra (redundant) copy. Pluto's value classes take advantage of this optimisation when used as method return types, and in the implementation of the new-instance creation function. In C++ compilers, a similar technique is known as the *named return value* (NRV) optimisation[68].

4.6.2 Headless layout and full value information

As stated in Section 4.3, instances of final value classes do not have a vptr field. In the generated C code, this is reflected in the declaration of the instance structure, which omits the vptr field that is present for other kinds of class. However, instances of final value classes may be coerced to a superclass that expects a vptr field at the object's head. In support of this, the C code generated for value classes contains some extra declarations (which can be seen in the example code of Section 4.6.6).

In addition to specifying the usual instance structure, the C code generated for final value classes contains a declaration of a *full value instance* structure (with the prefix `pto_fvi_`). This contains a vptr field in addition to those fields present in the ordinary instance structure, and hence matches the layout of its superclass. The structure declaration avoids repeating each instance field by simply declaring two members: one is the vptr field and the second is the plain instance structure used for final value objects, which in

turn contains all the remaining fields. This arrangement allows all the non-vptr members to be assigned as a unit, rather than with a complicated expression containing each member. Despite the nested structure, the layout of a full value instance is compatible with the flat superclass layout. Although the C standard does not mandate structure layout details, examination of gcc's assembly code output shows that a full value instance has the same representation as its superclass instance. Even though the Pluto compiler takes advantage of this property of gcc's structure layout, it is simply a notational short-hand. The Pluto compiler could easily be modified to generate C code that declares the full instance structure in a non-nested form, with all the instance fields listed individually.

For non-final value classes, the situation is reversed. The usual instance structure contains a vptr, and there is also a corresponding structure declaration in the generated code called a *headless value instance*, with the prefix `pto_hvi_`. This consists of all the members present in the instance structure except the vptr field. In effect, it declares the layout of all possible final value subclasses, since they are unable to add new instance fields.

The purpose of these extra structure declarations is to support the various coercions used for handling value class objects. The following section describes some of these, with Sections 5.4 and 5.7 detailing coercions used in the implementation of genericity. A related issue is the need for two type descriptor (vtable) instances to be created for final value classes, due to the full or headless layouts. This is explained further in Section 5.5 on type descriptors.

4.6.3 Coercions to supertypes

As Section 3.7.4 noted, coercing instances of reference-based classes simply involves C type casts, as they are handled indirectly via pointers. However, the coercion of value class objects is more complex because they are manipulated directly. In the generated code for value classes, the full-value and headless structures listed in the previous section are used by the value class coercions. Due to the different layouts of final and non-final value classes, there are several cases to consider:

Non-final value class to value superclass A straight unit copy can be made of the object, as value subclasses cannot add new instance fields and therefore have the same layout as their superclass.

Non-final value class to abstract superclass As the objects of non-final value classes already have their vptr field in place, this operation simply involves taking the address of the object. The issue of safe pointers to stack storage is discussed in Section 5.8.

Final value class to value superclass This coercion requires the joining of the final value object with the address of a vtable instance, using a GNU C constructor expression. The effect of this operation is to create a full-value instance. As the object now has a vptr field, the address of the full-info type descriptor is used (Section 5.5 explains the use of two type descriptors for final value classes).

Final value class to abstract superclass The implementation of this coercion is done in two stages: the object is first coerced to a full value (as in the previous coercion)

whose address is then taken. As gcc does not allow the addressing of constructor expressions, an intermediate temporary variable is used and the address of that is taken.

The implementation of coercions from value classes to an interface type involves the pairing of an interface vtable address with the object coerced to `Object` (as discussed in Section 3.7.4). The latter operation is expressed by using one of the coercions listed above, with the abstract class `Object` as the target. For details of the coercions used to implement genericity, see Section 5.4.

A minor complication in the implementation of coercions by the Pluto compiler is that gcc does not allow code that takes the address of certain constructs (in C terminology those that are not l-values). As noted above, this applies to GNU C constructor expressions, among others. The code generation phase of the Pluto compiler splits any coercions that would otherwise require the address of a non-lvalue, creating an intermediate variable.

4.6.4 Passing the receiver

Section 3.5.7 discussed how the Pluto-generated code represents the method receiver as a parameter called `this` in the translated C function. As explained in that section, those methods that may need dynamic dispatch have an entry in the vtable, and the C type of the `this` parameter is always `Object`. In translating methods that do not need vtable support (i.e. because they do not override and cannot be overridden) the type of `this` is the actual class in which the method is defined.

For value classes, this would mean passing a copy of the receiver by value at each call to such a method. An optimisation made by the Pluto compiler is to pass the receiver by reference for those value classes whose instance size is larger than the size of a pointer value. In these cases, the `this` parameter has a pointer type, and all uses of it have a dereference operation applied. This is similar to the use of the C++ reference facility. In the example generated code in Section 4.6.6, the definition of the `pto_fun_fvpoint__equalsequals` function shows that the `this` parameter's type is a pointer to an instance of `fvpoint`. This is because the class is a final value one, whose instance size is that of two integers, and hence pass-by-reference is used.

Due to the immutability of value classes, the choice of pass-by-value or pass-by-reference can be made without affecting the semantics of the translation. Code within the body of the called function cannot modify the `this` parameter's value, so the original receiver does not need to be copied by value.

4.6.5 Constructor optimisation

In order to eliminate unnecessary overheads in using value classes, some optimisations can be made to the functions used to create new instances of them. Since a final value class cannot be extended, there is no need to have separate constructor and creation functions: they can be merged. Even if this were not done, both functions are declared `inline`, which allows gcc to incorporate the body of the constructor within the creation function.

In most cases a constructor first calls its superclass constructor, but this can be omitted if the superclass has a trivial constructor. A trivial constructor is one that simply calls its superclass constructor, which must also be trivial. This test continues until the `Object` class is reached, which has an empty constructor. For example, in the case of a final value class that extends Object, there is no need to make the superclass call and the Pluto compiler omits it from the generated code. This means that the new object under construction does not need to be coerced to its superclass type — a saving of one local variable and a copy operation.

A further improvement to the code for creating new final value classes can be made by using a GNU C constructor expression to build the new object. The example code in Section 4.6.6 shows the sum effect of these optimisations in the `pto_fun_fvpoint__new` function definition. In a Pluto program, a `new fvpoint` expression is translated into a call to this function. The function consists of just a constructor expression that builds a new instance of the class using the supplied argument values. As the function is declared `inline`, calls to it can be replaced by this expression, resulting in efficient and compact assembly code.

4.6.6 Example generated code

This section shows the C code produced by the Pluto compiler for the following simple code which declares a `final value` class:

```
final value class fvpoint(x:int, y:int) {
  field x : int = x;
  field y : int = y;
  method ==(other:fvpoint) : boolean {
    return this.x == other.x && this.y == other.y;
  }
}
```

The Pluto class is very similar to that given in the example code of Section 3.7.3, except that it is a value class instead of a reference class. As discussed above, the fact that it is a value class means that in the generated C code there are differences in the type declarations and the coercion macros. The use of the `final` modifier for the class also results in different code generation, as there is a full-value instance structure, two type descriptor instances and the implicitly-final method no longer occupies an entry in the vtable. The C code produced is as follows:

```
/* Header file 'pto_hd1_fvpoint.h' – generated code */
/*

final value class fvpoint[](x : int[], y : int[]) extends Object[] implements []
 field x : int[]
 field y : int[]
 original_final method _equalsequals[](other : fvpoint[]) : boolean[]

*/
```

```
#ifndef PTO_DEF_fvpoint
#define PTO_DEF_fvpoint

#include "pluto.h"

/* Typedefs for this class */
typedef struct pto_ins_fvpoint pto_typ_fvpoint;
typedef const struct pto_ins_fvpoint pto_cty_fvpoint;
typedef struct pto_fvi_fvpoint pto_vit_fvpoint;

/* Other types used by this one */
#include "pto_hd1_int.h"
#include "pto_hd1_Object.h"

/* Class info type */
typedef const struct pto_cls_fvpoint* pto_tti_fvpoint;

/* Class instance (object) structure */
struct pto_ins_fvpoint {
  pto_typ_int pto_fld_x;
  pto_typ_int pto_fld_y;
};

/* Full value instance structure */
struct pto_fvi_fvpoint {
  pto_tti_fvpoint vptr;
  pto_typ_fvpoint object;
};

/* Class info structure/vtable */
struct pto_cls_fvpoint {
  const unsigned short imm_size;
  const short type_kind;
  pto_tti_fvpoint full_info;
  itable (*lookup_itable)(pto_cty__gentype itable_tag);
};

/* Declaration of type info instance(s) */
extern const struct pto_cls_fvpoint pto_inf_fvpoint;
extern const struct pto_cls_fvpoint pto_fif_fvpoint;

/* Declaration of interface views */

/* Coercion macros to and from other types */
#define pto_cvt_fvpoint_fromThis ((*(pto_vit_fvpoint*)pto_var_this).object)
#define pto_cvt_fvpoint_toFullValue(VALUE) ((pto_vit_fvpoint) { &pto_fif_fvpoint, VALUE
})
#define pto_cvt_fvpoint_toValue(CLASS,VALUE) ({ pto_cty_fvpoint tmp = VALUE;
((pto_cty_ ## CLASS) { (pto_tti_ ## CLASS)&pto_fif_fvpoint, tmp.pto_fld_x, tmp.pto_fld_y
}); })
#define pto_cvt_fvpoint_toAbstract(CLASS,VALUE) ((pto_cty_ ## CLASS)&VALUE)
```

```
#define pto_cvt_fvpoint_toAbstractHeap(CLASS,VALUE) ((pto_cty_ ## CLASS)({
pto_vit_fvpoint* nu = malloc(sizeof(pto_vit_fvpoint)); *nu = ((pto_vit_fvpoint) {
&pto_fif_fvpoint, VALUE }); nu; }))
#define pto_cvt_fvpoint_toInvGeneric(VALUE) ((pto_cty__genvalue)&VALUE)
#define pto_cvt_fvpoint_toVarGeneric(VALUE) ((pto_cty__genvalue)&VALUE)
#define pto_cvt_fvpoint_toInvGenericHeap(VALUE) ((pto_cty__genvalue)({ pto_typ_fvpoint*
nu = malloc(sizeof(pto_typ_fvpoint)); *nu = VALUE; nu; }))
#define pto_cvt_fvpoint_toVarGenericHeap(VALUE)
((pto_cty__genvalue)pto_cvt_fvpoint_toAbstractHeap(Object,VALUE))
#define pto_cvt_fvpoint_fromInvGeneric(VALUE) (*(pto_cty_fvpoint*)VALUE)
#define pto_cvt_fvpoint_fromVarGeneric(VALUE) ((*(pto_vit_fvpoint*)VALUE).object)
#define pto_cvt_fvpoint_dynFromInvGeneric(VALUE,TYPEARG)
(*(pto_cty_fvpoint*)VALUE)
#define pto_cvt_fvpoint_dynFromVarGeneric(VALUE,TYPEARG)
pto_cvt_fvpoint_fromVarGeneric(VALUE)
#define pto_cvt_fvpoint_toIfc(INTERFACE,OBJ) ((pto_cty_ ## INTERFACE) { (pto_tti_
## INTERFACE)&pto_ivu_fvpoint_ ## INTERFACE, OBJ })
#define pto_cvt_fvpoint_toInterface(INTERFACE,VALUE)
pto_cvt_fvpoint_toIfc(INTERFACE,pto_cvt_fvpoint_toAbstract(Object,VALUE))
#define pto_cvt_fvpoint_toInterfaceHeap(INTERFACE,VALUE)
pto_cvt_fvpoint_toIfc(INTERFACE,pto_cvt_fvpoint_toAbstractHeap(Object,VALUE))

/* Aliases for functions + globals */

/* Function prototypes */
extern pto_cty_int pto_fun_fvpoint__equalsequals(pto_cty_fvpoint* pto_var_this,
pto_cty_fvpoint pto_var_other);

/* Inline function prototypes */

/* Vtable entries */

/* Static initialiser function prototype */
extern void pto_ini_fvpoint(void);

/* New instance function prototype */
static inline pto_cty_fvpoint pto_fun_fvpoint__new(pto_cty_int pto_var_x, pto_cty_int
pto_var_y);

/* Global variable declarations */

#endif
```

```
/* Inline header file 'pto_hd2_fvpoint.h' - generated code */
#ifndef PTO_IDEF_fvpoint
#define PTO_IDEF_fvpoint

/* Other inline code used by this type */
#include "pto_hd2_int.h"
#include "pto_hd2_Object.h"

/* New instance creation function */
```

```
static inline pto_cty_fvpoint pto_fun_fvpoint__new(pto_cty_int pto_var_x, pto_cty_int
pto_var_y) {
  return ((pto_cty_fvpoint) { pto_var_x, pto_var_y });
}

/* Inline function definitions */

#endif
```

```
/* Source file 'pto_src_fvpoint.c' – generated code */
#include "pto_hd1_fvpoint.h"
#include "pto_hd2_fvpoint.h"

#define pto_ilu_fvpoint default_lookup_itable

/* Definition of type info instance(s) */
const struct pto_cls_fvpoint pto_inf_fvpoint = {
  sizeof(pto_cty_fvpoint),
  TYPEKIND_FINAL_VALUE,
  &pto_fif_fvpoint,
  pto_ilu_fvpoint
};

const struct pto_cls_fvpoint pto_fif_fvpoint = {
  sizeof(pto_vit_fvpoint),
  TYPEKIND_NON_FINAL_VALUE,
  &pto_fif_fvpoint,
  pto_ilu_fvpoint
};

/* Global variable definitions */

/* Static initialiser flag and function */
static int need_init = 1;
void pto_ini_fvpoint(void) {
  if (need_init) {
    need_init = 0;
    pto_ini_int();
    pto_ini_Object();
  }
}

/* Function definitions */
pto_cty_int pto_fun_fvpoint__equalsequals(pto_cty_fvpoint* pto_var_this, pto_cty_fvpoint
pto_var_other) {
  { pto_cty_int pto_var__tmp1 = (*pto_var_this).pto_fld_x;
  { pto_cty_int pto_var__tmp2 = pto_var_other.pto_fld_x;
  { pto_cty_int pto_var__tmp3 = pto_fun_int__equalsequals(pto_var__tmp1, pto_var__tmp2);
  { pto_typ_int pto_var__tmp7;
  { if (pto_var__tmp3)
```

```
    {
      { pto_cty_int pto_var__tmp4 = (*pto_var_this).pto_fld_y;
      { pto_cty_int pto_var__tmp5 = pto_var_other.pto_fld_y;
      { pto_cty_int pto_var__tmp6 = pto_fun_int__equalsequals(pto_var__tmp4,
      pto_var__tmp5);
      { pto_var__tmp7 = pto_var__tmp6; }}}}
    } else {
      pto_var__tmp7 = false;
    }
  { return pto_var__tmp7; }}}}}}
}
```

4.7 Summary

This chapter describes how value types are supported in the Pluto language, and details the implementation techniques used by the compiler. As shown, value types can be provided without excessive restrictions when compared with reference types. For example, a value class can define type parameters (making it generic), have subclasses, implement interfaces and participate in dynamic method dispatch. This allows closer integration of value and reference classes by incorporating them in a single inheritance hierarchy, rather than separate hierarchies for each kind. The importance of integration becomes more apparent when genericity is present — as described in Chapter 5.

Chapter 5

Second addition: genericity

5.1 Benefits of generic programming

In Section 2.4 of the background chapter, the concepts of genericity are outlined. There are a number of primary reasons why support for genericity in a programming language is of benefit:

- Code can be made more widely applicable, by abstracting it to work with many types of data instead of just one. Container classes and generic algorithms mean that code duplication is minimised.
- Use of genericity often eliminates the need to bypass the language's type system. For example, type casts down the class hierarchy can be avoided and compiler type checking is improved.
- Libraries of reusable code can be built, by taking advantage of generalised data structures and algorithms. An example of this is clearly seen in the powerful features provided by the C++ Standard Template Library[87].

In this chapter, the design and implementation of genericity in Pluto is detailed, building on the basic language translation scheme covered in Chapter 3. As will be shown, genericity support in the language does not have to be significantly compromised by the presence of value types. Implementing the combination of these two mechanisms can be achieved using the techniques described in this chapter.

5.2 Type parameters

Genericity in Pluto is achieved via the definition of type parameters. These may appear in class or interface definitions, or individual methods. A type parameter defines a new type name that can be used within the context of its associated class, interface or method. Example Pluto code in Figure 5.1 shows several uses of type parameters. The `Cell` class defines an unbounded type parameter called `T` that can represent any type, and uses this new type name in various constructs within the rest of the class definition. Similarly, the interface `Comparable` defines a type parameter named `E` whose role is to represent

```
class Cell[T](initialData:T) {
  field data : T = initialData;
  method get() : T {
    return this.data;
  }
  method set(newData:T) {
    var this.data = newData;
  }
}

interface Comparable[E] {
  abstract method ==(other:E) : boolean;
}

final value class point(x:int, y:int) implements Comparable[point] {
  field x : int = x;
  field y : int = y;
  method ==(other:point) : boolean {
    return this.x == other.x && this.y == other.y;
  }
  method print() {
    String.println("[" + $this.x + "," + $this.y + "]");
  }
}

class BackupCell[X <: Comparable[X]](initialData:X) extends Cell[X](initialData) {
  field backupData : X = initialData;
  method set(newData:X) {
    if (!(newData == this.data)) {
      var this.backupData = this.data;
    }
    super.set(newData);
  }
  method reset() {
    var this.data = this.backupData;
  }
}

abstract class Main() {
  static method run() : int {
    let cell : BackupCell[point] = new BackupCell[point](new point(3, 4));
    cell.get().print();
    cell.set(new point(5, 6));
    cell.get().print();
    cell.reset();
    cell.get().print();
    return 0;
  }
}
```

Figure 5.1: Pluto program using type parameters

the type of data to be compared against. A class or interface may contain a list of these type parameter definitions, written inside square brackets directly after the class/interface name.

Once a type parameter is defined, its name may be used in most places that a type name is expected: e.g. when declaring local variables and instance fields, instance method parameters and return types. Section 5.2.3 lists some situations in which a type parameter cannot be used as a type name, and explains the reason for these restrictions.

A class or interface that defines one or more type parameters is often known as a *generic type*. For example, in the code of Figure 5.1, `Cell` is a generic class. When this class name is used in the declarations of other constructs, such as variables and `extends` clauses, matching type arguments must be supplied that correspond with the type parameters defined by the class. In the `Main.run` method, a local variable `cell` is declared with type `BackupCell[point]` where `point` is a type argument. A similar situation can be seen in the same line of code, where a type argument is supplied when creating a new instance of the generic `BackupCell` class. For unbounded type parameters, any valid type name can be used as a type argument. Even the name of another type parameter can be used, as shown in the `extends` clause of the `BackupCell` class.

In most contexts, the name of a generic class or interface must be followed by a suitable list of type arguments. The only exception is that static members of a generic class can be accessed using only the class name. Section 5.2.3 explains how static members of a class are independent of class type parameters.

5.2.1 Bounded type parameters

The example `Cell` class defines a type parameter that is unbounded, in the sense that it can represent any type. This allows the class to be used for any type of data: a flexibility that is desirable in this case. For some generic classes however, the disadvantage of unbounded type parameters is that no assumptions can be made about the actual types supplied as type arguments. The `BackupCell` class works by comparing its existing and new data, using a call to the `==` method of one of the objects with the other as an argument. Therefore, it needs to ensure that only types supporting such a method are used as type arguments to match its type parameter `X`.

The definition of a type parameter may have an optional bound, which is specified using the syntax `<:`TypeName after the parameter's name. The `BackupCell` class uses this facility in the specification of its type parameter `X`, which can represent any subtype of the `Comparable` interface with `X` itself as a type argument. This expresses a constraint on type arguments, that they must be comparable with themselves. The `point` class fulfils this requirement, as it implements the `Comparable` interface with itself as a type argument.

In the Pluto type system, type parameters may be bounded by any class or interface, which may itself be generic. Although the name of a type parameter cannot be used as the actual bound, it may appear within its own bound as a type argument. This is the case for the `X` parameter in the `BackupCell` class. In fact, type parameter names

can be used within the bound of other type parameters defined in that class or interface, not just its own. As discussed in Section 2.4.2 this flexibility (known as F-bounded type parameters[25]) provides useful expressive power.

An unbounded type parameter is effectively the same as one with the type `Object` as its bound. This is because `Object` is the root class of the inheritance hierarchy in Pluto, and is also considered as a supertype of all interfaces. As such, any type in Pluto is a subtype of `Object`.

A generic class that defines a bounded type parameter can then rely on all actual type arguments being subtypes of the bound. For example, this ensures that the expression `newData == this.data` in the `set` method of `BackupCell` is type safe. A related assumption is made when a type parameter name is used as a type argument: this is acceptable provided its bound is a subtype of the matching type parameter's bound. For example, `BackupCell` safely extends `Cell` using `X` as a type argument, because the bound of `Cell`'s type parameter is `Object`, which is the supertype of every other type.

One possible extension to the bounding of type parameters would be to allow multiple bounds — this is proposed in the Java genericity extension document[14], which supports additional bounds that must all be interfaces. From a type checking perspective, Pluto could easily be extended to allow multiple bounds: type arguments would simply be checked to ensure that they are subtypes of all the bounds. The reason that this facility was not included in Pluto as described in this dissertation is due to the technique used to implement method dispatch on generic receivers. As Section 5.7 explains in detail, generic values are coerced to the type of their bound before the method call is made. With multiple bounds, the appropriate coercion operation could not easily be determined.

5.2.2 Method type parameters

Besides classes and interfaces, Pluto allows individual methods to define type parameters. The example code in Figure 5.2 shows two static methods, `id` and `minimumByHash`, that each define type parameters — they are *generic methods*. The former is the classic polymorphic identity function, which can be used with any type of data. The latter uses a bound on its type parameter `X` to allow the method body to make method calls on its two generic arguments.

The `Main.run` method shows how these generic methods are used. Type arguments are supplied when the method is called, with the ordinary method call arguments given after. Type checking of these calls is done by conceptually replacing the type parameter name (`X` or `T`) with the supplied type argument (`int`, `float` or `point`) before checking the ordinary arguments against those types.

5.2.3 Limitations of use

There are three situations in which the name of a type parameter cannot be used as a type name in Pluto, for reasons discussed below. In these places, the name of an actual type must be used: either a class or interface name.

The first restriction is that when creating a new object using the `new` expression, only

```
interface Hashable {
  abstract method hashcode() : int;
}

final value class point(x:int, y:int) implements Hashable {
  field x : int = x;
  field y : int = y;
  method hashcode() : int {
    return this.x + this.y;
  }
}

class Main() {
  static method run() : int {
    let i : int = Main.id[int](43);
    let f : float = Main.id[float](3.14159);
    let p : point = Main.minimumByHash[point](new point(5, 6), new point(3, 7));
    return 0;
  }
  static method id[T](ob:T) : T {
    return ob;
  }
  static method minimumByHash[X <: Hashable](first:X, second:X) : X {
    if (first.hashcode() < second.hashcode()) {
      return first;
    } else {
      return second;
    }
  }
}
```

Figure 5.2: Generic methods, using type parameters

the name of a non-abstract class can be used as the type to be instantiated. The reason for disallowing the use of `new T`, where `T` is a type parameter, is that there is no way of knowing the number of constructor parameters or their types. The type parameter `T` can represent many different types, with some having constructor parameters that require matching values to be supplied when creating instances. Therefore, it would not be possible to provide arguments to the above `new` call that could be type checked at compile time. Furthermore, the types represented by `T` can include interfaces and abstract classes, neither of which can be instantiated. In essence, the `new` operation is a fundamentally monomorphic one.

A second restriction on the use of type parameter names is that they cannot appear as the superclass or superinterface in an `extends` or `implements` clause of a class/interface definition. For example, code such as `class Cell[T]() extends T` is not valid. This is necessary because Pluto is a statically-typed language, and allowing such code would effectively require some form of dynamic inheritance: the class hierarchy would not be

known at compile-time, or even be constant.

Neither of these restrictions prevent type parameter names from being used as type arguments in those situations, however. In the example code of Figure 5.1, the class `BackupCell` uses its type parameter `X` as a type argument in the name of its superclass. Similarly, although Pluto does not allow `new T` when `T` is a type parameter, an expression such as `new Stack[T]()` is permissible.

Despite the presence of the two restrictions listed above, programs can often achieve the desired effects (creating new objects of a generic type, and appearing to change the subclass relationships at run-time) by applying suitable design patterns[43].

The third restriction is that type parameters defined by a class cannot be used by its static members: this is because the actual type arguments used in place of the type parameters may be different for each specific instance of the class. As static members can be accessed without reference to any instance of that class, they cannot therefore be dependent on the type arguments, which are instance-specific. Nevertheless, this restriction does not prevent static methods from defining their own type parameters. Indeed, many examples of methods that are generic also happen to be static.

5.3 Implementation overview

While the previous sections have described how genericity is supported in Pluto at the language level, the remainder of this chapter explains how genericity is implemented by the Pluto compiler. The major factor affecting implementation complexity is the support of value types with genericity. If all types that could be used in a generic context were reference-based, implementation would be much simpler. This is shown by the translation technique used for GJ[17] (an extension to Java) in which genericity is "compiled away" by inserting pointer-type casts where needed. These casts do not cause any run-time object conversion operations, such as boxing, because all objects are manipulated indirectly anyway. Since Pluto supports the use of value types in generic contexts, the implementation is more complicated. Using the code of Figure 5.1 where necessary to give examples, these points summarise the basic approach used:

- A generic value (e.g. in class `Cell`, the `data` instance field or the `initialData` and `newData` variables) is represented by a pointer to the actual object. As stated back in Section 3.5.1, one of the design principles in the Pluto implementation is that the size of immediate data such as fields and variables must be known at compile-time. Because of this, a generic value cannot be stored directly since its size is unknown at compile-time, and may be different for each use of a generic type. Therefore, generic values are handled indirectly through a pointer.

- When a non-generic value is passed into a generic context (usually as an argument to a method call: e.g. in the call to `cell.set` in the `Main.run` method) a coercion operation is inserted, that converts the value into a generic value (i.e. a pointer). Likewise, when a generic value is returned to a non-generic context (e.g. from a method call such as the various `cell.get` calls) the reverse coercion is applied, yielding a value whose type is that of the corresponding type argument. In the example code, the `cell` variable's type is `BackupCell[point]` so the result of the

`get` method call will be an object of type `point`. Section 5.4 explains these coercions, with Section 5.8 describing an optimisation that helps to reduce their cost.

- To enable code in a generic class to manipulate generic values (e.g. copying, coercing or invoking methods on them — see Section 5.7) it must be able to determine certain information about the objects to which generic values point: their size, and whether they are instances of a value type or a reference type, for example. Again, this information is not available at compile-time and can vary with each use of a generic class. Consequently, generic code consults run-time type information as required to support operations on generic values. This meta-information is provided in the form of type descriptors: see Section 5.5 for details.

- In order to locate the correct type descriptor at run-time, the Pluto implementation uses a *type-passing* technique. A generic method's type parameters are represented by extra parameters in the generated C code function, and calls to the generic method are translated so that the type arguments become extra function call arguments that are pointers to type descriptors. A similar scheme is used for generic classes, whereby these type descriptor pointers are stored within each object of the generic class. For example, objects of the `Cell` and `BackupCell` classes contain an extra field that is a pointer to the descriptor for whatever type argument was used to construct the object. This translation process is detailed in Section 5.6.

5.4 Coercing to and from generic values

As stated above, the representation of a generic value is a pointer in C. Since the type of the object pointed to cannot be determined at compile-time, the generated C code uses an integer pointer type `int*` for variables of all generic values. For readability, a synonym for this generic value type is used in the generated code: `pto_cty__genvalue`, which follows the naming convention outlined in Section 3.7.2.

For example, when the Pluto compiler translates the `Cell` class, uses of the type parameter `T` as type names are converted to the genvalue type in the generated C code. This includes the `data` instance field, the `get` method return type and the `set` method parameter type. For any type parameter, regardless of its name or its bound, all occurrences of it as a type name appear in the generated code with the genvalue type.

When calling a method that has a generic value parameter (such as the `Cell` class constructor or `set` method) a non-generic argument value must be converted to a generic value. Likewise, a method call or field access whose result type is generic requires the opposite conversion, from generic value back to non-generic. To implement these conversions, the Pluto-generated C code for each class and interface defines several coercions using preprocessor macros. For example, the `point` class defines a pair of macros called `pto_cvt_point_toInvGeneric` and `pto_cvt_point_fromInvGeneric` that coerce a `point` object to and from a generic value, respectively. The `Inv` portion of the name refers to *invariant*: see Section 6.5.1 for details of the effects of variance on generic coercions.

Since objects of reference-based types are always manipulated via pointers, the implementation of these generic value coercions in C is just a cast to and from the genvalue type. For value type objects, the coercion to a generic value simply takes the address of the object and casts it to the genvalue type. When coercing back from a generic value,

the pointer type is cast and dereferenced. A potential problem with taking the address of value type objects is that they may be allocated on the stack, which could lead to a dangling pointer if the object has a shorter lifetime than the pointer to it. Section 5.8 explains how this issue is managed by the Pluto compiler.

A minor complication in taking the address of values occurs with literals such as integers and strings: for example, C code cannot take the address of `43` or `"foo"`. For those literal values whose address is required, the Pluto compiler solves this problem by converting them into `static const` local variables (with unique names) whose address can then be taken.

5.5 Type descriptors

Each class and interface defined in Pluto has a type descriptor in the generated C code. As explained in Section 3.5.3, the form of these is a structure whose fields contain meta-information about the type. This meta-information is crucial to the implementation of genericity in Pluto, as it enables the correct interpretation of the objects that generic values point to. The contents of type descriptors, and the reasons why the information they contain is useful, are outlined in this section. How the information is actually used is described in Section 5.7.

5.5.1 Meta-information

There are three fields of meta-information in each type descriptor. These are explained below, under the name of the structure member as it appears in the generated C code:

imm_size The contents of this field is the amount of immediate storage required for an instance of the type, as measured by using the C `sizeof` operator. The value stored here indicates how much space would be required for a variable declared with this type. For reference-based types this is just the size of a pointer, whereas for other kinds of type it is the size of the instance structure. The imm_size field is used by operations that copy the objects pointed to by generic values (detailed in Section 5.7.3).

type_kind In this field is an indicator used to classify the type. There are four possible values, to reflect the following kinds of type:

- Abstract or reference class
- Non-final value class
- Final value class
- Interface

This information is used by the generic copying and coercion operations (described in Section 5.7) to control their interpretation of objects that generic values point to. For example, when the Pluto Array type is handling a generic value, it uses the type_kind information to decide whether to store the pointer itself or the object it points to.

full_info This field is a pointer to a type descriptor — for interfaces and most classes it simply points back to the same type descriptor in which it appears. Section 5.5.2 below explains what this field is used for.

5.5.2 Type descriptor instances

There is exactly one type descriptor instance for each interface or class, except for final value classes: these have exactly two instances. Since the information in type descriptors does not change, these instances are declared in C as `extern const` so that gcc can place them in the read-only data section when producing object files.

For final value classes, two instances of the type descriptor are needed: one that describes objects in their normal headless layout, and a second that describes the layout of full-value objects (i.e. those with a vptr field). The first of these uses its full_info field to point to the second instance. In the second instance, as for other kinds of classes, this field just points to its containing type descriptor instance. Naturally, the second instance has a larger value in the imm_size field, to reflect the presence of the vptr field. The type_kind field of the second instance uses the indicator for a non-final value class since the object has a vptr field at its head.

The need for locating the full-value type descriptor is to support the dynamic coercion of generic values, as detailed in Section 5.7.

5.6 Type arguments

An important property of the Pluto implementation of genericity is that in every context where a generic value is used, there is some way of finding the correct type descriptor for the object that the generic value points to. While the previous section has described the form of these type descriptors, this section explains how they are actually located by the code in a generic class that needs to use the information they contain.

The Pluto compiler uses a type-passing approach, whereby type information is passed into a generic context to enable the manipulation of objects through generic values. The term *type argument* is used here to describe the actual values that are passed around: these are implemented as a pointer to a type descriptor (i.e. similar to an object's vptr field, which is a pointer to the vtable/type descriptor for its class).

5.6.1 The type of type descriptors

All type descriptors are laid out in the same format, with the three fields of meta-information at the beginning in the same order. Although the descriptors for most classes contain extra fields for the vtable of function pointers, only the meta-information fields are of interest when the descriptor's address is used as a type argument. In Pluto-generated code, the C type of a type argument is a pointer to a structure that declares these three fields in the same way that all type descriptors do. This appears in the generated code with the synonym `pto_cty__gentype` — effectively it is the common type of all type descriptors.

In the generated C code, a type argument is constructed by simply taking the address of a type descriptor and casting it to this `gentype`. The pointer value is then passed around and stored, as described in the next sections.

5.6.2 Type argument passing

To best illustrate how the Pluto compiler implements type-passing, the translation of generic methods is detailed first. Each type parameter is represented in the generated C function by a parameter whose type is `gentype`, and each type argument used in method calls is represented by an extra function call argument that gives the address of a type descriptor. The general approach is similar to the implementation of polymorphic functions as described in Appel[10, page 382].

For example, the generic `id` method in Figure 5.2 is defined and used in Pluto like this:

```
static method id [T](ob:T):T { return ob; }

let p:point = new point(3,4);

let q:point = Main.id[point](p);
```

The Pluto compiler translates that code into the following C (where the coercion macros have been expanded to show the underlying operation):

```
pto_cty__genvalue pto_fun_Main_id(pto_cty__gentype pto_var_T,
                                  pto_cty__genvalue pto_var_ob) {
  return pto_var_ob;
}

pto_cty_point pto_var_p = pto_fun_point__new(3,4);

pto_cty_point pto_var_q =
  *(pto_cty_point*) pto_fun_Main_id((pto_cty__gentype)&pto_inf_point,
                                    (pto_cty__genvalue)&pto_var_p);
```

This code shows the generic `id` method in Pluto is represented by the `pto_fun_Main_id` function in C, where the type parameters and value parameters of the original method have been joined together in the C function's parameters. Likewise, the call to this function joins the type and value arguments to match. In this particular example the function makes no use of the type argument (available via the parameter `pto_var_T`) but it shows how the mechanism of type-passing is achieved.

An important observation is that the coercion of the C local variable `pto_var_p` to a generic value is done by simply taking its address. This is an example of the lazy boxing strategy used by the Pluto compiler, and is discussed further in Section 5.8. The opposite coercion, from a generic value back to a `point` object, is implemented by casting the generic value to a pointer of the expected type and dereferencing it. If the `id` generic method were called with a reference-based type, these two coercions would just be C type casts.

The example above illustrates how objects are coerced to and from generic values, and shows the way in which type information is passed to functions that may need it. Besides individual methods, classes can be generic also: their definition may contain a list of type parameters. When creating an instance of a generic class, type arguments must be supplied to the `new` call. Essentially, the use of `new` can be considered the same as

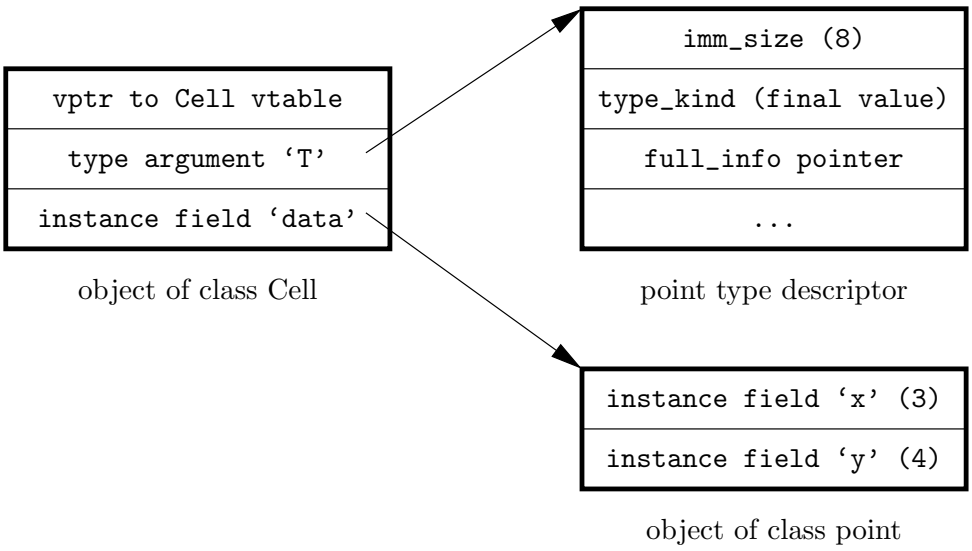

Figure 5.3: Memory layout of generic class instance

invoking a static method, so the translation example above is applicable. For example, in the translation of the `Cell` class from Figure 5.1 there is a generated C function that creates a new instance of the class. The signature of this function is:

```
pto_cty_Cell pto_fun_Cell__new(pto_cty__gentype pto_var_T,
                               pto_cty__genvalue pto_var_initialData);
```

where the parameter types are exactly the same as for the `pto_fun_Main_id` function in the example above. Assuming the definition of the `point` object `p` in the Pluto program above, an expression `new Cell[point](p)` would be translated into the following C code:

```
pto_fun_Cell__new((pto_cty__gentype)&pto_inf_point,
                  (pto_cty__genvalue)&pto_var_p)
```

Here, the arguments are identical to those used in the call to the translated `id` function.

These two examples show how the type arguments are initially created and passed into a generic context, whether at the point of a call to a generic method or the creation of a generic class.

5.6.3 Class type argument storage

In the previous section, the passing of type arguments in a translated `new` expression was shown. However, the need to access this type information may persist for as long as the object exists, not just for the duration of its construction. For example, instance methods may require access to a type descriptor in order to perform one of the operations described in Section 5.7. To support this, the Pluto translation scheme stores the type arguments as part of the object itself[1]. The instance layout for a generic class contains an extra

[1] In this respect, the storage of type arguments as object fields is reminiscent of the *virtual types* research[112] that supports genericity using type variables as object attributes. This is explicit at the source level, whereas Pluto uses the technique for implementation, but the similarity remains.

field to hold each type argument that was passed to the `new` call that created it. These synthesised fields are named using an integer slot number, and appear in the C code with names such as `pto_fld__tp1` for the first type parameter, and so on.

Taking the example creation expression `new Cell[point](p)` used in the previous section, Figure 5.3 shows the layout of the `Cell` object and the other data it refers to. As for all reference-based objects, there is a vptr field. The `data` instance field is a generic value, and so is represented by a pointer to the actual underlying object. The other field (shown in the middle of the object) is a pointer to a type descriptor: it is the type argument that was used to create the object. These genvalue and gentype fields complement each other: one points to an object, while the other describes its layout. In some classes, such as Arrays, there may be several generic values that are described by one gentype field.

The Pluto compiler inserts these extra instance fields directly after the vptr, and before those instance fields that were actually declared in the class. During construction of a new object, the gentype fields are initialised first because they may be referenced during the initialisation code for normal instance fields. As these extra fields form part of the layout of a class instance, they will also be present in all subclasses.

This approach to the storage of type descriptors is a trade-off between extra space in each object compared to a more complex and potentially slower technique that factors the information into separate storage. Such alternatives are compared in [59], which explores the use of dynamically-created vtable/type descriptors that contain the type arguments. The Pluto design described in this dissertation fixes the number of type descriptors at compile-time, thereby eliminating the need for run-time support. As a consequence, objects of a generic class must keep track of their type arguments by means other than through the vptr. Nevertheless, alternative implementation techniques could be used by Pluto: some are considered in Section 8.2.

5.6.4 Type argument aliasing

As discussed in Section 3.5.2, the layout of a subclass instance builds upon its superclass by appending any new instance fields to the layout. This includes any extra instance fields created to store new type arguments that were introduced by the subclass. However, it is common for a generic class to be extended by another generic class: the `BackupCell` class in Figure 5.1 is an example. Although that class defines a type parameter `X`, there is no need to create an extra field in the instance layout for `BackupCell` objects as the inherited field from its superclass `Cell` can be used. This reuse is possible because the type parameter `X` is really just an alias for the `T` type parameter defined by its superclass, `Cell`.

The Pluto compiler uses a straightforward rule to determine whether a type parameter is an alias for one in its superclass, by checking the type arguments used in the `extends` clause of a class. If a type argument is the name of one of the class's own type parameters, it can then be treated as an alias for the corresponding type parameter in the superclass. The type parameter will be able to share the same instance field slot as that used by the superclass (which may in turn be sharing the slot with one defined by its superclass). For

example, in the definition of the `BackupCell` class, the type name in its `extends` clause is `Cell[X]`. The one and only type argument is `X` which is the name of a type parameter, so the instance field slot used by `Cell`'s first (and also only) type parameter can be reused for `X`.

This aliasing and reuse of type argument storage slots helps to reduce the amount of extra space required for objects of generic classes. For class hierarchies where one generic class extends another and simply passes up the name of one of its type parameters as a type argument, reuse prevents redundant storage of duplicate information. It is also important for value classes that are generic, since value subclasses are unable to add extra instance fields (for reasons explained in Section 4.5.1).

5.6.5 Eliminating unused type arguments

In several common uses of genericity, especially with unbounded type parameters, the passing and storage of type arguments is unnecessary. For example, a generic `Pair` class would just store two generic values (pointers) as instance fields. Within its constructor it would use the supplied type arguments to ensure that stack-allocated data is boxed (as described in Section 5.8). However, after construction it would have no further need for the type arguments, assuming the class is simply a container for opaque data and just returns its generic values when asked. In such situations, storage for type arguments within objects is never used and could be removed.

A possible extension to the analysis done in the Pluto compiler would be to detect classes that do not make use of their type arguments, and omit the extra instance fields. This would not affect subclasses that define new code making use of generic values, since their field layout could add the necessary instance field to store the now-required type argument. As described in this dissertation however, the Pluto compiler does not implement such an optimisation, although Section 8.2 includes discussion on alternatives to the storage of type arguments.

5.7 Use of run-time type information

Previous sections have shown how type arguments are created, passed and stored, thereby ensuring that type descriptors can be accessed at run-time. This section describes situations where the run-time type information (RTTI) is actually used, with references to the Pluto code listed in Figure 5.4 to illustrate each.

The most straightforward use of RTTI is when a generic class or method uses the name of one of its type parameters as a type argument to another generic context, such as a method call or `new` expression. For example, the `asArray` method of the `Container` class passes the class type parameter `T` as a type argument in its call to the `makeSingletonArray` method. In the generated C code, the type argument is found by accessing the extra instance field of the current object — the translated code for that function call would be like this:

```
pto_fun_Factory_makeSingletonArray(pto_var_this->pto_fld__tp1,
                                   pto_var_this->pto_fld_element)
```

```
abstract class Printable() {
  abstract method toString() : String;
}

final class Container[T <: Printable](e:T) {
  field element : T = e;
  method asArray() : Array[T] {
    return Factory.makeSingletonArray[T](this.element);
  }
  method asPrintable() : Printable {
    let p : Printable = this.element;
    return p;
  }
  method print() {
    let s : String = this.element.toString();
    String.println(s);
  }
}

abstract class Factory() {
  static method makeSingletonArray[X](e:X) : Array[X] {
    let a : Array[X] = new Array[X](1);
    a.set(0, e);
    return a;
  }
}
```

Figure 5.4: Pluto program needing run-time type information

The `makeSingletonArray` method in turn passes its type parameter `X` as a type argument to the creation of an `Array` object in the `new` expression. In that situation, the type argument is accessible in the C code as a function parameter, so the translation of the expression `new Array[X](1)` is:

```
pto_fun_Array__new(pto_var_X, 1)
```

These examples show the two ways in which type arguments are extracted when needed, even though they are only used for subsequent type-passing in these situations. The operations described below build on this by dereferencing type arguments to read information from the type descriptors they point to.

5.7.1 Dynamic coercions

The coercions to and from generic values described in Section 5.4 do not require any use of RTTI, since they operate in a non-generic context where the actual type of the object is known at compile-time. However, some coercions of generic to non-generic values take place in a context where the actual object type is not known at compile-time. To support these operations, known as *dynamic coercions*, use of RTTI is needed. In Pluto-generated C code, these coercions have the name `dynFromInvGeneric`.

For example, in the `asPrintable` method of the `Container` class, the generic instance field `element` is assigned to a variable of type `Printable`. This is permitted in the Pluto type system because the generic value's type is `T` which is a type parameter bounded by `Printable`. To implement this assignment, a coercion is needed that produces an object of type `Printable` from the generic value. If the actual type argument used to create the `Container` object was a reference or non-final value class, then the coercion can simply use a C type cast. However, if the type argument was a final value class, the object pointed to by the generic value will not have a vptr field, so a more complex coercion is needed (in this case using the `value_copy` utility described in Section 5.7.3). This decision cannot be made at compile-time, so the RTTI is consulted to determine the nature of the type argument dynamically. One of the arguments to the coercion macro is the necessary type argument:

```
pto_cty_Printable pto_var_p =
  pto_cvt_Printable_dynFromInvGeneric(pto_var_this->pto_fld_element,
                                      pto_var_this->pto_fld__tp1);
```

The macro examines the `type_kind` field of the type descriptor to see whether the type argument is a final value class or not, and applies either a `value_copy` or a cast, respectively.

A similar approach is used when the type parameter bound is an interface type. In this situation, the actual type argument could be either a class or an interface type, so the behaviour of the coercion operation is guided at run-time by the value of the `type_kind` field.

5.7.2 Method calls on generic receivers

The major reason for using bounded type parameters, as described in Section 5.2.1, is to enable a generic class or method to make method calls with a generic value as the receiver. To implement these calls, the generated C code must locate the appropriate function to call and pass the receiver in the expected format. This is the standard approach used for the translation of any dynamically-dispatched method call.

In the Pluto-generated code, these calls are handled by first coercing the generic value and storing it in a temporary variable whose type is the bound of the corresponding type parameter. Then the usual dynamic dispatch mechanism detailed in Section 3.5.6 is applied, using this temporary variable as the receiver. For example, in the `print` method of the `Container` class there is a call to the `toString` method, using the generic field `this.element` as the receiver. The generated C code for this call first uses a dynamic coercion of the generic value to type `Printable`, then makes an indirect function call. The coercion operation is identical to that explained in the previous section.

5.7.3 Object copying utilities

There are a small number of situations where objects must be copied in a context where their size is not known at compile-time. To implement these operations, various entries in

the RTTI are used to control the copying. The following C functions are provided by the Pluto library to implement these copying utilities:

`value_copy` This function was mentioned in the discussion of dynamic coercions, as it is used when a final value object must be converted into its full form, i.e. with a vptr field at the head of the object. The arguments to this function are a genvalue and a gentype: the former points to a final value object, while the latter provides the RTTI needed to manipulate it. To store the new full value object, a heap allocation is made whose size is enough for a vptr field plus the amount obtained from the `imm_size` entry in the type descriptor. The object is copied into this new storage, and the `full_info` entry used as the vptr. The pointer returned by this function can then be used to obtain an object of the type required by the dynamic coercion operation, with a C cast and possibly a dereferencing operation if the eventual type is a value class. Note that this function is only used for dynamic coercions — Section 4.6.3 explained how final value class objects are coerced when compile-time information is available.

`generic_copy` The need for this function and the next one is due to the use of lazy boxing, which is explained in Section 5.8. This function returns a generic value that is guaranteed to point to an object stored on the heap. Like the `value_copy` function, its arguments are a genvalue and a gentype. If the actual object pointed to by the generic value argument is an instance of a reference class, then no further action is needed and the generic value argument is returned. Otherwise, the generic value could be a pointer to a stack-allocated object so a boxing operation is performed. Heap space is allocated and the object is copied into this new storage, using the `imm_size` entry to determine the object's size. The pointer to this boxed object is then returned.

`abstract_copy` This function is similar in purpose and operation to the previous one, except that it deals with a pointer representing an abstract class instead of a generic value. No gentype argument is needed for this function, since the supplied object pointer always refers to an object that has a vptr field: the type descriptor can be obtained from that.

An unfortunate limitation of the `generic_copy` and `abstract_copy` functions is that they cannot detect when their generic value argument already points to a heap-allocated object. When dealing with a value class object, the functions assume that it may be stack-allocated and use a boxing operation on it. In certain circumstances, this conservative behaviour can result in unnecessary copying and memory usage, as witnessed in the test results: see Section 7.5 for a discussion of this issue.

5.8 Lazy versus eager boxing

The term *boxing* is commonly used to refer to the action of copying an object stored on the stack into heap storage. For reasons listed in Section 4.1.1, use of stack allocation is preferable to heap storage. In addition, the act of boxing costs time to copy the object, and uses extra memory for the copy. One of the optimisation techniques used by the Pluto compiler is to postpone this boxing of objects in certain circumstances — in effect using a lazy strategy instead of an eager one. The results shown in Section 7.4.4 demonstrate the benefits of this.

The potential advantage of using lazy boxing is that in some situations the boxing operation is never needed, so an eager strategy would have caused unnecessary time and memory usage. For example, the generic `Main.id` and `Main.minimumByHash` methods defined in Figure 5.2 operate on generic values, and both can safely work with values that are pointers to stack-allocated objects. The following sections describe how this safe operation is ensured by the Pluto compiler.

5.8.1 Stack pointer safety

Some of the type coercion operations described in this work involve handling a value class object through a pointer. For example, Section 4.6.3 explains that a value class object is coerced to the type of an abstract class by taking its address. This is because a variable of abstract class type is represented as a pointer to an object. Similarly, the coercion of a value class object to a generic value (covered in Section 5.4) involves taking the address of the object.

Although some value class objects are stored on the heap (e.g. those that are instance fields of heap-allocated objects) others are stack-allocated, including local variables and method parameters. Taking the address of a stack-allocated object can be a potential problem if the object no longer exists when its address is used. For example, the parameters and local variables of a C function only exist while that function is being executed. If the address of one of these objects were to be returned or stored for access after the function call has completed, it would point to a non-existent object. The common term for this situation is a *dangling pointer*. To prevent them from occurring, the Pluto compiler adheres to the following two safety rules for the C code it generates:

1. A pointer to a stack-allocated object cannot be stored as an instance field in any object. Therefore, all instance fields that are represented with a C pointer will point to a heap-allocated object.

2. A pointer to a stack-allocated object cannot be stored in a stack frame that is older than the one in which the object is stored.

These rules ensure that a pointer is always safe, meaning it does not outlive the object that it points to. Similar restrictions are made and formalised in [47], which explains how a core fragment of the Microsoft IL is type checked to prevent dangling pointers.

5.8.2 Boxing operations

In order to avoid breaking the safety rules for pointers into the stack, the Pluto compiler inserts boxing operations at those places where a violation would otherwise occur.

There are two sources of unsafe pointers, and both must be replaced by operations that produce a safe pointer instead. The first source is an existing pointer that may be unsafe: these can be replaced using one of the copy utilities described in Section 5.7.3. The second source is type coercions that involve taking the address of a stack-allocated object: these are termed *lazy boxing* operations. In the Pluto-generated C code, there are also variations of these coercions with the suffix `Heap`, e.g. `pto_cvt_point_toInvGenericHeap`. These

are *eager boxing* versions that do not take the address of stack-allocated objects, and hence produce pointers that are safe.

To ensure safety of pointers to the stack, the Pluto compiler places the necessary copying or coercion operations in two contexts:

- When initialising or updating the value of an instance field whose type is generic or abstract. If the new value is already a pointer, the `generic_copy` or `abstract_copy` function is applied, as appropriate. These operations return a pointer that is guaranteed to point to a heap-allocated object. Otherwise, if the new value is a stack-allocated object, the `toInvGenericHeap` or `toAbstractHeap` coercion is applied.
- Returning a value from a method whose declared return type is generic or abstract. The same copying function or heap coercion is applied as in the previous situation. However, the Pluto compiler omits a copying operation if it can determine that the existing pointer does not refer to an object in the current stack frame.

5.8.3 Analysis for return value boxing

As stated above, the storage of pointers inside objects as instance fields must ensure that they do not point into the stack. The Pluto compiler inserts a call to one of the copy utility functions to ensure this. However, when returning a pointer value from a function such copying is not always necessary. Only when the pointer may contain the address of an object in the current stack frame (a *local* object) is a boxed copy required. The Pluto compiler uses the following information to decide whether a pointer may contain the address of a local object:

- An incoming pointer value (supplied to a function as a parameter) does not hold the address of a local object. Pluto-generated C code uses `const` pointers for function parameter types, so the parameter's value cannot be changed.
- A pointer obtained as the result of a function call could not hold the address of a local object, unless any of the call arguments (including the receiver object, if applicable) are pointers to local objects.

A boxing copy operation only needs to be applied when returning a pointer that is not covered by either of these invariants. The intent of this is to prevent the address of a local object from being passed back to an older stack frame, which would violate pointer safety.

For example, the C function produced by translation of the `Main.id` generic function is shown in Section 5.6.2. The return value is a pointer, but no copying is required because it is an incoming pointer value: the parameter `pto_var_ob`. Use of lazy boxing means that calls to this function can pass the address of a local object as an argument. These two factors means that the `id` method is efficient: no unnecessary boxing or copying is used.

5.9 Example generated code

As in previous chapters, some example generated C code is presented to show the implementation of the techniques discussed throughout the chapter. In this case, the example

used is the `Container` class defined in the Pluto code of Figure 5.4. The generated code is listed below, showing the files generated for the `Container` class only:

```
/* Header file 'pto_hd1_Container.h' – generated code */
/*

final class Container[T <: Printable[]](e : T[]) extends Object[] implements []

 field element : T[]
 original_final method asArray[]() : Array[T[]]
 original_final method asPrintable[]() : Printable[]
 original_final method print[]() : void[]

*/
#ifndef PTO_DEF_Container
#define PTO_DEF_Container

#include "pluto.h"

/* Typedefs for this class */
typedef struct pto_ins_Container* pto_typ_Container;
typedef struct pto_ins_Container*const pto_cty_Container;

/* Other types used by this one */
#include "pto_hd1_Object.h"
#include "pto_hd1_Printable.h"
#include "pto_hd1_String.h"
#include "pto_hd1_Factory.h"
#include "pto_hd1_Array.h"

/* Class info type */
typedef const struct pto_cls_Container* pto_tti_Container;

/* Class instance (object) structure */
struct pto_ins_Container {
   pto_tti_Container vptr;
   pto_typ__gentype pto_fld__tp1;
   pto_typ__genvalue pto_fld_element;
};

/* Class info structure/vtable */
struct pto_cls_Container {
   const unsigned short imm_size;
   const short type_kind;
   pto_tti_Container full_info;
   itable (*lookup_itable)(pto_cty__gentype itable_tag);
};

/* Declaration of type info instance(s) */
extern const struct pto_cls_Container pto_inf_Container;

/* Declaration of interface views */
```

```
/* Coercion macros to and from other types */
#define pto_cvt_Container_fromThis ((pto_cty_Container)pto_var_this)
#define pto_cvt_Container_toReference(CLASS,VALUE) ((pto_cty_ ## CLASS)VALUE)
#define pto_cvt_Container_toAbstract(CLASS,VALUE) ((pto_cty_ ## CLASS)VALUE)
#define pto_cvt_Container_toAbstractHeap(CLASS,VALUE) ((pto_cty_ ## CLASS)VALUE)

#define pto_cvt_Container_toInvGeneric(VALUE) ((pto_cty__genvalue)VALUE)
#define pto_cvt_Container_toVarGeneric(VALUE) ((pto_cty__genvalue)VALUE)
#define pto_cvt_Container_toInvGenericHeap(VALUE) ((pto_cty__genvalue)VALUE)
#define pto_cvt_Container_toVarGenericHeap(VALUE) ((pto_cty__genvalue)VALUE)
#define pto_cvt_Container_fromInvGeneric(VALUE) ((pto_cty_Container)VALUE)
#define pto_cvt_Container_fromVarGeneric(VALUE) ((pto_cty_Container)VALUE)
#define pto_cvt_Container_dynFromInvGeneric(VALUE,TYPEARG)
((pto_cty_Container)VALUE)
#define pto_cvt_Container_dynFromVarGeneric(VALUE,TYPEARG)
pto_cvt_Container_fromVarGeneric(VALUE)
#define pto_cvt_Container_toIfc(INTERFACE,OBJ) ((pto_cty_ ## INTERFACE) { (pto_tti_
## INTERFACE)&pto_ivu_Container_ ## INTERFACE, OBJ })
#define pto_cvt_Container_toInterface(INTERFACE,VALUE)
pto_cvt_Container_toIfc(INTERFACE,pto_cvt_Container_toAbstract(Object,VALUE))
#define pto_cvt_Container_toInterfaceHeap(INTERFACE,VALUE)
pto_cvt_Container_toIfc(INTERFACE,pto_cvt_Container_toAbstractHeap(Object,VALUE))

/* Aliases for functions + globals */

/* Function prototypes */

/* Inline function prototypes */
static inline pto_cty_Array pto_fun_Container_asArray(pto_cty_Container pto_var_this);
static inline pto_cty_Printable pto_fun_Container_asPrintable(pto_cty_Container
pto_var_this);
static inline pto_cty_void pto_fun_Container_print(pto_cty_Container pto_var_this);

/* Vtable entries */

/* Static initialiser function prototype */
extern void pto_ini_Container(void);

/* Constructor function prototype */
static inline void pto_fun_Container__con(pto_cty_Object pto_var_this, pto_cty__gentype
pto_var_T, pto_cty__genvalue pto_var_e);

/* New instance function prototype */
static inline pto_cty_Container pto_fun_Container__new(pto_cty__gentype pto_var_T,
pto_cty__genvalue pto_var_e);

/* Global variable declarations */

#endif
```

```
/* Inline header file 'pto_hd2_Container.h' – generated code */
#ifndef PTO_IDEF_Container
#define PTO_IDEF_Container

/* Other inline code used by this type */
#include "pto_hd2_Object.h"
#include "pto_hd2_Printable.h"
#include "pto_hd2_String.h"
#include "pto_hd2_Factory.h"
#include "pto_hd2_Array.h"

/* The class constructor function */
static inline void pto_fun_Container__con(pto_cty_Object pto_var_this, pto_cty__gentype
pto_var_T, pto_cty__genvalue pto_var_e) {
  { }
  pto_cvt_Container_fromThis->pto_fld__tp1 = pto_var_T;
  pto_cvt_Container_fromThis->pto_fld_element =
  generic_copy((pto_typ__genvalue)pto_var_e, (pto_cty__gentype)pto_var_T);
}

/* New instance creation function */
static inline pto_cty_Container pto_fun_Container__new(pto_cty__gentype pto_var_T,
pto_cty__genvalue pto_var_e) {
  pto_cty_Container nu = malloc(sizeof(struct pto_ins_Container));

  nu->vptr = &pto_inf_Container;
  pto_fun_Container__con((pto_cty_Object)nu, pto_var_T, pto_var_e);
  return nu;
}

/* Inline function definitions */
static inline pto_cty_Array pto_fun_Container_asArray(pto_cty_Container pto_var_this) {
  { pto_cty__genvalue pto_var__tmp1 = pto_var_this->pto_fld_element;
  { return
  pto_fun_Factory_makeSingletonArray((pto_cty__gentype)pto_var_this->pto_fld__tp1,
  pto_var__tmp1); }}
}

static inline pto_cty_Printable pto_fun_Container_asPrintable(pto_cty_Container pto_var_this)
{
  { pto_cty_Printable pto_var_p =
  pto_cvt_Printable_dynFromInvGeneric(pto_var_this->pto_fld_element,
  pto_var_this->pto_fld__tp1);
  { return pto_var_p; }}
}

static inline pto_cty_void pto_fun_Container_print(pto_cty_Container pto_var_this) {
  { pto_cty__genvalue pto_var__tmp2 = pto_var_this->pto_fld_element;
  { pto_cty_Printable pto_var__tmp5 = pto_cvt_Printable_dynFromInvGeneric(pto_var__tmp2,
  pto_var_this->pto_fld__tp1);
  { pto_cty_String pto_var_s =
  pto_var__tmp5->vptr->pto_fnp_toString(pto_cvt_Printable_toAbstract(Object,
  pto_var__tmp5));
```

```
  { pto_fun_String_println(pto_var_s); }}}}
}

#endif
```

```
/* Source file 'pto_src_Container.c' – generated code */
#include "pto_hd1_Container.h"
#include "pto_hd2_Container.h"

#define pto_ilu_Container default_lookup_itable

/* Definition of type info instance(s) */
const struct pto_cls_Container pto_inf_Container = {
  sizeof(pto_cty_Container),
  TYPEKIND_ABSTRACT_OR_REF,
  &pto_inf_Container,
  pto_ilu_Container
};

/* Global variable definitions */

/* Static initialiser flag and function */
static int need_init = 1;
void pto_ini_Container(void) {
  if (need_init) {
    need_init = 0;
    pto_ini_Object();
    pto_ini_Printable();
    pto_ini_String();
    pto_ini_Factory();
    pto_ini_Array();
  }
}

/* Function definitions */
```

5.10 Summary

This chapter describes the support for genericity in the Pluto language, and shows how it is implemented by the compiler. This includes discussion of the use of run-time type information (RTTI) and type-passing to handle value types with genericity. Type-passing allows the combination of genericity and value types to be implemented efficiently — generic features need not be restricted for use by reference-based types only.

Chapter 6

Third addition: variance

In this chapter, the support and implementation of variance in Pluto is detailed. As explained in the background Section 2.7, variance increases the flexibility of the type system by relaxing certain rules about the compatibility of types.

6.1 Overview

There are two main areas of the Pluto type system that allow variance:

1. Generic class and interface type parameters can be annotated with a variance indicator. These control the allowable subtyping relationships among different uses of a generic type. For example, a generic `List` class with a type parameter marked as covariant would enable the type `List[Circle]` to be a subtype of `List[Shape]`, if `Circle` is a subtype of `Shape`. Without variance, this potentially useful subtyping relationship would not be permitted.

2. When an instance method overrides an inherited method, the two signatures do not have to match exactly. For example, a method declared to return a `Shape` can be overridden by one that specialises the return type to `Circle`. This facility is also used when an inherited method with generic types in its signature is overridden by a method using non-generic types — a common situation when subclassing generic types.

In both these cases, the safety of the type system is retained by making certain restrictions on the use of variance: these are discussed in Sections 6.3 and 6.4.

Although the focus of the work in this dissertation is the efficient implementation of value types and genericity, there are a number of reasons why support for variance is also included:

- Most of the benefits of variance are associated with genericity: in effect, introducing variance is a generalisation of the genericity design in Pluto. Some languages already have variance as part of their genericity support[27, 72], so including it in Pluto shows that the design and implementation techniques described in this work are applicable to the more general case.

```
interface Collection[+T] {
  abstract method size() : int;
  abstract method get(index:int) : T;
}

final value class List[E](size:int) implements Collection[E] {
  field elements : Array[E] = new Array[E](size);
  method set(index:int, ob:E) {
    this.elements.set(index, ob);
  }
  method size() : int {
    return this.elements.size();
  }
  method get(index:int) : E {
    return this.elements # index;
  }
}
```

Figure 6.1: Pluto types using variant type parameter

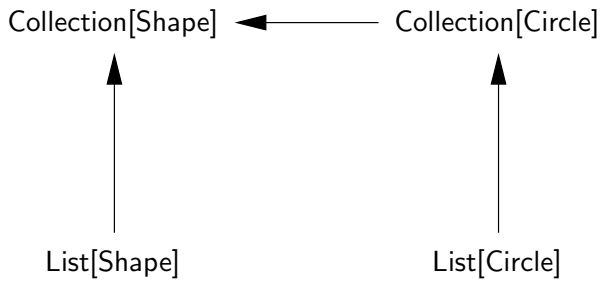

Figure 6.2: List and Collection subtyping relationships

- As discussed in Section 6.4.1, common uses of genericity involve method signature variance anyway, so generalised support can be provided easily. This contributes to the aims (stated in Section 2.8) of using expressive power to avoid subverting the type system or introducing run-time checks.
- Supporting variance in the language can have consequences for the implementation, especially with the existing combination of value types and genericity. Section 6.5 explains how the Pluto compiler deals with variance, and the implications for code generation.

6.2 Type parameter variance

In the discussion of genericity in Chapter 5, all of the type parameters were invariant. This means that, when considering whether one type is a subtype of another, type arguments must be identical. For example, the type parameter `E` in the `List` class of Figure 6.1 is invariant: different uses of the generic `List` class have no subtyping relationship. Assuming that `Circle` is a subtype of `Shape`, the diagram in Figure 6.2 shows no arrow between `List[Circle]` and `List[Shape]`. So an object of the former cannot be assigned to a

variable whose type is the latter. In some situations, this invariant behaviour restricts useful subtyping of generic types, thereby imposing unnecessary limitations.

To provide flexibility, the Pluto language allows an optional variance annotation in the definition of each type parameter. These specify the variance behaviour of a class or interface type parameter, either covariant `+` or contravariant `-`. If no annotation is present, the type parameter is invariant. The `Collection` interface defines a covariant type parameter `T`, which enables a subtyping relationship between different uses of the interface. As the horizontal arrow in Figure 6.2 shows, `Collection[Circle]` is a valid subtype of `Collection[Shape]`.

The Pluto type checker uses the variance of the type parameter to establish whether one use of a generic type is a subtype[1] of another use, e.g. whether

```
GenericType[X] <: GenericType[Y]
```

for any given type arguments `X` and `Y`. The three cases are as follows:

invariant The type `X` must be exactly the same as `Y`, i.e. `X == Y`.

covariant (+) Type `X` must be a subtype of `Y`, i.e. `X <: Y`.

contravariant (-) Type `Y` must be a subtype of `X`, i.e. `Y <: X`.

This process extends to more than one type parameter: each pair of type arguments is compared using the variance of the corresponding type parameter.

Together with subclassing, the use of variance allows flexibility in subtyping. The diagram in Figure 6.2 shows the relationships between various uses of the generic types defined in Figure 6.1. Vertical arrows show subtyping due to inheritance, and horizontal arrows show subtyping due to variance. Because the `List` class is declared to implement the `Collection` interface, the type `List[Circle]` is a subtype of `Collection[Circle]` and then also a subtype of `Collection[Shape]` due to variance.

Note that the direction of the horizontal arrow is determined by the variance of the type parameter. If a contravariant annotation had been used for the `Collection` interface's type parameter, the arrow would be reversed. In general, the choice of variance annotation is controlled by the desired subtyping behaviour for the generic type, subject to the restrictions discussed below.

6.3 Restrictions to maintain type safety

As highlighted in Section 2.7, the issue of variance in a type system can be controversial. Eiffel's conformance (subtyping) relation, and Java's handling of array subtyping are examples of variance where static type safety has been given up, usually requiring run-time checks. The support for variance in Pluto is statically type safe, which means that some restrictions are placed on the use of variance to avoid introducing checks that cannot be made at compile-time. These restrictions are based on type system theory, such as [1, 18, 19], and similar approaches in other languages[16, 27].

[1] the symbol `<:` is used here to mean "is a subtype of", similar to its use in the Pluto syntax for type parameter bounds

6.3.1 Variant types in method signatures

The rules for method signature overriding discussed in Section 2.7 (and below, in Section 6.4) show that type safety would be compromised if return types could be generalised (widened) or parameter types specialised (narrowed). To ensure the safe use of types, there are two restrictions on the use of variant type parameters in method signatures:

1. The name of a covariant type parameter cannot appear in a method parameter type.
2. The name of a contravariant type parameter cannot appear in a method return type.

These rules apply to the use of a type parameter as a type name in itself, or as a type argument. For example, the covariant type parameter `T` in the `Collection` interface of Figure 6.1 is safely used as the return type of the `get` method. However, it would be a violation of the type rules for that interface to declare a method with a parameter of type `T` or `Array[T]` etc. This explains why the `Collection` interface cannot include a `set` method. The `List` class is able to define such a method because its type parameter is invariant. Hence, there is a trade-off between extra functionality (unrestricted use of type parameters in method signatures) and loss of subtyping flexibility (due to invariance).

6.3.2 Variant types in fields

Besides methods, a variant type parameter may also be used in the type of an instance field defined in a Pluto class. To ensure that field access and update (which could be viewed as equivalent to `get`/`set` methods) is type safe, instance fields are only visible to methods defined within the class in which that field is defined or inherited. Using C++ terminology, all instance fields have protected visibility. The Pluto compiler checks an instance field usage such as `obj.fieldName` to ensure that the class in which the code appears (i.e. the static type of the `this` variable) is a subtype of `obj`'s static type.

6.3.3 Variance subtyping rules

Since the name of a type parameter may be used as a type argument, the safe use of variance must be ensured in these situations. For example, the `List` class in Figure 6.1 uses its type parameter named `E` as a type argument in its `implements` clause. In this case `E` is an invariant type parameter, and is passed as a type argument to match the covariant type parameter `T` of the `Collection` interface. The type safety rules for variance matching are given in [1, page 110], and can be summarised as follows:

- If the type argument is an invariant type, it always matches regardless of the variance of the type parameter.
- Otherwise, the variance of the type argument must be the same as the variance of the type parameter.

Therefore the example given above is type safe, due to the first rule. The Pluto compiler applies this check to all type arguments, treating those that are not the name of a type parameter as being invariant.

6.3.4 Method type parameter variance

The reason for allowing variance annotations in class and interface type parameters is to increase the flexibility of subtyping relationships, as shown in Figure 6.2. However, as explained in Section 5.2.2, individual methods can define their own type parameters. Use of variance annotations would not provide any benefit for method type parameters, so they must be invariant in the Pluto type system.

6.4 Method signature variance

The other area of support for variance in Pluto is the rules for method overriding. As explained in Section 2.7, an overriding method can safely specialise the return type and generalise any parameter types. For example, given a class hierarchy and method signatures as listed in Figure 6.3, the method `Sub.m` can override `Super.m`. In the method diagram, `Super.m` is shown with bold lines and `Sub.m` is shown with thin lines. This shows that the overriding is type safe, because `Sub.m` can be substituted for `Super.m` using only upcasts. The method parameter can be upcast from type `B` to type `A`, and the return type `C` can be upcast to type `B`.

This example shows the general rules for allowable method overriding in the Pluto type system: return types can be specialised and parameter types can be generalised. Although genericity was not used in the example code, the rule for a method's type parameters is that the bounds can be generalised in the overriding method. This ensures that any type argument that would be acceptable to the original method signature is also acceptable to the overriding method.

6.4.1 Generic types in signatures

Inheritance of methods that contain a generic type in their signature also involves variance. For example, the Pluto program in Figure 5.1 on page 82 defines a generic `Comparable` interface, whose `==` method parameter is generic. The `point` class is defined as implementing this interface, with `point` as a type argument. Therefore, the class inherits (and provides an implementation for) an `==` method signature where the parameter type is now `point` instead of a generic value.

This signature variance on inheritance is based on the type arguments used by classes and interfaces in their definition. An overriding method is type checked by comparison with the inherited signatures, using the normal variance flexibility described in Section 6.4. In the example above, the overriding method in the `point` class happens to have an identical signature to the inherited one. Nevertheless, it could have used any supertype of `point` as is permitted by the rule for generalisation of parameter types.

The significance of this discussion is that variance can occur when method signatures with generic types are inherited. This means that the Pluto compiler must account for the change, regardless of whether general variance of method signatures is permitted. For example, the Pluto-generated C code must support the invocation of an `==` method call on `point` objects through variables of either `Comparable[point]` or just `point` type. This implementation issue is covered in Section 6.5.

```
class A() { }

class B() extends A() { }

class C() extends B() { }

class Super() {
   method m(arg:B) : B {
      return arg;
   }
}

class Sub() extends Super() {
   method m(arg:A) : C {
      return new C();
   }
}
```

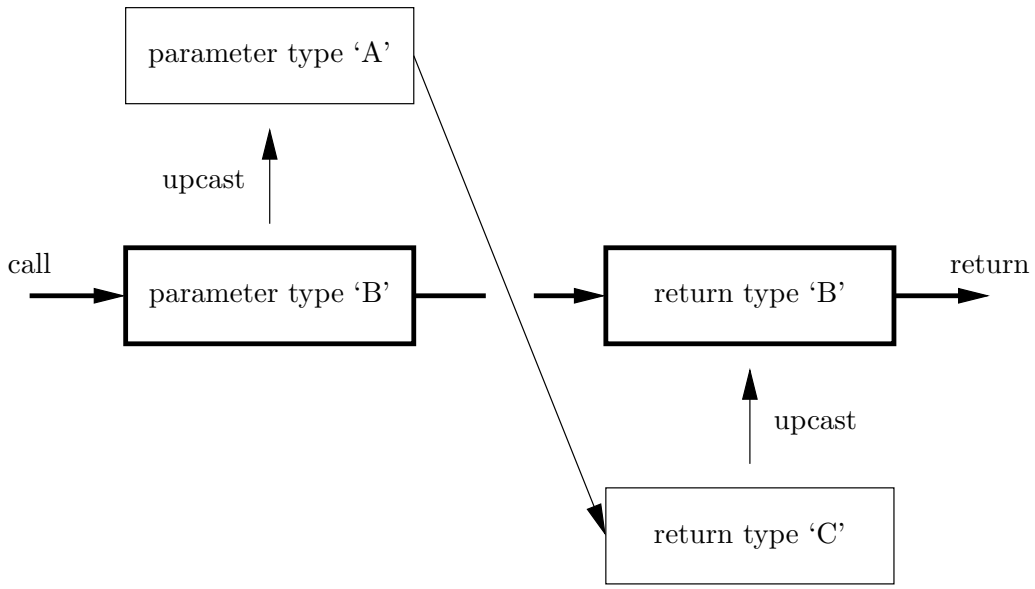

Figure 6.3: Pluto classes using method signature variance

6.4.2 Interface method signatures

In the Pluto type system, only methods defined in classes can use variance on overriding. The reason for this is due to the possible multiple inheritance of method signatures, and to avoid significant complications in the Pluto code generation (see Section 6.5 below). Classes can implement multiple interfaces that in turn can extend multiple superinterfaces, so there may be join points in the type hierarchy where more than one method of a given name is inherited. At these junctions, the Pluto type checker ensures that all of the methods with a given name have identical signatures: no variance is permitted. Nevertheless, once a method signature has been inherited by a class, the class can override the method using the variance rules as before.

These two restrictions (interfaces cannot change the signature of an inherited method, and multiple inheritance of signatures must match exactly) are made to simplify the implementation of variance by the Pluto compiler. The aim is to ensure that variance only occurs along the single inheritance path of the class hierarchy, enabling the implementation to use wrapper functions as described in Section 6.5. Despite this compromise, the Pluto type checker could in principle by extended to deal with multiple inheritance of differing signatures by selecting the most general one.

6.5 Implications for code generation

This section details how the back end of the Pluto compiler is affected by the presence of variance in the language. The use of variant type parameters only affects the coercion of objects to generic type, as explained in Section 6.5.1 below. Implementation of method signature variance involves wrapper functions and extra vtable entries, which are discussed in Sections 6.5.2 and 6.5.3 respectively.

6.5.1 Object coercions to and from variant generic types

The coercion operations described in Section 5.4 show how objects are converted to generic values when passed into a generic context (i.e. where they are handled through a variable whose type is the name of a type parameter). The operations detailed in that section apply when the type parameter in question is invariant, hence the mnemonic `Inv` in the coercion names. When the type parameters are variant however, an alternative set of coercions are used: these are similarly named but with `Var` instead of `Inv`.

For all kinds of type except final value classes, these variant coercions are identical to their invariant counterparts. The coercions of final value class objects to and from variant generic values differ in that the object is first converted to its full-value form. This ensures that generic values of variant type always point to an object with a vptr field at its head.

The need for this property is due to the flexibility introduced by variance. For example, when using a generic collection type whose type parameter is invariant (such as the `List` class in Figure 6.1) the compile-time type information can be used to predict the actual type of object stored in the collection. A `List[Circle]` can never be assigned to a variable whose type is `List[Shape]`, so the `get` method must return a generic value that points to a `Circle` object. With a variant type parameter, this prediction cannot be made: a

`Collection[Shape]` may actually refer to a `Collection[Circle]`, so the `get` method returns a pointer whose compile-time type is only a `Shape`. This causes no problems for most objects, as they have a vptr and are all represented in a compatible way. However, for final value objects this is not the case, so the Pluto compiler ensures that they are stored in their full form when handled as variant generic values.

The following code gives an example of why this special handling of final value objects is used by the Pluto implementation:

```
value class Val () { ... }

final value class FVal() extends Val () { ... }

let lfv : List [FVal] = new List[FVal](1);
lfv . set (0, new FVal());

let cv: Collection [Val] = lfv ;
let v:Val = cv.get (0);
```

In this code, a `List[FVal]` object is accessed through a variable of type `Collection[Val]` to retrieve an element. Since the result of this is a `Val` object, it is expected to have a vptr field. However, the actual collection is a `List[FVal]` whose elements are `FVal` objects. If these were stored in their headless form (as would be the case for an invariant type parameter) the result of the `get` operation would be not be suitable for storage in a `Val` variable. Consequently, the `FVal` object is coerced into its full form in the `set` method call above, using a `pto_cvt_FVal_toVarGeneric` coercion macro in the generated C code.

Final value objects are stored in their full form in any situation where use of variance could result in them being accessed as some other type. To ensure this, the Pluto compiler uses the variant coercion whenever the corresponding type parameter is itself variant, or is used as an alias for one that is. As explained in Section 5.6.4, a class or interface may use one of its type parameters as a type argument in its superclass, implements or superinterface clause. For example, the `List` class in Figure 6.1 uses its invariant type parameter as a type argument in the implements clause, where the corresponding `Collection` type parameter is covariant. Therefore, objects coerced to generic values for use by the `List` class will use the variant version of the coercion, because a `List` can be used as a `Collection` (as demonstrated in the code above).

Related to this topic, Section 8.2 considers a design alternative for final value objects, whereby they are always stored in their full form — even when the associated type parameter is invariant.

Another design alternative that could be considered is to convert final value objects into their full form lazily, i.e. only when they are accessed as another type due to variance. This approach contrasts with the eager coercion described in this section, where the conversion is done as soon as a final value object is passed into a variant generic context. For example, considering the code discussed above, the `List[FVal]` container could store its elements in their headless form (as they would be for an invariant generic context) and the coercion of an element to its full form would be applied only when it is accessed as a `Val` type,

as in the last line of the code above. The benefit of this is that some coercions of final value objects to their full form could be eliminated when variance is not actually used. One consequence of such a design would be that certain coercions of variant generic values to a non-generic type would require examination of run-time type information (RTTI), to account for the possibility that a generic value points to a headless object. In this respect, the `fromVarGeneric` coercions would need to operate similar to their dynamic counterparts, the `dynFromVarGeneric` coercions. In the current Pluto implementation, coercions from generic to non-generic type do not require any use of RTTI.

This design alternative could be worth exploring, since one of the aims of the present work is that features such as variance should only be paid for when used. In addition, another important aim is to ensure that the cost of certain language features is minimised for the common cases. In this respect, the design alternative just described would be beneficial for the (perhaps more common) code that uses a generic container such as `List[FVal]` and does not access it as a `Collection[Val]` type.

6.5.2 Wrapper functions

As first explained in Section 3.5.5, each method defined in a Pluto program is translated into a C function in the generated code. In addition, an entry in the class vtable is made for each new method signature that may require dynamic dispatch. The discussion in that section assumed that method signatures were invariant, and hence the C function signature in the vtable would be identical to the signatures of all the actual function implementations. Because of this, the vtable instance for a class could simply store the address of the function defined or inherited in that class. With variance of method signatures this straightforward mapping is not sufficient, as the C function signature is no longer identical to the function signature specified by the vtable entry.

For example, in the generated C code for the `Sub` class in Figure 6.3 there will be the following function definition that represents the `m` method:

```
pto_cty_C pto_fun_Sub_m(pto_cty_Object pto_var_this,
                        pto_cty_A pto_var_arg) {
  return pto_fun_C__new();
}
```

However, the vtable for the `Sub` class contains an entry for the `m` method that was inherited from its superclass. Since the vtable for `Sub` must be compatible with the layout of its superclass, the entry for `m` has the same signature as it had in `Super`, namely:

```
pto_cty_B (*pto_fnp_m)(pto_cty_Object pto_var_this,
                       pto_cty_B pto_var_arg)
```

Consequently, the address of the function `pto_fun_Sub_m` cannot be stored in that `Sub` vtable entry, because the signatures do not match. The Pluto compiler resolves this mismatch by generating *wrapper* functions[2] that have the signature expected by the vtable entry. The body of these wrappers calls the defined function (such as `pto_fun_Sub_m`

[2]Similar to the *trampoline procedures* used to implement the Theta object model[80]

above) after coercing its arguments, then coerces the result. In effect, it implements the upcasting operations shown in Figure 6.3, thereby adapting the defined function so that it can be used via the vtable. The wrapper used for the example given above is:

```
pto_cty_B pto_vte_Sub_m(pto_cty_Object pto_var_this,
                        pto_cty_B pto_var_arg) {

  return (pto_cty_B)pto_fun_Sub_m(pto_var_this,
                                  (pto_cty_A)pto_var_arg);
}
```

As can be seen, the wrapper simply forwards its argument (suitably coerced) to the defined function, then coerces the return value. The signature of this wrapper function is identical to the vtable entry, so its address is stored in the `Sub` vtable instance.

The Pluto compiler only creates wrapper functions when method signature variance is used. Although the use of a wrapper incurs some overhead, gcc is often able to inline the call to the defined function if it is small. When the coercions are simple C type casts, there is no run-time effect so the wrapper function code is then essentially identical to the defined function. Also, as explained in Section 3.5.6, the Pluto compiler makes use of static dispatch to implement some method calls. In these situations, the actual function defined by the class is called directly: wrapper functions are only used when a dynamically-dispatched call is made.

6.5.3 New vtable entries

The purpose of wrapper functions, as described in the previous section, is to adapt a function signature so that it can be used via dynamic dispatch. This is essential to ensure that the correct method implementation is called, since each class can only provide one implementation for a method of a given name. The example given previously showed that, for code such as:

```
let s:Super = new Sub();

let b:B = s.m(new B());
```

the correct implementation of the `m` method (that in class `Sub`) is invoked, by using a wrapper to adapt it to the expected signature.

In the case of the code just given, the static type of the receiver is `Super`. So the expected signature of `m` is the one known at that point in the class hierarchy, i.e. a parameter and return both of type `B`. However, if the static type was `Sub`, then the expected signature of a dynamically-dispatched call to method `m` would be the one with a parameter of type `A` and a return of type `C`. Therefore, the vtable for class `Sub` needs to introduce a new entry for this altered signature. To ensure unique naming, the Pluto compiler generates a derived name for these overloaded vtable entries, by prefixing the method name with an integer to distinguish it from others. For example, the vtable for the `Sub` class will contain these two entries:

```
pto_cty_B (*pto_fnp_m)(pto_cty_Object pto_var_this,
                       pto_cty_B pto_var_arg);

pto_cty_C (*pto_fnp_1m)(pto_cty_Object pto_var_this,
                        pto_cty_A pto_var_arg);
```

The first is the one inherited from `Super`, whereas the second represents the new signature for method `m` introduced by the `Sub` class. By the usual vtable rules, it will appear in all subclasses of `Sub` and is used when a dynamically-dispatched call is made on a receiver whose static type is `Sub`.

Even though a class may have multiple vtable entries for a given method name, such as those listed above for the `Sub` class, there is still only one implementation of that method in each class. Therefore, wrappers are created to adapt the C function to the signatures of all but one of these vtable entries. The last entry will have the same signature as the function, so a wrapper is not needed and the function's address can be stored directly. For example, the vtable instance for the `Sub` class will have the address of a wrapper stored in the `pto_fnp_m` entry, and the address of the actual function stored in the `pto_fnp_1m` entry.

Like the wrapper functions, these new vtable entries are only created when method signature variance is used. Furthermore, the Pluto compiler avoids the introduction of a new vtable entry when the new method definition is declared `final`, because dynamic dispatch is not necessary and the entry would never be used.

6.6 Summary

This chapter describes how variance is supported in the Pluto language, through type parameter annotations and by relaxing the rules of method signature matching when overriding. The aims of this support are to increase the expressive power of genericity by enabling more subtyping relationships, and to examine how this flexibility affects the task of generating code. The discussion of code generation, in Section 6.5, shows that the combination of variance, genericity and value types can be implemented in a straightforward manner with minimal run-time overhead.

Chapter 7

Evaluation and results

The purpose of the work presented in this dissertation is to support the central thesis, namely: "An object-oriented language with value types and genericity can be implemented efficiently using type-passing". Chapters 3–6 describe the Pluto language (which supports value types and genericity), and the compiler which implements this language using type-passing. In order to verify the thesis, the efficiency of this implementation must be demonstrated. This chapter details how the Pluto compiler was evaluated using a suite of test programs, and provides analysis of the results obtained. A concluding summary discusses how the results meet the test objectives and hence support the thesis.

7.1 Test objectives

Perhaps the most fundamental purpose of writing the Pluto compiler is to show that the design and implementation schemes outlined in the previous chapters have been successfully applied to produce a complete working compiler. As stated above however, the purpose of the testing process is then to establish the efficiency of this implementation of the Pluto language — more specifically, in terms of executable size, memory usage and running time of the generated code. The definition of what is an efficient implementation is naturally subjective, so the following list defines those qualities that can be expected of an implementation of value types and genericity. The extent to which the implementation of Pluto is efficient should be considered with respect to these objectives:

- Use of simple types such as integers and floating-point values in a monomorphic context should not result in significant performance penalties when compared with a language such as C++ that supports these types directly as primitives. Put simply, adding two integers together should not incur unnecessary overhead.
- More generally than the previous objective, any costs associated with using abstraction mechanisms such as polymorphism or genericity should only be paid for when used. Monomorphic code should not incur any unnecessary overhead.
- A generic class such as a container, used with a particular element type, should not impose a large overhead compared to a non-generic version that is hardwired for

Test Program	Description	Page
gen_compare	A comparator method for generic values	140
gen_tree	Binary search tree of generic comparable values	141
word_tree	Binary search tree of fixed-type comparable values	145
matrix_mult	Matrix multiplication with complex numbers	148
ackermann	Computes the Ackermann function	154
prime_sieve	Finds prime numbers using Sieve of Eratosthenes	156
gen_stack	Exercises a generic stack with different element types	159
gen_comptr_tree	Generic binary search tree using comparator object	167
word_comptr_tree	Fixed-type binary search tree using comparator object	176

Table 7.1: Summary of test programs

that element type. This is related to the previous objective in that polymorphic code will typically have some overhead when compared to a monomorphic version — the essence of this objective is that the cost of genericity should be minimised.

- Even when manipulated in a generic context, use of a value class for small data types should provide appreciable benefits in terms of memory usage and running time compared with the same class declared as a reference type. If this objective were not met, there would be little benefit to having value types in addition to reference types.

- For a given program, the size of the generated executable file should be comparable with that generated for an equivalent implementation of the program in a language such as C++ or Eiffel that offers similar features. In other words, Pluto-generated code should not be unnecessarily large or suffer from code bloat.

7.2 Choice of test programs

The programs used in these tests are briefly summarised in Table 7.1. Full details of each program and its source code can be found at the indicated page in Appendix B.

One of the most immediate difficulties when evaluating or comparing the implementation of a new language is that there is no established body of source code that can be used for benchmarking. For this reason, the test programs used in this chapter are so-called *micro-benchmarks* such as the classic ackermann, prime_sieve and matrix_mult programs. Some slightly larger programs, such as the binary search tree variations, were designed specifically to show the handling of value types and genericity. Despite the apparently simple nature of the test programs, they can be viewed as typical of the building blocks of more complex programs. The results presented in this chapter could in principle be applied to larger programs, by considering these smaller unit tests as general indicators.

As one of the aims of testing is to compare the use of value types and reference types, many of the test programs were built with a certain *key class* declared as either a reference, a value or a final value class. The results tables shown later in this chapter contain separate columns for each of these cases: *Ref*, *Val* and *FVal* respectively. Where appropriate, the

Processor	Intel Pentium III 500 MHz
Processor L2 Cache	512 Kilobytes
Motherboard Chipset	Intel 440BX 100 MHz
Physical Memory	384 Megabytes SDRAM
Operating System	Redhat Linux 6.1
Linux Kernel	Version 2.2.12
C/C++ Compiler	GNU gcc Version 2.95.2
Eiffel Compiler	SmallEiffel Version -0.78
Java Compiler	GNU gcj Version 2.95.2
Assembler	GNU as Version 2.9.5
Static Linker	GNU ld Version 2.9.5
Dynamic Linker	Version ld-2.1.2.so
C Library	GNU glibc Version 2.1.2
Garbage Collector	Boehm/Demers Version 5.3
Timing Tool	GNU time Version 1.7

Table 7.2: Test environment details

test program descriptions in Appendix B indicate which class is the key class for that program. To avoid the possibility of dynamic method dispatch being replaced by static dispatch due to use of the `final` keyword in the FVal case, methods were marked `final` where appropriate.

7.3 Methodology

This section details how the testing process was conducted, and how the measurements presented later in the chapter were made.

7.3.1 Test environment

The test programs were all built and run on the same PC. The hardware specification and software environment of this machine is shown in Table 7.2.

The reliance of the Pluto system on using gcc to compile the generated C code means that the quality of gcc itself is a significant factor in the results. At the time of writing, a major new version of gcc (3.0) had just been released. Naturally, the entire set of tests reported here could be run again using gcc 3.0, with potential differences in the areas of executable size and running time particularly.

In addition to the different versions of gcc, there are many possible variations in the level of optimisation, extent of inlining and other options that could be examined. While these comparisons would be interesting in themselves, they are effectively evaluations of gcc itself and so are not central to the work described in this dissertation. For this reason, and to avoid the large number of option combinations, all the tests were conducted using the same set of options (described in the next section).

7.3.2 Building Pluto programs

The Pluto test programs were built by first invoking the Pluto compiler on the source, then compiling and linking the resulting C code using the following options:

```
gcc -O3 -fomit-frame-pointer -s pto_src_*.c -lpluto
```

The `-s` option simply strips symbolic information from the resulting executable, in the same way that the `strip` utility does. The use of `-O3` gives the maximum level of optimisation in gcc, including the inlining of functions. This takes advantage of the definitions of inline functions produced by the Pluto compiler, as discussed in Section 3.7.5. The omission of the frame pointer is possible when compiling the Pluto-generated C code because there are no uses of dynamic allocation within stack frames. This option frees the frame pointer for use in register allocation, reduces code size and improves the running time of the program.

Test programs requiring the use of garbage collection were built by making three additions to the C compilation process:

1. The garbage collector header file `gc.h` was included.
2. All uses of `malloc` were replaced with calls to the garbage collector's allocation function, by declaring a macro in the `pluto.h` header file that is included at the beginning of all Pluto-generated C code compilation. None of the Pluto-generated code uses the `free` function, so no changes were needed for deallocation.
3. The resulting executable was statically linked with the garbage collector library `libgc.a`, by adding the option `-lgc` when invoking gcc.

Some of the utility functions defined in the Pluto library (described in Section 5.7.3) allocate heap memory. Therefore, when building Pluto programs that use garbage collection, the Pluto run-time library was also rebuilt to use it. This is important to ensure that as many heap allocations as possible are managed by the garbage collector.

7.3.3 Building non-Pluto programs

For test programs written in C++, their source was compiled using the following command line:

```
g++ -O3 -fomit-frame-pointer -s code.cpp
```

Eiffel programs were compiled using the command below, which first generates C code then invokes the local C compiler on the result. In the test environment, gcc was used to ensure consistent comparison with Pluto and C++.

```
compile -clean -boost -no_split -no_gc -O3 -fomit-frame-pointer code.e
```

Here, the options `-O3` and `-fomit-frame-pointer` are passed to gcc to compile the C code generated by the SmallEiffel compiler. The compilation process also strips symbols from the resulting executable, so a `-s` option is unnecessary.

The Java test programs were compiled using the GNU native Java compiler (gcj) with the following options:

```
gcj --main=Main -O3 -fomit-frame-pointer -s Main.java
```

It is important to note that this process results in an executable that is dynamically linked to the gcj run-time support libraries for garbage collection and Java standard classes. This should be borne in mind when comparing the cross-language test results, especially with respect to the size of executables.

7.3.4 Measurement techniques

Throughout the results tables shown later in this chapter, there are three primary measurements: executable size, memory usage and running time. The simplest of these to measure is executable size, as it is just the size of the executable file produced by the build process discussed above. As previously mentioned, the executable files are all stripped of symbol information using the `-s` option to gcc.

Measurement of program running time was conducted using the `time` utility, which reports the elapsed time taken to run another program. This utility also reports the amount of CPU time consumed by the process, divided into user space and kernel space execution. The PC used for testing was a lightly-loaded single-user machine, and for all the results reported in this chapter the elapsed time was the same as the amount of CPU time (i.e. the `time` utility reported 99% or 100% CPU usage). Hence the elapsed time is a true reflection of each test program's running time, as no other activities impinged significantly on its execution.

For each test configuration the program was run three times, and the median value of elapsed time chosen. This reduces the likelihood of spurious results due to caching, dynamic linking or other kernel activities. In all cases, the elapsed times for the three runs varied only slightly, with at most 5% difference.

The measurement of program memory usage relied on use of the Linux `/proc` virtual filesystem, which provides access to status information about running processes. For a given process identifier *pid*, the virtual file `/proc/`*pid*`/status` contains memory usage fields for that process's data segment and stack segment, among others. In the results tables of this chapter, the memory usage figures refer to the sum of the data and stack segment sizes. This reflects the amount of heap and stack space used by each program. In all completed test runs, the memory consumption was met from physical RAM — no swap space was needed.

In order to retrieve the process status information for a test program, the Pluto-generated C code was hand-modified with the following extra code inserted just before the end of the `main` function:

```
char cmd[30];
sprintf(cmd, "cat /proc/%d/status", getpid());
system(cmd);
```

The `getpid` function returns the process identifier for the process that called it: the test program itself, in this case. The `system` function is then used to invoke a command which displays the required status information. After recompiling with this addition, the program was run again (with the same input, if applicable) to produce the memory status

information for the test program just before it ends. Note that this extra code was not present when measuring the program's executable size or running time: it was used only when measuring memory usage.

The same memory measurement technique was used for the C++ and SmallEiffel-generated C code. For the Java tests however, there was no C code produced by gcj; a different technique had to be used. The test program Java source was modified with the following code added to the end of the `main` method:

```
System.err.println("Done");
while (true);
```

After recompilation, the program was run and the memory status information gathered once the program had reached the endless loop.

7.4 The tests

The following sections show the results of running the test programs in various configurations and with different problem sizes in some cases. Accompanying each table of results is a commentary that highlights some key observations about the data.

7.4.1 Reference, value and final value objects

The first group of test results (listed in Table 7.3) show the effects of using either a reference class (Ref), a value class (Val) or a final value class (FVal) for the key class in a given program. The test programs used for this group are all those that have a key class.

Garbage collection was not used for any of the test runs listed in this table, hence the large amounts of memory used in many of the tests. The purpose of this is to examine the raw memory allocation total for each program, by isolating them from the effects of the garbage collector. See Section 7.4.2 for test results using garbage collection.

With reference to Table 7.3, there are several detailed observations that can be made:

- In the matrix_mult tests, use of Ref instead of Val or FVal results in a dramatic increase in memory consumption, and a significant slowdown. This can be attributed to the large number of temporary objects created inside the innermost loop of the multiplication routine. With Ref class objects and no garbage collection, the memory usage becomes prohibitive: the test of a 250 square matrix exhausted all the available memory and had to be aborted.

  With Val and FVal for the key class, the memory usage is manageable due to the intermediate objects being stack allocated, with the memory being reused at each iteration of the inner loop. The extra memory consumed by the Val version is caused by the storage of vptr fields in each object, which are omitted in the FVal version. Assuming the `complex` class requires 16 bytes for its two instance fields, and the vptr field takes another 4 bytes, this accounts for the 25% extra storage used by the Val version compared to FVal.

  Running times for the matrix_mult test show that the FVal version is roughly three times faster than for Val, with the Ref version slower still. Interestingly, the running times of the Val and Ref versions are closer than might be expected. The speed gain for FVal over Val is likely to be due to the constructor optimisation for final value classes (described in Section 4.6.5) being exploited for the `complex` class.

Test Program	Executable Size (Kilobytes)			Problem Size	Memory Usage (Kilobytes)			Running Time (seconds)		
	Ref	*Val*	*FVal*		*Ref*	*Val*	*FVal*	*Ref*	*Val*	*FVal*
matrix_mult	5.9	6.1	5.9	80 square	24392	404	332	0.56	0.41	0.13
				150 square	159540	1340	1076	3.93	2.71	0.81
				250 square	fail	3692	2960	fail	14.10	4.97
gen_comptr_tree	7.0	7.2	7.0	32460 words	1352	936	936	3.25	3.32	3.16
word_comptr_tree	6.7	6.8	6.7	32460 words	1352	844	844	3.19	2.95	2.92
gen_compare	4.5	4.6	4.5	1 million	46892	16	23456	0.90	0.62	0.59
				2 million	93768	16	46892	1.82	1.24	1.20
				4 million	187516	16	93768	3.65	2.48	2.42
word_tree	7.5	7.8	7.6	500 clones	35028	34516	34512	2.68	2.59	2.53
				1000 clones	68940	68424	68416	5.23	5.09	4.95
				2000 clones	136768	136248	136232	10.30	10.04	9.78
gen_tree	7.8	8.0	7.8	500 clones	35028	57116	66148	2.81	3.27	3.65
				1000 clones	68940	113580	122608	5.49	6.47	6.97
				2000 clones	136768	226512	235532	10.82	12.64	13.56

Table 7.3: Comparing reference, value and final value types

- The figures for the gen_comptr_tree program and the non-generic word_comptr_tree version give an idea of the performance cost of using a generic data structure compared with a fixed one, especially when the element type is a value type. With a Ref element type, the memory consumption is identical — this is unsurprising as the same representation is used for both. The speed slowdown is marginal for Ref, at only 2%. With Val and FVal element types, the difference is more pronounced: memory consumption increases by 11% for the generic version, and the speed slowdown averages around 10%.

  Viewing the figures for these two programs along the other axis shows how the different kinds of element type affect performance. Memory usage with Val and FVal elements is the same, for both the generic and fixed-type programs. In both programs, use of Ref elements requires more memory than non-Ref: by 44% in the generic case and 60% for the fixed-type program. The latter figure indicates the space-saving advantage of embedding value type fields within their parent object, rather than as distinct separately-allocated objects (an advantage highlighted in Section 4.1.1).

  With regard to running time of the generic program, the figures for each kind of element type vary only slightly, by 2–3%. For the non-generic version of the program, the use of Val and FVal elements shows a distinct improvement over Ref: running times are around 8% faster.

- Comparing the Ref and FVal versions of the gen_compare test program shows a consistent 50% slowdown in running time, and double the memory usage. However, this program is primarily intended for use in the lazy/eager boxing tests (covered in Section 7.4.4), and it is the Val and FVal versions that are of main interest.

  The running times for the Val version are only 3–4% slower than for FVal. However, the most significant observation is that the memory usage for the FVal version is vastly greater than for Val. In addition, the memory use is linear in the problem size, whereas the Val figures are constant. This highlights a significant inefficiency in the Pluto compiler's handling of FVal objects in a generic context. The source of this memory consumption is within the generic `max` method, when one of the generic value arguments is coerced to the type of the bound in order to invoke a method on it. For Val objects, this simply involves a type cast, whereas FVal objects are headless and must be rebuilt with a vptr field. This results in an extra boxing operation, which is the cause of the extra memory consumption for FVal. The summary in Section 7.5 discusses this issue.

- The word_tree test figures show only minor differences between the Ref, Val and FVal versions. Memory usage is very slightly higher for Ref, and running times are broadly similar, with FVal marginally the quickest. An interesting effect can be seen in the gen_tree test program figures, where the memory and running time trends are reversed. Here, the Val and FVal versions use more memory and suffer slowdowns compared to the Ref version. Again, this highlights an inefficiency in the Pluto compiler's handling of non-Ref objects in a generic context — but for different reasons than that described in the previous comment.

  Here, the source of the extra memory consumption is within the cloning of internal tree nodes. A new node is created using the generic value held by the original, but the constructor clones the generic value when its run-time type is Val or FVal because it may be stack-allocated. For Ref data, the cloning is known to be unnecessary and so the generic value is shared. This shows a limitation in the lazy-boxing strategy used by the Pluto compiler: it is overly-conservative in this situation. Making manual

modifications to the generated C code to bypass this redundant copying resulted in memory usage comparable with the Ref version, proving the source of the inefficiency. Again, this issue is discussed in the summary (Section 7.5).

Some general patterns emerge from the figures in this group of tests:

1. The use of Ref for the key classes results in increased memory consumption compared with Val or FVal data, except in the anomalous gen_tree test program.
2. Running times for FVal beat those for Val in all cases except gen_tree. In turn, the running times for Ref are worse than Val in almost all of the tests.
3. The size of executables for a given program are virtually the same whether the key class is Ref or FVal. The Val class executable is only slightly larger, by a few hundred bytes. However, the small size of all the test programs means it is unclear whether this difference is a constant or linear in the program size.
4. Comparing the executable size of a generic test such as gen_tree or gen_comptr_tree against its corresponding non-generic version (word_tree or word_comptr_tree) shows a small overhead for genericity: around 4–5% for these programs.
5. Certain situations involving value types in a generic context give rise to inefficiencies, due to limitations in the Pluto compiler implementation. Possible remedies for these are discussed in the summary (Section 7.5).

7.4.2 Use of garbage collection

The tests in this group are the same set of programs as in the previous group, but with garbage collection built into the executables. In these tests, results were only compiled for one problem size: the middle of the three sizes used in some of the previous group's tests. Again, each program was built with a key class declared as either Ref, Val or FVal. The results of this group of tests are listed in Table 7.4.

The purpose of these tests is to evaluate the cost of using garbage collection (GC), and to see how the observations made in the previous test group carry over to an environment that reclaims dead memory allocations. In most programming language environments, long-running programs incorporate some form of memory deallocation, either explicit or managed automatically. Therefore, this group of tests is more realistic in terms of memory consumption and running time than the previous group of tests that have no deallocation at all.

Some observations about the figures shown in Table 7.4:

- As before, executable size is largely the same regardless of the kind of the key class. By linking with the garbage collector, the size of executables increases by approximately 36 Kilobytes, compared with the figures in the previous group of tests. However, as the test programs are all quite small, this fixed overhead results in a large relative increase in program size. Naturally, for larger programs the extra size of the garbage collector would become less significant.
- For three of the tests, gen_comptr_tree, word_comptr_tree and gen_compare, use of GC has actually improved the running times (by around 4%) as well as lowering the memory consumption.

Test Program	Executable Size (Kilobytes)			Problem Size	Memory Usage (Kilobytes)			Running Time (seconds)		
	Ref	*Val*	*FVal*		*Ref*	*Val*	*FVal*	*Ref*	*Val*	*FVal*
matrix_mult	41.5	41.7	41.5	150 square	3472	2068	1708	7.07	2.79	0.85
gen_comptr_tree	43.4	43.5	43.4	32460 words	704	704	704	3.14	3.18	2.92
word_comptr_tree	43.0	43.0	43.0	32460 words	704	616	616	3.11	2.80	2.83
gen_compare	40.8	40.9	40.8	2 million	180	52	180	1.59	1.24	1.18
word_tree	43.8	44.1	43.9	1000 clones	820	816	660	6.30	5.95	5.81
gen_tree	44.1	44.3	44.1	1000 clones	816	812	1028	6.47	8.80	7.61

Table 7.4: Comparing types, using garbage collection

- In almost all cases, the memory consumption and running times of the Val and FVal versions show improvements over the Ref version. One exception is the gen_tree program which is worse in the Val and FVal versions, for reasons stated in the previous test group section.

- Comparing the generic tree test program versions against their non-generic counterpart shows a small slowdown for using genericity when the element type is Ref (about 2%), and nearly identical memory usage. With Val and FVal element types, the slowdown is more pronounced: 3–14% for the gen_comptr_tree and word_comptr_tree versions, with a 14% increase in memory use for the generic version.

- For the tree test programs (other than the problematic gen_tree) use of FVal element types instead of Ref gives a speedup of 9% on average. In the non-generic versions, memory consumption is 19% higher for Ref, whereas for gen_comptr_tree it is the same. This shows that data structures do not impose overhead when used with value types.

- The matrix_mult test program shows some interesting results, especially when compared with the previous (non-GC) group. In the Ref element type version, the huge memory usage of the non-GC test has been significantly reduced but with the cost of a much slower running time. Profiling shows that this is due to the load placed on the GC routines to reclaim the large number of intermediate result objects. In the Val and FVal versions, the use of GC has hardly affected the running time (slower by about 4%) and has actually increased the memory usage (by 56% on average). This indicates that the matrix_mult program is better off using FVal type elements without GC.

- Extra memory is still consumed by the FVal version of gen_compare compared to the Val version, for the reasons stated in the previous section. In the gen_tree program, the Val version no longer has higher memory usage than the Ref version (as it did without GC) but suffers a slowdown due to the extra GC activity involved. As in the non-GC tests, the FVal version still uses more memory and is slower than the Ref version. The summary in Section 7.5 discusses these problems.

7.4.3 Cross-language comparison

In this group of tests, some Pluto test programs are compared with translations written in other programming languages that support similar features. A selection of five programs were implemented in Pluto, C++, Eiffel and Java (except program gen_comptr_tree for which no Eiffel version was written). In those Pluto programs with a key class, it was declared as an FVal type. Table 7.5 lists the results obtained.

Section 7.3.3 described how the non-Pluto programs were built. None of the Pluto, C++ and Eiffel programs used garbage collection, but the compiled Java programs were dynamically linked with a run-time library providing garbage collection. The gcj Java compiler does not appear to provide any way of avoiding this dependency.

Making comparisons between different programming languages can be problematic due to the many possible variations in compilation, optimisation and level of run-time support. These factors can make it difficult to compare "like with like", and run the risk of producing invalid comparisons. The tests discussed in this section attempt to minimise the effect of these issues in the following ways:

Test Program	Executable Size (Kilobytes)				Problem Size	Memory Usage (Kilobytes)				Running Time (seconds)			
	Pluto	*C++*	*Eiffel*	*Java*		*Pluto*	*C++*	*Eiffel*	*Java*	*Pluto*	*C++*	*Eiffel*	*Java*
ackermann	3.1	3.2	4.2	5.1	(3,9)	196	196	212	240	0.75	0.80	1.20	2.03
					(3,10)	388	388	420	464	3.57	3.74	5.38	8.75
					(3,11)	772	772	836	912	17.64	18.52	28.73	42.53
prime_sieve	4.3	3.3	4.5	5.3	10 x 2M	7832	1960	1972	2320	9.21	5.63	5.45	7.10
matrix_mult	5.9	3.9	6.8	8.9	80 square	332	304	328	1460	0.13	0.12	0.18	1.73
					150 square	1076	1060	1072	3524	0.81	0.93	1.23	13.38
					250 square	2960	2936	2956	8412	4.97	4.82	6.67	66.10
gen_stack	5.5	68.3	6.6	8.7	10000	276	276	272	628	13.85	7.40	13.17	42.19
gen_comptr_tree	7.0	103.5	—	12.3	32460 words	936	400	—	1060	3.16	1.56	—	3.25

Table 7.5: Cross-language comparison

1. The languages being compared offer broadly similar features: all are object-oriented, provide genericity (except Java, which can simulate it) and all make a distinction between value and reference types.

2. The programs used in the tests are small, mostly self-contained examples that can often be translated into each language in a straightforward manner.

3. All the language implementations compared employ variations of the same back end to produce executables: the gcc system. Both Pluto and SmallEiffel generate C code that is compiled by gcc, whereas the C++ and Java programs are compiled with the g++ and gcj tools which are both front ends into the basic gcc infrastructure.

With reference to the figures listed in Table 7.5, the following comments can be made about the results:

- In the ackermann tests, the Pluto code has the smallest executable, the lowest running time and the joint lowest memory usage. This clearly shows that Pluto meets those objectives from Section 7.1 relating to zero overhead for monomorphic and basic arithmetic code.

- The Pluto version of the prime_sieve program uses significantly more memory than the others, because it lacks a small numeric data type. The C++, Eiffel and Java versions use arrays of 8-bit bytes to represent the set of flags, whereas Pluto only has 32-bit integers, resulting in the four-fold increase in memory consumption. Naturally, this could be remedied by adding a `byte` data type to the Pluto standard library, or using bitwise operations to represent several flags in one integer value.

  The relatively slow running time of the Pluto version of this program can be attributed to the use of the Array data type, which is generic in Pluto. The versions in other languages use type-specific arrays, as they are either supported directly in the language (C++ and Java) or expanded in the generated code (Eiffel). The use of generic arrays in Pluto therefore imposes an overhead for access speed and memory usage: this topic is discussed in the summary (Section 7.5).

- For the matrix_mult tests, Pluto compares well with Eiffel: using almost identical amounts of memory, and with smaller executable and faster running time (Eiffel being around 40% slower). The C++ implementation is only marginally faster than Pluto, and uses slightly less memory. The smaller size of the C++ executable is likely to be due to the use of a non-generic procedural implementation rather than a generic object-oriented style. The Java implementation fares badly: it is ten times slower than Eiffel and consumes three times as much memory. This is due to its use of reference types for the `complex` matrix elements, with the resultant performance loss arising from garbage collection activity.

- For the gen_stack and gen_comptr_tree test programs, the C++ executable size is far larger than for the other tests. This difference is due to these two programs making use of standard library facilities, whereas the other tests are self contained programs. Brief investigations with the gen_stack program show that the problem is caused by use of the `iostream` output facilities, and worsened by the version of gcc used. Removing the formatted output statements, e.g. `cerr << "msg"`, reduces the program size to 32.6 Kilobytes. Using gcc version 3.0 instead of version 2.95.2 on this modified program brings the size down to 5.1 Kilobytes. While the issue of C++ program size could be examined in further depth, it is not central to the work presented in this dissertation and is not pursued further.

- The C++ implementations of the generic programs (gen_stack and gen_comptr_tree) show a distinct advantage over Pluto in terms of running time particularly, and memory usage in the case of gen_comptr_tree. This is most likely due to the C++ approach of expanding templates to give non-generic, type-specific code in the generated executable. The Pluto version of the gen_stack program is implemented using its generic Array type, which involves the same slowdown discussed above for the prime_sieve program.

  The faster running time of the C++ version of gen_comptr_tree is mainly due to its use of the standard C++ library `string` class, which implements the string length operation in constant time by caching the text length inside the object as an extra field. In the Pluto version, its length operation is built on top of the C `strlen` function, which takes linear time. Modifying the Pluto version of this program so that Word objects cache their length reduces its running time to 1.89 seconds, without increasing memory consumption.

7.4.4 Eager versus lazy boxing

This final group of tests show the effects of eagerly boxing value type objects when passing them into a generic context, in contrast to the normal lazy boxing used by the Pluto compiler. Each test was conducted with and without garbage collection (GC). For those test programs that have a key class, this was defined as an FVal class. The gen_compare program was tested with both FVal and Val versions of the key class. Testing Ref versions would be irrelevant, as reference type objects are already boxed. The results of this group of tests are shown in Table 7.6.

As detailed in Section 5.8, lazy boxing involves passing pointers to potentially stack-allocated data when value type objects are used in a generic context. In situations where a pointer to a stack-allocated object could be made invalid due to the object's stack frame being destroyed, the object is boxed by copying it to the heap using an operation called `generic_copy` (see Section 5.7.3 for details). This is the normal strategy used by the Pluto compiler, and is shown by the figures listed in columns headed *Lazy*. All previous group tests have been built with lazy boxing.

A simpler approach is to box eagerly, by copying value type objects onto the heap at the point they are passed into a generic context. In order to measure the effect of using eager boxing, the Pluto-generated C code used for the lazy version was systematically modified by replacing uses of the `toInvGeneric` coercion operation with the `toInvGenericHeap` operation. Corresponding replacements were made for the `VarGeneric` coercions. These modifications effectively insert eager boxing operations at the point of coercion to a generic context. The final modification is then to eliminate the now-redundant lazy boxing operations, by removing the calls to `generic_copy`. The columns headed *Eager* in the table show the results for these modified programs.

Examination of the figures in Table 7.6 shows the following:

- Without garbage collection, the memory usage increases dramatically in all test programs except gen_tree. The total figures, which sum the memory usage across all the test programs, show that eager boxing uses three times as much memory as the lazy strategy. With regard to running times, the use of eager boxing causes a significant slowdown in most test programs. Again, the figures for the gen_tree

Test Program	Executable Size (Kilobytes)		Problem Size	Memory Usage (Kilobytes)				Running Time (seconds)			
				No GC		*With GC*		*No GC*		*With GC*	
	No GC	*With GC*		*Lazy*	*Eager*	*Lazy*	*Eager*	*Lazy*	*Eager*	*Lazy*	*Eager*
gen_stack	5.5	41.9	4000	148	250200	392	308	2.23	6.91	2.23	5.23
matrix_mult	5.2	41.5	150 square	1076	2656	1708	2036	0.81	0.83	0.85	0.89
gen_compare (FVal)	4.5	40.8	2 million	46892	140644	180	180	1.20	2.65	1.18	2.45
gen_compare (Val)	4.6	40.9	2 million	16	93768	52	180	1.24	2.92	1.24	2.86
gen_comptr_tree	7.0	43.4	32460 words	936	20960	704	988	3.16	4.05	2.92	3.58
word_tree	7.6	43.9	1000 clones	68416	158636	660	660	4.95	6.79	5.81	8.24
gen_tree	7.8	44.1	1000 clones	122608	77960	1028	816	6.97	5.83	7.61	7.41
			Total:	240092	744824	4724	5168	20.56	29.98	21.84	30.66

Table 7.6: Comparing lazy and eager boxing

program are contrary to the overall trend. Overall, the speedup for using lazy boxing in the non-GC case averages 31% across all the tests. This extra running time cost for eager boxing is due to the large number of heap allocation requests.

- When garbage collection is applied to the test programs, the difference in memory usage between lazy and eager versions is far less pronounced. Some programs use more, some less while others use exactly the same. Overall, memory usage only increases by 9% when eager boxing is used. Nevertheless, there is still a significant improvement in running time: around 29% speedup for using lazy boxing, which can be attributed to reduced GC activity.

- In both GC and non-GC versions, the gen_tree test program shows a drop in memory consumption when eager boxing is used, with a corresponding speedup. This can be explained by recalling the root cause of the problematic memory consumption for gen_tree with value type elements (first discussed above, in Section 7.4.1). In the lazy boxing version, the cloning operation makes redundant copies of the generic value stored in each internal tree node. However, one of the modifications made to the generated C code to produce the eager boxing version is removal of the `generic_copy` operations used to construct the fields of new tree nodes. Thus, the boxing operation is only performed once (eagerly) and no redundant copies made — this results in the eager boxing version using less memory overall. The test summary, in Section 7.5, discusses Pluto compiler implementation issues prompted by this and other results for the gen_tree test program.

7.5 Summary of results

The purpose of the testing and analysis described in this chapter is to provide quantitative data that can be used to evaluate the efficiency of the Pluto compiler implementation. Based on the detailed figures shown, and the comments made in the preceding sections, some general points about the test programs can be summarised:

- The size of Pluto-generated executable files are typically smaller than those produced for equivalent Java and SmallEiffel programs, and are comparable with those generated for C++ implementations of the programs.

- Code using simple types and operations such as integer and floating-point arithmetic is as efficient as that written in C++ or Eiffel. There is no overhead incurred for having types such as `int` and `float` defined as final value classes in Pluto.

- For small immutable data types, use of value classes instead of reference classes results in substantial savings in memory usage, and enables significant speedups. In addition, load on the garbage collector is reduced.

- When using value type objects in a generic context, a lazy boxing strategy gives a 30% speedup compared to the simple eager boxing approach. Lazy boxing provides three-fold memory savings without garbage collection in use, and 9% savings even when it is included.

- A generic container data structure imposes only a 2% slowdown and zero memory overhead compared to a non-generic version, when used with reference type elements. With value type elements, the cost of using genericity is a 10% slowdown and 11% memory increase.

All of the above points show that the Pluto compiler can be considered an efficient implementation of a language with value types and genericity. Nevertheless, the testing process and results show some areas in which the Pluto implementation is not as efficient as it could be:

- Array accessing and updating can be slower than in languages such as C++ and Eiffel which support type-specific arrays directly. This issue is most visible in those test programs that make heavy use of arrays: prime_sieve and gen_stack. The problem is caused by Pluto's use of a single generic Array data type, which means elements are accessed using a scaling factor that is stored in the Array object. As the immediate size of an Array element is actually known at compile-time, the Pluto code generation could be improved to use a constant value for the scaling factor.

- When a generic value is used as the receiver for a method invocation, it must be converted into a form that can be used to locate the method via its vtable, and then be passed as the hidden `this` argument for that method. In the case of final value class objects, an extra boxing operation is used to convert the object into its full value form. As the gen_compare test program shows, this can cause a potentially large increase in memory usage compared to the use of non-final value classes. The reason for this situation is that objects of a final value class are stored in their headless form when used in a generic context. This saves the space that would otherwise be needed for the extra vptr fields, but does introduce the inefficient extra boxing if the object is needed in its full form. A design alternative would be to store the full value instance whenever these objects are converted to a generic value. This strategy is in fact used when the generic type is variant (as detailed in Sections 5.4 and 6.5.1).

- The various test results for the gen_tree program have highlighted one area in which the lazy boxing approach causes worse performance than using simple eager boxing. This problem (and others) could be remedied by improving the boxing analysis that the Pluto compiler uses to decide when a generic value should be copied to the heap. More extensive analysis could distinguish those generic values that might refer to stack-allocated objects from those that definitely do not. This would remove the redundant copying that afflicts the gen_tree test program, for example.

Some possible design variations and alternatives are briefly explored in Section 8.2, such as non-generic arrays and a different handling of final value objects in a generic context.

Chapter 8

Conclusions

This chapter reviews the work presented in this dissertation and summarises the main results of the research. In addition, some design alternatives are considered followed by discussion of possible further work. These suggest ways in which the research could be improved or extended, and highlight some wider applications of run-time type information.

8.1 Summary of contributions

The aim of this work is to support the thesis: that value types and genericity can be implemented efficiently using type-passing. The following specific results have been achieved:

- A language with extensive support for value types, genericity and variance has been designed and implemented, with analysis and test results that demonstrate the efficiency of this implementation.
- The level of support for value types in Pluto shows that they need not be heavily restricted in order to retain efficiency. Value classes exist in the inheritance hierarchy, can implement interfaces and may be generic by defining type parameters.
- Generic classes and interfaces can be used uniformly for all types, whether value or reference. Thus, value types and genericity are fully integrated in the Pluto type system, allowing wide application of generic types.
- Value types used in a monomorphic context do not incur unnecessary overhead: features such as genericity and variance are only paid for by code that actually uses them. Simple arithmetic code in Pluto is as fast as the equivalent in C++ or Eiffel.
- Use of type information passed at run-time enables the effective implementation of genericity when value types are present, without imposing a significant burden on program size, running time or memory usage.
- Adopting a lazy strategy for boxing objects is an important optimisation technique that can be applied to improve performance of generic code. Test results show a 30% speedup together with lower memory usage across a range of test programs.

Together, these results demonstrate that type-passing is indeed a suitable technique for implementing value types and genericity. The success of the Pluto compiler shows that

language support for value types and genericity does not have to be significantly compromised by concerns about implementation efficiency. Furthermore, the translation scheme used by the compiler proves that sophisticated run-time services or virtual machine environments are not necessarily required to support such language features.

8.2 Design variations

The Pluto compiler embodies a variety of design decisions regarding the implementation approach used. These choices are by no means the only ones, but they reflect a trade-off chosen during the original design of the Pluto compiler. The results (summarised in Section 7.5) highlight some areas in which the choice of a design feature gave rise to an undesirable loss of efficiency. In the light of such information, this section reviews some of the design decisions and assesses possible alternatives.

8.2.1 Final value objects in a generic context

The decision to maintain final value objects in their headless form when passed to a generic context had significant consequences for other aspects of the implementation. As usual, this is a trade-off between extra memory use and running time. The headless form is smaller, and coercing to a generic value does not incur the expense of creating the full form of the object with its vptr field. This was the approach taken in the Pluto compiler as described in this work. However, one implication of this decision is that generic code must account for the possibility that generic values can point to objects that may or may not have a vptr field. As a result, extra run-time checks must be made to deal with both cases — these can be seen in the complicated dynamic coercions used for generic values, and are also needed in the generic copying utilities described in Section 5.7.3. Furthermore, the dynamic coercion operation can involve heap allocation to create a full object, producing the the overhead discussed in the results Section 7.5.

A solution to this source of inefficiency would be to ensure that generic values always point to objects that have a vptr field. This would simplify many of the coercions and copying utilities and remove the need for the `full_info` entry in the type descriptors. Perhaps most importantly, it would eliminate the extra heap allocation used in some dynamic coercion operations. This approach is actually used when the generic context is associated with a variant type parameter, as explained in Section 6.5.1. The proposal discussed in this section is to extend this to invariant type parameters as well. Section 6.5.1 also examines the opposite design alternative to that being considered here, where final value objects could be stored in their headless form in a variant generic context.

More speculatively, it may even be possible to eliminate the storage of type arguments as object fields (explained in Section 5.6.3) if all generic values point to objects with vptr fields, as run-time type information can be obtained via each object. Support for reflection, as explored in Section 8.3, might preclude this however.

To implement the alternative design discussed in this section, coercion of final value objects to any generic context would involve first converting them into their full form. The disadvantage of this is clear: storage requirements for final value objects in generic

form is increased. For example, an array of integers would store every integer headed by its own vptr field (which would be the same vptr for all). The current design avoids this by storing just the integer, providing a significant space saving for large arrays. Potential savings for individual objects are less likely, as the granularity of heap-allocated space in typical C library `malloc` implementations means that a minimum size is incurred — eliminating the extra vptr field would typically not affect the actual allocation size.

It would be interesting to run the tests of Chapter 7 again, after modifying the Pluto compiler to use the full form of final value objects in all generic situations. This would show whether the trade-off chosen for final value objects was justified. Ideally, optimising the common case should take priority: the extra boxing inefficiency occurs only in limited situations. By comparing results for the range of test programs, it could be seen which alternative is best overall.

8.2.2 Non-generic arrays

The results comparing Pluto programs against similar code written in C++ and Eiffel (see Section 7.4.3) show that array access in Pluto is not as efficient as in the other languages. Since many other data structures can be built on the foundation of arrays, it is important to provide a good implementation. The primary reason for the efficiency loss is that the array type in Pluto is a generic class defined by the standard library, rather than being built into the language syntax. This means that array element access (either reading or writing) involves offset calculations computed at run-time, where the index is scaled by the element size. Improvements to the Pluto compiler could eliminate these run-time operations by handling arrays as a special case. Since the `Array` class has an invariant type parameter, the Pluto translation can take advantage of compile-time knowledge of the element type.

A radical alternative to Pluto's support for arrays would be to allow non-generic instantiations of a generic class, similar to the effect of C++ templates. For example, a program could request an expansion of the generic array class with a given element type. This would be a selective specialisation, allowing type-specific code to be produced — a manual optimisation, in effect.

8.3 Possible extensions

The Pluto language was designed specifically for use in this work, as a vehicle to demonstrate the implementation techniques detailed in previous chapters. The purpose of the language design is to support the features of interest to this research: value types, genericity and variance. To this end, the language does not provide many other features found in production languages such as access control, multi-threading, nested classes, exceptions or sophisticated control flow constructs. These were omitted because they were not considered essential to the goals of the research.

Beyond the language itself, the Pluto library of predefined classes contains just a minimal set of types needed to support basic numeric operations, simple input/output of strings and to provide an Array container. Naturally, these facilities could be built upon

or extended to give more advanced libraries of reusable utilities.

Despite the restricted set of features embodied in the Pluto system as defined in this dissertation, there are many possible extensions that could be made. Typically, these would be independent of the features already supported and could be added by enhancements to the Pluto compiler. The following sections briefly discuss how some interesting features might be added to the Pluto language, and in particular how they could be implemented using the existing infrastructure. In keeping with the use of type-passing to implement genericity and value types, the features chosen for discussion below are ones that can make use of run-time type information.

8.3.1 Reflection

Object-oriented languages such as Java, C++ and Eiffel provide simple reflective capabilities such as type tests (e.g. `instanceof` in Java) and checked downcasts (the C++ `dynamic_cast` for example). These constructs could be implemented in Pluto by making use of the type descriptors already accessible through an object's vptr field. To support type tests and checked downcasts, each type descriptor could be extended with an extra field that points to a null-terminated array of type descriptor addresses, one for each supertype. As Section 3.5.4 explained, the address of a type descriptor uniquely identifies a particular type. A type test could then be implemented by searching this list at run-time, comparing each supertype descriptor address to find the required one. This process is similar to that employed by the Pluto compiler to implement interface vtable lookup (see Section 3.6).

To fully implement type tests and downcasts, run-time type information about generic types is also needed. For example, testing whether an object is an instance of a generic type such as `List[Circle]` involves checking that the object is a `List`, then checking that its type argument is exactly `Circle` (since the `List` class type parameter is invariant). Implementation of such tests requires access to type arguments at run-time. This is already possible: Section 5.7 showed how the run-time type information infrastructure is used for certain operations. The storage of type arguments within objects provides the type information necessary to implement these operations, and could also be applied to support reflection.

Beyond type tests and checked downcasts, Java supports a powerful general reflection mechanism[106]. Java programs can obtain information about any object's fields and methods, which are themselves represented as ordinary objects. Although more complicated, such facilities could in principle be added to Pluto by using the type descriptors accessible through an object's vptr. Reflection information could be built at run-time with a C function generated by the Pluto compiler for each class or interface. These would be invoked in a similar manner to a dynamically-dispatched method call implementation, via a function pointer entry in each vtable. Naturally, each class or interface need only create their reflection information once, and it can be done lazily to avoid unnecessary memory usage.

8.3.2 Cooperative garbage collection

The Pluto language provides no ability for programmers to explicitly deallocate storage. Stack-based objects are reclaimed automatically, but heap-based objects are not deallocated by Pluto-generated C code. For some of the evaluation tests (detailed in Chapter 7) the Boehm/Demers/Weiser garbage collector library[13] was linked with Pluto-generated C code. This library manages the heap automatically, reclaiming objects stored there when they are no longer needed.

The BDW garbage collector does not require any cooperation from the Pluto compiler or the generated C code because it uses a conservative algorithm. This means that use of the library is extremely simple, as it can be just plugged in, but does introduce the possibility of undetected garbage. Non-conservative garbage collector techniques (such as most of those surveyed in [122]) typically require support from the compiler in order to locate potential pointers to heap objects. The Pluto compiler could provide this information, again using the type descriptors that are accessible through the vptr field of objects. An extra entry could be added to each type descriptor that contains a list of offsets within objects of that type, one for each instance field that is a pointer. This would enable garbage collectors to locate the pointers within objects accurately, rather than using a conservative analysis.

8.3.3 Object serialisation and persistence

In order for objects to be made persistent by saving them on disk, transmitting them over a network or storing them in a database, they must be serialised. Such facilities are provided in Eiffel[72] and Java[107]. This process involves creating a flat representation for a graph of potentially many objects, in a way that the original structure can be rebuilt when the flat form is deserialised. The implementation of serialisation requires similar information to that discussed in the previous sections: reflective information about an object, and the location of pointer fields to determine the object graph structure.

As the serialisation and deserialisation operations are type-specific, the Pluto language could support them by having the compiler generate C functions for each type, to convert an object of that type to and from a serialised form. Like the reflection implementation outlined above, these functions would be dispatched dynamically and will typically call other serialisation functions for their instance fields. Again, the presence of run-time type information is critical to the support of this feature.

8.4 Conclusion

Programming languages are becoming more sophisticated, supporting powerful features for abstraction and enabling the development of reusable code libraries. Higher-level language constructs allow programmers to express their designs in a more intuitive way, while also increasing the ability of compilers to do type checking and perform optimisations. One of the tasks of compiler researchers is to ensure that these language features can be translated into efficient code for execution. In many cases, however, it is the combination of language features that causes difficulty for compiler writers.

The work presented in this dissertation first shows that a language combining three useful and powerful features — user-defined value types, genericity and variance — does indeed present a challenge to efficient implementation. The compilation approach subsequently described in this work demonstrates that passing type information at run-time is a useful way of implementing these language features. Wider application of this run-time type infrastructure should enable compiler writers to improve support for advanced language features.

Appendix A

Grammar for Pluto core language

The abstract syntax shown in Figures A.1 and A.2 summarises the core Pluto language.

Program	::=	(*TypeDefinition*)$^+$
TypeDefinition	::=	*ClassDefinition* | *InterfaceDefinition*
ClassDefinition	::=	[`final`] *ClassKind* `class` *AlphaName* *TypeParameterList* *ValueParameterList* `extends` *TypeName* *CodeBlock* *ValueList* `implements` *TypeNameList* `{` (*ClassMember*)* `}`
ClassKind	::=	`abstract` | `reference` | `value`
InterfaceDefinition	::=	`interface` *AlphaName* *TypeParameterList* `extends` *TypeNameList* `{` (*AbstractMethod*)* `}`
TypeNameList	::=	(*TypeName*)*
TypeName	::=	*AlphaName* `[` *TypeNameList* `]`
TypeParameterList	::=	(*TypeParameter*)*
TypeParameter	::=	[*Variance*] *AlphaName* `<:` *TypeName*
Variance	::=	`+` | `-`
ValueParameterList	::=	(*ValueParameter*)*
ValueParameter	::=	*AlphaName* `:` *TypeName*
ClassMember	::=	*Method* | *AbstractMethod* | *Field*
Method	::=	[`static` | `final`] `method` *MethodName* *TypeParameterList* *ValueParameterList* `:` *TypeName* *CodeBlock*
AbstractMethod	::=	`abstract` `method` *MethodName* *TypeParameterList* *ValueParameterList* `:` *TypeName*
Field	::=	[`static`] `field` *AlphaName* `:` *TypeName* `=` *CodeBlock* *Value*
CodeBlock	::=	`{` (*Declaration*)* `}`

Figure A.1: Pluto core abstract syntax (type-level and member-level)

Declaration	::=	*IfDeclaration* | *WhileDeclaration* | *ReturnDeclaration* | *LetDeclaration* | *VarDeclaration* | *AssignDeclaration*
IfDeclaration	::=	`if` *Value CodeBlock* `else` *CodeBlock*
WhileDeclaration	::=	`while` *CodeBlock Value CodeBlock*
ReturnDeclaration	::=	`return` [*Value*]
LetDeclaration	::=	`let` *AlphaName* `:` *TypeName* `=` *Binding*
VarDeclaration	::=	`var` *AlphaName* `:` *TypeName*
AssignDeclaration	::=	`var` [*Value* `.`] *AlphaName* `=` *Binding*
Binding	::=	*StaticMethodCall* | *InstanceMethodCall* | *SuperMethodCall* | *StaticFieldAccess* | *InstanceFieldAccess* | *ObjectCreation* | *Value*
StaticMethodCall	::=	*AlphaName* `.` *MethodName TypeNameList ValueList*
InstanceMethodCall	::=	*Value* `.` *MethodName TypeNameList ValueList*
SuperMethodCall	::=	`super` `.` *MethodName TypeNameList ValueList*
StaticFieldAccess	::=	*AlphaName* `.` *AlphaName*
InstanceFieldAccess	::=	*Value* `.` *AlphaName*
ObjectCreation	::=	`new` *TypeName ValueList*
MethodName	::=	*AlphaName* | *SymbolicName*
ValueList	::=	(*Value*)*
Value	::=	*AlphaName* | *SpecialName* | *StringLiteral* | *NumericLiteral*
SpecialName	::=	`true` | `false` | `this`
StringLiteral	::=	`" ... "`
NumericLiteral	::=	(`0` | `1` | `2 ... 9`)$^+$ [`.` (`0` | `1` | `2 ... 9`)$^+$]
AlphaName	::=	(`a` | `b` | `c ... z` | `A` | `B` | `C ... Z`)$^+$
SymbolicName	::=	(`!` | `#` | `$` \| `%` \| `&` \| `*` \| `+` \| `-` \| `_` \| `.` \| `:` \| `/` \| `<` \| `=` \| `>` \| `\` \| `^` \| `\|` \| `~` \| `?` \| `@`)$^+$

Figure A.2: Pluto core abstract syntax (low-level)

Appendix B

Test programs

This chapter contains the actual program source code used for the testing in Chapter 7. All of the Pluto, C++ and Java programs, and some of the Eiffel programs, were written from scratch. The Eiffel programs for ackermann, prime_sieve and matrix_mult are based on original source code obtained from a benchmarking site[11], modified to match the requirements of the tests.

B.1 Program gen_compare: generic comparator method

In this test program, a method with a type parameter is used to implement a generic maximum function. The type parameter bound specifies that only classes that are comparable with themselves can be used. A `complex` number class is also defined, and the `max` function called repeatedly with two complex number objects.

The purpose of this program is to test the overhead of passing value and reference types into a generic context, and to examine the cost of invoking methods on values of a generic type. In addition, this program is used in the comparison between lazy and eager boxing.

The `complex` class is the key class for this program. The problem size is the number of calls to the generic method.

B.1.1 Pluto source

```
interface Comparable[X] {
  abstract method <(other:X) : boolean;
}

final value class complex(re:float, im:float) implements Comparable[complex] {
  field re : float = re;
  field im : float = im;
  final method <(other:complex) : boolean {
    return !(other.re < this.re || other.im < this.im);
  }
}
```

```
abstract class Main() {
  static method run() : int {
    let PROBLEMSIZE : int = 2000000;
    var count : int = 0;
    while (count < PROBLEMSIZE) {
      let c1 : complex = new complex(1.0, 0.0);
      let c2 : complex = new complex(2.0, 0.0);
      let c3 : complex = Main.max[complex](c1, c2);
      var count = count + 1;
    }
    return 0;
  }
  static method max[X <: Comparable[X]](one:X, other:X) : X {
    if (one < other) {
      return other;
    } else {
      return one;
    }
  }
}
```

B.2 Program gen_tree: generic binary search tree

In this program a binary search tree data structure is defined, using generic classes to allow the tree to store elements of any type that can be compared with itself. A file of text strings is read in, and each is used to construct an object of class `word`. The words are added to a binary search tree object, where the element type is `word`. This tree is then repeatedly cloned, and each copy accepts three *visitor*[43] objects that perform various operations involving traversal of the tree.

This test program is designed to show the cost of abstractions such as method calls requiring dynamic dispatch, generic data structures that do not use arrays, and the use of value type objects stored in a generic data structure. In addition, this program can be compared with the fixed-type version described below, in Section B.3, to examine the possible overhead of using genericity.

The input to this program is a file containing 32460 unsorted words, one per line. The words are not unique: there may be several duplicates. The file is only read once.

The `word` class is the key class for this program. The problem size is the number of times the original tree is cloned and operated on.

B.2.1 Pluto source

```
interface Comparable[X] {
  abstract method <(other:X) : boolean;
}

abstract class Visitor[X]() {
```

```
  abstract method visit(d:X);
}

final class Tree[X <: Comparable[X]]() {
  field root : Node[X] = new EmptyNode[X]();
  method add(e:X) {
    var this.root = this.root.insert(e);
  }
  method accept(v:Visitor[X]) {
    this.root.accept(v);
  }
  method depth() : int {
    return this.root.depth();
  }
  method clone() : Tree[X] {
    let copy : Tree[X] = new Tree[X]();
    var copy.root = this.root.clone();
    return copy;
  }
}

abstract class Node[T <: Comparable[T]]() {
  abstract method clone() : Node[T];
  abstract method insert(e:T) : Node[T];
  abstract method accept(v:Visitor[T]);
  abstract method depth() : int;
}

class EmptyNode[T <: Comparable[T]]() extends Node[T]() {
  method clone() : Node[T] {
    return this;
  }
  method insert(e:T) : Node[T] {
    return new InternalNode[T](e, this, this);
  }
  method accept(v:Visitor[T]) { }
  method depth() : int {
    return 0;
  }
}

class InternalNode[T <: Comparable[T]](d:T, l:Node[T], r:Node[T]) extends Node[T]() {
  field data : T = d;
  field left : Node[T] = l;
  field right : Node[T] = r;
  method clone() : Node[T] {
    return new InternalNode[T](this.data, this.left.clone(), this.right.clone());
  }
  method insert(e:T) : Node[T] {
    if (e < this.data) {
      var this.left = this.left.insert(e);
```

```
      } else if (this.data < e) {
        var this.right = this.right.insert(e);
      }
      return this;
    }
    method accept(v:Visitor[T]) {
      this.left.accept(v);
      v.visit(this.data);
      this.right.accept(v);
    }
    method depth() : int {
      let ld : int = this.left.depth();
      let rd : int = this.right.depth();
      if (ld < rd) {
        return rd + 1;
      } else {
        return ld + 1;
      }
    }
  }

  final value class Word(s:String) implements Comparable[Word] {
    field str : String = s;
    final method <(other:Word) : boolean {
      return this.str < other.str;
    }
    final method $() : String {
      return this.str;
    }
    final method length() : int {
      return this.str.length() − 1;
    }
  }

  class WordPrinter() extends Visitor[Word]() {
    method visit(w:Word) {
      String.print("Got word: ");
      String.print($w);
    }
  }

  class LongWordFinder() extends Visitor[Word]() {
    field longest : Word = new Word("");
    method visit(w:Word) {
      if (this.longest.length() < w.length()) {
        var this.longest = w;
      }
    }
    method getLongest() : Word {
      return this.longest;
    }
```

```
}

class WordLengthCounter(m:int) extends Visitor[Word]() {
  field totals : Array[int] = WordLengthCounter.init(m);
  static method init(max:int) : Array[int] {
    let a : Array[int] = new Array[int](max + 1);
    var i : int = 0;
    while (i < max) {
      a.set(i, 0);
      var i = i + 1;
    }
    return a;
  }
  method visit(w:Word) {
    let entry : int = w.length();
    this.totals.set(entry, (this.totals # entry) + 1);
  }
  method printTotals() {
    var i : int = 1;
    while (i < this.totals.size()) {
      String.print("Words of length ");
      String.print($i);
      String.print(": ");
      String.println($(this.totals # i));
      var i = i + 1;
    }
  }
}

abstract class Main() {
  static method run() : int {
    let PROBLEMSIZE : int = 1000;
    let dictionary : Tree[Word] = new Tree[Word]();
    var line : String = String.readln();
    while (line.length() != 0) {
      dictionary.add(new Word(line));
      var line = String.readln();
    }
    var repeat : int = 0;
    while (repeat < PROBLEMSIZE) {
      let copy : Tree[Word] = dictionary.clone();
      let lwf : LongWordFinder = new LongWordFinder();
      copy.accept(lwf);
      let longest : Word = lwf.getLongest();
      let wlc : WordLengthCounter = new WordLengthCounter(longest.length());
      copy.accept(wlc);
      let depth : int = copy.depth();
      var repeat = repeat + 1;
    }
    return 0;
  }
```

```
}
```

B.3 Program word_tree: fixed-type binary search tree

This program is a conversion of the generic binary search tree program described in Section B.2, where the element type is fixed as `word` instead of being generic.

The input to this program is the same as for the generic version.

Like the generic version, the `word` class is the key class for this program. The problem size is the number of times the original tree is cloned and operated on.

B.3.1 Pluto source

```
interface Comparable[X] {
  abstract method <(other:X) : boolean;
}

abstract class Visitor[X]() {
  abstract method visit(d:X);
}

final class WordTree() {
  field root : WordNode = new EmptyWordNode();
  method add(e:Word) {
    var this.root = this.root.insert(e);
  }
  method accept(v:Visitor[Word]) {
    this.root.accept(v);
  }
  method depth() : int {
    return this.root.depth();
  }
  method clone() : WordTree {
    let copy : WordTree = new WordTree();
    var copy.root = this.root.clone();
    return copy;
  }
}

abstract class WordNode() {
  abstract method clone() : WordNode;
  abstract method insert(e:Word) : WordNode;
  abstract method accept(v:Visitor[Word]);
  abstract method depth() : int;
}

class EmptyWordNode() extends WordNode() {
  method clone() : WordNode {
    return this;
  }
```

```
  method insert(e:Word) : WordNode {
    return new InternalWordNode(e, this, this);
  }
  method accept(v:Visitor[Word]) { }
  method depth() : int {
    return 0;
  }
}

class InternalWordNode(d:Word, l:WordNode, r:WordNode) extends WordNode() {
  field data : Word = d;
  field left : WordNode = l;
  field right : WordNode = r;
  method clone() : WordNode {
    return new InternalWordNode(this.data, this.left.clone(), this.right.clone());
  }
  method insert(e:Word) : WordNode {
    if (e < this.data) {
      var this.left = this.left.insert(e);
    } else if (this.data < e) {
      var this.right = this.right.insert(e);
    }
    return this;
  }
  method accept(v:Visitor[Word]) {
    this.left.accept(v);
    v.visit(this.data);
    this.right.accept(v);
  }
  method depth() : int {
    let ld : int = this.left.depth();
    let rd : int = this.right.depth();
    if (ld < rd) {
      return rd + 1;
    } else {
      return ld + 1;
    }
  }
}

final value class Word(s:String) implements Comparable[Word] {
  field str : String = s;
  final method <(other:Word) : boolean {
    return this.str < other.str;
  }
  final method $() : String {
    return this.str;
  }
  final method length() : int {
    return this.str.length() - 1;
  }
```

```
}

class WordPrinter() extends Visitor[Word]() {
  method visit(w:Word) {
    String.print("Got word: ");
    String.print($w);
  }
}

class LongWordFinder() extends Visitor[Word]() {
  field longest : Word = new Word("");
  method visit(w:Word) {
    if (this.longest.length() < w.length()) {
      var this.longest = w;
    }
  }
  method getLongest() : Word {
    return this.longest;
  }
}

class WordLengthCounter(m:int) extends Visitor[Word]() {
  field totals : Array[int] = WordLengthCounter.init(m);
  static method init(max:int) : Array[int] {
    let a : Array[int] = new Array[int](max + 1);
    var i : int = 0;
    while (i < max) {
      a.set(i, 0);
      var i = i + 1;
    }
    return a;
  }
  method visit(w:Word) {
    let entry : int = w.length();
    this.totals.set(entry, (this.totals # entry) + 1);
  }
  method printTotals() {
    var i : int = 1;
    while (i < this.totals.size()) {
      String.print("Words of length ");
      String.print($i);
      String.print(": ");
      String.println($(this.totals # i));
      var i = i + 1;
    }
  }
}

abstract class Main() {
  static method run() : int {
    let PROBLEMSIZE : int = 1000;
```

```
    let dictionary : WordTree = new WordTree();
    var line : String = String.readln();
    while (line.length() != 0) {
      dictionary.add(new Word(line));
      var line = String.readln();
    }
    var repeat : int = 0;
    while (repeat < PROBLEMSIZE) {
      let copy : WordTree = dictionary.clone();
      let lwf : LongWordFinder = new LongWordFinder();
      copy.accept(lwf);
      let longest : Word = lwf.getLongest();
      let wlc : WordLengthCounter = new WordLengthCounter(longest.length());
      copy.accept(wlc);
      let depth : int = copy.depth();
      var repeat = repeat + 1;
    }
    return 0;
  }
}
```

B.4 Program matrix_mult: matrix multiplication with complex numbers

This program defines a generic `matrix` class and a `complex` number class, then implements the multiplication of two matrices to produce a third. The purpose of this program is to simulate a typical scientific subroutine, and to compare the relative efficiency of using a value type versus a reference type for `complex`. In order to examine the cost of using generic classes against a procedural style, the C++ implementation is written using basic C features.

The key class for this test program is `complex`. The problem size is the number of elements in the matrix, which is square.

Note that the Pluto `float` type is double precision, so the corresponding type is used in the C++, Eiffel and Java versions of this program.

B.4.1 Pluto source

```
final value class complex(re:float, im:float) {
  static field zero : complex = new complex(0.0, 0.0);
  field re : float = re;
  field im : float = im;
  final method re() : float {
    return this.re;
  }
  final method im() : float {
    return this.im;
  }
  final method +(other:complex) : complex {
    return new complex(this.re + other.re, this.im + other.im);
```

```
  }
  final method *(other:complex) : complex {
    return new complex((this.re * other.re) − (this.im * other.im), (this.re * other.im) +
    (this.im * other.re));
  }
  final method $() : String {
    return $this.re + "i" + $this.im;
  }
}

final class Matrix[T](x:int, y:int) {
  field xdim : int = x;
  field ydim : int = y;
  field data : Array[T] = new Array[T](x * y);
  method get(x:int, y:int) : T {
    return this.data # ((x * this.xdim) + y);
  }
  method set(x:int, y:int, d:T) {
    this.data.set((x * this.xdim) + y, d);
  }
}

abstract class Main() {
  static field PROBLEMSIZE : int = 150;
  static method run() : int {
    let m1 : Matrix[complex] = Main.makeMatrix();
    let m2 : Matrix[complex] = Main.makeMatrix();
    let m3 : Matrix[complex] = new Matrix[complex](Main.PROBLEMSIZE,
    Main.PROBLEMSIZE);
    var i : int = 0;
    while (i != Main.PROBLEMSIZE) {
      var j : int = 0;
      while (j != Main.PROBLEMSIZE) {
        var sum : complex = complex.zero;
        var k : int = 0;
        while (k != Main.PROBLEMSIZE) {
          var sum = sum + (m1.get(i, k) * (m2.get(k, j)));
          var k = k + 1;
        }
        m3.set(i, j, sum);
        var j = j + 1;
      }
      var i = i + 1;
    }
    return 0;
  }
  static method makeMatrix() : Matrix[complex] {
    let m : Matrix[complex] = new Matrix[complex](Main.PROBLEMSIZE,
    Main.PROBLEMSIZE);
    var i : int = 0;
    while (i != Main.PROBLEMSIZE) {
      var j : int = 0;
```

```
        while (j != Main.PROBLEMSIZE) {
          m.set(i, j, new complex(@i, @j));
          var j = j + 1;
        }
        var i = i + 1;
      }
      return m;
    }
}
```

B.4.2 C++ source

```
struct complex {
  double re;
  double im;
};

#define cadd(X,Y) ((struct complex) { (X).re+(Y).re, (X).im+(Y).im })

#define cmult(X,Y) \
((struct complex) { (X).re*(Y).re - (X).im*(Y).im, (X).re*(Y).im + (X).im*(Y).re })

const int PROBLEMSIZE=150;

int main() {

  struct complex m1[PROBLEMSIZE][PROBLEMSIZE];
  struct complex m2[PROBLEMSIZE][PROBLEMSIZE];
  struct complex m3[PROBLEMSIZE][PROBLEMSIZE];

  struct complex result ;

  int i=0;
  for (; i < PROBLEMSIZE; i++) {
    int j=0;
    for (; j < PROBLEMSIZE; j++) {
      m1[i][j] = (struct complex){ (double)i,(double)j };
      m2[i][j] = (struct complex){ (double)i,(double)j };
    }
  }

  for ( i=0; i < PROBLEMSIZE; i++) {
    int j=0;
    for (; j < PROBLEMSIZE; j++) {
      struct complex sum = { 0.0, 0.0 };
      int k=0;
      for (; k < PROBLEMSIZE; k++) {
        struct complex one=m1[i][k];
        struct complex two=m2[k][j];
        struct complex prod=cmult(one,two);
```

```
        sum = cadd(sum, prod);
      }
      m3[i][j] = sum;
    }
  }

  return 0;
}
```

B.4.3 Eiffel source

```
expanded class COMPLEX

feature
   build(r, i:DOUBLE) is
      do
         re := r
         im := i
      end

feature
   re:DOUBLE
   im:DOUBLE

feature
   to_string :STRING is
      do
         Result := re. to_string + "i" + im. to_string
      end

   infix"+" (other:COMPLEX) : COMPLEX is
      do
         !!Result
         Result. build(re+other.re, im+other.im)
      end

   infix"*" (other:COMPLEX) : COMPLEX is
      do
         !!Result
         Result. build((re*other. re)-(im*other.im),
                       (re*other. im)+(im*other.re))
      end
end

class MATRIX

creation make

feature
   make is
      local
         m1, m2: like m3
```

```
        do
            m1 := makematrix
            m2 := makematrix

            !! m3.make(0, problemsize − 1, 0, problemsize − 1)

            multiplymatrix (m1, m2)
        end

    m3: ARRAY2[COMPLEX]

    problemsize : INTEGER is 150

    makematrix: like m3 is
        local
            i , j : INTEGER
            entry : COMPLEX
        do
            !! Result.make(0, problemsize − 1, 0, problemsize − 1)
            from i := 0 until i = problemsize loop
                from j := 0 until j = problemsize loop
                    !! entry
                    entry . build ( i , j )
                    Result.put(entry , i , j )
                    j := j + 1
                end
                i := i + 1
            end
        end

    multiplymatrix ( first , second : like m3) is
        local
            i , j , k : INTEGER
            sum: COMPLEX
        do
            from i := 0 until i = problemsize loop
                from j := 0 until j = problemsize loop
                    !! sum
                    from k := 0 until k = problemsize loop
                        sum := sum + first.item( i , k ) ∗ second.item(k , j )
                        k := k + 1
                    end
                    m3.put(sum, i , j )
                    j := j + 1
                end
                i := i + 1
            end
        end

end
```

B.4.4 Java source

```
final class complex {
  private final double re;
  private final double im;

  complex(double re, double im) {
    this.re=re;
    this.im=im;
  }

  public static complex zero = new complex(0.0, 0.0);

  public double getRe() { return this.re; }

  public double getIm() { return this.im; }

  public complex plus(complex other) {
    return new complex(this.re+other.re, this.im+other.im);
  }

  public complex times(complex other) {
    return new complex((this.re*other.re)-(this.im*other.im),
                       (this.re*other.im)+(this.im*other.re));
  }

  public String toString() {
    return "" + this.re + "i" + this.im;
  }
}

final class complexMatrix {

  private final int xdim;
  private final int ydim;
  private final complex[] data;

  complexMatrix(int x, int y) {
    this.xdim=x;
    this.ydim=y;
    this.data=new complex[x*y];
  }

  public complex get(int x, int y) {
    return this.data[(x*this.xdim)+y];
  }

  public void set(int x, int y, complex d) {
    this.data[(x*this.xdim)+y] = d;
  }
}
```

```
public class Main {
  static final int PROBLEMSIZE = 150;

  public static void main(String [] args) {

    complexMatrix m1 = Main.makeMatrix();
    complexMatrix m2 = Main.makeMatrix();
    complexMatrix m3 = new complexMatrix(Main.PROBLEMSIZE,Main.PROBLEMSIZE);

    for (int i=0; i != Main.PROBLEMSIZE; i++) {
      for (int j=0; j != Main.PROBLEMSIZE; j++) {
        complex sum = complex.zero;
        for (int k=0; k != Main.PROBLEMSIZE; k++) {
          sum = sum.plus(m1.get(i,k).times(m2.get(k,j )));
        }
        m3.set(i , j , sum);
      }
    }
  }

  static complexMatrix makeMatrix() {
    complexMatrix m = new complexMatrix(Main.PROBLEMSIZE,Main.PROBLEMSIZE);

    for (int i=0; i != Main.PROBLEMSIZE; i++) {
      for (int j=0; j != Main.PROBLEMSIZE; j++) {
        m.set(i ,j , new complex((double)i, (double)j));
      }
    }

    return m;
  }
}
```

B.5 Program ackermann: the Ackermann function

This program computes the Ackermann function, which makes a large number of recursive calls. The only data type used in this program is `int`. The purpose of this test is to show the efficiency of integer arithmetic and function calls, and to examine any performance loss across different language implementations.

There is no key class for this test. The problem size for this program is the second argument to the initial function call.

B.5.1 Pluto source

```
abstract class Main() {
  static method run() : int {
    Main.Ackermann(3, 10);
    return 0;
  }
```

```
  static method Ackermann(m:int, n:int) : int {
    if (m == 0) {
      return n + 1;
    } else if (n == 0) {
      return Main.Ackermann(m − 1, 1);
    } else {
      return Main.Ackermann(m − 1, Main.Ackermann(m, n − 1));
    }
  }
}
```

B.5.2 C++ source

```
static int Ackermann(int m, int n) {
  if (m == 0)
    return n + 1;
  else if (n == 0)
    return Ackermann(m − 1, 1);
  else
    return Ackermann(m − 1, Ackermann(m, n − 1));
}

int main () {
  Ackermann(3,10);

  return 0;
}
```

B.5.3 Eiffel source

```
class ACKERMANN

creation make

feature
   make is
      local
         total :INTEGER
      do
         total := ack(3,10)
      end;

feature {NONE}
   ack(m, n:INTEGER):INTEGER is
      do
         if m = 0 then
            Result := n + 1
         elseif n = 0 then
            Result := ack(m − 1, 1)
         else
```

```
            Result := ack(m − 1, ack(m, (n − 1)))
          end
        end;
end
```

B.5.4 Java source

```
public class Main {
  public static void main(String [] args) {
    Main.Ackermann(3,10);
  }

  static int Ackermann(int m, int n) {
    if (m == 0)
      return n + 1;
    else if (n == 0)
      return Main.Ackermann(m − 1, 1);
    else
      return Main.Ackermann(m − 1, Main.Ackermann(m, n − 1));
  }
}
```

B.6 Program prime_sieve: the Sieve of Eratosthenes

This test program counts how many prime numbers there are within a certain range of integers, using the Sieve of Eratosthenes method. The test makes heavy use of integer arithmetic and array accesses. Using different language implementations of this program highlights any inefficiencies in their support for arrays.

Like the ackermann test, this program has no key class. The problem size for this program is 10 iterations of the sieving process, for a range of 2 million integers.

B.6.1 Pluto source

```
abstract class Main() {
  static method run() : int {
    Main.sieve();
    return 0;
  }
  static method sieve() : int {
    let PROBLEMSIZE : int = 2000000;
    let flags : Array[int] = new Array[int](PROBLEMSIZE);
    var iterations : int = 0;
    var count : int = 0;
    while (iterations < 10) {
      var iterations = iterations + 1;
      var count = 0;
      var i : int = 0;
      while (i < PROBLEMSIZE) {
        flags.set(i, 1);
        var i = i + 1;
```

```
      }
      var i = 0;
      while (i < PROBLEMSIZE) {
        if (flags # i != 0) {
          let prime : int = i + i + 3;
          var k : int = i + prime;
          while (k < PROBLEMSIZE) {
            flags.set(k, 0);
            var k = k + prime;
          }
          var count = count + 1;
        }
        var i = i + 1;
      }
    }
    return count;
  }
}
```

B.6.2 C++ source

```
const int PROBLEMSIZE=2000000;

int sieve () {
  unsigned char flags[PROBLEMSIZE];
  int iterations ;
  int count;

  for ( iterations = 0; iterations < 10; iterations ++) {
    int i ;
    int prime;
    int k;
    count = 0;

    for ( i = 0; i < PROBLEMSIZE; i++)
      flags [ i ] = 1;

    for ( i = 0; i < PROBLEMSIZE; i++) {
      if ( flags [ i ]) {
        prime = i + i + 3;
        k = i + prime;
        while (k < PROBLEMSIZE) {
          flags [k] = 0;
           k += prime;
        }
        count++;
      }
    }
  }
```

```
  return count;
}


int main() {
  sieve ();

  return 0;
}
```

B.6.3 Eiffel source

```
class SIEVE

creation make

feature
   problemsize:INTEGER is 2000000

   make is
      local
         count: INTEGER;
         flags : ARRAY[CHARACTER];
         i : INTEGER;
         iterations : INTEGER;
         prime: INTEGER;
         k: INTEGER;
      do
         !! flags.make(0, problemsize)

         from iterations := 0 until iterations = 10 loop
            count := 0

            from i := 0 until i = problemsize loop
               flags.put('t', i)
               i := i + 1
            end

            from i := 0 until i = problemsize loop
               if flags.item(i) = 't' then
                  from
                     prime := i + i + 3
                     k := i + prime
                  until k > problemsize loop
                     flags.put('f', k)
                     k := k + prime
                  end
                  count := count + 1
               end
               i := i + 1
            end
```

```
            iterations := iterations + 1
        end
    end;

end
```

B.6.4 Java source

```
public class Main {
  public static void main(String [] args) {
    Main.sieve ();
  }

  static int sieve () {
    final int PROBLEMSIZE = 2000000;

    byte flags [] = new byte[PROBLEMSIZE];
    int iterations ;
    int count=0;

    for ( iterations = 0; iterations < 10; iterations ++) {
      int i ;
      int prime;
      int k;
      count = 0;

      for ( i = 0; i < PROBLEMSIZE; i++)
        flags [ i ] = 1;

      for ( i = 0; i < PROBLEMSIZE; i++) {
        if ( flags [ i ] == 1) {
          prime = i + i + 3;
          k = i + prime;
          while (k < PROBLEMSIZE) {
            flags [k] = 0;
            k += prime;
          }
          count++;
        }
      }
    }

    return count;
  }
}
```

B.7 Program gen_stack: generic stack

For this program, a generic `stack` class is defined and used for two different element types: `int` and `float`. The stack is implemented using an array, and handles resizing as needed.

Both stack objects are exercised as follows: for each `i` between 1 and the problem size, a constant value of the element type is pushed `i` times, then popped `i` times. This program is based on an example used in [59].

This program is designed to exercise a generic data structure, and to examine the cost of using a generic class for elements of a value type. The Java implementation uses the `java.lang.Object` type in lieu of genericity, and employs typecasts when retrieving elements. The constant value to be pushed and popped is pre-boxed.

There is no key class in this program. The problem size is the maximum depth that the stack will reach.

Note that the Pluto `float` type is double precision, so the corresponding type is used in the C++, Eiffel and Java versions of this program.

B.7.1 Pluto source

```
abstract class Visitor[X]() {
  abstract method visit(d:X);
}

final class Stack[T]() {
  field elements : Array[T] = new Array[T](10);
  field size : int = 0;
  method push(e:T) {
    if (!(this.size < this.elements.size())) {
      let bigger : Array[T] = new Array[T](this.size * 2);
      var count : int = 0;
      while (count < this.size) {
        bigger.set(count, this.elements # count);
        var count = count + 1;
      }
      var this.elements = bigger;
    }
    this.elements.set(this.size, e);
    var this.size = this.size + 1;
  }
  method pop() : T {
    if (this.size == 0) {
      String.printAbort("Attempted pop of empty stack");
    }
    var this.size = this.size - 1;
    return this.elements # this.size;
  }
  method clone() : Stack[T] {
    return this;
  }
  method isEmpty() : boolean {
    return this.size == 0;
  }
  method accept(v:Visitor[T]) {
    var index : int = this.size;
    while (0 < index) {
```

```
        var index = index − 1;
        v.visit(this.elements # index);
      }
    }
}

class intPrinter() extends Visitor[int]() {
  method visit(i:int) {
    String.print("Got integer: ");
    String.println($i);
  }
}

abstract class Main() {
  static method run() : int {
    let PROBLEMSIZE : int = 10001;
    let s : Stack[int] = new Stack[int]();
    var i : int = 1;
    while (i < PROBLEMSIZE) {
      var j : int = 0;
      while (j < i) {
        s.push(43);
        var j = j + 1;
      }
      var j = 0;
      while (j < i) {
        s.pop();
        var j = j + 1;
      }
      var i = i + 1;
    }
    let sf : Stack[float] = new Stack[float]();
    var i = 1;
    while (i < PROBLEMSIZE) {
      var j : int = 0;
      while (j < i) {
        sf.push(3.14159);
        var j = j + 1;
      }
      var j = 0;
      while (j < i) {
        sf.pop();
        var j = j + 1;
      }
      var i = i + 1;
    }
    return 0;
  }
}
```

B.7.2 C++ source

```
#include <cstdlib>
#include <iostream>

using namespace std;

template <class T> class Visitor {
public:
  virtual void visit (T& d) = 0;
};

template <class T> class Stack {
private:
  T* elements;
  int size;
  int current;

public:
  Stack () : elements(new T[10]), size (10), current (0) { }

  void push(T& e);

  T& pop();

  bool isEmpty() {
    return this->current == 0;
  }

  virtual void accept( Visitor <T>* v) {
    int index = this->current;
    while (0 < index)
      v->visit(this->elements[--index]);
  }
};

template <class T> void Stack<T>::push(T& e) {
  if (this->current >= this->size) {
    const int oldSize = this->size;
    T* bigger = new T[oldSize * 2];
    for (int count=0; count < oldSize; count++)
      bigger[count] = this->elements[count];

    this->elements = bigger;
    this->size = oldSize * 2;
  }
  this->elements[this->current++] = e;
}

template <class T> T& Stack<T>::pop() {
  if (this->current == 0) {
    cerr << "Attempted pop of empty stack"<< endl;
```

```
    exit(EXIT_FAILURE);
  }
  return this->elements[--this->current];
}

class intPrinter : public Visitor<int> {
public:
  virtual void visit(int& i) {
    cout << "Got integer: " << i << endl;
  }
};

int main() {
  const int PROBLEMSIZE = 10001;
  Stack<int> s;
  int val = 43;

  for (int i=1; i < PROBLEMSIZE; i++) {
    for (int j=0; j < i; j++) {
      s.push(val);
    }
    for (int j=0; j < i; j++) {
      s.pop();
    }
  }

  Stack<double> sd;
  double dval = 3.14159;

  for (int i=1; i < PROBLEMSIZE; i++) {
    for (int j=0; j < i; j++) {
      sd.push(dval);
    }
    for (int j=0; j < i; j++) {
      sd.pop();
    }
  }

  return 0;
}
```

B.7.3 Eiffel source

```
class STACK[T]

creation make

feature
   make is
      do
         !!elements.make(0,9)
```

```
            position := 0
        end

feature
    elements:ARRAY[T]
    position :INTEGER

feature
    push(e:T) is
        local
            counter:INTEGER
            nu:ARRAY[T]
        do
            if position >= elements.count then
                !!nu.make(0,(elements.count*2)-1)
                from counter := 0 until counter = elements.count loop
                    nu.put(elements.item(counter), counter)
                    counter := counter + 1
                end
                elements := nu
            end
            elements.put(e, position)
            position := position + 1
        end

    pop : T is
        do
            if position = 0 then
                std_output. put_string ("Attempted pop of empty stack")
            end
            position := position - 1
            Result := elements.item( position )
        end
end

class MAIN

creation make

feature
    problemsize:INTEGER is 10000
    make is
        local
            s:STACK[INTEGER]
            sf:STACK[DOUBLE]
            i,j,value:INTEGER
            dval:DOUBLE
        do
            !!s.make
            from i := 1 until i > problemsize loop
                from j := 0 until j = i loop
                    s.push(43)
```

```
            j := j + 1
        end
        from j := 0 until j = i loop
            value := s.pop
            j := j + 1
        end
        i := i + 1
    end

    !! sf.make
    from i := 1 until i > problemsize loop
        from j := 0 until j = i loop
            sf.push(3.14159)
            j := j + 1
        end
        from j := 0 until j = i loop
            dval := sf.pop
            j := j + 1
        end
        i := i + 1
    end
  end

end
```

B.7.4 Java source

```
import java.io.*;

abstract class Visitor {
  abstract void visit(Object d);
}

final class Stack {
  private Object elements[] = new Object[10];
  private int current=0;

  void push(Object e) {
    if (this.current >= this.elements.length) {
      final int oldSize = this.elements.length;
      Object bigger[] = new Object[oldSize * 2];
      for (int count=0; count < oldSize; count++)
        bigger[count] = this.elements[count];

      this.elements = bigger;
    }
    this.elements[this.current++] = e;
  }

  Object pop() {
    if (this.current == 0) {
```

```
      System.err. println ("Attempted pop of empty stack");
      System.exit (1);
    }
    return this.elements[--this.current];
  }

  boolean isEmpty() {
    return this. current == 0;
  }

  void accept( Visitor v) {
    int index = this. current;
    while (0 < index)
      v. visit (this.elements[--index]);
  }
}


class intPrinter extends Visitor {
  void visit (Object o) {
    System.out. print ("Got integer: " );
    System.out. println (((Integer)o). intValue ());
  }
}

public class Main {
  public static void main(String[] args) throws Exception {
    final int PROBLEMSIZE = 10001;
    Stack s=new Stack();
    Integer val = new Integer(43);

    for (int i=1; i < PROBLEMSIZE; i++) {
      for (int j=0; j < i; j++) {
        s.push(val);
      }
      for (int j=0; j < i; j++) {
        s.pop();
      }
    }

    Stack sd=new Stack();
    Double dval = new Double(3.14159);

    for (int i=1; i < PROBLEMSIZE; i++) {
      for (int j=0; j < i; j++) {
        sd.push(dval);
      }
      for (int j=0; j < i; j++) {
        sd.pop();
      }
    }
```

```
  }
}
```

B.8 Program gen_comptr_tree: generic binary search tree with comparator

In this test program, a generic binary search tree is defined and populated much the same as for the gen_tree program in Section B.2. However, instead of placing a bound on the type parameter, the ordering of two elements is found by passing a comparator object into the insertion method. Two different comparators are defined: one based on lexical word ordering, and the second based on word length with lexical ordering used as a tie breaker.

Instead of repeated cloning, this program makes a new tree by traversing the existing one and inserting each word into the new tree. The copied tree uses the same comparator algorithm, resulting in a degenerate unbalanced tree as the words being inserted are already in order. The purpose of this is to exercise the comparison of generic values, without being obscured by the effects of cloning as used in gen_tree.

The input to this program is the same file of words as used in the gen_tree test program, described in Section B.2.

The key class for this program is `word`. The problem size is the number of words read from the file, which is 32460.

B.8.1 Pluto source

```
abstract class Comparator[T]() {
  abstract method compare(one:T, other:T) : int;
}

abstract class Visitor[X]() {
  abstract method visit(d:X);
}

final class Tree[X](c:Comparator[X]) {
  field comparisonAlgorithm : Comparator[X] = c;
  field root : Node[X] = new EmptyNode[X]();
  method add(element:X) {
    var this.root = this.root.insert(element, this.comparisonAlgorithm);
  }
  method accept(v:Visitor[X]) {
    this.root.accept(v);
  }
}

abstract class Node[T]() {
  abstract method insert(element:T, algorithm:Comparator[T]) : Node[T];
  abstract method accept(v:Visitor[T]);
}

class EmptyNode[T]() extends Node[T]() {
```

```
  method insert(element:T, algorithm:Comparator[T]) : Node[T] {
    return new InternalNode[T](element, this, this);
  }
  method accept(v:Visitor[T]) { }
}

class InternalNode[T](d:T, l:Node[T], r:Node[T]) extends Node[T]() {
  field data : T = d;
  field leftChild : Node[T] = l;
  field rightChild : Node[T] = r;
  method insert(element:T, algorithm:Comparator[T]) : Node[T] {
    let order : int = algorithm.compare(element, this.data);
    if (order < 0) {
      var this.leftChild = this.leftChild.insert(element, algorithm);
    } else if (0 < order) {
      var this.rightChild = this.rightChild.insert(element, algorithm);
    }
    return this;
  }
  method accept(v:Visitor[T]) {
    this.leftChild.accept(v);
    v.visit(this.data);
    this.rightChild.accept(v);
  }
}

final value class Word(s:String) {
  field text : String = s;
  final method <(other:Word) : boolean {
    return this.text < other.text;
  }
  final method $() : String {
    return this.text;
  }
  final method length() : int {
    return this.text.length();
  }
}

class WordLexicalComparator() extends Comparator[Word]() {
  method compare(one:Word, other:Word) : int {
    if (one < other) {
      return 0 − 1;
    } else if (other < one) {
      return 1;
    } else {
      return 0;
    }
  }
}
```

```
class WordLengthComparator() extends Comparator[Word]() {
  field secondLevel : Comparator[Word] = new WordLexicalComparator();
  method compare(one:Word, other:Word) : int {
    let lengthDiff : int = one.length() − other.length();
    if (lengthDiff != 0) {
      return lengthDiff;
    } else {
      return this.secondLevel.compare(one, other);
    }
  }
}

class WordPrinter() extends Visitor[Word]() {
  method visit(w:Word) {
    String.print("Got word: ");
    String.println($w);
  }
}

class WordTreeCopier() extends Visitor[Word]() {
  field tree : Tree[Word] = new Tree[Word](new WordLengthComparator());
  method visit(w:Word) {
    this.tree.add(w);
  }
  method getTree() : Tree[Word] {
    return this.tree;
  }
}

abstract class Main() {
  static method run() : int {
    let dictionary : Tree[Word] = new Tree[Word](new WordLengthComparator());
    var line : String = String.readln();
    while (line.length() != 0) {
      dictionary.add(new Word(line));
      var line = String.readln();
    }
    dictionary.accept(new WordTreeCopier());
    return 0;
  }
}
```

B.8.2 C++ source

```
#include <iostream>
#include <string>

using namespace std;

template <class T> class Comparator {
```

```
public:
  virtual int compare(T& one, T& other) = 0;
};

template <class T> class Visitor {
public:
  virtual void visit (T& d) = 0;
};

template <class T> class Node {
public:
  virtual Node<T>* insert(T& element, Comparator<T>* algorithm) = 0;
  virtual void accept( Visitor <T>* v) = 0;
};

template <class T> class InternalNode : public Node<T> {
private:
  T data;
  Node<T>* leftChild;
  Node<T>* rightChild;

public:
  InternalNode(T& d, Node<T>* l, Node<T>* r)
    : data(d), leftChild (l), rightChild (r) { }

  virtual Node<T>* insert(T& element, Comparator<T>* algorithm) {
    const int order = algorithm->compare(element,this->data);
    if (order < 0) {
      this->leftChild = this->leftChild->insert(element,algorithm);
    }
    else if (0 < order) {
      this->rightChild = this->rightChild->insert(element,algorithm);
    }
    return this;
  }

  virtual void accept( Visitor <T>* v) {
    this->leftChild->accept(v);
    v->visit(this->data);
    this->rightChild->accept(v);
  }
};

template <class T> class EmptyNode : public Node<T> {
public:
  virtual Node<T>* insert(T& element, Comparator<T>* algorithm) {
    return new InternalNode<T>(element, this, this);
  }

  virtual void accept( Visitor <T>* v) { }
};
```

```
template <class T> class Tree {
private:
  Comparator<T>* comparisonAlgorithm;
  Node<T>* root;

public:
  Tree(Comparator<T>* c) : comparisonAlgorithm(c), root(new EmptyNode<T>()) { }

  void add(T& element) {
    this->root = this->root->insert(element, this->comparisonAlgorithm);
  }

  void accept( Visitor <T>* v) {
    this->root->accept(v);
  }
};


class Word {
private:
  string text;

public:
  Word(string& s) : text(s) { }

  bool lessThan(Word& other) {
    return this->text.compare(other.text) < 0;
  }

  int length() {
    return this->text.length();
  }

  string & toString() {
    return this->text;
  }
};

class WordLexicalComparator : public Comparator<Word> {
public:
  virtual int compare(Word& one, Word& other) {
    if (one.lessThan(other)) {
      return -1;
    } else if (other.lessThan(one)) {
      return 1;
    } else {
      return 0;
    }
  }
};
```

```
class WordLengthComparator : public Comparator<Word> {
private:
  Comparator<Word>* secondLevel;

public:
  WordLengthComparator() : secondLevel(new WordLexicalComparator()) { }

  virtual int compare(Word& one, Word& other) {
    const int lengthDiff = one.length() - other.length();
    if ( lengthDiff != 0) {
      return lengthDiff;
    }
    else {
      return this->secondLevel->compare(one,other);
    }
  }
};

class WordPrinter : public Visitor<Word> {
public:
  virtual void visit (Word& w) {
    cout << "Got word: " << (w.toString()) << endl;
  }
};

class WordTreeCopier : public Visitor<Word> {
private:
  Tree<Word>* tree;

public:
  WordTreeCopier() : tree(new Tree<Word>(new WordLengthComparator())) { }
  virtual void visit (Word& w) {
    this->tree->add(w);
  }

  Tree<Word>* getTree() {
    return this->tree;
  }
};

int main() {
  Tree<Word> dictionary(new WordLengthComparator());

  string line;

  while (cin >> line) {
    Word newWord(line);
    dictionary.add(newWord);
  }
```

```
    dictionary .accept(new WordTreeCopier());

  return 0;
}
```

B.8.3 Java source

```
import java.io.*;

abstract class Comparator {
  abstract int compare(Object one, Object other);
}

abstract class Visitor {
  abstract void visit (Object d);
}

final class Tree {
  private final Comparator comparisonAlgorithm;
  private Node root = new EmptyNode();

  public Tree(Comparator c) {
    this.comparisonAlgorithm = c;
  }

  public void add(Object element) {
    this.root = this.root. insert (element, this.comparisonAlgorithm);
  }

  public void accept( Visitor v) {
    this.root.accept(v);
  }
}

abstract class Node {
  abstract Node insert(Object element, Comparator algorithm);
  abstract void accept( Visitor v);
}

class EmptyNode extends Node {

  public Node insert(Object element, Comparator algorithm) {
    return new InternalNode(element, this, this);
  }

  public void accept( Visitor v) { }
}


class InternalNode extends Node {
  private final Object data;
```

```
  private Node leftChild;
  private Node rightChild;

  public InternalNode(Object d, Node l, Node r) {
    this.data=d;
    this.leftChild=l;
    this.rightChild=r;
  }

  public Node insert(Object element, Comparator algorithm) {
    final int order = algorithm.compare(element,this.data);
    if (order < 0) {
      this.leftChild = this.leftChild.insert(element,algorithm);
    }
    else if (0 < order) {
      this.rightChild = this.rightChild.insert(element,algorithm);
    }
    return this;
  }

  public void accept(Visitor v) {
    this.leftChild.accept(v);
    v.visit(this.data);
    this.rightChild.accept(v);
  }
}


final class Word {
  private final String text;

  public Word(String s) {
    this.text=s;
  }

  public boolean lessThan(Word other) {
    return this.text.compareTo(other.text) < 0;
  }

  public int length() {
    return this.text.length();
  }

  public String toString() {
    return this.text;
  }
}

class WordLexicalComparator extends Comparator {
  public int compare(Object one, Object other) {
    final Word wordOne = (Word)one;
```

```
      final Word wordOther = (Word)other;
      if (wordOne.lessThan(wordOther)) {
        return −1;
      } else if (wordOther.lessThan(wordOne)) {
        return 1;
      } else {
        return 0;
      }
    }
  }

  class WordLengthComparator extends Comparator {
    private final Comparator secondLevel = new WordLexicalComparator();

    public int compare(Object one, Object other) {
      final Word wordOne = (Word)one;
      final Word wordOther = (Word)other;
      final int lengthDiff = wordOne.length() − wordOther.length();
      if ( lengthDiff != 0) {
        return lengthDiff ;
      }
      else {
        return this.secondLevel.compare(one,other);
      }
    }
  }


  class WordTreeCopier extends Visitor {
    private final Tree tree = new Tree(new WordLengthComparator());
    public void visit (Object w) {
      this.tree.add(w);
    }
    public Tree getTree() {
      return this.tree;
    }
  }

  class WordPrinter extends Visitor {
    public void visit (Object w) {
      System.out.print("Got word: ");
      System.out.println (w.toString ());
    }
  }

  public class Main {
    public static void main(String[] args) throws IOException {
      final Tree dictionary = new Tree(new WordLengthComparator());

      BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
```

```
    String line=in.readLine();
    while ( line != null ) {
      dictionary .add(new Word(line));
      line = in.readLine();
    }

    dictionary .accept(new WordTreeCopier());
  }
}
```

B.9 Program word_comptr_tree: fixed-type binary search tree with comparator

This program is a conversion of the generic one described above, in Section B.8, but with an element type fixed as `word` instead of being generic.

The input to this program, the key class and the problem size are all the same as for the generic version.

B.9.1 Pluto source

```
abstract class WordComparator() {
  abstract method compare(one:Word, other:Word) : int;
}

abstract class WordVisitor() {
  abstract method visit(d:Word);
}

final class WordTree(c:WordComparator) {
  field comparisonAlgorithm : WordComparator = c;
  field root : Node = new EmptyNode();
  method add(element:Word) {
    var this.root = this.root.insert(element, this.comparisonAlgorithm);
  }
  method accept(v:WordVisitor) {
    this.root.accept(v);
  }
}

abstract class Node() {
  abstract method insert(element:Word, algorithm:WordComparator) : Node;
  abstract method accept(v:WordVisitor);
}

class EmptyNode() extends Node() {
  method insert(element:Word, algorithm:WordComparator) : Node {
    return new InternalNode(element, this, this);
  }
  method accept(v:WordVisitor) { }
}
```

```
class InternalNode(d:Word, l:Node, r:Node) extends Node() {
  field data : Word = d;
  field leftChild : Node = l;
  field rightChild : Node = r;
  method insert(element:Word, algorithm:WordComparator) : Node {
    let order : int = algorithm.compare(element, this.data);
    if (order < 0) {
      var this.leftChild = this.leftChild.insert(element, algorithm);
    } else if (0 < order) {
      var this.rightChild = this.rightChild.insert(element, algorithm);
    }
    return this;
  }
  method accept(v:WordVisitor) {
    this.leftChild.accept(v);
    v.visit(this.data);
    this.rightChild.accept(v);
  }
}

final value class Word(s:String) {
  field text : String = s;
  final method <(other:Word) : boolean {
    return this.text < other.text;
  }
  final method $() : String {
    return this.text;
  }
  final method length() : int {
    return this.text.length();
  }
}

class WordLexicalComparator() extends WordComparator() {
  method compare(one:Word, other:Word) : int {
    if (one < other) {
      return 0 − 1;
    } else if (other < one) {
      return 1;
    } else {
      return 0;
    }
  }
}

class WordLengthComparator() extends WordComparator() {
  field secondLevel : WordComparator = new WordLexicalComparator();
  method compare(one:Word, other:Word) : int {
    let lengthDiff : int = one.length() − other.length();
    if (lengthDiff != 0) {
      return lengthDiff;
```

```
      } else {
        return this.secondLevel.compare(one, other);
      }
    }
}

class WordPrinter() extends WordVisitor() {
  method visit(w:Word) {
    String.print("Got word: ");
    String.println($w);
  }
}

class WordTreeCopier() extends WordVisitor() {
  field tree : WordTree = new WordTree(new WordLengthComparator());
  method visit(w:Word) {
    this.tree.add(w);
  }
  method getTree() : WordTree {
    return this.tree;
  }
}

abstract class Main() {
  static method run() : int {
    let dictionary : WordTree = new WordTree(new WordLengthComparator());
    var line : String = String.readln();
    while (line.length() != 0) {
      dictionary.add(new Word(line));
      var line = String.readln();
    }
    dictionary.accept(new WordTreeCopier());
    return 0;
  }
}
```

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] ACM Conference. *Proceedings of OOPSLA'89*, volume 24(10) of *SIGPLAN Notices*, October 1989.

[3] ACM Conference. *Proceedings of OOPSLA'95*, volume 30(10) of *SIGPLAN Notices*, October 1995.

[4] ACM Conference. *Proceedings of ICFP'98*, volume 34(1) of *SIGPLAN Notices*, Baltimore, Maryland, September 1998.

[5] ACM Conference. *Proceedings of OOPSLA'98*, volume 33(10) of *SIGPLAN Notices*, October 1998.

[6] ACM Symposium. *Proceedings of POPL'97*, Paris, France, January 1997.

[7] Ole Agesen. Design and implementation of Pep, a Java just-in-time translator. *Theory and Practice of Object Systems*, 3(2):127–155, 1997.

[8] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Proceedings of OOPSLA'97*, volume 32(10) of *SIGPLAN Notices*, pages 49–65. ACM Conference, October 1997.

[9] B Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[10] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, January 1998.

[11] Doug Bagley. The Great Computer Language Shootout. Web site. http://www.bagley.org/~doug/shootout/.

[12] Hans-J Boehm. Space efficient conservative garbage collection. *ACM SIGPLAN Notices*, 28(6):197–206, June 1993.

[13] Hans-J Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9):807–820, September 1988.

[14] Gilad Bracha et al. Adding Generics to the Java Programming Language: Participant Draft Specification, April 2001. JSR-14 Public Draft http://java.sun.com/aboutJava/communityprocess/jsr/jsr_014_gener.html.

[15] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. *GJ: Extending the Java Programming Language with Type Parameters*, 1998. Project Documentation, available from http://www.cs.bell-labs.com/~wadler/pizza/gj or http://www.math.luc.edu/pizza/gj.

[16] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. *GJ Specification*, May 1998. Project Documentation, see [15].

[17] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of OOPSLA'98* [5], pages 183–200.

[18] Kim Bruce. Typing in object-oriented languages: Achieving expressiveness and safety. Technical report, Williams College, September 1996.

[19] Kim Bruce, Luca Cardelli, et al. On binary methods. *Theory and Practice of Object Systems*, 1(3):217–238, 1995.

[20] Kim Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *Proceedings of ECOOP'98*, number 1445 in LNCS, Brussels, Belgium, 1998. Springer-Verlag.

[21] Kim Bruce, Leaf Petersen, and Adrian Fiech. Subtyping is not a good "Match" for object-oriented languages. In *Proceedings of ECOOP'97* [99], pages 104–127.

[22] Kim Bruce and Joseph Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. Technical report, Williams College, February 1999.

[23] Zoran Budimlic and Ken Kennedy. Optimizing Java: Theory and practice. *Concurrency: Practice and Experience*, 9(6):445–463, June 1997.

[24] S.J. Bushell, A. Dearle, A.L. Brown, and F.A. Vaughan. Using C as a compiler target language for native code generation in persistent systems. Technical report, University of Adelaide, Australia, 1994.

[25] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of FPCA'89*, pages 273–280. ACM Conference, September 1989.

[26] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[27] Robert Cartwright and Guy L. Steele Jr. Compatible genericity with run-time types for the Java programming language. In *Proceedings of OOPSLA'98* [5], pages 201–215.

[28] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[29] Craig Chambers. *The Design and Implementation of the SELF compiler, an Optimizing Compiler for Object-Oriented Languages*. PhD thesis, Stanford University, 1992.

[30] Craig Chambers. The Cecil language: Specification and rationale. Technical report, University of Washington, 1997.

[31] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *Proceedings of OOPSLA '91, ACM SIGPLAN Notices*, pages 1–15, November 1991.

[32] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. In *Proceedings of OOPSLA'89* [2], pages 49–70.

[33] Juan Chen and Andrew W. Appel. Dictionary passing for polytypic polymorphism. *To appear*, March 2001.

[34] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. *ACM SIGPLAN Notices*, 34(10):1–19, October 1999.

[35] Suzanne Collin, Dominique Colnet, and Olivier Zendra. Type inference for late binding: The SmallEiffel compiler. In *Joint Modular Languages Conference*, number 1204 in Lecture Notes in Computer Science, pages 67–81. Springer-Verlag, 1997.

[36] Douglas E. Comer. *Computer Networks and Internets*. Prentice-Hall, 2001. Third Edition.

[37] R. C. H. Connor, A. Dearle, R. Morrison, and A. L. Brown. An object addressing mechanism for statically typed languages with multiple inheritance. In *Proceedings of OOPSLA'89* [2], pages 279–285.

[38] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *Proceedings of ICFP'98* [4], pages 301–312.

[39] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Proceedings of OOPSLA'95* [3], pages 156–168.

[40] Julian Dolby. Automatic inline allocation of objects. In *Proceedings of PLDI'97*, volume 32(5) of *SIGPLAN Notices*, pages 7–17. ACM Conference, May 1997.

[41] Julian Dolby and Andrew A. Chien. An evaluation of automatic object inline allocation techniques. In *Proceedings of OOPSLA'98* [5], pages 1–20.

[42] Robert Fitzgerald, Todd Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for Java. Technical report, Microsoft Research, October 1998.

[43] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[44] David Gay and Bjarne Steensgaard. Stack allocating objects in Java. Technical report, Microsoft Research, 1998.

[45] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[46] B. Gomes, D. Stoutamire, B. Weissman, and J. Feldman. Using value semantic abstractions to guide strongly typed language design. Technical report, International Computer Science Institute, December 1997.

[47] Andrew Gordon and Don Syme. Typing a multi-language intermediate code. In *Proceedings of POPL 2001*, pages 248–260, London, England, January 2001. ACM Symposium.

[48] James Gosling. The evolution of numerical computing in Java. Web page, October 1998. http://java.sun.com/people/jag/FP.html.

[49] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, August 1996. First Edition.

[50] Jan Gray. Under the hood of C++. Microsoft Developer Network Library article, 1998. http://www.geocities.com/SiliconValley/Foothills/7592/oop/Gray-article.html.

[51] Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs, and Koen G. Langendoen. *Modern Compiler Design*. John Wiley & Sons, 2000.

[52] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of POPL'95*, pages 130–141, San Francisco, California, January 1995. ACM Symposium.

[53] A. Hejlsberg and S. Wiltamuth. C# language reference. Microsoft Developer Network Library, 2000.

[54] Atsushi Igarashi and Benjamin Pierce. Foundations for virtual types. In *6th Workshop on Foundations of Object-Oriented Languages*. FOOL6, 1999.

[55] ISO/IEC 14882:1998. *International Standard: Programming Languages — C++*, September 1998.

[56] ISO/IEC 9899:1999. *International Standard: Programming Languages — C*, December 1999.

[57] Java Grande Forum. Homepage. http://www.javagrande.org.

[58] Numerics Working Group Java Grande Forum. Improving Java for numerical computation. Online proposal, October 1998. http://math.nist.gov/javanumerics/jgfnwg-01.html.

[59] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET Common Language Runtime. In *Proceedings of PLDI 2001*, 2001. To appear.

[60] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1988. Second Edition.

[61] Andrew Koenig and Barbara Moo. *Ruminations on C++*. Addison-Wesley, 1997.

[62] Andreas Krall and Reinhard Grafl. CACAO — A 64-bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, November 1997.

[63] Glenn Krasner, editor. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, 1983.

[64] Christopher League, Zhong Shao, and Valery Trifonov. Representing Java classes in a typed intermediate language. *ACM SIGPLAN Notices*, 34(9):183–196, September 1999.

[65] Xavier Leroy. Efficient data representation in polymorphic languages. Technical Report 1264, INRIA Rocquencourt, August 1990.

[66] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proceedings of POPL'92*, pages 177–188, Albequerque, New Mexico, January 1992. ACM Symposium.

[67] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, September 1996. First Edition.

[68] Stanley Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996.

[69] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Andrew C. Myers. Theta reference manual. Technical Report Programming Methodology Group Memo 88, MIT, February 1995.

[70] C. E. McDowell. Reducing garbage in Java. *ACM SIGPLAN Notices*, 33(9):84–86, September 1998.

[71] Roland McGrath, Ulrich Drepper, et al. *The GNU C Library Reference Manual*. The GNU Project. Version 2.1.

[72] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[73] Scott Meyers. *More Effective C++*. Addison-Wesley, 1996.

[74] Microsoft Corporation. The .NET Common Language Runtime. Microsoft Developer Network Library, 2000.

[75] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *Proceedings of ECOOP'99* [100].

[76] Yasuhiko Minamide. Full lifting of type parameters. Technical report, RIMS, Kyoto University, 1997.

[77] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, December 1995. CMU-CS-95-226.

[78] R. Morrison, A. Dearle, R.C.H. Conner, and A.L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Transactions on Programming Languages and Systems*, 13(3):342–371, July 1991.

[79] Andrew Myers and Barbara Liskov. Efficient implementation of parameterized types despite subtyping. Thor Note 9, July 1994.

[80] Andrew C. Myers. Bidirectional object layout for separate compilation. In *Proceedings of OOPSLA'95* [3], pages 124–139.

[81] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proceedings of POPL'97* [6], pages 132–145.

[82] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of POPL'97* [6], pages 146–159.

[83] John Peterson and Mark P. Jones. Implementing type classes. In *Proceedings of PLDI'93*, volume 28(6) of *SIGPLAN Notices*, pages 227–236. ACM Conference, June 1993.

[84] Simon L. Peyton Jones, Thomas Nordin, and Dino Oliva. `C--`: A portable assembly language. In *Proceedings of the 1997 Workshop on Implementing Functional Languages*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

[85] Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. `C--`: A portable assembly language that supports garbage collection. In *International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 1–28. Springer-Verlag, September 1999.

[86] Markus Pizka. Design and implementation of the GNU INSEL-Compiler gic. Technical Report TUM-I9713, SFB 342/09/97, Technical University of Munich, October 1997.

[87] P. J. Plauger, Alexander A. Stepanov, Meng Lee, and David R. Musser. *The C++ Standard Template Library*. Prentice-Hall, 2000.

[88] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications — A Way Ahead of Time (WAT) compiler. In *Proceedings of COOTS'97*, 1997.

[89] Didier Rémy and Jérôme Vouillon. On the (un)reality of virtual types. Technical note, INRIA Rocquencourt, 1998.

[90] SafetyNet project homepage. University of Sussex. http://www.cogs.susx.ac.uk/projects/safetynet/index.html.

[91] Bratin Saha and Zhong Shao. Optimal type lifting. Technical Report Number YALEU/DCS/TR-1159, Yale University, August 1998.

[92] David Shang. Covariant specification. *ACM SIGPLAN Notices*, 29(12), December 1994.

[93] David Shang. Transframe annotated reference manual. Technical report, Software Systems Research Lab, Motorola, 1997.

[94] Zhong Shao. Flexible representation analysis. In *Proceedings of ICFP'97*, volume 32(8) of *SIGPLAN Notices*, pages 85–98, Amsterdam, The Netherlands, June 1997. ACM Conference.

[95] Zhong Shao. An overview of the FLINT/ML compiler. In *Workshop on Types in Compilation*. TIC'97, 1997.

[96] Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proceedings of PLDI'95*, volume 30(6) of *SIGPLAN Notices*, pages 116–129. ACM Conference, June 1995.

[97] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *Proceedings of ICFP'98* [4], pages 313–323.

[98] Jose H. Solorzano and Suad Alagić. Parametric polymorphism for Java: A reflective solution. In *Proceedings of OOPSLA'98* [5], pages 216–225.

[99] Springer-Verlag. *Proceedings of ECOOP'97*, number 1241 in LNCS, Jyväskylä, Finland, 1997.

[100] Springer-Verlag. *Proceedings of ECOOP'99*, number 1628 in LNCS, Lisbon, Portugal, 1999.

[101] Richard Stallman et al. *Using and Porting GNU CC*. The GNU Project. Version 2.95.

[102] Guy Steele. Growing a language. Keynote talk at OOPSLA'98, October 1998.

[103] David Stoutamire and Stephen Omohundro. The Sather 1.1 specification. Technical Report tr-96-012, University of California, Berkeley, August 1996.

[104] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[105] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997. Third Edition.

[106] Sun Microsystems. Java Core Reflection API and Specification. JDK 1.1 Documentation, February 1997.

[107] Sun Microsystems. Java Object Serialization Specification. JDK 1.2 Documentation, 1998.

[108] Sun Microsystems/Metamata. The JavaCC parser generator tool. Version 1.0.

[109] Clemens Szypersky, Stephen Omohundro, and Stephan Murer. Engineering a programming language: The type and class system of Sather. Technical Report tr-93-064, University of California, Berkeley, November 1993.

[110] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proceedings of PLDI'96*, volume 31(5) of *SIGPLAN Notices*, pages 181–192. ACM Conference, May 1996.

[111] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings of ECOOP'97* [99], pages 444–471.

[112] Kresten Krab Thorup and Mads Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. In *Proceedings of ECOOP'99* [100], pages 186–204.

[113] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen. Programming with regions in the ML Kit. Technical Report DIKU-TR-97/12, University of Copenhagen, 1997.

[114] Mads Tofte and Jean-Pierre Talpin. A theory of stack allocation in polymorphically typed languages. Technical Report DIKU-report 93/15, University of Copenhagen, 1993.

[115] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[116] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of 1994 ACM Conference on LISP and Functional Programming*, pages 1–11, Orlando, Florida, June 1994.

[117] Mads Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages*. FOOL5, January 1998.

[118] David Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, 1986.

[119] David Ungar, Randall B. Smith, Craig Chambers, and Urs Hölzle. Object, message and performance: How they coexist in SELF. *IEEE Computer*, 25(10):53–65, October 1992.

[120] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: An approach to translation based on reflective features. *ACM SIGPLAN Notices*, 35(10):146–165, October 2000.

[121] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java through the homogeneous translation LM: Gathering type descriptors at load-time. Technical Report DEIS-LIA-00-001, University of Bologna, April 2000.

[122] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, 1994.

[123] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. *ACM SIGPLAN Notices*, 32(10):125–141, October 1997.