USING DYNAMIC MIXINS FOR SOFTWARE DEVELOPMENT

USING DYNAMIC MIXINS FOR SOFTWARE DEVELOPMENT

By

RONALD EDEN BURTON, M.Sc., B.Math.

A Thesis

Submitted to the Department of Computing and Software

and the School of Graduate Studies

of McMaster University

in Partial Fulfilment of the Requirements

for the Degree of

Doctor of Philosophy

McMaster University © Copyright by Ronald Eden Burton, July 2018

Doctor of Philosophy (2018) McMaster University

(Computing and Software) Hamilton, Ontario, Canada

TITLE:	Using Dynamic Mixins For Software Development
AUTHOR:	Ronald Eden Burton M.Sc., (Information Systems) Athabasca University, Athabasca, Canada B.Math., (Computer Science) University of Waterloo, Waterloo, Canada
SUPERVISOR:	Dr. Emil Sekerinski
NUMBER OF PAGES:	xiii, 121

To Mom and Dad

Abstract

Object-oriented programming has gained significant traction in the software development community and is now the common approach for developing large, commercial applications. Many of these applications require the behaviour of objects to be modified at run-time.

Contemporary class-based, statically-typed languages such as C++ and Java require collaboration with external objects to modify an object's behaviour. Furthermore, such an object must be designed to order to support such collaborations. Dynamic languages such as Python which natively support object extension do not guarantee type safety.

In this work, using dynamic mixins with static typing is proposed as a means of providing type-safe, object extension. A new language called mix is introduced that allows a compiler to syntactically check the type-safety of an object extension. A model to support object-oriented development is extended to support dynamic mixins.

The utility of the approach is illustrated using sample use cases. Finally, a compiler was implemented to validate the practicality of the model proposed.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor Dr. Emil Sekerinski for taking me under his wing. His mentorship has proven to be invaluable not only in the completion of this work but also to my professional and academic development.

I would also like to thank the members of my supervisory committee: Dr. Frantisek Franek, Dr. Ridha Khedri and Dr. Tom Maibaum. They have provided constructive feedback throughout my time at here at McMaster and were critical to me seeing this project through to its completion.

To my external examiner Dr. Jeremy Bradbury, I am tremendously grateful for your review of this thesis. Your feedback has strengthened the quality of the final product.

Thanks also goes to the Computing and Software faculty, staff and my fellow graduate students. We have a small but special group of people here. Our informal discussion and interactions have enriched my life greatly. Of particular note are my lab-mates Dr. Bojan Nokovic, Dr. Tian Zhang and Shucai Yao, for their constructive feedback during our group meetings.

Last but not least, I reserve a special thanks for my family. Tracy, it has been a long voyage but the end has finally arrived. We made it! Ashley, Simone, Renee and Maceo...Daddy's done.

Contents

Abstract **iii**

Acknowledgements **iv**

Declaration of Academic Achievement **xii**

1 Introduction **1**

1.1 Object Composition to Facilitate Code Reuse 2

1.2 Alternatives to Object Composition 4

1.3 Mixins as a Reuse Tool 5

1.4 Current Issues with Mixins 6

1.4.1 Type Safety 6

1.4.2 Interference 7

1.4.3 Inefficient Method Lookup 10

1.5 Summary of Contributions 10

2 mix, a Statically Typed Language for Dynamic Mixins **12**

2.1 Language Goals 12

2.2 mix Abstract Syntax 13

2.3 Language Definition 14

2.4 Differentiating Language Features 16
2.5 Formal Definition of mix 17
2.5.1 Abstract Syntax for Core Language 17
2.5.2 Core Language Semantics 19
2.5.3 Translation of Classes and Modules 22

3 mix Implementation **26**
3.1 Memory Layout 26
3.2 Program Initialization 28
3.3 Object Creation 29
3.4 Object Extension 29
3.5 Type Test and Type Cast 32
3.6 Method Calls 32
3.7 Analysis 33
3.8 Translation to Executable Code 34
3.9 Related Work 39
3.10 Evaluation 41
3.11 Discussion 43

4 Program Correctness **44**
4.1 Module Consistency 44
4.2 Mixin Refinement 45
4.3 Correctness of Mixin Composition 46
4.3.1 Refinement and Augmentation 48
4.3.2 Class Invariants 51
4.3.3 Compositional Reasoning with Dynamic Mixins 55
4.4 Discussion 62

5 Use Case - Intrusive Data Structures **64**
5.1 Abstract Specification 65
5.2 Specification Refinement 65
5.3 Correctness Proof 66
5.4 Machine Automated Proofs Using Boogie 68
5.5 Discussion 77

6 Use Case - Implementing Design Patterns **78**
6.1 Decorator Pattern 78
6.2 Proxy Pattern 79
6.3 Chain of Responsibility Pattern 82
6.4 Strategy Pattern 84
6.5 Patterns to Support Object Extension 85
6.5.1 Dynamic Object Model Pattern 85
6.5.2 Extension Objects Pattern 87
6.5.3 Role Object Pattern 89
6.6 Discussion 90

7 Conclusion **92**

A mix Concrete Syntax **100**

B mix Program Generated Code **102**

C Code Used to Gather Timing Results **113**

D Proofs **118**

List of Tables

3.1 Timing Results. This table shows the number of seconds (in CPU time) required to execute the program specified in Listing 3.6. 40

List of Figures

3.1 Memory Layout of the Example in Section 1.4.2 28

3.2 A Class Hierarchy 31

3.3 Method Resolution Time 42

5.1 Boogie Model of Associations Program 69

Listings

1.1 Modifying an Object Composition 2

1.2 Program With Access Outside Collaboration 3

1.3 Mixin Potential Safety Issues 6

1.4 Flawed Design with Mixins 7

2.1 A Class Implementing Another Class 14

3.1 A *Point* class, with related mixins 35

3.2 Memory Representation of *Point* class, with related mixins 36

3.3 Initializing the program from Listing 3.1 37

3.4 Examples of "up" method calls 38

3.5 Library function for type casting 38

3.6 Adding Features Using Mixins 40

4.1 Correct Design with Mixins, with class *Stack* as in Listing 1.4 53

4.2 Flawed Mixins 61

4.3 Corrected Mixins 61

4.4 More Flawed Mixins 62

5.1 Type Definitions in Boogie 70

5.2 Association Module Variable Declarations 70

5.3 Specification Verification 72

5.4 Coupling Invariant Function 72

5.5 New Associations Object 73

5.6 Setting Association Object Fields 75

5.7 Clearing Association Object Fields 76

6.1 Decorator Pattern-Object Composition Solution 80

6.2 Decorator Pattern-Mixin Composition Solution 80

6.3 Proxy Pattern-Object Composition Solution 81

6.4 Proxy Pattern-Mixin Composition Solution 81

6.5 CofR Pattern-Object Composition Solution 83

6.6 CofR Pattern-Mixin Composition Solution 83

6.7 Strategy Pattern-Object Composition Solution 84

6.8 Strategy Pattern-Mixin Composition Solution 84

6.9 Dynamic Object Model Pattern-Type Code 86

6.10 Dynamic Object Model Pattern-Object/Client Code 86

6.11 Extension Object Pattern 88

6.12 Role Object Pattern 88

B.1 C Code Generated by Compiler of *Point* Class Example in Listing 3.1 102

C.1 C++ Sample 114

C.2 Java Sample 115

C.3 Python Inheritance Sample 116

C.4 Python Mixin Sample 117

Declaration of Academic Achievement

Burton E. and Sekerinski E. (2013) Correctness of Intrusive Data Structures Using Mixins In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, 6 pages, pp. 53–58.

Burton E. and Sekerinski E. (2014) Using Dynamic Mixins to Implement Design Patterns In *Proceedings of the 19th European Conference on Pattern Languages of Programs*, 19 pages, pp. 14:1–14:19.

Burton E. and Sekerinski E. (2015) The Safety of Dynamic Mixin Composition In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 8 pages, pp. 1992–1999.

Burton E. and Sekerinski E. (2016) An Object Model for a Dynamic Mixin Based Language In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 7 pages, pp. 1986–1992.

Burton E. and Sekerinski E. (2017) An Object Model for Dynamic Mixins In

Journal of Computer Languages, Systems & Structures, 12 pages, pp. (in print)

Chapter 1

Introduction

Any non-trivial software system is typically composed of small, discrete units with well-defined interfaces. This lets software developers focus on implementing solutions to manageable problems at the unit level. In theory, multiple units can be developed independently yet in parallel. System architects can abstract away low-level implementation considerations and concentrate on composing units to obtain the desired system properties. From an engineering standpoint, the benefits of such an approach are well known. Individual units are easier to maintain and evolve. They can also be reused in other systems requiring the same functionality. Furthermore, a unit can be replaced by one providing equivalent functionally without disrupting the entire system (Parnas 1972).

An object-oriented design approach is one method of decomposing such systems. In object-oriented programs, the primary means of decomposition is at the data level. A system is divided into entities, known as objects, which contain a set of related data and a set of operations that are used for data access and mutation. These objects emulate units described above. Properties such as data type and visibility are attached to each element within an object and externally visible elements define an object's interface. This interface is then used by other "client" objects in the system to access the object's internally stored data or initiate its operations.

The primary method of code reuse in object-oriented development has been "inheritance via subtyping". Mainstream statically typed, class-based OOP languages such as Java, C# and C++ support code reuse by allowing a developer to define superclass-subclass relationships. Classes are used to define the structure of these objects and act as blueprints for object construction at run-time. Each object is an instance of a particular class. A class specifies the data members and methods associated with that class's related objects. A subclass is then defined in terms of its parent so only a developer only needs to specify how it differs from its parent class.

All other parts of the subclass specification are inherited from its parent.

1.1 Object Composition to Facilitate Code Reuse

Despite the popularity of object-oriented development in industry, there are shortcomings about the paradigm that have been identified. When using inheritance for code reuse, a new class is created and defined as a subclass of the class which contains the code being reused. This approach becomes problematic when code is to be reused for multiple, potentially unrelated classes. In object-oriented design, the intent of the class hierarchy is to describe relationships between objects. Critics argue that subclassing solely for the purpose of reuse pollutes the class hierarchy making it difficult to understand (Szyperski 1992; Flatt, Krishnamurthi, and Felleisen 1998a; Van Limberghen and Mens 1996). Another limitation of relying on inheritance for reuse is that object modifications can only be done at compile-time and the class level, thus objects cannot change structure or behaviour during program execution.

These limitations can be addressed by using object composition. Object composition allows reuse by having individual objects play specific roles in a multiple-object collaboration. Here, objects store references to other objects they are dependent on in the collaboration and access additional functionality provided by forwarding messages to the appropriate object. An advantage of object composition is that the functionality provided can be easily changed at run-time. This is done by changing one of the objects in the collaboration. In Listing 1.1, the methods of *ExtObj* are available to *obj*. By replacing the *obj.ext* field with a reference to a different object, the behaviour of the composition has been modified. This example illustrates the more extreme situation where reference type of *obj.ext* has changed, so the method implementation that *obj.ext.someMethod*() refers to is different. The behaviour of the method called may be different even if the reference type is unchanged due to the values of the object's (*obj.ext*) internal fields.

Listing 1.1: Modifying an Object Composition

```
class BaseObject
   var ext : ExtObj
   ...

class ExtObj
   method someMethod()
      ....

class ExtObj2 extends ExtObj
```

```
    method someMethod()
        ....
begin
    BaseObject obj
    obj := new BaseObject
    obj.ext := new ExtObj // save the object reference
    obj.ext.someMethod()
    dispose obj.ext // destroy the extension object
    obj.ext := new ExtObj2 // create/save a new extension
    obj.ext.someMethod() // behaviour has changed from earlier invocation
end
```

Object composition allows run-time object behaviour changes but the features it provides are spread across multiple objects which breaks encapsulation. The developer is charged with ensuring that references to the objects in the collaboration are valid. Furthermore, since objects in the collaboration remain independent, the developer must ensure that objects receiving forwarding messages that are not modified in unexpected ways by external objects. In Listing 1.2, assume the functionality provided by *ExtObj* is intended to be accessed via *objA*. Copying the reference to *e* (or accessing *objA.ext* directly) allows the collaboration to be modified nefariously. Another important point is that in this collaboration, *objA* must have been designed to support the *ExtObj* extension in advance if a statically typed language is being used.

Listing 1.2: Program With Access Outside Collaboration

```
begin
    ...
    ExtObj e
    objA.ext := new ExtObj // create an extension obj, save the reference
    e := objA.ext // extension can be accessed outside collaboration
end
```

Also note that memory management is left to the developer. When *extObj* is released from the collaboration, they must ensure that it is destroyed to prevent memory leaks.

1.2 Alternatives to Object Composition

Various approaches have been developed in order to support reuse. Unlike composition, which uses multiple external run-time objects to modify an object's behaviour, these approaches bind code to an existing object at the class level (where all objects of the class type are affected) or at the object level (where only the current object is affected). There are two benefits that a developer would like to gain with such code segments.

- **object modification**. The code shall alter a set of objects' behaviour in a predictable way.
- **applicable in multiple contexts**. The same code can be applied to multiple, potentially unrelated objects.

Aspects

In aspect-oriented development, code segments are placed in advices which can be invoked at programmer defined join points (Kiczales, Hilsdale, Hugunin, Kersten, Palm, and Griswold 2001). It was developed to provide functionality required across many different features in a system.

Subjects

In subject-oriented development domain modelling accounts for the fact that objects can be viewed from different perspectives within a system (Ossher, Kaplan, Katz, Harrison, and Kruskal 1996). This results in a set of implementations associated with an interface's method declarations. The method implementation used when invoked is based on user-defined composition rules. Subjects change the behaviour of objects depending on current context but are not really designed for reuse.

Contexts

Context-Oriented Programming allows method invocation to be directed to different code based on the current receiving object's type and the current user-defined layer of execution (Hirschfeld, Costanza, and Nierstrasz 2008). Object behaviour modification is done by changing contexts at different points of execution as oppose to explicit object extension.

Traits/Talents

Traits are a group of methods that can be added to a class (Ducasse, Nierstrasz, Schärli, Wuyts, and Black 2006). In the event that a method name is equivalent to

one in the class it is being applied to, the programmer is tasked with providing a resolution at compile time via aliasing (renaming). Talents are traits applied at the object as opposed to the class level (Ressia, Gîrba, Nierstrasz, Perin, and Renggli 2014). Unlike traits however, objects can both acquire and relinquish talents at run-time.

These approaches to changing object behaviour follow seminal work on the topic in the late 1970s at MIT. A project called Flavors (Moon 1986) defines a type of object to be a *flavor*. Such flavors can be "mixed" together to define an object's behaviour.

This work focuses on dymamic mixins as a viable alternative for object extension and code reuse. The following section describes dynamic mixins in detail.

1.3 Mixins as a Reuse Tool

Mixins are design elements that support the maintainability and incremental development of large programs. Mixins do not stand on their own but have to be bound to provide functionality. They were originally introduced as abstract classes, which are subsequently composed with superclasses in order to create objects (Moon 1986); more generally, mixins allow a unifying treatment of diverse inheritance mechanisms (Bracha and Cook 1990).

Static mixins are code fragments that can be composed with classes. Typically they are used to encapsulate behaviour that can be reused with different classes. They contain methods and fields but are incomplete because they can refer to methods that have no associated implementation. Static mixin composition can be viewed as the application of a function that takes in a class and a set of static mixins and returns a class containing a union of the fields and methods from each of its constituents. This operation provides bindings for any unresolved method calls found in the mixins provided. Semantics vary per implementation but some linearization scheme is used to order the mixins applied to a class. This determines how a program searches for an object's particular field or method when referenced. Some mixins can add their intended behaviour without assistance from the objects they are bound to. These are known as *free mixins* (Simons 2004). Methods included in *bound* (or incomplete) *mixins* have references to their associated objects contained within them. Dynamic mixins are applied to individual objects at run-time, extending the object with its contained fields and methods. For the purposes of this work, the term "mixin" refers to a dynamic mixin (applied to an object at run-time) unless explicitly stated otherwise.

Dynamic mixins allow the composition and modification of objects at run-time; this is in contrast to the composition at compile-time, as is done through inheritance. Dynamic mixins can be used for the modelling of roles that objects may acquire and relinquish (VanHilst and Notkin 1996), for efficient data structure implementations (Burton and Sekerinski 2013), and for more flexibly adding features to a base product

(Apel, Leich, and Saake 2006). Dynamic mixins can be expressed in several recent languages. They are directly supported in Groovy (Subramaniam 2008), Perl 6, and Ruby (Fitzgerald 2007). Dynamic mixins can be expressed in JavaScript and Python through object augmentation (Harmes and Diaz 2007) by adding and overwriting object fields that are methods. In Python, static mixins are directly expressed as classed and composed through multiple inheritance (Lutz 2008).

1.4 Current Issues with Mixins

1.4.1 Type Safety

Dynamic mixins add run-time flexibility but can introduce safety problems in dynamically typed languages. This is due to the fact that one cannot ensure that dynamic extensions to objects are available when accessed at run-time. In Listing 1.3, object *a* will always be able to call the method *provideServiceA*() because its class definition *CA* has guaranteed that it will implement the *ServiceA* interface. At the point where the statement (*b* **as** *ServiceA*).*provideServiceA*() is executed however, it is unknown whether the object *b* has been extended to include the *provideServiceA*() method. This depends on the value of *someCondition* earlier.

Listing 1.3: Mixin Potential Safety Issues

```
class ServiceA
    provideServiceA()
    ...

class CA extends ServiceA
    ...

class CB
    ...

class MA extends ServiceA
    ...

begin
    var someCondition : boolean
    var extendObj : ServiceA
    CA a := new CA()
    CB b := new CB()
```

```
    ...
    if someCondition then
        extend b with MA
        ...
    a.provideServiceA () // safe
    (b as ServiceA).provideServiceA () // unsafe
end
```

1.4.2 Interference

Dynamic mixins support *separation of concerns*, a term that was used by Dijkstra in the 70's and underlies Parnas' principles for modularization. However, interaction among dynamic mixins can be subtle and lead to unexpected consequences as their order of composition is not statically fixed. The example in Listing 1.4, adapted from (Prehofer 2001), illustrates this. We specify a stack using sequences, writing $\langle\rangle$ for the empty sequence and $e \rightarrow s$ for prepending element e to sequence s. Method parameters are passed by value and the type of a method's return value is specified as part of its signature.

Listing 1.4: Flawed Design with Mixins

```
class Stack
    var s : seq(integer) := ⟨⟩
    method push(val e : integer)
        s := e → s
    method pop() : integer
        var e : integer
        e, s := head(s), tail(s)
        return e
    method size() : integer
        return len(s)

class Lock extends Stack
    var l : boolean := false
    method lock()
        l := true
    method unlock()
        l := false
    method push(val e : integer)
        if ¬l then Stack.push(e)
```

```
    method pop() : integer
        if ¬l then return Stack.pop()

class Counter extends Stack
    var c : integer
    method init()
        Stack.size(c)
    method push(val e : integer)
        c := c + 1 ; Stack.push(e)
    method pop() : integer
        c := c − 1 ; return Stack.pop()
    method size() : integer
        return c

class Encrypt extends Stack
    method push(val e : integer)
        Stack.push(9 − e)

var log : integer := 0

class Logging extends Stack
    method push(val e : integer)
        Stack.push(e) ; log := log + 1
    method pop() : integer
        val e : integer
        e := Stack.pop() ; log := log + 1
        return e
    method size() : integer
        val n : integer
        n := Stack.size() ; log := log + 1
        return n
```

Indentation is used instead of explicit bracketing. All methods are assumed to be *public* and all fields are assumed to be *private*. Any class can be a mixin, which can be added to an object at run-time. Mixins, however, may need certain features to be present, hence they are "abstract subclasses." The **extends** clause is used to explicitly specify that dependency. A mixin accesses external features by invoking method calls in the form $C.m()$, where C is the feature's class name.

We write $r :=$ **new** C for creating an object x of class C and **extend** r **with** D for extending object r with D. This extension requires that classes needed by mixin D are *implemented* by r. For example, *Stack* can be implemented by a class that uses arrays and pointers instead of sequences, as long as it provides the methods *push*, *pop*, and *size*. Object r can also include other mixins. Thus, r does not have to be

"exactly" as needed by D, simply "at least" as needed by D. For example,

$x :=$ **new** *Stack*() ;
extend x **with** *Counter* ;
extend x **with** *Lock*

will first create a *Stack* object and then add *Counter* and *Lock* mixins to it. The purpose of *Counter* is to keep an explicit count of the number of elements in the stack such that the number can more efficiently be queried. The purpose of *Lock* is to provide additional functionality by allowing the stack to be unmodifiable.

Method calls are resolved in linear order beginning with the last mixin bound to the object and ending with the object's base class. Above, after the creation of x, the call $x.push(a)$ leads to the calling sequence $Lock.push \rightsquigarrow Counter.push \rightsquigarrow Stack.push$ and the effect is $x.c := x.c + 1 \; ; \; x.s := a \to x.s$ (because $Lock.l$ is false). Along the same lines, the statement

$$x.push(a) \; ; \; x.lock() \; ; \; x.push(b) \; ; \; x.size(n) \tag{1}$$

also has the effect of $x.c := x.c + 1 \; ; \; x.s := a \to x.s$ and additionally sets n to 1 because the call $x.push(b)$ will resolve to *Lock.push*, which will return immediately as the field l has been set to **true** by the call $x.lock()$. However, if x is composed in a different order,

$x :=$ **new** *Stack*() ;
extend x **with** *Lock* ;
extend x **with** *Counter*

then (1) would also have the effect of $x.c := x.c + 1 \; ; \; x.s := a \to x.s$ but sets n to 2. In this case, the call $x.push(b)$ resolves to $Counter.push \rightsquigarrow Lock.push$, which increments c in *Counter.push* and returns immediately in *Lock.push*. Now the call $x.size()$ does not return the size of the stack as $x.c$ does not reflect the size of the stack, contrary to the purpose of *Counter*. Intuitively, the problem is the assumption that $Counter.c$ always equals the size of the stack is broken. This assumption can be formally expressed by the invariant $c = len(s)$ in *Counter*.

The *interference* of *Lock* with *Counter* violates the separation of concerns principle as *Counter* and *Lock* are not truly independent. Suppose that in the development of a large system, stacks with locks are needed to implement the functionality of module L, stacks with counters are needed to implement the functionality of module M, and modules L and M are developed independently by different developers. Also, assume that each developer is not aware that the other may extend stacks with mixins. The extension of a stack object with a counter mixin only works as the developers of M intended if no lock mixin is present in that stack object. The existence of a lock mixin, however, is unknown to the developers of module M. Even if it were known, the developers would have to agree to the order in which the mixins are applied.

The purpose of the class *Logging* in Listing 1.4 is to count the number of calls to *push* and *pop* across objects. Now consider the sequence

> x := **new** *Stack*() ;
> **extend** x **with** *Lock* ;
> **extend** x **with** *Logging* ;
> *x.lock*() ; *x.push*(a)

which will increment *log* by 1. However, if *Lock* and *Logging* are added in reverse order,

> x := **new** *Stack*() ;
> **extend** x **with** *Lock* ;
> **extend** x **with** *Logging* ;
> *x.push*(a)

then *log* will not be incremented. Thus, no invariant is violated, but the order in which mixins are added changes the result of the computation.

1.4.3 Inefficient Method Lookup

Static, object-oriented languages that do not support dynamic mixins can have a static object memory layout. Once an object is created, its fields and their types are known. A fixed offset to fields provides constant time access to an object's fields at run-time. Languages that provide object extension must have some lookup code added to each field access request since field location cannot be calculated at run-time. Dynamically typed languages allow object extension by storing object members using some abstract data structure. A scheme is also required to handle name clashes. The time required to access to field members is bounded by the type of data structure used.

A linearization order determines the order that components within an object are inspected when searching for an object's member. It also is used for method combination. The linearization tree formed with inheritance (or static mixins) is determined at compile-time so location of all subclasses within an object is known. On the other hand, the order in which the dynamic mixins are added is not known until run-time. This complicates the compiler's object model as this information must be encoded in the object.

Finally, casting an object to a type involves finding the location of a particular mixin within an object. Since a mixin location is not fixed, an efficient means of traversing the object to find this information is required.

1.5 Summary of Contributions

The primary goal of this work is to address the concerns listed in Section 1.4. In particular, the thesis addresses the question of whether dynamic mixins can be used

as a type-safe, efficient method of code reuse. To accomplish this, the following contributions have been documented:

Mixin Formalization Current theory in literature about object-oriented design focuses on subclassing and subtyping. In this work, we extend a theory which models object-oriented programs to formalize dynamic mixins (Burton and Sekerinski 2014). (Chapter 2)

Mixin Refinement Data refinement is used to ensure that a given module is a correct specialization of a data specification. In object-oriented development, there has been much work in ensuring that subclasses are refinements of their parent classes. A definition for mixin-based refinement is provided and rules to help ensure that mixin composition is "safe" are presented (Burton and Sekerinski 2015). (Chapter 4)

Mixin Language The survey of current languages supporting dynamic mixins as a language construct shows that most are dynamically typed languages which do not support type safety. To address this shortcoming, a statically typed language mix[1] is presented (Burton and Sekerinski 2014). (Chapter 2)

Object Model Current languages that support dynamic mixins implement classes as a set of attributes stored in an abstract data structure. In this work, an alternative object model is presented which supports dynamic mixins while providing constant time access to attributes and methods (Burton and Sekerinski 2016). (Chapter 3)

Mixin Implementation A prototype compiler has been developed in order to test the practicality of mix and implement a concrete version of the mixin-refinement theory. The implementation also uses the object model developed (Burton and Sekerinski 2017). (Chapter 3)

Use Cases Finally, we present a set of use cases to validate the applicability of the work for practitioners (Burton and Sekerinski 2013), (Burton and Sekerinski 2014).(Chapter 5, 6)

[1] pronounced "mix"

Chapter 2

mix, a Statically Typed Language for Dynamic Mixins

Mixin features are presented here as implemented in mix, a language developed to support the flexible, modular, and safe extension of objects. The distinguishing feature of mix is that dynamic mixins are the primary means of code reuse; dynamic mixins allow reuse that can be achieved by static mixins and inheritance. Despite the dynamic nature of the language, mix statically ensures that method calls are defined. Only mixin-related aspects of the mix language are discussed.

2.1 Language Goals

The purpose of mix is to support the structured use of dynamic mixins. With this in mind, the following are design goals of the language.

1. *support for implementing roles and features*
2. *simple object evolution*
3. *object safety*

A subset of an object's properties are relevant in a particular collaboration can be seen as a *role* (Kristensen and Osterbye 1996). In this context, a collaboration is a group of objects that work together to provide a service or implement an application feature. An object can take on many roles and participate in multiple collaborations over its lifetime, some of them concurrently. In mix, the intention is to give developers the ability to express the fact that objects can routinely enter and exit collaborations via adding and dropping roles during program execution.

Developers should be able to easily update the behaviour of an object based on the system state or environment. This is essential for the development of self-aware and context-aware systems (Salehie and Tahvildari 2009) (Ceri, Daniel, Matera, and

Facca 2007). Furthermore, feature improvement should be applied in a modular fashion. The goal is to make the development and support of such systems as simple as possible.

Dynamic mixins provide flexibility but we wish to prevent object composition that violates object invariants. An *object invariant* is defined as a property of its state space (set of fields) that is expected to hold throughout its lifetime. In particular, we concern ourselves with bound mixins that require access to their associated object's fields.

2.2 mix Abstract Syntax

The abstract syntax for mix is presented below. The non-terminal *id* is a sequence of letters. A line over a term stands for that term being repeated zero to many times. A term in square brackets is optional and occurs at most once. Text in courier font is a terminal keyword in the language. The concrete syntax in Appendix A includes extra terminal symbols and distinguishes between lists and single terms. Note that the abstract syntax uses result variables to store values returned from methods. Their associated result values can be used in program correctness proofs. The concrete syntax passes method results to its caller using the **return** statement as is common in many programming languages.

$\langle program \rangle$ ::= $\textbf{program}\ id\ \overline{\langle class \rangle}\ \langle statement \rangle$

$\langle class \rangle$ ::= $\textbf{class}\ id\ [\textbf{extends}\ id]\ [\textbf{implements}\ id]\ \overline{\langle member \rangle}$

$\langle member \rangle$ ::= $\langle constant \rangle \mid \langle variable \rangle \mid \langle method \rangle \mid \langle init \rangle$

$\langle constant \rangle$ ::= $\textbf{const}\ \overline{id}\ \langle type \rangle\ \langle expression \rangle$

$\langle variable \rangle$ ::= $\textbf{var}\ \overline{id}\ \langle type \rangle$

$\langle method \rangle$ ::= $\textbf{method}\ id\ \textbf{(}\overline{\textbf{val}\ \overline{id} : \langle type \rangle}\textbf{)}\ [\textbf{res}\ \text{id}\ \langle type \rangle]\ \overline{\langle variable \rangle}\ \langle statement \rangle$

$\langle init \rangle$ ::= $\textbf{initialization}\ \textbf{(}\overline{\textbf{val}\ \overline{id} : \langle type \rangle}\textbf{)}\ \overline{\langle variable \rangle}\ \langle statement \rangle$

$\langle designator \rangle$ ::= $id\ \overline{.id}$

$\langle statement \rangle$ ::= $\langle designator \rangle := \langle expression \rangle$
| $\textbf{if}\ \langle expression \rangle\ \textbf{then}\ \langle statement \rangle\ \textbf{else}\ \langle statement \rangle$
| $\textbf{extend}\ \langle expression \rangle\ \textbf{with}\ id$
| $\textbf{implement}\ \langle expression \rangle\ \textbf{with}\ id$

	|	**dispose** ⟨*expression*⟩
	|	⟨*expression*⟩ ($\overline{\langle \mathit{expression} \rangle}$,$\overline{id}$)
	|	⟨*statement*⟩ **;** ⟨*statement*⟩
⟨*expression*⟩	::=	**nil** | **true** | **false** | 0 | 1 | ...
	|	**new** *id* $\overline{\langle \mathit{expression} \rangle}$
	|	*id* **as** *id* | *id* **has** *id*
	|	⟨*expression*⟩ ∧ ⟨*expression*⟩ | ⟨*expression*⟩ + ⟨*expression*⟩
		...
⟨*type*⟩	::=	**boolean** | **integer** | ...

2.3 Language Definition

Classes serve for both creating and adding mixins to objects. Class *fields* are assumed to be private to the class and class *methods* are assumed to be public. Classes may *extend* another class and may *implement* another class; at most one class can be extended and one class implemented. Classes that don't extend another class can be directly instantiated or used as a free mixin. Classes that extend another class require objects of other classes to be present and are used as bound mixins.

If class D extends C, then D may add *new* methods and must *override* methods of C. In order to reuse methods of C, an overriding method in D has a method body that makes a "super-call" to the corresponding C method. If a method m of C is not explicitly overridden in D, it is assumed to be defined as the "super-call" $C.m()$ in D. If class D' implements class D, then D' must define all methods of D: type-checking allows objects of class D' to be used wherever objects of class D are expected. Fields of D' are only those declared in D', fields of D are not "inherited". If D extends C, then D' must also extend C. Listing 2.1 gives an example of how the state space is replaced in an implementing class. In this example, *IntLock* implements *Lock* (found in Listing 1.4) by using an integer field instead of a boolean field. The method *length*() uses a default implementation from the *Stack* object that the class is bound to at runtime.

Listing 2.1: A Class Implementing Another Class

```
class IntLock extends Stack implements Lock
  var n : integer
  initialization()
    n := 0
  method push(t : integer)
    if n = 0 then
      Stack.push(t)
  method lock(t : integer)
    n := 1
```

```
method length() : integer
  return Stack.length()
```

If D extends C, we write $extends(D) = C$. The function *extends* is defined for all classes except class *Object*, a predefined class. If a class declaration does not include an **extends** clause, it implicitly extends *Object*. The notions of sub- and superclasses as well as sub- and supertypes are the transitive and reflexive extensions of the *extends* and *implements* relations:

$$\begin{aligned} superclasses(C) &= \text{set of all classes that } C \text{ transitively extends, including } C \\ subclasses(C) &= \text{set of all classes that transitively extend } C\text{, including } C \\ supertypes(C) &= \text{set of all classes that } C \text{ transitively implements, including } C \\ subtypes(C) &= \text{set of all classes that transitively implement } C\text{, including } C \end{aligned}$$

The declaration of an object variable specifies the type to be a class, as in **var** *s*: *Stack*. Variable *s* may refer to a *Stack* object or to one that implements *Stack*. The object can have other mixins, like *Lock*, but through *s* only the functionality of the *Stack* mixin can be accessed, even if a call to *s.push* may lead to another mixin, like *Lock*. The statements and expressions in mix relating to mixins are:

$x :=$ **new** C create object x of class C: allocate the object and execute its initializer. Class C must be or implement the declared class of x and C must not extend another class.

$y :=$ **extend** x **with** D extend object x with mixin D and let y refer to the new mixin; if mixin D is already present in x, raise an exception; if D extends a class E distinct from *Object* and an E mixin is not present in x, raise an exception. This ensures that mixin D occurs in x at most once and that all mixins extended by D are already contained in the object. The order in which mixins are added to an object is maintained. This is used for method call resolution. If used as a statement, like **extend** *s* **with** *Lock* for the example in Listing 1.4, the returned reference to *Lock* is discarded.

$z :=$ **implement** x **with** D given the static type of s of object x, replace the mixin that implements s with a new mixin segment of type D and let z refer to this new segment. If mixin D does not implement s, raise an exception. The new mixin segment of type D assumes the same place in the linearization order as the one it replaced.

x **has** D test if one of the mixins of x is of class D or implements D; x may be declared of any class.

x **as** D return an object of class D by casting x, if x has D, otherwise raise an exception; object x may be declared of any class. For example, given s : *Stack*

and $l : Lock$, if $l := s$ **as** *Lock* succeeds, *l.lock* may be called. Converting to *Object* will always succeed. For example, s and l cannot be compared for equality, as they have different types, but s **as** *Object* $= l$ **as** *Object* tests if s and l refer to mixins of the same object.

$x.m()$ execute the *last* mixin that was added to x and that implements a class that extends C, assuming x is declared to be of class C, $x : C$ and C includes method m; that is, call the last added method m. (If unrelated classes define methods m, these are unrelated methods.) Within a class, the "self-call" to another method of the same class is written as $m()$.

$C.m()$ "super-call" m by executing the *previous* mixin that was added and implements m, provided that the class in which $C.m()$ is called is a proper subclass of C and m is a method of C.

The *extends* relation between related mixins forms a tree. This relation is *linearized* according to the order of extension. Self-calls in one mixin go to the last mixin, super-calls go to the previous mixin. Both self-calls and super-calls are dynamically bound.

Since an object may contain several unrelated mixins, meaning that they don't have a common superclass except *Object*, each set of related classes has it own linearization order. Mixins don't have to be of the exact extended type, only of an implemented type.

Consider the example in Listing 1.4 where a base stack data structure is decorated with both a counter and a locking feature mixin. The call *s.push*(5) goes to the end of the linearization chain, here to mixin *Counter*. By making the super-call *Stack.push*() in each of the object extensions that extend *Stack*, all mixins in the linearization chain rooted in *Stack* will be called and can reestablish their own local mixin invariant (Burton and Sekerinski 2015): the call *Stack.push*(t) in *Lock* goes to *Counter* and the call *Stack.push*(t) in *Counter* goes to *Stack*.

2.4 Differentiating Language Features

Unlike in dynamically typed languages, adding individual fields to objects in mix is not allowed. The rationale is that developers must organize their classes into features or roles before they are composed with objects. Such functionality can be obtained using the Decorator pattern in a statically typed language, however a base object must anticipate its extension and declare a pointer to a specific class extension type at compile time (Burton and Sekerinski 2014). Future extensions involve modifying the base object's class code. In mix, the base object is unaware of any extension. "Super-calls" are made without knowing the actual type of the class that will receive it. This call is safe because the receiver is guaranteed to implement the interface that the sender needs.

Statically typed object-oriented languages such as C++ do not support dynamic super-calls natively. The Extension Object Pattern provides a means of developing object extensions but does not consider super-calls between extensions (Gamma 1997). In Java, one can simulate this by generating a linked list of collaborating objects. The reflection API can then be used to obtain class type and interface information about the objects. Method calls would then be routed to the appropriate class based on this information. Type safety is not guaranteed because class and method name would have to be passed by the caller.

While statically typed languages like C++ and Java don't directly support mixins, extensions of such languages with static mixins have been proposed (Ancona, Lagorio, and Zucca 2003; Flatt, Krishnamurthi, and Felleisen 1998b; Allen, Bannet, and Cartwright 2003). Dynamically typed languages are well suited to support dynamic mixins. Flavors is an extension of Lisp in which objects are created by composing mixins, called "flavors" (Moon 1986). It includes multiple means of method combination where a programmer can define a sequence of method calls from an object's different constituent flavors when one of its methods is invoked. XOTcl, an extension of OTcl (Neumann and Zdun 1999), provides a simpler method combination technique where an object's methods with the same name are linearized and the keyword **next** is used to invoke the next method in the list. In Python and JavaScript, mixins are not natively available but since attributes can be added to objects dynamically, mixins can be simulated using additional code (Harmes and Diaz 2007). Ruby supports dynamic mixins (Fitzgerald 2007) but does not allow them to contain state or provide a way to remove them. Dynamic mixins can be added to Groovy objects using the **metaClass** attribute at runtime (Subramaniam 2008) or via the static **Class.mixin** attribute (CodeHaus 2014).

2.5 Formal Definition of $\dot{m}$ix

The formal definition of $\dot{m}$ix is in two steps: first we define a core language by the weakest precondition predicate transformer. Then we define $\dot{m}$ix by translation to the core language.

2.5.1 Abstract Syntax for Core Language

The abstract syntax of the core language is as follows. A line over a term stands for that term being repeated zero or more times. The core language adds statements such as non-deterministic assignment and composition which allows a programmer to build non-executable program specifications. Guards and preconditions are also added. A procedure type is introduced so that methods can be passed as parameters.

$$
\begin{array}{rcll}
\langle \mathit{statement} \rangle & ::= & \mathbf{skip} & \text{(empty statement)} \\
& | & \mathbf{abort} & \text{(failed statement)} \\
& | & \mathbf{assume}\ \langle \mathit{expression} \rangle & \text{(guard)} \\
& | & \mathbf{assert}\ \langle \mathit{expression} \rangle & \text{(precondition)} \\
& | & \overline{\mathit{id}} := \overline{\langle \mathit{expression} \rangle} & \text{(multiple assignment)} \\
& | & \overline{\mathit{id}} :\in \langle \mathit{expression} \rangle & \text{(nondeterministic assignment)} \\
& | & \langle \mathit{expression} \rangle (\overline{\langle \mathit{expression} \rangle}, \overline{\mathit{id}}) & \text{(call)} \\
& | & \langle \mathit{statement} \rangle \mathbin{[\!]} \langle \mathit{statement} \rangle & \text{(nondeterministic composition)} \\
& | & \langle \mathit{statement} \rangle ; \langle \mathit{statement} \rangle & \text{(sequential composition)} \\
& | & \overline{\mathit{declaration}}\ \mathbf{in}\ \langle \mathit{statement} \rangle & \text{(declaration)}
\end{array}
$$

$$
\begin{array}{rcll}
\langle \mathit{declaration} \rangle & ::= & \mathbf{const}\, \overline{\mathit{id}} = \langle \overline{\mathit{expression}} \rangle & \text{(constant)} \\
& | & \mathbf{var}\, \overline{\mathit{id}} : \overline{\mathit{type}} \mid \langle \mathit{expression} \rangle & \text{(variable)}
\end{array}
$$

$$
\begin{array}{rcll}
\langle \mathit{expression} \rangle & ::= & \mathbf{val}\, \overline{\mathit{id}} : \overline{\mathit{type}}\ \ \mathbf{res}\, \overline{\mathit{id}} : \overline{\mathit{type}} \bullet \langle \mathit{statement} \rangle & \text{(procedure)} \\
& | & \langle \mathit{expression} \rangle \wedge \langle \mathit{expression} \rangle \mid \langle \mathit{expression} \rangle + \langle \mathit{expression} \rangle \ldots & \\
& & & \text{(other expressions)}
\end{array}
$$

$$
\begin{array}{rcll}
\langle \mathit{type} \rangle & ::= & \overline{\mathit{type}} \mapsto \overline{\mathit{type}} & \text{(procedure type)} \\
& | & \mathbf{boolean} \mid \mathbf{integer} \mid \ldots & \text{(other types)}
\end{array}
$$

$$
\langle \mathit{program} \rangle ::= \mathbf{program}\, \mathit{id}\ \langle \mathit{statement} \rangle \qquad \text{(main program)}
$$

A program declares the extent variable *ref* and initializes it to the empty set:

$$
\mathbf{program}\, P\ S \mathrel{\hat{=}} \mathbf{var}\, \mathit{ref} : \mathit{Ref} := \{\}\ \mathbf{in}\, S
$$

2.5.2 Core Language Semantics

The core language is similar to those of other refinement calculi, e.g. (Back and Wright 1998; Morgan 1998), with the exception of considering statements as values (Naumann 1995), which we treat here syntactically by defining the syntax of statements as expressions, rather than semantically, as this is sufficient for our purpose. Procedures are expressions: a *procedure* is formally a triple, a statement with formal value and result parameters, of procedure type, e.g. $\mathbf{val}\ i : \mathit{integer}\ \mathbf{res}\ b : \mathbf{boolean} \bullet b := \mathit{even}(i)$ is of type $\mathbf{integer} \mapsto \mathbf{boolean}$. In the call $f(e, x)$, expression f is a procedure and e, x are the actual value and result parameters. We assume that all programs are well-typed and may leave out the types for brevity.

The correctness assertion $\{p\}S\{q\}$ means that if predicate p holds initially, then statement S terminates and predicate q holds finally. This is equivalent to stating that p implies the *weakest precondition* of S to establish q, written as $p \Rightarrow wp(S, q)$. The predicate transformer wp is defined in the standard way for common statements. Let x be an identifier, X a type, b, e, q expressions, and S, T statements. An overline indicates a list, e.g. $\overline{x}$ is a list of identifiers:

$$
\begin{aligned}
&wp(\mathbf{abort}, q) \hat{=} \mathit{false}\\
&wp(\mathbf{skip}, q) \hat{=} q\\
&wp(\mathbf{assume}\, b, q) \hat{=} b \Rightarrow q\\
&wp(\mathbf{assert}\, b, q) \hat{=} b \wedge q\\
&wp(\overline{x} := \overline{e}, q) \hat{=} q[\overline{x} \backslash \overline{e}]\\
&wp(\overline{x} :\in e, q) \hat{=} \forall \overline{x} \in e \bullet q\\
&wp(f(\overline{e}, \overline{x}), q) \hat{=} \forall \overline{w} : \overline{W} \bullet wp(S, q[\overline{x} \backslash \overline{w}])[\overline{v} \backslash \overline{e}] \quad \text{where} \quad f = \mathbf{val}\, \overline{v} : \overline{V}\, \mathbf{res}\, \overline{w} : \overline{W} \bullet S\\
&wp(S \,[\!]\, T, q) \hat{=} wp(S, q) \wedge wp(T, q)\\
&wp(S\, ;T, q) \hat{=} wp(S, wp(T, q))\\
&wp(\mathbf{const}\, \overline{x} = \overline{e}\, \mathbf{in}\, S, q) \hat{=} wp(S[\overline{x} \backslash \overline{e}], q)\\
&wp(\mathbf{var}\, \overline{x} : \overline{X} \mid b\, \mathbf{in}\, S, q) \hat{=} \forall \overline{x} : \overline{X} \bullet b \Rightarrow wp(S, q) \quad \text{provided } x \text{ not free in } q
\end{aligned}
$$

The **abort** statement guarantees nothing about the program's state or termination. The **skip** statement leaves the program's state unmodified and always terminates. The multiple assignment $\overline{x} := \overline{e}$ updates all variables $\overline{x}$ simultaneously; type-checking ensures that the expressions $\overline{e}$ are of the correct types. The nondeterministic assignment $x :\in e$ assigns any element of the set e to x and *blocks* if e is empty; it generalizes to a tuple $\overline{x}$ of variables. The assumption $\mathbf{assume}\, b$ skips if b is true and blocks otherwise. The assertion $\mathbf{assert}\, b$ skips if b is true and *aborts* otherwise. The nondeterministic choice $S \,[\!]\, T$ selects either operand, whichever is not blocked. If either operand aborts, the whole statement aborts. For the procedure call $f(\overline{e}, \overline{x})$, type-checking ensures that f is of the appropriate form. The term $wp(S, q[\overline{x} \backslash \overline{w}])[\overline{v} \backslash \overline{e}]$ implies that finally $\overline{w}$ is assigned to $\overline{x}$ and that initially $\overline{e}$ is assigned to $\overline{v}$. The universal quantification of $\overline{w}$ expresses that $\overline{w}$ is initialized arbitrarily. The sequential composition is as usual. The constant declaration $\mathbf{const}\, \overline{x} = \overline{e}\, \mathbf{in}\, S$ is defined by substituting

$\overline{x}$ by $\overline{e}$ in S, which is well-defined as S is an expression. Both constant and variable declarations introduce bound "variables", which follow the usual rules of nesting. The variable declaration $\textbf{var}\, \overline{x} : \overline{X} \mid b\, \textbf{in}\, S$ initializes variables $\overline{x}$ to any element such that b holds, or blocks if none exist.

Two statements are semantically equal (rather than equal as terms) if they always establish the same postcondition:

$$S = T \mathrel{\hat{=}} \forall q \bullet wp(S, q) = wp(T, q)$$

Following example illustrates parameter passing, procedures as values, and substitution in statements; the types are left out:

$$\begin{array}{ll}
\quad wp(m := (\textbf{val}\, x\, \textbf{res}\, y \bullet y := 2 \times x)\ ;\ m(3, a), a = 6) & \\
\equiv wp(m(3, a), a = 6)[m\backslash\, \textbf{val}\, x\, \textbf{res}\, y \bullet y := 2 \times x] & (wp \text{ of } ;, :=) \\
\equiv wp((\textbf{val}\, x\, \textbf{res}\, y \bullet y := 2 \times x)(3, a), a = 6) & \text{(substitution)} \\
\equiv \forall y \bullet wp(y := 2 \times x, (a = 6)[a\backslash y])[x\backslash 3] & (wp \text{ of call}) \\
\equiv \forall y \bullet wp(y := 2 \times x, y = 6)[x\backslash 3] & \text{(substitution)} \\
\equiv\ true & (wp \text{ of } :=\text{, substitution, logic})
\end{array}$$

As a note, this treatment of procedures as values is general enough to allow "self-modifying" programs; for example, $wp(m := (m := \textbf{skip})\ ;\ m(), m = \textbf{skip})$ is indeed *true*. Statements of the core language can be extended as needed, for example with **require** and **if** statements:

$$\begin{array}{lll}
\langle statement \rangle & ::= & \ldots \\
& \mid & \textbf{require} \langle expression \rangle\ \textbf{then} \langle statement \rangle \\
& \mid & \textbf{if} \langle expression \rangle\ \textbf{then} \langle statement \rangle\ \textbf{else} \langle statement \rangle \\
& \mid & \textbf{if} \langle expression \rangle\ \textbf{then} \langle statement \rangle
\end{array}$$

Those are defined in terms of the core statements:

$$\begin{array}{l}
\textbf{require}\, b\, \textbf{then}\, S \mathrel{\hat{=}} \textbf{assert}\, b\ ;\ S \\
\textbf{if}\, b\, \textbf{then}\, S\, \textbf{else}\, T \mathrel{\hat{=}} (\textbf{assume}\, b\ ;\ S)\ [\!]\ (\textbf{assume}\, \neg b\ ;\ T) \\
\textbf{if}\, b\, \textbf{then}\, S \mathrel{\hat{=}} \textbf{if}\, b\, \textbf{then}\, S\, \textbf{else}\, \textbf{skip}
\end{array}$$

Loops can be defined in terms of fixed points, e.g. (Back and Wright 1998; Morgan 1998). We skip their treatment as they are not needed for the formalization of mixins. Object fields are defined as functions from object references to their values. We extend

expressions and assignments accordingly:

$$\langle expression \rangle ::= \ldots \\ \quad \mid \langle expression \rangle.id \qquad \text{(field selection)}$$

$$\langle statement \rangle ::= \ldots \\ \quad \mid \langle expression \rangle.id := \langle expression \rangle \qquad \text{(field assignment)} \\ \quad \mid \langle expression \rangle.id :\in \langle expression \rangle \text{(nondeterministic field assignment)}$$

The dot notation $x.f$, used for referring to field f of object x, is synonymous to $f(x)$. For example, $x.f := x.f + 1$ is synonymous to $f(x) := f(x) + 1$. We write $f[x \leftarrow e]$ for modifying function f to return e given the argument x:

$$f(d) := e \mathrel{\hat{=}} f := f[d \leftarrow e]$$
$$f(d) :\in e \mathrel{\hat{=}} \mathbf{var}\, h \mid h \in e \bullet f := f[d \leftarrow h]$$

We add procedure declarations, class declarations, and for convenience, variations of variable declarations:

$$\langle declaration \rangle ::= \ldots$$
$$\mid \mathbf{var}\, \overline{id} : \overline{type}$$
$$\mid \mathbf{var}\, \overline{id} : \overline{type} := \overline{\langle expression \rangle}$$
$$\mid \mathbf{procedure}\, id(\mathbf{val}\, \overline{id} : \overline{\langle type \rangle}\, \mathbf{res}\, \overline{id} : \overline{type})\ \langle statement \rangle$$
$$\mid \mathbf{class}\, id\, \mathbf{var}\, \overline{id} : \overline{\langle type \rangle}$$
$$\quad \overline{\mathbf{method}\, id(\mathbf{val}\, \overline{id} : \overline{\langle type \rangle}\, \mathbf{res}\, \overline{id} : \overline{\langle type \rangle})\ \overline{\langle statement \rangle}}$$
$$\mid \mathbf{class}\, id\, \mathbf{extends}\, \overline{id}\, \mathbf{var}\, \overline{id} : \overline{\langle type \rangle}$$
$$\quad \overline{\mathbf{method}\, id(\mathbf{val}\, \overline{id} : \overline{\langle type \rangle}\, \mathbf{res}\, \overline{id} : \overline{\langle type \rangle})\ \langle statement \rangle}$$
$$\mid \mathbf{class}\, id\, \mathbf{implements}\, \overline{id}\, \mathbf{var}\, \overline{id} : \overline{\langle type \rangle}$$
$$\quad \overline{\mathbf{method}\, id(\mathbf{val}\, \overline{id} : \overline{\langle type \rangle}\, \mathbf{res}\, \overline{id} : \overline{\langle type \rangle})\ \langle statement \rangle}$$

Not specifying an initializing predicate, as in $\mathbf{var}\, x : X$ means that x is initialized to an arbitrary value; alternatively, a variable may be initialized to a specific value, as in $\mathbf{var}\, x : X = e$, or $\mathbf{var}\, x = e$ if the type can be inferred.

$$\mathbf{var}\, x : X \mathrel{\hat{=}} \mathbf{var}\, x : X \mid true$$
$$\mathbf{var}\, x : X := e \mathrel{\hat{=}} \mathbf{var}\, x : X \mid x = e$$

Declaring a variable $\mathbf{var}\, x : C$, where C is a class, makes x of type *Ref*, the set of all possible object references, and associates the class C to x. A procedure declaration is just a shorthand for a constant declaration:

$$\mathbf{procedure}\, m(\mathbf{val}\, v : V\ \mathbf{res}\, w : W)\ S \mathrel{\hat{=}} \mathbf{const}\, m = \mathbf{val}\, v : V\ \mathbf{res}\, w : W \bullet S$$

A *module* A is formally a structure with a set of variables, an initialization predicate, an invariant, and a set of methods (Abrial 1996; Morgan 1998). Variables and methods are either private or public; if a variable or method is not marked as private, it is understood to be public:

```
module A
  var v : V = v0
  invariant I
  method m (u: U) w: W
    require b then S
  ...
```

We refer to the invariant I as A_{inv} and to the initialization $v = v0$ as A_{init}. Each method is a pair, with the precondition b of method m referred to as m_{pre} and the body S as m_{body}; if the precondition is left out, it is understood to be **true**.

2.5.3 Translation of Classes and Modules

Classes are declared within modules but the notion of independent modules are removed during translation. Classes themselves are either *free* or *extend* another class, in which case the methods of the extended class are *overridden*. The declaration of any class, say C, leads to the declaration of a number of variables, constants, and procedures: an *extent variable* $C.ref$ for all objects (or mixins) of that class, for each field f, a *field variable* $C.f$ mapping C references to field values, and for each method m, a *method variable* $C.m$ mapping C references to procedures. A *method* is a function that takes an object reference, by convention called *self*, and returns a procedure. Free class C also declares constants $C{::}init$ for the initialization ("constructor"), which is taken to be **skip** by default, and $C{::}m$ for each method m. The procedure $C.init$ assigns the methods $C{::}m$ to the method variables of the created object and calls the initialization. The procedure $C.new$ creates an object before calling $C.init$. The procedure $C.extend(r)$ checks if r is an object before calling $C.init$. Procedures $C.has$ and $C.as$ perform class ("type") checks and class ("type") casts. With $ref : Ref$ being the set all created objects, we define:

```
    class C
      var f : F
      method init(val u : U) P
      method m(val v : V res w : W) M
  ≙
```

```
var C.ref : set(Ref) := {}
var C.f : Ref → F
var C.m : Ref → (V ↦ W)
const C::init(self : Ref) = val u : U • P
const C::m(self : Ref) = val v : V res w : W • M
procedure C.init(val r: Ref, u : U)
  C.ref := C.ref ∪ {r} ; C.m(r) := C::m(r) ; C::init(r)(u)
procedure C.new(val u : U res r : Ref)
  r :∉ ref ; ref := ref ∪ {r} ; C.init(r, u)
procedure C.extend(val r : Ref, u : U)
  require r ∈ ref then C.init(r, u)
procedure C.has(val r : Ref res b : boolean)
  b := r ∈ C.ref
procedure C.as(val r : Ref res s : ref)
  require r ∈ C.ref then s := r
```

This definition generalized to classes with more (or less) than one field or method. We allow fields to be initialized at declaration, writing **var** $f : F := e$, which is equivalent to adding $self.f := e$ to the $C::init$ method.

If class D extends C, then overridden methods of C are stored in the method variables of C. For the fields g of D and the new methods $D::n$ of D, new field and method variables are introduced. Method $D.extend(r)$ checks if r is a C object and if so, mixes D into the object:

```
    class D extends C
      var g : G
      method init(z : Z) Q
      method m(val v : V res w : W) M
      method n(val x : X res y : Y) N
≙
    var D.ref : set(Ref) := {}
    var D.g : Ref → G
    var D.n : Ref → (X ↦ Y)
    const D::init(C.m)(self : Ref) = val z : Z • Q
    const D::m(C.m)(self : Ref) = val v : V res w : W • M
    const D::n(self : Ref) = val x : X res y : Y • N
    procedure D.init (val r : Ref, z : Z)
      D.ref := D.ref ∪ {r} ;
      C.m(r), D.n(r) := D::m(C.m(r))(r), D::n(r) ;
      D::init(r, z)
    procedure D.extend(val r : Ref, z : Z)
      require r ∈ C.ref then D.init(r, z)
    procedure D.has(val r : Ref res b: boolean)
```

$b := r \in D.ref$
procedure $D.as$(**val** $r : Ref$ **res** $s : Ref$)
require $r \in D.ref$ **then** $s := r$

Here, $D{::}init$, $D{::}m$ are functions that take $C.m$, a procedure that is distinct from the method variable $C.m$, as a parameter. There are two kinds of method calls. The *down-call* $c.m(x, e)$, is resolved to method m of object c at the time of the call; as a special case, c can be *self*, the receiver of a call. If c is of class C, a call to a method m of c refers to $C.m(c)$:

$$c.m(e, x) \mathrel{\hat{=}} C.m(c)(e, x)$$

The other method call is the *up-call* (*super-call*) $C.m$, which when occurring in mixin D, is resolved to the value of *self.m* at the time when the mixin D is applied to *self*. The assignment to $C.m(r)$ in $D.init$ reflects this.

A number of conventions are used to make programs look familiar. Access to field f of object r is written as $r.f$. Furthermore, if r is declared as being of class C and D is another class (which may or may not need C):

$$\begin{aligned}
&r.f \mathrel{\hat{=}} C.f(r)\\
&r := \textbf{new}\, C \mathrel{\hat{=}} C.new(r)\\
&\textbf{extend}\, r\, \textbf{with}\, C \mathrel{\hat{=}} C.extend(r)\\
&b := r\, \textbf{has}\, D \mathrel{\hat{=}} D.has(r, b)\\
&s := r\, \textbf{as}\, D \mathrel{\hat{=}} D.as(r, s)
\end{aligned}$$

Within methods, field access *self.f* is abbreviated as f, as in Listing 1.4.

Finally, a mix *program* is a named statement:

$$program ::= \textbf{program}\, id\, \langle statement \rangle \qquad \text{(main program)}$$

A program declares the extent variable *ref* and initializes it to the empty set:

$$\textbf{program}\, P\ S \mathrel{\hat{=}} \textbf{var}\, ref : Ref := \{\}\, \textbf{in}\, S$$

A program is syntactically *well-defined* (1) if all classes that it uses are declared on top level within the program, (2) all extent variables and method variables are "hidden", i.e. only modified through **new** and **extend**, (3) all field variables are "private", i.e. each field f is modified only through assigning *self.f* in methods, (4) object initializations do not modify any variables except *self.f*. Well-defined programs may access and modify global variables. These are used for observing the program's behaviour.

var $Lock.ref : set(Ref) := \{\}$
var $Lock.l : Ref \rightarrow$ **boolean**
var $Lock.lock : Ref \rightarrow (() \mapsto ())$

```
var Lock.unlock : Ref → (() ↦ ())
const Lock::init(self : Ref) = self.l := true
const Lock::lock(self : Ref) = self.l := true
const Lock::unlock(self : Ref) = self.l := false
const Lock::push(Stack.push)(self : Ref) =
  val e : integer res d : boolean •
    if ¬self.l then Stack.push(e, d) else d := false
const Lock::pop(Stack.pop)(self : Ref) =
  res e : integer, d : boolean •
    if ¬self.l then Stack.pop(e, d) else d := false
procedure Lock.init(val self : Ref)
  Lock.ref := Lock.ref ∪ {self} ;
  Stack.push(self), Stack.pop(self) :=
    Lock::push(Stack.push(self))(self), Lock::pop(Stack.pop(self))(self) ;
  Lock.lock(self), Lock.unlock(self) := Lock::lock(self), Lock::unlock(self) ;
  Lock::init(self)
procedure Lock.new(res r : Ref)
  r :∉ ref ; ref := ref ∪ {r} ; Lock.init(r)
procedure Lock.extend(val r : Ref)
  require r ∉ Stack.ref then Stack.init(r)
procedure Lock.has(val r : Ref res b : boolean)
  b := r ∈ Lock.ref
procedure Lock.as(var r : Ref res s : Ref)
  require r ∈ Lock.ref then s := r
```

Chapter 3

mix Implementation

Dynamic mixins complicate the compiler's object model. The order in which the mixins are added is not known until runtime, so this information must be encoded in the model. This linearization order is required to determine where to look up method definitions when a message is received by the base object. As with the Decorator pattern, dynamic mixins can be used for method combination. This linearization order defines how methods are combined. Finally, casting an object to a type involves finding the location of a particular mixin within an object. Since a mixin location is not fixed, an efficient means of traversing the object to find this information is required.

The implementation of the dynamic mixin language mix is discussed in this chapter. In particular, a memory model to support the language implementation is described. Static, object-oriented languages that do not support dynamic mixins can have a static object memory layout. Once an object is created, its fields and their types are known. Fixed field offsets provide constant time access to an object's fields at runtime. Languages that provide object extension must have some lookup code added to each field access request since field locations cannot be calculated at compile-time. The proposed memory layout allows constant time access to fields and methods defined in the static type of the object, while supporting functionality found in the Decorator pattern. Pointer management is left to the language implementer as opposed to the application developer.

3.1 Memory Layout

Despite allowing an object to be extended with an arbitrary number of mixins, methods are always accessed using a statically known offset; no searching for methods by name takes place. This is realized by two kinds of structures, *type descriptors* and *object segments*.

For every class D, including *Object*, a type descriptor *D_type* is created. This is a record that contains a pointer to a method implementation for every method

defined in that class. If class D extends C, then D_type contains all methods defined in D, whether new or overridden. The layout of type descriptors is such that if class C' implements C, then the type descriptor of C' first has pointers to its own implementation of methods of C and then possibly pointers to new methods. Type descriptors are used to identify types and for method call resolution. We use the type *TypePtr* for pointers to type descriptors.

Each object consists of a number of object segments, of which one is the *head segment* and the remaining are *mixin segments*. The head segment determines the object's identity. Mixin segments can be added (and potentially removed) as needed. Each object segment is a record with following fields:

type The field *type* points to the type descriptor of the mixin's type. The head segment points to *Object_type*, the type descriptor of *Object*.

cycle The field *cycle* points to the next segment in order of extension, regardless of its type. The head segment points to the first mixin segment and the last mixin segment points to the head segment.

bottom For every class C, except *Object*, all subclasses of C, including C, have a field C_bottom that points to the last added mixin that extends C.

up For every class C, except *Object*, all proper subclasses of C, i.e. excluding C, have a field C_up to the last previously added mixin that extends C.

fields This field is itself a record, containing all the fields declared in the mixin's class

The order of fields is such that *type* and *cycle* are first, then the *bottom* and *up* fields for each superclass, in order of the subclass relation, starting with the one that only needs *Object*, and finally concludes with the *fields* record.

Figure 3.1 illustrates this.

This memory layout also applies in presence of subtyping: if some class C' implements C and has extra methods $m'()$, then C'_type has fields for $m()$ and $m'()$. If D extends C', and has extra methods $n()$, then D_type will only have fields for $m()$ and $n()$ because class D is a subtype of C but not C'. The type *ObjPtr* is the type of pointers to object segments.

The auxiliary function *implements*(t, C) tests if the type pointed to by t implements C, which means that it is of type C or any of its subtypes:

$$\mathit{implements}(t, C) \mathrel{\hat{=}} \bigvee B \in \mathit{subtypes}(C) \bullet t = B_type$$

For example, if only class C' implements C, we have that *implements*(t, C) is $t = C_type \vee t = C'_type$. Calls to *implements* in the code found in the upcoming sections are *inlined* as at run-time. Class names, the *implements* relation, and the *extends* relation are not explicitly represented.

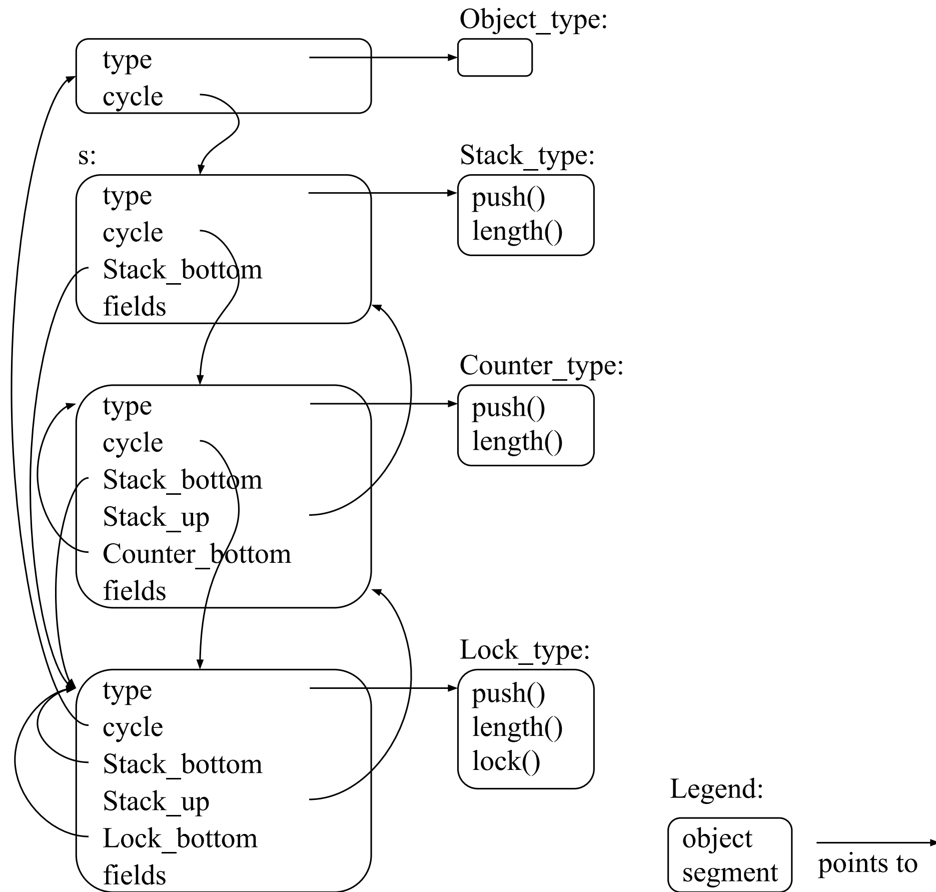

Figure 3.1: Memory Layout of the Example in Section 1.4.2

Figure 3.2 gives an example with multiple extensions of a class and "interleaved" addition of mixins from different branches.

3.2 Program Initialization

When a program in mix begins, a type descriptor *C_type* for each class, which includes declared classes and the predefined class *Object*, is allocated:

```
for C ∈ all classes do
    // allocate type descriptor C_type for C
    for m ∈ methods defined in C do
        C_type.m := implementation of m
```

3.3 Object Creation

Creating an object of class C first creates the head segment and then extends it with C's segment. The statement $x :=$ **new** C translates to the call $x := C_new()$, assuming that C does not extend any class other than *Object*:

```
procedure C_new(): ObjPtr
    // allocate header segment h
    h.type := Object_type
    h.cycle := h
    return C_extend(h)
```

3.4 Object Extension

When extending an object by class D that extends C, first the absence of a D mixin and the presence of a C mixin are checked. In existing segments, the *bottom* pointers need to be updated: each *bottom* pointer in a segment that goes to a segment that implements a superclass of D is set to point to the extension. As updating *bottom* pointers when extending by D is also required when extending by a class that extends D, the code for that is factored out in the procedure *D_updateBottoms*. The new segment has *up* pointers for all proper superclasses of D. If A is a proper superclass, *A_up* is set to the last mixin that extends A. The last mixin is found by iterating through all mixins from the first segment downward and recording *A_up* each time a segment is found that implements a subclass of A. In order to start at the first segment, the header is located using the *cycle* pointers; the first segment is the one after the header. The updates of the *bottom* pointers of the segment and the *up*

pointers of the extension can be merged into one loop. Finally, the *cycle* pointers are updated to make e the new bottom and the *bottom* pointers of e are set to itself. The statement $y :=$ **extend** x **with** D translates to the call $y := D_extend(x)$:

```
procedure D_extend(p: ObjPtr): ObjPtr
    if D_has(p) ∨ ¬ C_has(p) then raise
    b := p
    while b.cycle.type ≠ Object_type do // find bottom b
        b := b.cycle
    h := b.cycle // header
    // allocate segment e for extension D
    q := h.cycle // first segment
    repeat
        D_updateBottoms(q, e)
        for A in superclasses(D) − {D} do
            if ⋁ B ∈subclasses(A) • implements (q.type, B) then
                e.C_up := q
        q := q.cycle
    until q = b
    b.cycle := e // new bottom is e
    e.cycle := h
    e.type := D_type
    for C in superclasses(D) do
        e.C_bottom := e
    return q
```

If D extends only *Object*, the body of $D_updateBottoms(q,\ e)$ is empty. Otherwise, if D extends C we have:

```
procedure D_updateBottoms(q, e: ObjPtr)
    for B ∈ subclasses(C) − subclasses(D) do
        if implements(q.type, B) then q.B_bottom := e
    C_updateBottoms(q, e)
```

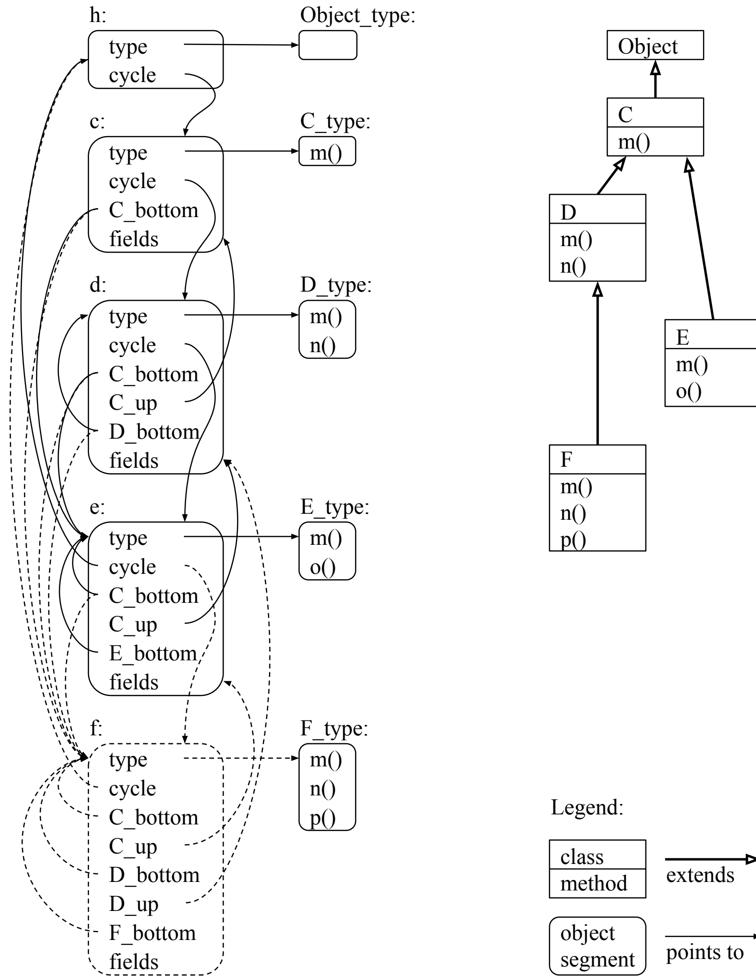

Figure 3.2: Top right: a class hierarchy; class C only extends *Object*, thus C mixins have only a *C_bottom* field; class D extends C, thus D mixins have *C_bottom*,*C_up*, *D_bottom* fields and similarly for class E; class F extends D, which itself extends C, thus F mixins have *C_bottom*, *C_up*, *D_bottom*, *D_up*, *F_bottom* fields.

Left: in solid lines, a C object to which D and E mixins were added, in that order; in dashed lines, changes after extending with F. The corresponding mixins are labelled c, d, e, f: fields *C_bottom* in c, d, e point to E, the last mixin extending C, after extension with F, point to F; field *D_bottom* in d points to d, after insertion of F, the last mixin extending D, points to F; in e, field *E_bottom* is unaffected by the extension with F.

3.5 Type Test and Type Cast

The expression x **has** D translates to the call $D_has(x)$; for each type, one such procedure is generated. The *cycle* fields connect all segments of an object and the parameter x points to one of those. The segments are traversed and if a segment is found that implements D, the type test returns **true**, otherwise **false**:

```
procedure D_has(p: ObjPtr): boolean
    q := p
    repeat
        if implements(q.type, D) then return true
        q := q.cycle
    until p = q
    return false
```

The expression x **as** D translates to the call $D_as(x)$. The implementation is similar to that of type tests, except that a pointer to a segment that implements D is returned if one is found, otherwise an exception is raised:

```
procedure D_as(p: ObjPtr): ObjPtr
    q := p
    repeat
        if implements(q.type, D) then return q
        q := q.cycle
    until p = q
    raise
```

3.6 Method Calls

Suppose x is declared to be of class D and D has method m. The call $x.m(args)$ goes to the last mixin of x that extends D, which is determined by following the D_bottom pointer of x. The segment $x.D_bottom$ is used for both selecting the method to be called and as the first parameter, *self*, in the method call:

$$x.m(args) \;\hat{=}\; x.D_bottom.type.m(x.D_bottom,\ args)$$

Recall that the field D_bottom is present in every mixin segment that extends D and is allocated at the same offset in every such segment; the segment D_bottom may be of class D or a class that implements D. The "self-call" $m(args)$ within a class with method m is a shorthand for $self.m(args)$:

$$m(args) \;\hat{=}\; self.D_bottom.type.m(self.D_bottom,\ args)$$

In class D, a "super-call" $C.m(args)$ to any of its proper superclasses where

$C \in superclasses(D) - \{D\}$ translates to an up-call following the C_up field. Variable *self* is the pointer to the segment in which the call occurs; $self.C_up$ is used for both selecting the method to be called and as the *self*-parameter in the method call:

$$C.m(args) \ \hat{=} \ self.C_up.type.m(self.C_up,\ args)$$

Recall that the field C_up is present in every mixin segment that extends C and is allocated at the same offset in every such segment.

3.7 Analysis

A compiler emitting C code has been built to evaluate this model. The run-time complexity of the generated code is as follows:

Program initialization is linear in the number of classes and linear in the total number of methods (as in other compiled languages)

Object creation needs constant time, plus the time for memory allocation (as in other compiled languages)

Object extension is linear in the number of mixins (assuming that the presence of the extending class, absence of the extended class is checked while searching for header), plus quadratic in the depth of the extension hierarchy (for updating the *up* and *bottom* pointers for existing mixins, assuming that *implements* requires constant time), plus linear in the number of mixins of the object, plus linear in the depth of the extension hierarchy (for updating the bottom pointers of the extension); plus the time for allocation (in Python object extension is linear in the number of methods, accessed by hashing)

Type test, type cast needs time linear in the number of mixins (assuming that the *implements* function needs constant time) (in Python, test of set membership, in case mixins are recorded as a set in a field; in compiled languages, if the class hierarchy is fixed, can be optimized to constant time)

Down-call needs constant time (one indirection to the *bottom* mixin and one to the method table, accessing the method at a fixed offset); (in compiled languages, one indirection to the method table, accessing the method at a fixed offset)

Super-call needs constant time (one indirection to the *up* mixin and one to the method table, accessing the method at a fixed offset); (in compiled languages,a direct call)

The assumption that *implements* requires constant time can be guaranteed if the class hierarchy is fixed. The dependencies can be summarized as follows:

- The number of free mixins does influence type casts, type test linearly and object extension double linearly, but neither influences method lookup nor object creation time.
- The number of method and fields per mixin does not influence method lookup and field access time (like C++, Java, but unlike languages that require hashing)
- The depth of the extension hierarchy influences object extension quadratically; (in C++, Java, Python there is no influence)

The memory overhead is as follows:

- For each free mixin: three pointers (*cycle*, *type*, *bottom*)
- For every object: one header segment with two pointers (*type*, *cycle*)
- For every directly or indirectly extended class: two pointers (*up*, *bottom*)

(In C++, Java, Python, one type pointer for each object)

3.8 Translation to Executable Code

The compiler that has been built to implement mix emits C code as its target. A target of C code allows for fair comparison with C program implementations as the same compiler will be used to optimize and generate executable code. The source code is available in the following git repository - http://www.github.com/bmellow/mx.

Listing 3.1 contains a program which defines a *Point* class. It is used to explain key parts of the translation as described earlier in the chapter. For brevity, the complete translation is found in Appendix B.

Along with the core *Point* class, the following are included.

- *ArrayPoint*, an alternative data representation of a point. This provides a different implementation of all methods that *Point* exposes. Since it implements *Point*, this mixin can be used wherever a *Point* mixin is. In Listing 3.1, the statement **implement** *p* **with** *ArrayPoint* replaces the object segment in *p* of type *Point* with a new one of type *ArrayPoint*. The new segment takes the place of the discarded segment in the object *p*'s linearization order. In order to accomplish this, the new segment copies the *cycle*, *Point_up* and *Point_bottom* from the discarded segment.
- *MultiPoint*, a bound mixin that adds a feature to *Point*. It adjusts any move of a point by some defined linear factor. Unlike *ArrayPoint*, this mixin requires an object segment of type *Point* to exist in the object before it can be added to an object. Augmenting an object with this segment requires that it is inserted into

the *cycle* and *Point* linked lists. Thus, *Point_up* field must be updated to store the pointer of the previously added *Point* extending or implementing segment. The *Point_bottom* fields of all *Point* typed segments must be updated to store a pointer to this newly added segment.

- *Member*, a free mixin that records the fact an object is a member of some arbitrary set. Free mixins only require that the *cycle* field of the previously added object segment (of any type) stores the pointer to this newly added *Member* segment. The newly added segment should initialize its *cycle* field to store the pointer to the head segment of the object.

Listing 3.1: A *Point* class, with related mixins

```
program P
  class Point
    var x : integer
    var y : integer
    initialization(x0, y0 : integer)
      x := x0; y := y0
    method move(x0, y0 : integer)
      x := x + x0; y := y + y0

  class MultiPoint extends Point
    var m : integer
    initialization (sz2 : integer)
      m := sz2
    method setMultiplier(sm : integer)
      m := sm
    method move(x0, y0 : integer)
      var x1 : integer
      var y1 : integer
      x1 := x0 * m; y1 := y0 * m
      Point.move(x1, y1)

  class ArrayPoint implements Point
    var ar : array of integer
    initialization(x0, y0 : integer)
      ar := new integer[2]
      ar[0] := x0; ar[1] := y0
    method move(x0, y0 : integer)
      ar[0] := ar[0] + x0; ar[1] := ar[1] + y0

  class Member
```

```
    var g : boolean
    method setg(x : boolean)
      g := x

  begin
    var p : Point
    var m : Member
    p := new Point;
    extend p with Member; extend p with MultiPoint;
    implement p with ArrayPoint;
    p.move(5,7); m := p as Member; m.setg(true)
  end
```

Object Structure

Each class is translated into a C structure containing its attributes and pointers to other segments as described in Section 3.1. Another structure is created for pointers to the class's methods. By convention, for class C, these two structures are named *C_Impl* and *C_Methods* respectively. Since a unique *C_Method* type is created for each class declared in the program, the *methods* pointer type in the *C_Impl* class can explicitly be declared.

In addition to function pointers, the *C_Impl* structure also contains an *implements* attribute. Since the C emitted code only instantiates one instance of each *C_Method* structure per class, they can be used as type descriptors for run-time type checks and casts. The implements relation forms a tree among different classes so the *implements* attribute used to store it is left untyped (i.e. void pointer). This allows storage of different pointer types (representing different classes) for simple tree traversal.

Listing 3.2: Memory Representation of *Point* class, with related mixins

```
        Object_Interface* Object_cycle;
        struct Point_Impl* Point_bottom;
        int x;
        int y;
} Point_Impl;

void Point_construct ( Point_Impl* self, int y0 , int x0){
        self->x = x0;
        self->y = y0;
}
```

```
void Point_move ( Point_Impl* self, int y0 , int x0){
        self->x = self->x+x0;
        self->y = self->y+y0;
}

void Point_init() {
```

Initialization

The set of mix statements at the end of Listing 3.1 enclosed with the **begin** and **end** keywords are translated and placed inside the *main*() function.

An initialization procedure is generated for each class. It allocates memory for the appropriate *C_Method* and assigns its reference to a global variable. Function pointers are then initialized to point to class method implementations.

The generated class initialization procedure invocation statements are placed at the beginning of the *main*() function.

Listing 3.3: Initializing the program from Listing 3.1

```
void* point_MethodTable;

void Point_init() {
  point_MethodTable = malloc(sizeof(Point_Methods));
  ((Point_Methods*)point_MethodTable)->construct =
      (void(*)(void *, int y0 , int x0)) &Point_construct;
  ((Point_Methods*)point_MethodTable)->move =
      (void(*)(void *, int y0 , int x0)) &Point_move;
}

int main() {
  Object_init(); Point_init(); MultiPoint_init();Member_init();
  Point_Impl* p; Member_Impl* m;
  p = Point_new();
  Member_extend((Object_Interface *)p);
  MultiPoint_extend((Object_Interface *)p);
  (((Point_Methods*)(p->Point_bottom)->methods)->move(p->Point_bottom,5,7));
  m = (Member_Impl *) castObject( (void *) p,member_MethodTable);
  (((Member_Methods*)(m->Member_bottom)->methods)->setg(m->Member_bottom,1));
}
```

Method Calls

Method resolution is done by finding the last mixin added which implements or extends the object's static type and invoking that mixin's implementation of the named method.

To invoke a method of an object, a pointer to the receiving object is obtained and a pointer to the last extending mixin segment is obtained using the static type's "down" pointer. A pointer to the method implementation is accessed using the *methods* attribute. This pointer is then casted to the static type's method structure. Finally, the function is invoked in the usual way for object-oriented implementations where the first parameter is a "self" reference. The mix statement *m.setg*(**true**) is translated into the last statement of Listing 3.3. Up-calls within bound mixin methods are handled similarly with the "up" pointer of the "self" reference being used as the object pointer argument as shown in Listing 3.4.

Listing 3.4: Examples of "up" method calls

```
void MultiPoint_move ( MultiPoint_Impl* self, int y0 , int x0){
  int x1; int y1;
  x1 = x0*self->m; y1 = y0*self->m;
  (((Point_Methods*)(self->Point_up)->methods)->move(self->Point_up,x1 , y1));
}
```

Type Casting

Casting an object to another type is done with a type-agonstic library function. The mix statement *m* := *p* **as** *Member* is translated into the following C statement.

```
m = (Member_Impl *) castObject( (void *) p,member_MethodTable);
```

The definition of *castObject*() is found in Listing 3.5.

Listing 3.5: Library function for type casting

```
void* castObject(void* m, void* t) {
  Object_Interface* temp = ((Object_Interface*) m)->Object_cycle;
  while ((temp->methods != object_MethodTable) && (temp->methods != t))
    temp = temp->Object_cycle;

  if ((temp->methods == object_MethodTable) && (t != object_MethodTable)) {
    printf("exception"); return NULL;
  }
  return temp;
}
```

3.9 Related Work

Object field or method access in Smalltalk requires at least one level of indirection as access to the object's metaclass object is required. Metaclass objects store pointers to a class's method implementations so method resolution for objects done by searching a list of class objects in the current class hierarchy until one is found (Goldberg and Robson 1983). In mix, type descriptors serve the purpose of specifying class types. Similar to Smalltalk, object class types are encapsulated with class method definitions. A type hierarchy, however, is not formed with metaclasses but is encoded directly into objects.

Self is another dynamically typed language which has a prototype-based object model (Ungar and Smith 1991). At the implementation level, objects are represented as assignable slots and a pointer to an appropriate map (Chambers, Ungar, and Lee 1989). When an object is extended, a new map to reflect its new structure must be generated. The mix language does not support extension by individual fields but only by complete class. Since the object layout for all classes is known at compile time, no map structures are generated at runtime.

Python is a dynamically typed language, which unlike other class based languages, allows one to add and remove fields at runtime. The CPython implementation stores fields in a *dictionary*. This has the disadvantage of requiring that the field identifiers are stored in every object and field access requires some type of search. Hash tables are typical implementations, which must be sparsely populated to work efficiently. In (Ishizaki, Ogasawara, Castanos, Nagpurkar, Edelsohn, and Nakatani 2012), caching field accesses is proposed as a means of reducing access time. The PyPy implementation uses maps like in Self to optimize field access (Bolz 2011). When an object is created, it is assigned a map that stores the instance offsets in a linked list. Common objects share the same map, eliminating the memory wasted by storing this information on a per-object basis. Adding fields to or removing fields from an object requires that object to be assigned to a new map that matches its structure. A similar concept, called hidden classes, is used to optimize field access in Google's V8 implementation of its JavaScript engine (Google 2015). As in Python methods are fields, dynamic mixins can be expressed by adding all fields and methods for a mixin as new fields, with appropriate names to avoid name clashes. In this case, the depth of the extension hierarchy is not influenced, but the total number of fields and methods influences the efficiency of the hash table used for field and method lookups.

In strongly, statically typed languages such as C++, field access is accomplished by adding an offset to an object pointer. Since this offset is known at compile time using the object's type information, and objects are stored in contiguous memory, field access is significantly faster than any technique involving lookup tables (Pugh and Weddell 1990). This constant time access is preserved even in the context of objects with non-virtual base classes since the memory required to store such objects and a deterministic ordering of their fields can be established at compile time (Stroustrup

Table 3.1: Timing Results. This table shows the number of seconds (in CPU time) required to execute the program specified in Listing 3.6.

Call Depth	Inheritance-Based				Mixin-Based		
	PyPy	CPython	Java	C++	PyPy	CPython	mix
2	0.398	13.1	0.099	0.169	0.514	12.5	0.182
4	0.635	20.6	0.117	0.287	0.920	21.1	0.320
8	1.26	37.8	0.191	0.522	1.7	36.9	0.650

1999). Polymorphic field access is done by maintaining class offset pointers for all of a class's superclasses. Polymorphic method calls are handled by dispatch tables which select method implementations based on the dynamic type of an object.

In languages which support multiple inheritance, a means of ensuring static offsets to class fields becomes challenging when a superclass occurs multiple times in a class inheritance hierarchy[1] (Zibin and Gil 2003). Techniques to address this include:

Colouring Techniques based on (Dixon, McKee, Vaughan, and Schweizer 1989), where indices into a lookup table are assigned so that no field in the same object have the same index.

Bidirectional Object Layouts where objects can have both negative and positive indices (Pugh and Weddell 1990)

Both approaches are NP-hard to solve so algorithms proposed in (Ducournau 2011; Myers 1995) seek to produce acceptable yet suboptimal results.

In (Templ 1993), multiple inheritance is shown to be implementable using single inheritance and thus removing the space and speed concessions made by using the algorithms above. The approach involves allocating a separate block of memory for each of an object's superclasses. Each block would store instance variables inherited from that superclass and pointers to the object's other memory blocks. To reduce the number of memory blocks allocated in the general case, an optimization would involve grouping objects into compound objects and including static offsets to address the embedded ones. This approach however assumes that repeated inheritance is disallowed.

Listing 3.6: Adding Features Using Mixins

```
program MixinExtensionTest
    class C0
        var c0 : integer
        initialization(t : integer)
            c0 := 1
```

[1]We assume a shared model where only one copy of the superclass's fields is included when instantiating an instance of the subclass. This occurs in C++ when the **virtual** keyword is used

```
        method m(d : integer)
            c0 := (c0 + d) mod 99

    class C1 extends C0
        var c1 : integer
        initialization(t : integer)
            c1 := 1
        method m(d : integer)
            C0.m(d)
            c1 := (c1 + d) mod 99

    class C2 extends C1
        var c2 : integer
        initialization(t : integer)
            c2 := 1
        method m(d : integer)
            C1.m(d)
            c2 := (c2 + d) mod 99

    begin
        var c : C0
        var i : integer
        i := 0
        c := new C0
        extend c with C1
        extend c with C2
        while i < 100000000 do
            c.m(i)
            i := i + 1
    end
```

3.10 Evaluation

The proposed object model is designed to support object extension without the need to look up methods or fields by name; fields are accessed with fixed offsets and method calls, both up-calls and bottom-calls, require two indirections. In order to test the efficiency of the mix object model, a dedicated benchmark is employed: the assumption is that method calls are significantly more frequent than object creations, object extensions, type tests, or type casts. Method calls require one extra indirection compared to compiled languages, but field access does not, so the benchmark focuses on method calls and involves little computation, see Listing 3.6. The benchmark simulates the Decorator pattern, which requires upcalls through all extending mixins. The length of the call chain is varied from 2, 4 and 8. The comparison is done with Java, C++,

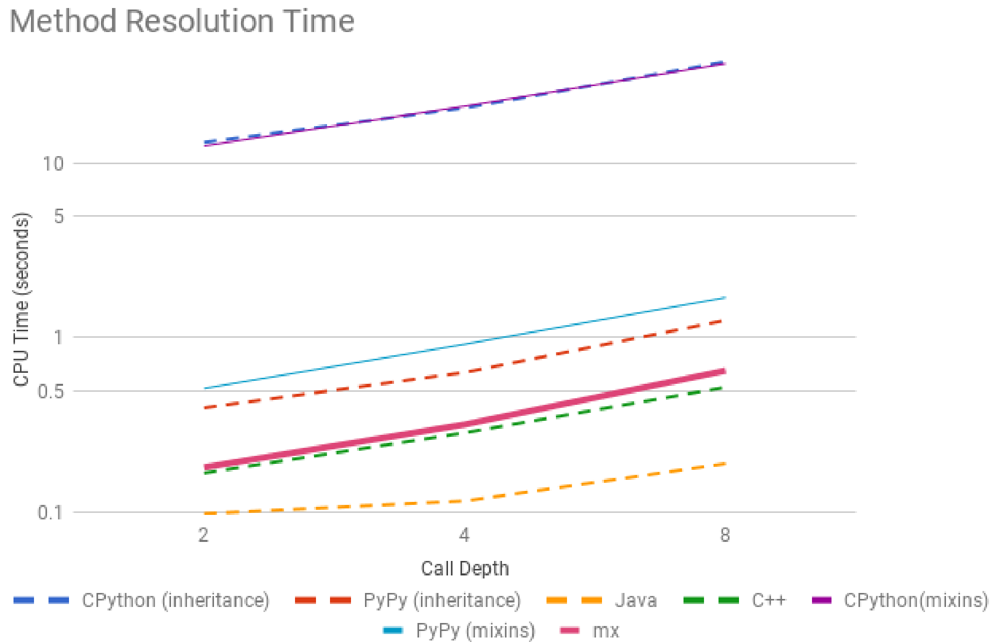

Figure 3.3: Method resolution in mx is not as fast as inheritance-based approaches but outperforms mixin-based ones.

and two Python implementations, CPython and PyPy. Code used for the tests can be found in Appendix C. The mx compiler employs the same LLVM code generator as the C++ compiler.

The C++ compiler uses fixed offsets for field access and one indirection (in the *virtual method table*) with a fixed offset to resolve method calls. The Java JIT compiler additionally optimizes method calls at run-time, replacing indirect calls with direct ones. CPython is an interpreter that looks up method names in a hash table. PyPy is a JIT compiler that improves name lookups.

The results are shown in Table 3.1 and graphically in Figure 3.3. The first four columns show timing results from inheritance-based programs. These programs do not express dynamic mixins, but are included here as a level of performance that we would like to approach. Addresses to class methods needed to access super-calls are known at compile-time. Since accessing methods can be done using offsets and table lookups in C++ and Java, the inheritance approaches in these languages provide the best performance results. Of the two Python implementations, the compiled PyPy is much faster than CPython. The next two columns show times when the program is implemented using dynamic mixins and the receiver of a super-call must be determined at runtime. In the Python-mixin based implementations, the receiver is found by using a name lookup at runtime. The difference between Python mixin vs inheritance based results can be explained by the extra time required to look up

the address of super calls. In summary, the object model used in mix incurs a penalty for method resolution over statically typed inheritance, but provides a significant improvement over mixin approaches which require name lookup.

3.11 Discussion

The fact that the set of mixins bound to an object changes at runtime makes a contiguous, fixed offset memory representation of an object impossible. Furthermore, "super-calls" need to be resolved at runtime. The memory model proposed deals with these issues by representing an object as a set of memory segments where each segment stores a mixin object. Method calls can be resolved with a single extra indirection as opposed to performing a name lookup. Our measurements on an artificial benchmark, consisting mainly of method calls with little computation, show that this is significantly more efficient than method lookup by name but slower than ahead-of-time compiled languages with single indirection (as in C++) by a factor of about 1.5. Given that typical programs would contain more computation and fewer method calls, chances are that the slowdown of dynamic mixins may not be significant, making them an attractive programming construct for the flexibility and safety gained. On the other hand, just-in-time compiled languages with single indirection (as in Java) are a factor of 2 to 3 more efficient in our artificial benchmark. This motivates us to study the run-time optimization of our required double indirection method call in the future.

The object model proposed could be implemented using a contiguous chunk of memory which varies in size as mixins are added. Implementing the model using this memory layout would mean using memory offsets instead of pointers to access the various embedded mixins. The allocated memory would have to be expanded as mixins are added. Such a layout may have potential benefits with respect to caching and memory fragmentation, but requires the cooperation of a garbage collector. An evaluation of the potential benefits of this memory layout is also left as future work.

Chapter 4

Program Correctness

Our approach is to establish the correctness of modules using mixins by the refinement of abstract modules. The theory of refinement has been studied extensively, e.g. (Back and Wright 1998; Morgan 1998), and applied to object-oriented programs, e.g. (Mikhajlov and Sekerinski 1998; Mikhajlova and Sekerinski 1997).

We use abstract classes for the specification of programs with dynamic mixins. Abstract classes are defined in terms of a programming language extended with specification constructs (abstract data types and abstract statements). The correctness of a (concrete) program with dynamic mixins is shown by refinement of an abstract class, or in general of an abstract module. Class refinement as defined here is based on data refinement (Hoare 1972).

4.1 Module Consistency

Recall the structure of a module from Section 2.5.2.

Definition 1. Module A is *consistent* if

(a) the initialization establishes the invariant,

$$A_{init} \Rightarrow A_{inv}$$

(b) each method m of A preserves the invariant under its precondition:

$$A_{inv} \wedge wp(m_{body}, true) \Rightarrow wp(m_{body}, A_{inv})$$

The definition implies that the precondition of every method must be strong enough to guarantee the termination of the body. In particular, if the body contains further method calls, then the precondition must be strong enough that the precondition of all the method calls are satisfied. The definition of consistency also

necessitates that public module variables can be inspected, but not modified outside of the module.

Statement S is refined by statement T, written $S \sqsubseteq T$ if for all postconditions c, whenever S terminates with c, so does T, formally:

$$S \sqsubseteq T \equiv \forall c \bullet wp(S, c) \Rightarrow wp(T, c) \tag{4.1}$$

Ordinary (algorithmic) refinement can be generalized to *data refinement*. Let S be a statement over "abstract" variables, say x, let T be a statement over "concrete" variables, say y, and let R be a predicate over x and y, known as the *coupling invariant*. In general, R may involve other variables that are common to S and T. Statement S is refined by T via R, written $S \sqsubseteq_R T$ if, provided that the variables of S and T are initially related by R, after their "simultaneous execution" the variables are again related by R, formally:

$$S \sqsubseteq_R T \equiv \forall c \bullet (\exists y \bullet R \wedge wp(S, c)) \Rightarrow wp(T, \exists x \bullet R \wedge c) \tag{4.2}$$

When refining modules, the invariant of the "concrete" module becomes the coupling invariant.

Definition 2. Let A and B be modules with the same public variables and public methods. Then A *is refined by* B if

(a) the joint initialization establishes the invariant of B,

$$A_{init} \wedge B_{init} \Rightarrow B_{inv}$$

(b) each method A.m is refined by B.m via B_{inv} under A_{inv}:

$$A_{inv} \Rightarrow A.m \sqsubseteq_{B_{inv}} B.m$$

Module refinement ensures that the observable behaviour through public variables and methods is preserved (Morgan 1998).

4.2 Mixin Refinement

Our proposal for safe mixin composition relies on the notion of refinement. Informally, for programs with statements that create and use objects, *class refinement* is understood as follows:

> *Suppose a program creates and uses an object of class C. Class D refines C means that creating an object of class D instead will preserve the behavior of the program.*

Thus C can be *substituted* by D for object creation. This implies that the public fields and methods of D have to be "syntactically compatible" with those of C. Classes C and D can have different private fields and methods. Their public methods can also have different bodies. Any extra public fields and methods of D that are not present in C are irrelevant as they will not be used. Class refinement is formally established by a *refinement relation*, that can be expressed as a *coupling invariant* between the fields of C and D.

Mixins can provide additional functionality through new fields and methods (as in *Lock* in Listing 1.4) or can re-implement existing functionality (as in *Counter*). Informally, *mixin refinement* is understood as follows:

> *Suppose a program creates and uses an object of class C. Assume class D, the mixin, needs C and provides extra functionality. Now consider an interleaving of statements of the original program with statements extending the object by D and using D's extra functionality. Then D refines C if the behaviour of the original program is preserved.*

Behavioral subtyping between classes also considers new methods in the subtype (Liskov and Wing 1994): new methods can only have an effect that is achievable through a combination of calls to existing methods. Mixin refinement is more permissive in that new methods can provide extra functionality, as long as the behavior of the existing methods is preserved.

4.3 Correctness of Mixin Composition

Compositionality is analysed through establishment and preservation of invariants and through augmentation and refinement of methods. Statement S *establishes* predicate p if, provided that S terminates, i.e. $wp(S, true)$ holds, at termination p holds. Statement S *preserves* p if S establishes p under p:

$$S \text{ establishes } p \mathrel{\hat{=}} wp(S, true) \Rightarrow wp(S, p)$$
$$S \text{ preserves } p \mathrel{\hat{=}} p \wedge wp(S, true) \Rightarrow wp(S, p)$$

Preservation of invariants can be shown over the structure of statements:

Lemma 1 (Preservation Over Structure). *For statements S, T and predicates b, p, q:*

$$
\begin{array}{rcll}
 & & \textbf{skip} \text{ preserves } p & \text{(a)}\\
(b \Rightarrow S \text{ preserves } p) & \equiv & \textbf{assume}\, b\, ;\, S \text{ preserves } p & \text{(b)}\\
(b \Rightarrow S \text{ preserves } p) & \equiv & \textbf{assert}\, b\, ;\, S \text{ preserves } p & \text{(c)}\\
(p \Rightarrow p[x \backslash e]) & \equiv & x := e \text{ preserves } p & \text{(d)}\\
(p \Rightarrow (\forall x \in e \bullet p)) & \equiv & x :\in e \text{ preserves } p & \text{(e)}\\
(S \text{ preserves } p) \wedge (T \text{ preserves } p) & \Rightarrow & S\, ;\, T \text{ preserves } p & \text{(f)}\\
(b \Rightarrow S \text{ preserves } p) \wedge (\neg b \Rightarrow T \text{ preserves } p) & \equiv & \textbf{if}\, b\, \textbf{then}\, S\, \textbf{else}\, T \text{ preserves } p & \text{(g)}\\
(T[x \backslash e] \text{ preserves } p) & \equiv & \textbf{const}\, x = e\, \textbf{in}\, T \text{ preserves } p & \text{(h)}\\
(\forall \overline{x} : \overline{X} \bullet b \Rightarrow T \text{ preserves } p) & \Rightarrow & \textbf{var}\, \overline{x} : \overline{X} \mid b\, \textbf{in}\, T \text{ preserves } p & \text{(i)}
\end{array}
$$

Preservation can be shown piecewise by breaking up the predicate:

Lemma 2 (Piecewise Preservation). *For statements S, T and predicates p, q:*

$$
\begin{array}{rcll}
(S \text{ preserves } p) \wedge (S \text{ preserves } q) & \Rightarrow & (S \text{ preserves } p \wedge q) & \text{(a)}\\
(q \Rightarrow S \text{ preserves } p) \wedge (p \Rightarrow S \text{ preserves } q) & \Rightarrow & (S \text{ preserves } p \wedge q) & \text{(b)}
\end{array}
$$

For example, Lemma 2 (b) allows to conclude

$$x, y := x + y, x + y \text{ preserves } x \geq 0 \wedge y \geq 0$$

from:

$$
\begin{array}{l}
y \geq 0 \Rightarrow x, y := x + y, x + y \text{ preserves } x \geq 0\\
x \geq 0 \Rightarrow x, y := x + y, x + y \text{ preserves } y \geq 0
\end{array}
$$

The syntactic notion of a statement not assigning to a variable is generalized to a statement not modifying an expression and further generalized to modifying a function only at one argument. Let e be an expression of type E:

$$
\begin{array}{l}
S \text{ does not modify } e \mathrel{\hat{=}} \forall q : E \to \mathit{bool} \bullet S \text{ preserves } q(e)\\
S \text{ modifies } f \text{ only at } r \mathrel{\hat{=}} \forall o \neq r \bullet S \text{ does not modify } f(o)
\end{array}
$$

Statement S does not modify predicate p is stronger than S preserves p, as not modifying implies that both p and $\neg p$ are preserved.

Lemma 3 (Statement Not Modifying Expression). *For statement S:*

$$S \text{ does not modify } p \;\Rightarrow\; S \text{ preserves } p$$

This follows immediately from the definition of S does not modify p by instantiating q with the identity. If the free variables of p are not assigned in S, then S obviously does not modify p. The next lemma states when a statement establishes a property for all elements of a set.

Lemma 4 (Statement Updating a Function)**.** *For statement S, variable f of function type, and boolean function p, assume that S modifies f only at d, that S does not modify s, and that S establishes $p(f(d))$:*

$$(\forall i \in s - \{d\} \bullet p(f(i))) \wedge wp(S, true) \;\Rightarrow\; wp(S, \forall i \in s \bullet p(f(i)))$$

4.3.1 Refinement and Augmentation

The following lemma includes a general rule for refinement of assignments.

Lemma 5 (Refinement Laws)**.** *Let S be a statement over variables that include x, the abstract variables, let T be a statement over variables that include y, the concrete variables, let $R(x)(y)$ relate x and y, and let z be among the global variables:*

$$\begin{array}{rcl}
(R(x)(y) \Rightarrow R(e)(f)) & \equiv & x := e \sqsubseteq_R y := f \qquad \text{(a)} \\
(R(x)(y) \Rightarrow e = f) & \equiv & z := e \sqsubseteq_R z := f \qquad \text{(b)} \\
(R(x)(y) \wedge x \in e \Rightarrow (\forall y \in f \bullet R(x)(y)) & \equiv & x :\in e \sqsubseteq_R y :\in f \qquad \text{(c)} \\
(R(x)(y) \Rightarrow e \supseteq f) & \equiv & z :\in e \sqsubseteq_R z :\in f \qquad \text{(d)} \\
(S_1 \sqsubseteq_R T_1) \wedge (S_2 \sqsubseteq_R T_2) & \Rightarrow & S_1 \, [\!] \, S_2 \sqsubseteq_R T_1 \, [\!] \, T_2 \qquad \text{(e)} \\
(S_1 \sqsubseteq_R T_1) \wedge (S_2 \sqsubseteq_R T_2) & \Rightarrow & S_1 \, ; S_2 \sqsubseteq_R T_1 \, ; T_2 \qquad \text{(f)}
\end{array}$$

The abstract and concrete variables can have the same name, but still be distinct. For example, $x := x+1 \sqsubseteq_R x := 1-x$ where $R(x)(x') \equiv x' = x \,\mathbf{mod}\, 2$. We adopt the convention of priming the concrete variables when they need to be distinguished from the abstract one. Formally, this requires substituting the variables in the concrete program with primed one before applying the definition of data refinement.

This definition of refinement does not cover the refinement of procedures, as *wp* is defined only for statements, not expressions. Refinement is extended to procedures in a natural way, with the value and result parameters becoming global variables:

$$\mathbf{val}\, v : V \;\mathbf{res}\, w : W \bullet S \sqsubseteq_R \mathbf{val}\, v : V \;\mathbf{res}\, w : W \bullet T \,\hat{=}\, S \sqsubseteq_R T$$

For example, with abstract variable x and concrete variable y, from Lemma 5 (b) it follows that $\mathbf{res}\, w \bullet w := x \,\mathbf{mod}\, 2 \sqsubseteq_R \mathbf{res}\, w \bullet w := y$ provided $R(x)(y) \equiv y = x \,\mathbf{mod}\, 2$.

Algorithmic refinement is always reflexive, $S \sqsubseteq S$ for any S, but data refinement, $S \sqsubseteq_R S$, not. For example, $x := x \cup \{3\} \sqsubseteq_R x := x \cup \{3\}$ holds for $R(x)(x') \equiv x \subseteq x'$, but does not hold for $R(x)(x') \equiv x \supseteq x'$. We say that a statement *preserves* a relation R if it refines itself under R:

$$S \text{ preserves } R \,\hat{=}\, S \sqsubseteq_R S$$

Preservation of a relation is equivalent to preservation of a predicate if the relation is a "partial identity".

Lemma 6. *For statement S and predicate p over variables x, let $R(x)(x') \equiv p \wedge x = x'$ relate abstract variables x to concrete variables x'. Then:*

$$S \text{ preserves } R \equiv S \text{ preserves } p$$

Proof.

$$\begin{array}{lll}
& S \text{ preserves } R & \\
\equiv & \forall q \bullet p \wedge x = x' \wedge wp(S, q) \Rightarrow wp(S[x \backslash x'], \exists x \bullet p \wedge x = x' \wedge q) & \text{(definitions)} \\
\equiv & \forall q \bullet p \wedge x = x' \wedge wp(S, q) \Rightarrow wp(S[x \backslash x'], p[x \backslash x'] \wedge q[x \backslash x']) & \text{(one-point rule)} \\
\equiv & \forall q \bullet p \wedge wp(S, q) \Rightarrow wp(S, p \wedge q) & \text{(renaming)} \\
\equiv & \forall q \bullet p \wedge wp(S, q) \Rightarrow wp(S, p) & \text{(conjunctivity, logic)} \\
\equiv & p \wedge wp(S, true) \Rightarrow wp(S, p) & \text{(“}\Rightarrow\text{” by taking } p \text{ for } q\text{, “}\Leftarrow\text{” by monotonicity)} \\
\equiv & S \text{ preserves } p & \text{(def. preserves)}
\end{array}$$

□

Considering a single statement of a mixin, in essence, “adds computation” to an existing computation. The notion of *augmentation* formalizes this. Let A be a function that takes a statement, say X, as a parameter and adds computation to X. Suppose that when applying A to some statement S, we want the existing computation to be preserved, so $A(S)$ to refine S. However, we allow S to be augmented already and want to preserve that behaviour as well, motivating following definition:

$$A \text{ augments } S \text{ under } R \mathrel{\hat{=}} \forall X \bullet S \sqsubseteq_R X \wedge X \text{ preserves } R \Rightarrow X \sqsubseteq_R A(X)$$

The term $X \sqsubseteq_R A(X)$ implies that the abstract variables, i.e. those modified by X, are among the concrete variables, but A may add more concrete variables. The following examples assume that S is over x, function A adds y, and $R(x)(x', y) \equiv x = x'$:

1. Assume $A(X) = X$. Then A augments $x := x + 3$ under R. Specifically, for $X = x, y := x + 3, 5$, we have that $x := x + 3 \sqsubseteq_R X$ and X preserves R (by Lemma 6 with $p \equiv true$), hence $X \sqsubseteq_R A(X) = X$.

2. Assume $A(X) = X \,;\, z := 7$. Then A augments $x := x + 3$ under R. Specifically, for $X = x, y := x + 3, 5$, we have that $x := x + 3 \sqsubseteq_R X$ and X preserves R, hence $X \sqsubseteq_R A(X) = X \,;\, z := 7$.

3. Assume $A(X) = y := 5 \,;\, X$. Then A augments $x := x + 3$ under R. Specifically, for $X = x, y := x + 3, 5$, we have that $x := x + 3 \sqsubseteq_R X$ and X preserves R, hence $X \sqsubseteq_R A(X) = y := 5 \,;\, X$.

4. Assume $A(X) = y := 5$. Then A does not augment $x := x + 3$ under R. As a counterexample, take $X = x := x + 3$. It follows $x := x + 3 \sqsubseteq_R X$, but $X \not\sqsubseteq_R A(X) = y := 5$.

5. Assume $A(X) = X\,;X$. Then A does not augment $x := x+3$ under R. As a counterexample, take $X = x := x+3$. It follows $x := x+3 \sqsubseteq_R X$, but $X \not\sqsubseteq_R A(X) = x := x+3\,;x := x+3$.

6. Assume $A(X) = X\,;x := 5$. Then A does not augment $x := x+3$ under R. As a counterexample, take $X = x := x+3$. It follows $x := x+3 \sqsubseteq_R X$, but $X \not\sqsubseteq_R x := x+3\,;x := 5$.

For the following examples, let $R(x)(x', y) \equiv x \subseteq x'$.

7. Assume $A(X) = X\,;x := x \cup \{3\}$. Then A augment $x := x \cup \{5\}$ under R. That is, if $x := x \cup \{5\} \sqsubseteq_R X$ and X preserves R, then $X \sqsubseteq_R X\,;x := x \cup \{3\}$.

8. Assume $A(X) = x := x \cup \{5\}$. Then A does not augment $x := x \cup \{5\}$ under R. As a counterexample, take $X = x := x \cup \{5, 7\}$. It follows $x := x \cup \{5\} \sqsubseteq_R X$ and X preserves R, but $X \not\sqsubseteq_R x := \{5\}$.

9. Assume $A(X) = \mathbf{skip}$. Then A does not augment $\mathbf{skip}$ under R. As a counterexample, take $X = x := x \cup \{5\}$. It follows $\mathbf{skip} \sqsubseteq_R x := x \cup \{5\}$ and X preserves R, but $X \not\sqsubseteq_R \mathbf{skip}$.

Intuitively, $A(X)$ augments S if A calls X exactly once, on all possible paths. Furthermore, the computation that $A(X)$ adds to X must not modify the outcome of S. Following theorem formalizes this.

Theorem 1 (Augmentation Laws)**.** *Let S, T_1, T_2 be statements, R a relation, b a Boolean expression, and A, B functions from statements to statements. Then:*

$$A(X) = X \;\Rightarrow\; A \text{ augments } S \text{ under } R \tag{a}$$

$$\begin{array}{l} A(X) = T_1\,;B(X)\,;T_2 \wedge \\ \mathbf{skip} \sqsubseteq_R T_1 \wedge \mathbf{skip} \sqsubseteq_R T_2 \wedge \\ B \text{ augments } S \text{ under } R \end{array} \;\Rightarrow\; A \text{ augments } S \text{ under } R \tag{b}$$

$$\begin{array}{l} A(X) = \mathbf{if}\, b\, \mathbf{then}\, B(X)\, \mathbf{else}\, C(X) \wedge \\ (b \Rightarrow B \text{ augments } S \text{ under } R) \wedge \\ (\neg b \Rightarrow C \text{ augments } S \text{ under } R) \end{array} \;\Rightarrow\; A \text{ augments } S \text{ under } R \tag{c}$$

Proof. For (a), assuming $A(X) = X$:

$$\begin{array}{lll} & A \text{ augments } S \text{ under } R & \\ \equiv & \forall X \bullet S \sqsubseteq_R X \wedge X \text{ preserves } R \Rightarrow X \sqsubseteq_R X & \text{def. of augments, assumption} \\ \equiv & \mathit{true} & \text{(def. of preserves, logic)} \end{array}$$

For (b), assuming $A(X) = T_1 \,;\, B(X) \,;\, T_2$:

$$
\begin{array}{ll}
& A \text{ augments } S \text{ under } R \\
\equiv & \forall X \bullet S \sqsubseteq_R X \wedge X \text{ preserves } R \Rightarrow X \sqsubseteq_R T_1 \,;\, B(X) \,;\, T_2 \\
& \hfill \text{def. of augments, assumption} \\
\Leftarrow & \forall X \bullet S \sqsubseteq_R X \wedge X \text{ preserves } R \Rightarrow \textbf{skip} \sqsubseteq_R T_1 \wedge X \sqsubseteq_R X \wedge \textbf{skip} \sqsubseteq_R T_2 \\
& \hfill (\text{as } X = \textbf{skip} \,; X \text{ and } X = X \,;\, \textbf{skip}, \text{ Lemma 5()}) \\
\equiv & \textbf{skip} \sqsubseteq_R T_1 \wedge \textbf{skip} \sqsubseteq_R T_2 \wedge A \text{ augments } S \text{ under } R \quad (\text{def. of augments, logic})
\end{array}
$$

For (c), assuming $A(X) = \textbf{if}\, b\, \textbf{then}\, B(X)\, \textbf{else}\, C(X)$:

$$
\begin{array}{ll}
& A \text{ augments } S \text{ under } R \\
\equiv & \forall X \bullet S \sqsubseteq_R X \wedge X \text{ preserves } R \Rightarrow \\
& \quad X \sqsubseteq_R \textbf{if}\, b\, \textbf{then}\, B(X)\, \textbf{else}\, C(X) \qquad (\text{def. of augments, assumption}) \\
\equiv & \forall X \bullet S \sqsubseteq_R X \wedge X \text{ preserves } R \Rightarrow \\
& \quad (b \Rightarrow X \sqsubseteq_R B(X)) \wedge (\neg b \Rightarrow X \sqsubseteq_R C(X)) \\
& \hfill (\text{def of if/then/else, Lemma 1(g)}) \\
\equiv & (b \Rightarrow (\forall X \bullet S \sqsubseteq_R X \wedge X \text{ preserves } R \Rightarrow X \sqsubseteq_R B(X))) \wedge \\
& (\neg b \Rightarrow (\forall X \bullet S \sqsubseteq_R X \wedge X \text{ preserves } R \Rightarrow X \sqsubseteq_R C(X))) \qquad (\text{logic}) \\
\equiv & (b \Rightarrow B \text{ augments } S \text{ under } R) \wedge (\neg b \Rightarrow B \text{ augments } S \text{ under } R) \\
& \hfill (\text{def. augments})
\end{array}
$$

□

Note that proofs of the lemmata used can be found in Appendix D.

4.3.2 Class Invariants

Listing 4.1 explicitly specifies the *object invariant* $c = len(s)$ of class *Counter*. In general, if free class C declares fields $C.f$, the object invariant is a predicate over $C.f(self)$; if class D extends C and declares fields $D.g$, the object invariant is a predicate over $D.g(self)$ and $C.f(self)$. The object invariant cannot refer to global variables, to fields of other objects of the same class, or to fields of other classes. The object invariant of *Counter* is a shorthand for $Counter.c(self) = len(Stack.s(self))$.

The *class invariant* is derived from the object invariant and involves the extent variable, the field variables, and the method variables of the class. Assume that C_1, C_2, ... with fields $C_1.f$, $C_2.f$, ... and methods $C_1.m$, $C_2.m$... are all the classes declared in the program. The refinement relation PRJ relates abstract variables ref, $C_1.ref$, $C_2.ref$, ..., $C_1.m$, $C_2.m$, ... to the same concrete variables. It is defined by predicate prj, with the concrete variables primed:

$$prj \mathrel{\widehat{=}} (\wedge i \bullet C_i.ref \subseteq C_i.ref') \wedge (\wedge i \bullet \forall o \in C_i.ref \bullet C_i.f(o) = C_i.f'(o))$$

For free class C with methods m the class invariant $C.inv$ is defined as:

$$\begin{aligned} C.inv \mathrel{\hat{=}} & C.ref \subseteq ref \wedge \\ & (\forall o \in C.ref \bullet C::inv(o)) \wedge \\ & (\forall o \in C.ref \bullet C::m(o) \sqsubseteq_{PRJ} C.m(o)) \end{aligned}$$

The inclusion $C.ref \subseteq ref$ implies *subsumption*: every C object is an object. The universal quantification $\forall o \in C.ref \bullet C::inv(o)$ lifts the object invariant to all objects of the class. The class invariant is "higher order" in the sense that the invariant has to be preserved by methods but also includes the refinement $\sqsubseteq_{PRJ}$ for methods, hence methods have to preserve method refinement. For class C, method variables $C.m(o)$ may "point" to methods $C::m$ or to methods of another class with which object o was extended. The refinement $C::m \sqsubseteq_{PRJ} C.m(o)$ allows the actual methods $C.m(o)$ to perform additional computation: the refinement relation PRJ permits the new methods to create additional objects, $C_i.ref \subseteq C_i.ref'$, but requires that the fields of "old" objects are unaffected, $C_i.f(o) = C_i.f'(o)$. The refinement relation is a projection that does not constrain global variables and "new" objects.

For class D extending class C and with new methods n, the class invariant $D.inv$ is defined as:

$$\begin{aligned} D.inv \mathrel{\hat{=}} & D.ref \subseteq C.ref \wedge \\ & (\forall o \in D.ref \bullet D :: inv(o)) \wedge \\ & (\forall o \in D.ref \bullet D::m(C::m)(o) \sqsubseteq_{PRJ} C.m(o)) \wedge \\ & (\forall o \in D.ref \bullet D::n(o) \sqsubseteq_{PRJ} D.n(o)) \wedge \\ & C.inv \end{aligned}$$

The inclusion $D.ref \subseteq C.ref$ again implies subsumption: every D object is a C object. The refinement $D::m(C::m)(o) \sqsubseteq_{PRJ} C.m(o))$ now requires that methods m to which the method variables "point" have to refine their definition in D, under relation PRJ, provided that up-calls in D go to C, which is expressed as $D::m(C::m)(o) \sqsubseteq_{PRJ} C.m(o)$. The new methods n in C have to refine their definition in D, $D::n(o) \sqsubseteq_{PRJ} D.n(o)$. Finally, the class invariant of C becomes part of the class invariant of D.

Free class C is *well-defined* if the initialization establishes the object invariant and all methods m preserve the object invariant:

$$\begin{aligned} & C \text{ is well-defined} \mathrel{\hat{=}} \\ & \quad r \in C.ref \Rightarrow C::init(r) \text{ establishes } C::inv(r) && \text{(a)} \\ & \quad r \in C.ref \Rightarrow C::m(r) \text{ preserves } C::inv(r) && \text{(b)} \end{aligned}$$

Class D needing class C is *well-defined* if the initialization establishes the object invariant, overriding methods m preserve the object invariant under the assumption that up-calls go to C, each overriding method augments the overridden one, and new

methods preserve the object invariant:

D is well-defined $\hat{=}$
I::$inv(r) \Rightarrow D$::$init(C$::$m)(r)$ establishes D::$inv(r)$ (a)
I::$inv(r) \Rightarrow D$::$m(C$::$m)(r)$ preserves D::$inv(r)$ (b)
I::$inv(r) \Rightarrow D$::$m(_)(r)$ augments C::$m(r)$ (c)
I::$inv(r) \Rightarrow D$::$n(r)$ preserves D::$inv(r)$ (b)

Listing 4.1 is an alternate design to the Stack problem illustrated in Listing 1.4. It contains well-defined classes: *Stack* and *Logging* have *true* as invariant, which is not specified explicitly as the invariant is trivially established and preserved.

Listing 4.1: Correct Design with Mixins, with class *Stack* as in Listing 1.4

```
class Stack
   var s : seq integer := ⟨⟩
   method push(val e : integer, res d : boolean)
      s, d := e → s, true [] d := false
   method pop(res e : integer, d : boolean)
      e, s, d := head(s), tail(s), true [] d := false
   method size(res n : integer)
      n := len(s)

class Lock extends Stack
   var l : boolean := false
   invariant s = s'
   method lock()
       l := false
   method unlock()
       l := true
   method push(val e : integer, res d : boolean)
      if ¬l then Stack.push(e, d) else d := false
   method pop(res e : integer, d : boolean)
      if ¬l then Stack.pop(e, d) else d := false

class Counter extends Stack
   var c : integer
   invariant c = len(s) ∧ s = s'
   method init()
      Stack.size(c)
   method push(val e : integer, res d : boolean)
      Stack.push(e, d) ; if d then c := c + 1
   method pop(res e : integer, d : boolean)
      Stack.pop(e, d) ; if d then c := c − 1
```

```
    method size(res n : integer)
        n := c

class Cache extends Stack
    var t : integer
    var v : boolean
    invariant (v ∧ s = t → s′) ∨ (¬v ∧ s = s′)
    method init()
        v := false
    method pop(val e : integer)
        if v then e, v := t, false
        else Stack.pop(e)
    method push(res e : integer)
        if ¬v then v := true
        else Stack.push(t) ;
        t := e
    method size(res n : integer)
        Stack.size(n) ;
        if v then n := n + 1

class Encrypt extends Stack
    invariant ∀ i • 0 ≤ i < len(s) ⇒ s(i) = s′(9 − i)
    push(val e : integer)
        Stack.push(9 − e)

var log : integer := 0

class Logging extends Stack
    invariant s = s′
    method push(val e : integer)
        Stack.push(e) ; log := log + 1
    method pop(res e : integer)
        Stack.pop(e) ; log := log + 1
    method size(res n : integer)
        Stack.size(n) ; log := log + 1

class Counter extends Stack
    var c : integer
    invariant c = len(s)
    method init()
        Stack.size(c)
    method push(val e : integer, res d : boolean)
```

```
        c := c + 1 ; Stack.push(e)
    method pop(res e : integer, d : boolean)
        c := c − 1 ; Stack.pop(e)
    method size(res n : integer)
        n := c

var log : integer := 0

class Logging extends Stack
    override method push(val e : integer)
        log := log + 1 ; Stack.push(e)
    override method pop(res e : integer, d : boolean)
        log := log + 1 ; Stack.pop(e, d)
```

Analyzing the example of Listing 1.4 reveals that *Lock* does not *not* refine *Stack*: the method *Stack*::*push* will always push an element on the stack, but *Lock*::*push* will not, so it does not augment *Stack*::*push*. This can be corrected by having *Stack*::*push* either add an element to the stack and set a success flag or clear the flag otherwise. The *Lock*::*push* method is modified to clear the flag if l is **true**, as in Listing 4.1. *Lock* refines *Stack* by reducing the nondeterminism, however that modification of *Stack* causes *Counter* to behave incorrectly, as *Counter*::*push* always increments c by one whether the call to *Stack.push* adds e to the stack or not. The solution is to increment c only if calls to *Stack.push* are successful. Now, if object x of class *Stack* is created, *Counter* and *Lock* can be added in any order, and the behaviour of *Stack* is preserved. As a side note, *Stack*::*push* could be equivalently expressed with pre- and postconditions,

```
    method push(val e : integer, res d : boolean)
        modifies s
        ensures (s = e → s₀ ∧ d) ∨ (s = s₀ ∧ ¬ d)
```

where s_0 refers to the "old" value of s, however, *Lock*::*push* and *Counter*::*push* cannot be specified solely with pre- and postconditions. These would have to literally include the pre- and postcondition of the called method *Stack*::*push*, leading to duplication or to the necessity of naming the postcondition of *Stack*::*push* so it can be referred to. More importantly, such a specification would allow implementations that do not call *Stack.push* if that effect can be achieved otherwise. The intention is that *Lock*::*push* calls *Stack*::*push*, as is later shown to be necessary.

4.3.3 Compositional Reasoning with Dynamic Mixins

Dynamic mixins have different compositional properties depending on if they are non-interfering, non-finalizing, or non-observable. A *non-interfering mixin* preserves invariants, a *non-finalizing mixin* augments inherited methods, and a *non-observable*

mixin refines the needed mixin. Every mixin should be non-interfering but not necessarily non-finalizing and non-observable.

Well-definedness of classes ensures *non-interference*, i.e. invariants are not invalidated. For example in Listing 4.1, class *Logging* is well-defined, but adding a *Logging* mixin is observable—which is the point of logging. Refinement of classes ensures *non-observability*, i.e. adding a refining mixin cannot be observed, under certain conditions. For example, the original design of the *Logging* class does not refine *Stack*. The following two theorems formalize this.

Theorem 2 (Non-interference). *Consider a well-formed program with well-defined classes. Assume that C, D are among those classes, C is a free class, and D extends C.*

(a) The program initialization establishes all class invariants.

(b) The object creation $r := \mathbf{new}\, C$ preserves all class invariants.

(c) The object extension $\mathbf{extend}\, r\, \mathbf{with}\, D$ preserves all class invariants.

(d) The method call $r.m(e, x)$ preserves all class invariants.

Proof. Let C_1, C_2, ... be all classes of the program. For (a), note that the program initializes ref to the empty set, that any class C_i is declared on the top level and initializes $C_i.\mathit{ref}$ to the empty set, hence all parts of $C_i.\mathit{inv}$ hold after program initialization.

For (b) to (d), let additionally C', D' be arbitrary distinct classes of the program such that C' is free and D' needs C'. We first determine $wp(C{::}init(r), C.inv)$:

$$\begin{array}{ll}
 & wp(C{::}init(r), C.inv) \\
\equiv & wp(C{::}init(r), C.ref \subseteq ref \wedge (\forall o \in C.ref \bullet C{::}m \sqsubseteq_{PRJ} C.m(o))) \wedge \\
 & wp(C{::}init(r), (\forall o \in C.ref \bullet C :: inv(o))) \qquad \text{(def. of } C.inv\text{, conjunctivity)} \\
\equiv & C.ref \subseteq ref \wedge (\forall o \in C.ref \bullet C{::}m \sqsubseteq_{PRJ} C.m(o)) \wedge wp(C{::}init(r), true) \wedge \\
 & wp(C{::}init(r), (\forall o \in C.ref \bullet C :: inv(o))) \qquad \text{(Lemma 3, (*), def. of preserves)} \\
\equiv & C.ref \subseteq ref \wedge (\forall o \in C.ref \bullet C{::}m \sqsubseteq_{PRJ} C.m(o)) \wedge wp(C{::}init(r), true) \wedge \\
 & (\forall o \in C.ref - \{r\} \bullet C :: inv(o)) \qquad \text{(Lemma 4, (**), def. of preserves)}
\end{array}$$

The step (*) relies on $C{::}init$ not modifying $C.ref, C.m$, which is given by well-formedness of the program. The step (**) relies on $C{::}init(self)$ establishing $I(self)$,

which is given by well-definedness of C. Now we determine $wp(C.init(r), C.inv)$:

$$
\begin{array}{ll}
& wp(C.init(r), C.inv) \\
\equiv & wp(C.ref := C.ref \cup \{r\} \,;\, C.m(r) := C::m \,;\, C::init(r), C.inv) \quad \text{(def. of } C.init\text{)} \\
\equiv & wp(C.ref := C.ref \cup \{r\} \,;\, C.m(r) := C::m, \\
& \quad C.ref \subseteq ref \wedge (\forall o \in C.ref \bullet C::m \sqsubseteq_{PRJ} C.m(o)) \wedge \\
& \quad wp(C::init(r), true) \wedge (\forall o - \{r\} \in C.ref \bullet C :: inv(o))) \\
& \hfill (wp \text{ of } ;, \text{ above calculation}) \\
\equiv & C.ref \cup \{r\} \subseteq ref \wedge (\forall o \in C.ref - \{r\} \bullet C::m \sqsubseteq_{PRJ} C.m(o)) \wedge \\
& \quad wp(C::init(r), true) \wedge (\forall o \in C.ref - \{r\} \bullet C :: inv(o))) \\
& \hfill (wp \text{ of } ;, \text{ Lemma 4})
\end{array}
$$

For the proof of (b), we first show that $r := \textbf{new } C$ preserves the invariant of C itself:

$$
\begin{array}{ll}
& wp(r := \textbf{new}\, C, C.inv) \\
\equiv & wp(r :\notin C.ref \,;\, ref := ref \cup \{r\}, wp(C.init(r), C.inv)) \quad \text{(def. of } \textbf{new}, ;) \\
\equiv & wp(r :\notin C.ref \,;\, ref := ref \cup \{r\}, \\
& \quad C.ref \cup \{r\} \subseteq ref \wedge (\forall o \in C.ref - \{r\} \bullet C::m \sqsubseteq_{PRJ} C.m(o)) \wedge \\
& \quad wp(C::init(r), true) \wedge (\forall o - \{r\} \in C.ref \bullet C :: inv(o))) \\
& \hfill \text{(above calculation)} \\
\equiv & C.ref \subseteq ref \wedge (\forall o \in C.ref \bullet C::m \sqsubseteq_{PRJ} C.m(o)) \wedge \\
& \quad wp(C::init(r), true) \wedge (\forall o \in C.ref \bullet C :: inv(o))) \\
& \hfill (wp \text{ of } ;, :=, :\in, \text{ simplification}) \\
\equiv & C.inv \wedge wp(r := \textbf{new}\, C, true) \quad \text{(by def. of } C.inv, \textbf{new})
\end{array}
$$

Hence $r := \textbf{new}\, C$ preserves $C.inv$. We continue with determining $wp(C::init(r), D'.inv)$:

$$
\begin{array}{ll}
& wp(C::init(r), D'.inv) \\
\equiv & wp(C::init(r), D.ref \subseteq Cref \wedge (\forall o \in D::inv(o)) \\
& \quad \wedge (\forall o \in D.ref \bullet D::m(C::m)(o) \sqsubseteq_{PRJ} C.m(o))) \\
& \quad \wedge (\forall o \in D.ref \bullet D::n(o) \sqsubseteq_{PRJ} D.n(o))) \wedge C.inv \quad \text{(def of D'.inv)} \\
\equiv & wp(C::init(r), C.inv) \wedge \\
& \quad wp(C::init(r), D.ref \subseteq C.ref \wedge (\forall o \in D::inv(o)) \\
& \quad \wedge (\forall o \in D.ref \bullet D::m(C::m)(o) \sqsubseteq_{PRJ} C.m(o))) \\
& \quad \wedge (\forall o \in D.ref \bullet D::n(o) \sqsubseteq_{PRJ} D.n(o))) \quad \text{(conjunctivity)} \\
\equiv & wp(C::init(r), C.inv) \quad \text{(r is not in D.}ref\text{, Lemma 3, def. of preserves)} \\
\equiv & C.ref \subseteq ref \wedge (\forall o \in C.ref \bullet C::m \sqsubseteq_{PRJ} C.m(o)) \wedge wp(C::init(r), true) \wedge \\
& (\forall o \in C.ref - \{r\} \bullet C::inv(o)) \quad \text{(previous calculation)}
\end{array}
$$

Now it is shown that $r := \textbf{new}\, C$ preserves $D'.inv$, assuming that the progam is

well-formed. For readability, DI is defined to be the $D'.inv$ minus the $C.inv$

$$\begin{aligned} DI \mathrel{\hat{=}} & D.ref \subseteq C.ref \wedge \\ & (\forall o \in D.ref \bullet D :: inv(o)) \wedge \\ & (\forall o \in D.ref \bullet D{::}m(C{::}m)(o) \sqsubseteq_{PRJ} C.m(o)) \wedge \\ & (\forall o \in D.ref \bullet D{::}n(o) \sqsubseteq_{PRJ} D.n(o)) \end{aligned}$$

$$\begin{aligned} & wp(r := \mathbf{new}\, C, D'.inv) \\ \equiv\ & wp(r := \mathbf{new}\, C, C.inv) \wedge wp(r := \mathbf{new}\, C, DI)) && \text{(conjunctivity)} \\ \equiv\ & wp(r := \mathbf{new}\, C, C.inv) \wedge \\ & wp(r :\notin C.ref \ ; ref := ref \cup \{r\}; \\ & \quad C.ref := C.ref \cup \{r\} \ ; C.m(r) := C{::}m, wp(C{::}init(r), DI)) \\ & && \text{(def. of C.}init\text{)} \\ \equiv\ & wp(r := \mathbf{new}\, C, C.inv) \wedge \\ & wp(r :\notin C.ref \ ; ref := ref \cup \{r\}; C.ref := C.ref \cup \{r\} \ ; C.m(r) := C{::}m, \\ & \quad C.ref \subseteq ref \wedge (\forall o \in C.ref \bullet C{::}m \sqsubseteq_{PRJ} C.m(o)) \wedge wp(C{::}init(r), true) \wedge \\ & \quad (\forall o \in C.ref - \{r\} \bullet DI) && \text{(calculation above)} \\ \equiv\ & C.inv \wedge wp(r := \mathbf{new}\, C, true) \wedge \\ & C.ref \cup \{r\} \subseteq ref \cup \{r\} \wedge (\forall o \in C.ref \cup \{r\} \bullet C{::}m \sqsubseteq_{PRJ} C.m(o)) \wedge \\ & wp(C{::}init(r), true) \wedge (\forall o \in C.ref \cup \{r\} - \{r\} \bullet DI) \\ & && \text{(}wp\text{ of ;, :=, :}\in\text{, simplification, Lemma 4, previous calculation)} \\ \equiv\ & C.inv \wedge wp(r := \mathbf{new}\, C, true) && \text{(simplification,def of C.}inv\text{)} \end{aligned}$$

Hence $r := \mathbf{new}\, C$ preserves $D'.inv$. As $D'.inv$ includes $C'.inv$, we have that $r := \mathbf{new}\, C$ preserves the invariant of all other classes as well.

(1) As the program initialization sets *Object* and *C.ref* for all classes C to the empty set, this follows vacuously.

(2) We need to consider parts (a) to (c) of the definition of class invariant for the newly created object, as all other objects are unmodified. Method $C.new(r)$ modifies *C.ref* and *ref* such that (a) is preserved and neither variable is modified elsewhere. For (b), we need to show that for the newly created object r, the object invariant holds, which holds by condition (a) of well-definedness. For (c), we observe that $C.init(r)$ and therefore $C.new(r)$ assign $C{::}\overline{m}$ to $C.m(r)$. Thus we need to show $C{::}\overline{m} \sqsubseteq_{I(C.\bar{f}(r))} C{::}\overline{m}$, which follows from $C{::}\overline{m}$ preserving $I(C.\bar{f}(r))$, which in turn is given as C is well-defined.

(3) We need to consider parts (a') to (e') of the definition of class invariant. Parts (a') and (b') hold for similar reasons as in (2). The key is to see why parts (c') and (d') are preserved for all classes and objects: suppose r, created as a C object, was extended with E, hence $C.\overline{m}$ refines $D{::}\overline{m}$. Now, **extend** r **with** D

replaces $C.\overline{m}$ with $D::\overline{m}(C.\overline{m}(r))$. However, by condition (d') of well-definedness, $D::\overline{m}$ always call $C.\overline{m}$, hence refine $C.\overline{m}$, and by transitivity part (c') and (d') are preserved. Part (e') is preserved by the same argument.

(4) As all modifications of object variables, method variables, and field variables are done within methods that preserve the object invariant, it follows that a method call preserves all class invariants.

□

The restriction that field variables $C.\bar{f}$ are modified only though methods $C::m$ is stronger than what is required for new methods in class refinement. For example, if D inherits C, the only requirement for new methods of D is that they preserve the invariant of C, but they may access the fields directly. This is not allowed when D needs C, as illustrated in Listing 4.2. The method *DirectStack::push2* updates field *Stack.s* directly, but *Counter* only overrides *push*, so combining these two mixins will break the invariant of *Counter*. This would not happen if *push2* would call *Stack.push* twice instead, as in Listing 4.3 (although with a different meaning here, if d is false, e could have been pushed once on the stack).

From the non-interference theorem follows *non-observability* in certain cases: Suppose a program operating on object r of class C is modified at some arbitrary point to include **extend** r **with** D. That object extension modifies $D.ref$, $C.\overline{m}$, $D.\overline{n}$ but is otherwise abstractly **skip**. Provided that the program does not access these variables directly, it follows from the class invariant that all modified methods $C.\overline{m}$ refine the declared ones. Furthermore, since all modified methods have to up-call the original ones, it follows by induction over the structure of the program that the program with the object extension is a refinement of the original program. Hence, as in the example of adding *Counter* to *Stack*, the object extension is not observable.

Definition 3 (Class Well-Definedness)**.** Free class C with class invariant P and methods $\overline{m}$ is well-defined if

(a) Method *init* of C abstractly has no effect,

$$\textbf{skip} \sqsubseteq_P \textbf{assume}\, r \in C.ref\ ; C.init(r)$$

(b) all methods of C preserve P,

$$C::\overline{m} \text{ preserves } P$$

If D needs class C, has class invariant Q, overrides methods m, and defines new methods n, then D is well-defined if

(a') Method *init* of C abstractly has no effect,

$$\textbf{skip} \sqsubseteq_P \textbf{assume}\, r \in C.ref\ ; C.init(r)$$

(b') all methods m of D must refine m of C under the assumption that up-calls in $D::m$ go to $C::m$:

$$P \Rightarrow C::\overline{m} \sqsubseteq_Q D::\overline{m}(C::\overline{m})$$

(c') all new methods n of D must preserve the class invariant under the assumption that up-calls in $D::n$ go to $C :: m$:

$$P \Rightarrow D::\overline{n}(C::\overline{m}) \text{ preserves } Q$$

(d') calls to $C.m$ must occur only in $D::m$ and $D::m$ always calls $C.m$.

Conditions (a) and (a') differ from the standard condition for initialization in class refinement, which would require that the initialization of D refines the initialization of C, by requiring that the initialization of D refines **skip**. This ensures objects can be extended at any time without the extension being observable. Conditions (b) and (c') are the standard condition for classes. Condition (b') is analogous to the case when D inherits C and calls $C.m$ in D are super-calls, except that here they may be bound to a mixin other than C.

Condition (d') is more subtle. In Listing 4.2, *OddStack* refines *Stack*, but restricts the elements of the stack to only odd integers (we sidestep the issue of the stack having even elements when *OddStack* is applied; where an exception could be raised). Mixin *SneakyStack* only adds a new method that calls an existing method of *Stack*. Now, given a stack object x, if first *SneakyStack and then OddStack* are applied to x, the sequence *x.alwaysPush*(2, *d*); *x.pop*(*e*, *d*) may cause the assertion to fail. The reason is that *x.alwaysPush*(2, *d*) makes an up-call to *Stack::push*, shortcutting *OddStack::push*, which would prevent 2 to be added to the stack. This problem is avoided if *SneakyStack* makes a down-call to *self.push* instead, as then that call is dynamically resolved to *OddStack.push*, see Listing 4.3.

Listing 4.2: Flawed Mixins

```
class DirectStack extends Stack
  method push2(val e : integer, res d : boolean)
    s, d := e → e → s, true [] d := false

class OddStack extends Stack
  invariant ∀ e ∈ s • odd(e)
  method push(val e : integer, res d : boolean)
    if odd(e) then Stack.push(e, d) else d := false
  method pop(res e : integer, res d : boolean)
    Stack.pop(e, d); if d then assert odd(e)

class SneakyStack extends Stack
  method alwaysPush(val e : integer, res d : boolean)
    Stack.push(e, d)
```

Listing 4.3: Corrected Mixins

```
class DirectStack extends Stack
  method push2(val e : integer, res d : boolean)
    self.push(e, d) ; if d then self.push(e, d)

class OddStack extends Stack
  invariant ∀ e ∈ s • odd(e)
  method push(val e : integer, res d : boolean)
    if odd(e) then Stack.push(e, d) else d := false
  method pop(res e : integer, res d : boolean)
    Stack.pop(e, d); if d then assert odd(e)

class SneakyStack extends Stack
  method alwaysPush(val e : integer, res d : boolean)
    self.push(e, d)
```

Listing 4.4: More Flawed Mixins

```
class Stack12
    var s : seq(integer) := ⟨⟩
    method push(val e : integer)
        s := e → s
    method push12(val e : integer)
        s := e → s [] s := e → e → s

class Only1 extends Stack
  method push12(val e : integer)
    self.push(e)

class Only2 extends Stack
  method push12(val e : integer)
    self.push(e) ; self.push(e)
```

The second part of condition (d') requires $D::m$ to up-call $C.m$. In Listing 4.4, *push12* of *Stack12* pushes e once or twice on the stack. Mixin *Only1* redefines *push12* to push only once and mixin *Only2* redefines *push12* always to push twice. Now, if *Only1* is first mixed into an object of class *Stack12* followed by *Only2*, users of the object's *Only1* feature would still expect only one push, even if a call to *push12* will always push twice. The nondeterminism present in *Stack12*::*push12* must not be reduced in a mixin, as then different mixins could reduce it differently, resulting in the described situation. Likewise, a mixin may not enlarge the domain of termination, as another mixin that would be called first may not do so. Here, both redefinitions of *push12* would have to call *Stack12*::*push12*. Hence mixins "cannot refine" the original behaviour. Mixins can superimpose computation (like *Counter*) or add new behavior (like *Lock*).

4.4 Discussion

Verification with object invariants has been studied extensively, in the context of behavioural subtyping (America 1990; Liskov and Wing 1994) and in verification languages such as JML (for Java) and Spec# (for C#), with rules to guide the design of behaviour preserving subclasses (Barnett, DeLine, Fähndrich, Leino, and Schulte 2004; Chalin, Kiniry, Leavens, and Poll 2006; Dhara and Leavens 1996; Ruby and Leavens 2000). In particular, the Spec# verification methods allow object invariants spanning several objects through tree-like ownership structures and layers of abstractions that prescribe when invariants have to hold (Leino and Müller 2010). When mimicking mixins in Java or C#, the mixins *Stack*, *Lock* and *Counter* in Listing 4.1 become objects on their own. The *Stack* object needs to maintain a list of its extensions which in turn need to refer to each other in order to resolve base calls like

Stack.push(e, d). It is not obvious how these verification techniques can be extended to such cyclic structures. A dedicated method for the correctness of mixin composition is called for.

Chapter 5

Use Case - Intrusive Data Structures

Mixins allow the simple implementation of intrusive data structures. Consider the example of a one-to-one association, which is a relation that allows one object of its domain to relate to exactly one object of its range (Burton and Sekerinski 2013). A conventional library implementation would maintain a data structure with each element storing pointers to the pair of related objects. Looking up an object which a given object in its domain or range is related to requires traversing that data structure. With mixins, a separate data structure can be avoided by mixing in a field to a domain object that points to its range object and vice versa as shown in the module below:

```
module LinkedAssociations
  class Link
    var l: Object
  class O2O
    method add(a: Object, b: Object)
      extend a with Link ; a.l := b ;
      extend b with Link ; b.l := a
    method to(a: Object) b: Object
      if a has Link then b := a.l else b := nil
    method delete(a: Object)
      remove Link from a.l ;
      remove Link from a
```

Such an approach provides the following benefits:

1. The elimination of a separate data structure reduces the number of objects required to store the structure. With an invasive approach, the number of objects is equal to the number elements stored.

2. Access to a particular stored object ensures direct access to ones related to it. Traversal of the data structure is not required.

5.1 Abstract Specification

Using refinement theory described in Chapter 4, we express the one-to-one associations specification in terms of relations. Writing $A \leftrightarrow B$ for the type of relations between A and B, the following specification is defined:

module *Associations*
  abstract class *O2O*
    var r: *Object* $\leftrightarrow$ *Object* $:= \{\}$
    invariant $r = r^{-1} \wedge r \cap id = \{\} \wedge nil \notin dom\ r$
    method *add*(a: *Object*, b: *Object*)
      require $a \neq nil \wedge b \neq nil \wedge a \neq b \wedge$
        $(\forall\ o \in O2O \bullet a,\ b \notin dom\ o.r)$ **then**
        $r := r \oplus \{a \mapsto b,\ b \mapsto a\}$
    method *to* (a: *Object*) b: *Object*
      if $a \in dom\ r$ **then** $b := r(a)$ **else** $b := nil$
    method *delete* (a: *Object*)
      require $a \in dom\ r$ **then**
        $r := \{a,\ r(a)\} \lhd r$
  invariant *injective* $(\cup\ o \in O2O \bullet o.r)$

Class *O2O* is defined as an **abstract class** since it is a specification not intended to be directly compiled into executable code. (It is not abstract in the sense that it declares methods without bodies.)

The consistency of a given specification is checked by ensuring that invariants are preserved after the execution of any method in the module.

The class invariant for the example above states that r must be symmetric, that no element refers to itself, and that it does not relate *nil*. Method *add*(a, b) requires that both a and b must not be *nil*, that they must be distinct, and that neither a nor b is in the domain of r of any *O2O* object. The *maplet* $a \mapsto b$ is a shorthand for the pair (a, b). The relation $q \oplus r$ stands for relation q overwritten by relation r and $s \lhd r$ stands for set s subtracted from the domain of r. The module invariant, which ranges over the properties all objects created by that module (Meyer 1997), states that the union of r of all *O2O* objects must be an injective relation.

5.2 Specification Refinement

Module *LinkedAssociations* is now defined as a refinement of *Associations*:

module *LinkedAssociations* **refines** *Associations*
  class *Link*
    var l: *Object*
  class *O2O*
    method *add*(a: *Object*, b: *Object*)

extend a **with** $Link$; $a.l := b$;
extend b **with** $Link$; $b.l := a$
method to (a: $Object$) b: $Object$
if a **has** $Link$ **then** $b := a.l$ **else** $b := nil$
method $delete$ (a: $Object$)
remove $Link$ **from** $a.l$;
remove $Link$ **from** a
invariant
$O2O = Associations.O2O \wedge$
$(\forall\ o \in O2O \bullet \forall\ a,\ b \in Object \bullet a \mapsto b \in o.r \Rightarrow a.l = b) \wedge$
$Link = (\cup\ o \in O2O \bullet dom\ o.r)$

The module invariant of *LinkedAssociations* is generalized to be the *coupling invariant* of the refinement: it states that $LinkedAssociations.O2O = Associations.O2O$. Every *O2O* object of *Associations* is implemented by exactly one *O2O* object of *LinkedAssociations*. Thus if $o.r$ relates a to b, then this corresponds to $a.l$ pointing to b, for any *O2O* object o, and that every object that is being related to by $o.r$, for some *O2O* object o, has a corresponding *Link* mixin.

5.3 Correctness Proof

For the purpose of proofs, the class declarations in *Associations* are eliminated and the keywords **private var** denote that the declared variable is only accessible from inside the defined module:

module $Associations$
private var $O2O$: $set(Object) = \{\}$
private var $O2O.r$: $Object \to (Object \leftrightarrow Object)$
invariant
$(\forall\ this \in O2O \bullet this.r = this.r^{-1} \wedge this.r \cap id = \{\} \wedge$
$nil \notin dom\ this.r) \wedge$
$injective\ (\cup\ o \in O2O \bullet o.r)$
method $O2O.new()$ $this$: $Object$
$this :\notin O2O \cup \{nil\}$; $O2O := O2O \cup \{this\}$; $this.r := \{\}$
method $O2O.add(this$: $Object$, a: $Object$, b: $Object)$
require $this \in O2O \wedge a \neq b \wedge a \neq nil \wedge b \neq nil \wedge$
$(\forall\ o \in O2O \bullet a,\ b \notin dom\ o.r)$ **then**
$this.r := this.r \oplus \{a \mapsto b,\ b \mapsto a\}$
method $O2O.to$ ($this$: $Object$, a: $Object$) b: $Object$
require $this \in O2O$ **then**
if $a \in dom\ this.r$ **then** $b := this.r(a)$ **else** $b := nil$
method $O2O.delete$ ($this$: $Object$, a: $Object$)

require $this \in O2O \wedge a \in dom\ this.r$ **then**
$this.r := \{a, this.r(a)\} \lhd this.r$

For the consistency of module *Associations*, abbreviated as A, we have to show:

(a) $A_{init} \Rightarrow A_{inv}$

(b) $A_{inv} \wedge new_{pre} \Rightarrow wp(new_{body}, A_{inv})$
$A_{inv} \wedge add_{pre} \Rightarrow wp(add_{body}, A_{inv})$
$A_{inv} \wedge to_{pre} \Rightarrow wp(to_{body}, A_{inv})$
$A_{inv} \wedge delete_{pre} \Rightarrow wp(delete_{body}, A_{inv})$

Here, the module initialization has to establish the module invariant and the *new* method has to preserve it. For example, for *O2O.new* this is shown by:

$wp(O2O.new_{body}, A_{inv})$
(by (field update),(wp, seq composition),(wp, choice),(wp, assignment))
$\equiv \forall this :\notin O2O \cup \{nil\} \bullet A_{inv}[r \backslash r[this \leftarrow \{\}]]$
$[O2O \backslash O2O \cup \{this\}]$ (by substitution, renaming)
$\equiv \forall this :\notin O2O \cup \{nil\} \bullet$
$(\forall o \in O2O \cup \{this\} \bullet o.r[this \leftarrow \{\}] = o.r[this \leftarrow \{\}]^{-1} \wedge$
$o.r[this \leftarrow \{\}] \cap id = \{\} \wedge nil \notin dom\, o.r[this \leftarrow \{\}]) \wedge$
$injective(\cup o \in O2O \cup \{this\} \bullet o.r[this \leftarrow \{\}])$
(cases o ∈ O2O and o = *this*, one-point rule)

$\equiv \forall this :\notin O2O \cup \{nil\} \bullet$
$(\forall o \in O2O \bullet o.r = o.r^{-1} \wedge o.r \cap id = \{\} \wedge nil \notin dom\, o.r) \wedge$
$this.r[this \leftarrow \{\}] = this.r[this \leftarrow \{\}]^{-1} \wedge$
$this.r[this \leftarrow \{\}] \cap id = \{\} \wedge nil \notin dom\, this.r[this \leftarrow \{\}] \wedge$
$injective((\cup o \in O2O \bullet o.r) \cup this.r[this \leftarrow \{\}])$ (by function modification)
$\equiv \forall this :\notin O2O \cup \{nil\} \bullet$
$(\forall o \in O2O \bullet o.r = o.r^{-1} \wedge o.r \cap id = \{\} \wedge nil \notin dom\, o.r) \wedge$
$injective(\cup o \in O2O \bullet o.r)$ (as *this* does not occur)
$\equiv A_{inv}$

For the refinement of *Associations* by *LinkedAssociations*, abbreviated as LA, we have to show:

(a) $A_{init} \wedge LA_{init} \Rightarrow LA_{inv}$

(b) $A_{inv} \Rightarrow A.new \sqsubseteq_{LA_{inv}} LA.new$
$A_{inv} \Rightarrow A.add \sqsubseteq_{LA_{inv}} LA.add$
$A_{inv} \Rightarrow A.to \sqsubseteq_{LA_{inv}} LA.to$
$A_{inv} \Rightarrow A.delete \sqsubseteq_{LA_{inv}} LA.delete$

The definition of refinement contains a quantification over all postconditions. This does not allow *wp* to be directly applied as for invariant proofs. A large collection of derived refinement laws that allow the above conditions to be shown are given in (Back and Wright 1998; Morgan 1998); these allow compact proofs but require careful selection and instantiations of the laws. More "streamlined" proofs can be obtained by using an alternative definition of refinement that avoids quantification over predicates (Chen and Udding 1989), which is also used by the B method (Abrial 1996):

(b') each method $A.m$ is refined by $B.m$ via B_{inv} under A_{inv}:

$$A_{inv} \wedge B_{inv} \wedge A.m_{pre} \Rightarrow wp(B.m, \overline{wp}(A.m_{body}, B_{inv}))$$

Here $\overline{wp}$ is the *conjugate weakest precondition*, defined by $\overline{wp}(S, c) \equiv \neg wp(S, \neg c)$. Using this rule immediately leads to a proof condition in (first order) logic; we have applied it to establish the correctness of *LinkedAssociations* by hand. As expected, the proofs are lengthy.

5.4 Machine Automated Proofs Using Boogie

To avoid generating lengthy, repetitive proofs by hand, the correctness of this program is checked by encoding the translation into an intermediate language called Boogie (Leino 2008).

Unlike theorem provers such as Isabelle and PVS, the Boogie language explicitly models statements found in imperative programming languages. In this language, classes, objects and mixins can be easily encoded as was done in the core language. The Boogie language also contains constructs to directly embed proof obligations in a program. The Boogie verifier uses weakest precondition semantics as we do in our language for correctness checking (Barnett, Chang, DeLine, Jacobs, and Leino 2006) by discharging all annotated proof obligations. Below, relevant syntax and semantics are stated.

- **type** keyword, gives the designer the ability to create custom types. Polymorphic types are also allowed.
- **const** keyword, allows the declaration of symbolic constants.
- **requires** statement, allows the user to specify preconditions, conditions which must hold before method execution.
- **ensures** statement, specifies a method's postconditions. Multiple requires (or ensures) expressions are conjoined.

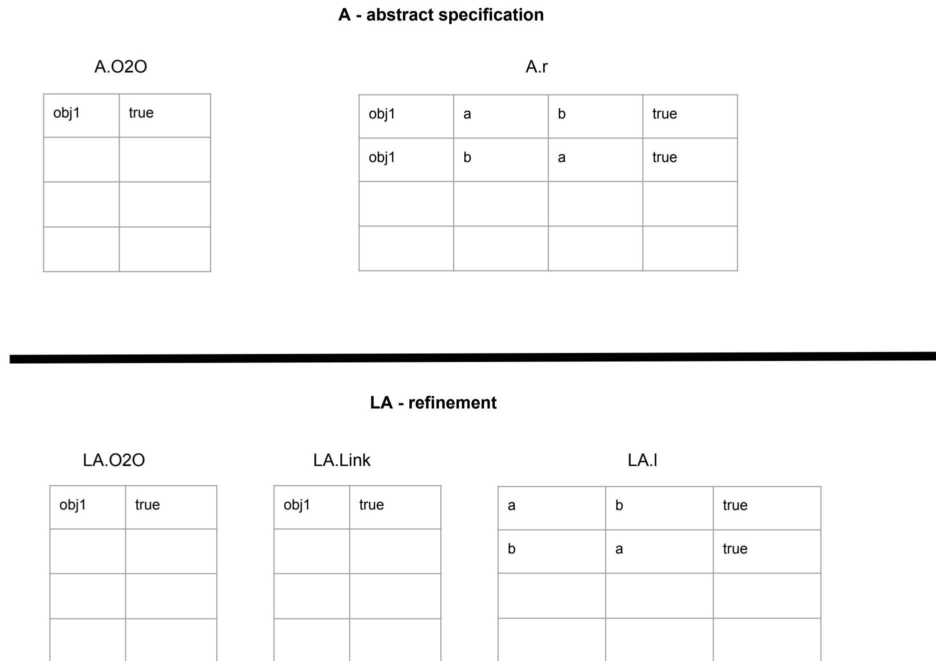

Figure 5.1: Boogie model of the Associations program. Relations are stored in maps. This illustrates the case where a single association between objects a and b are stored.

- **havoc** statement, is used to non-deterministically assign a properly-typed value to a variable. When used with the **assume** statement, a developer can further restrict the range of values that are chosen from.

- **function** declaration is used to introduce mathematical functions to a program.

- **axiom** declaration, defines restrictions on the range of values returned by functions.

In this section, the translation of the association example discussion in this chapter is described. Here, both the abstract specification and its refinement are both coded in the same module. Two sets of proof obligations are defined to check the correctness of the module. The first set is defined in terms of the specification's state space. It ensures that the abstract implementation of the module is correct relative to its implementation. The second is defined in terms of both the abstract and its refinement's state spaces. It defines a *coupling invariant* used to ensure that both state spaces are correctly related when the module's procedure is "executed".

Types

Sets are represented as a map of elements to booleans where elements mapping to **true** are in the set. Object references are of type *Object* and include the value *nil*. An object field can be encoded as a mapping from objects to values as defined in the *Field* type declaration below. Since the map value is polymorphic, an object can contain fields of arbitrary types.

Listing 5.1: Type Definitions in Boogie

```
type Set α= [α] bool;
type Relation αγ= [α] [γ] bool;
type Object;
const nil: Object;
type Field α= [Object] α;
```

Variable Declarations

Object references of a particular type or class can be stored in a predefined set. For this module, object references of type *Associations* are stored in the *A.O2O* relation with concrete objects of the refining class *LinkedAssociations* being stored in *LA.O2O*. Each variable is of type *Set* and declared to store elements of type *Object*. In this model, a reference points to both the abstract and concrete representations of an object.

The mixin *Link* is stored in a similar fashion within the *LA.Link* relation. Classes and mixins are treated similarily in this model. Class fields are encoded with a relation from object references to its field values as done with *A.r* and *LA.l*. The model is shown in Figure 5.1.

Listing 5.2: Association Module Variable Declarations

```
var LA.O2O: Set Object;
var LA.l: Relation Object Object;
var LA.Link: Set Object;

var A.O2O: Set Object;
var A.r: Field (Relation Object Object);
```

Using Invariants for Verification

The invariants of the program act as proof obligations. They are encapsulated in a function which can be checked before and after method execution. The verifier uses a method's body, preconditions and postconditions to ensure that class and module invariants are reestablished.

The class invariant found in Listing 5.3 is for verifying the correctness of the abstract specification.

Listing 5.3: Specification Verification

```
function classInvariant( specRelSet: Field (Relation Object Object),
     specObjSet: Set Object) returns (bool)
{

  !specObjSet[nil]

  // ∀o ∈ O2O • o.r = o.r⁻¹ (object associations stored separately)
  && (forall t:Object, x:Object, y:Object ::
       specObjSet[t] ==> (specRelSet[t][x][y] == specRelSet[t][y][x]))


  // r ∩ id = {}, (objects cannot be associated to themselves)
  && (forall t:Object, x:Object, y:Object :: specObjSet[t] ==>
       !specRelSet[t][x][y])

  // ∪ o ∈ O2O • o.r (objects can only participate in one association)
  && (forall x:Object, y:Object, z:Object, o:Object ::
       ( ( specObjSet[o] && specRelSet[o][x][z])
            && ( specObjSet[o] && specRelSet[o][y][z])
          ) ==>
          (x == y) )
}
```

Coupling Invariant

The coupling invariant asserts relationships between the state spaces of both the abstract specification and its refinement. Listing 5.4 illustrates the coupling invariant code for this problem.

Listing 5.4: Coupling Invariant Function

```
function couplingInvariant( relSet: Field (Relation Object Object),
     specObjSet: Set Object, refineObjSet: Set Object,
     refineCls: Set Object, refineClsAttr: Relation Object Object )
     returns (bool)
{
   // O2O=IO2O
  //(forall x:Object :: specObjSet[x] ¡==¿ refineObjSet[x] )
  specObjSet == refineObjSet
```

```
  // ∀o∈O2O • ∀a,b ∈ Object • a ↦ b ∈ o.r ⇒ a.l=b
  // (ensures that the if the relation exists in the abstract
  // state space, it also exists in the concrete one

   && (forall t:Object :: specObjSet[t] ==>
        ( forall x:Object, y:Object :: relSet[t][x][y] ==> (refineCls[x] && refineClsAttr
            [x][y]) )
        )

  // Link = (∪o ∈ O2O • dom o.r)
  // (ensure thats if the relation exists, the object in the domain has beeen extended)
  && (forall o:Object, a:Object, b:Object :: (relSet[o][a][b]) ==> (refineCls[a] &&
      refineCls[b]))
}
```

Creating a New Relation Instance Object

As shown in Listing 5.5, a new *O2O* object instance is used to store an association. An unused object reference is selected using the **havoc** − **assume** pair of statements. The **havoc** *a* statement nondeterministically chooses an object reference and assigns it to *a*. The following **assume** expression bounds the possible values that can be chosen. In this case, it guarantees that an unused object reference is selected. This new object is both an instance of the *Associations.O2O* and *LinkedAssociations.O2O* classes so the reference is stored in the *A.O2O* and *LA.O2O* maps respectively.

This procedure equates to running the constructor of the *O2O* in the abstract specification class (*Associations*) and the refining class (*LinkedAssociations*). Since no constructor implementation is provided for either, both are assumed to be **skip**.

Listing 5.5: New Associations Object

```
procedure new() returns (result: Object)

  requires classInvariant(A.r,A.O2O);
  requires couplingInvariant(A.r, A.O2O, LA.O2O, LA.Link, LA.l);

  modifies A.O2O; modifies A.r; modifies LA.O2O;
```

```
  ensures classInvariant(A.r,A.O2O);
  ensures couplingInvariant(A.r, A.O2O, LA.O2O, LA.Link, LA.l);

{
  var this: Object;

  // non-deterministically select an unused reference
  havoc this;
  assume ( !A.O2O[this] && (LA.O2O[this]!=true) && (this != nil) );

  // record that it is used in both state spaces
  A.O2O[this] := true; LA.O2O[this] := true;
  A.r[this] := relation_empty();

}
```

Adding Associated Objects to Relation Instance

The code found in Listing 5.6 models the program storing the association between two objects. In addition to the class and coupling invariants, the preconditions check to guarantee that the incoming *this* object is of the *O2O* type, that the a and b arguments are not equal and are not *nil*, and that both objects are not already involved in another association.

In Boogie, the **modifies** statement allows the user to specify which global variables are being modified in a method. This simplifies the work of the verifier, as it will only check proof obligations relating to these variables.

The abstract specification specifies that the relation a to b and b to a are recorded. These are added to the $A.r$ map. The **extend** statement in mix adds a mixin to be added to a currently existing object. The translation in Boogie equates to adding the object reference into the appropriate mixin map. Here, both objects a and b are added to the *LA.Link* map. The *Link.l* field is updated in the model by setting the *LA.l* where the first element of the map is the object reference extended by *Link*.

Listing 5.6: Setting Association Object Fields

```
procedure add(this: Object, a: Object, b: Object )
  // preconditions
  requires (A.O2O[this]==true); requires (LA.O2O[this]==true);
  requires (a!=b);
  requires (this != nil); requires (a != nil); requires (b != nil);
  requires (forall o:Object :: !(exists y: Object :: A.r[o][a][y] == true) );
  requires (forall o:Object :: !(exists x: Object :: A.r[o][x][a] == true) );
  requires (forall o:Object :: !(exists y: Object :: A.r[o][b][y] == true) );
  requires (forall o:Object :: !(exists x: Object :: A.r[o][x][b] == true) );


  requires classInvariant(A.r,A.O2O);
  requires couplingInvariant(A.r, A.O2O, LA.O2O, LA.Link, LA.l);

  modifies A.r, LA.Link, LA.l;

  // postcondition
  ensures A.r[this][a][b] == true;
  ensures A.r[this][b][a] == true;


  ensures classInvariant(A.r,A.O2O);
  ensures couplingInvariant(A.r, A.O2O, LA.O2O, LA.Link, LA.l);

{
  A.r[this][a][b] := true; A.r[this][b][a] := true;

  assume (forall y: Object :: y != b ==> !A.r[this][a][y]);
  assume (forall y: Object :: y != a ==> !A.r[this][b][y]);

  LA.Link[a] := true; LA.Link[b] := true;
  LA.l[a][b] := true; LA.l[b][a] := true;

}
```

Removing Associated Objects to Relation Instance

The removal of a relation works similarly as shown in Listing 5.7. The preconditions to the remove function ensure that the *O2O* object is a valid reference and that the object a being removed is currently in an association.

The method finds the association by using the **havoc** statement to find the object b associated with a. This object b is chosen arbitrarily but the assume statement ensures that the only object associated with the specified a is selected. The extension of both objects is eliminated by removing the elements from the *Link* set. Also, the field l of both *Link* objects is removed from the $LA.l$ map.

Listing 5.7: Clearing Association Object Fields

```
procedure remove(this: Object, a: Object )

  // preconditions
  requires (A.O2O[this]==true); requires (LA.O2O[this]==true);
  requires (exists x:Object :: A.r[this][a][x] && A.r[this][x] [a] );
  requires (LA.Link[a]==true);

  requires classInvariant(A.r,A.O2O);
  requires couplingInvariant(A.r, A.O2O, LA.O2O, LA.Link, LA.l);

  modifies A.r, LA.Link, LA.l;

  // postcondition
  ensures (forall x:Object :: A.r[this][a][x] != true);
  ensures (forall x:Object :: A.r[this][x][a] != true);

  ensures classInvariant(A.r,A.O2O);
  ensures couplingInvariant(A.r, A.O2O, LA.O2O, LA.Link, LA.l);

{
  var b: Object;

    havoc b;
    assume (A.r[this][a][b] && A.r[this][b][a] && LA.Link[b] && LA.l[a][b] && LA.l
        [b][a]);


    A.r[this][a][b] := false; A.r[this][b][a] := false;
    LA.Link[a] := false; LA.Link[b] := false;
    LA.l[a][b] := false; LA.l[b][a] := false;
}
```

5.5 Discussion

Modelling classes in Boogie relieves a developer from manually doing correctness proofs for class implementations but currently translation from mix code must be done explicitly. The translation to Boogie presented resembles the core language that describes mix as opposed to the programming language itself. Application developers may find this process tedious and error-prone. The code required to verify this program was approximately 180 lines long and took 6 seconds for the Boogie verifier to prove its correctness, so scalability may be a concern for larger applications. Some of the Boogie code, such as object management and field access is “boiler-plate” code which can be automatically generated so extending mix with specification statements and having its compiler emit Boogie code may further reduce any burden on application developers. This is left as future work.

Chapter 6

Use Case - Implementing Design Patterns

In sections 6.1-6.4, we use mixin composition idioms discussed thus far to implement some design patterns documented in (Gamma, Helm, Johnson, and Vlissides 1995). We focus on *object structural* and *object behavioural* patterns as they are used to provide run-time control over a system's feature composition and behaviour respectively. This is consistent with mix's goals of supporting role-based development and object evolution. The selected examples exploit the composition techniques described in Chapter 4.

Most of the documented implementations were originally coded in C++ (some alternate Smalltalk solutions are presented). We translate the pattern implementations into Java-like syntax so the differences between our proposed mixin-based and the corresponding documented object composition solutions are clear. The convention followed for each pattern presentation in this section is as follows. Initially, a description and summary of the pattern is stated, followed by the object composition and mixin-based implementations (the object composition one is always on the left) and is concluded with a short discussion.

Section 6.5 documents related work pertaining to object composition using design patterns.

The chapter concludes with a critical discussion about the differences between object and mixin composition.

6.1 Decorator Pattern

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Objects are dynamically added and removed from a collaboration in order to modify its behaviour. The objects form a linked list. For a specified method, objects in

the list contribute to the behaviour by defining an implementation which includes some object-specific code and an invocation of the method with the same name contained in the next object in the list. Ultimately, this results in a client's method call being transformed into a series of sequential method invocations within the collaboration.

The object composition approach creates a linked list of objects and uses method combination to deliver the desired functionality. In Listing 6.1, the *ConcreteDecoratorB* object contains a reference to a *ConcreteDecoratorA* which in turn contains a reference to a *ConcreteComponent* object. Since the *Decorator* classes and the *ConcreteComponent* implement the *Component* interface, a Decorator's *Operation()* method can call its stored object's *Operation()* method. The example results in the following calling sequence when *cdB.Operation()* is called: *ConcreteDecoratorB.Operation()* $\rightsquigarrow$ *ConcreteDecoratorA.Operation()* $\rightsquigarrow$ *ConcreteComponent.Operation()*. The equivalent behaviour in Listing 6.2 is obtained by extending a *ConcreteComponent* with Decorator mixins as done in the *Client* class. The *Component.Operation()* statement implicitly invokes the next method in the linearization chain.

The main difference between the two composition techniques is that the object composition solution relies on the developer explicitly creating the required object linked list. This does provide more flexibility. The object list can be generalized to trees where a decorator can spawn methods from multiple objects. The correctness of the list formation is left to the developer. In particular, she must ensure that no cycles are inadvertently created and all objects' "next object" reference pointers are properly initialized. The mixin-based approach ensures that the mixin list is formed without cycles. It is also simpler because there is no need to include code which checks the integrity of the mixin list. Removing decorators at runtime in the mixin-based approach involves just using the **remove** statement, unlike the object composition approach where you would have to modify references embedded in the objects.

Having multiple objects implement the Decorator pattern means that the identity of the pattern is spread across all participating objects. This makes identifying the pattern instance complex as one must track each object involved and their relationship to each other. Since mixin composition results in a single object, its reference uniquely identifies the pattern instance.

The object composition solution gives the developer the flexibility to easily share decorators between pattern instances but extra management is required if the decorators rely on state for execution. Finally, another criticism of the object composition approach is that it results in many small, similar objects.

6.2 Proxy Pattern

Provide a surrogate or placeholder for another object to control access to it.

This pattern gives the designer the ability to separate management of an object's access from its core functionality. Examples of management concerns include object

Listing 6.1: Decorator Pattern-Object Composition Solution

```
class Component { Operation(); }

class ConcreteComponent implements
    Component
  Operation() { ... };
  ...

class Decorator implements
    Component
  Component comp;
  Decorator(Component c) {comp = c
      };
  Operation() { comp.Operation() };
  ...

class ConcreteDecoratorA extends
    Decorator
  ConcreteDecoratorA(Component c) {
      super(c); };
  Operation() { ... comp.Operation() };
  ...

class ConcreteDecoratorB extends
    Decorator
  ConcreteDecoratorB(Component c) {
      super(c); };
  Operation() { ... comp.Operation() };
  ...

class Client {
  main() {
    ConcreteComponent cc = new
        ConcreteComponent();
    ConcreteDecoratorB cdB = new
        ConcreteDecoratorB(new
        ConcreteDecoratorA (cc));
    cdB.Operation();
     ...
```

Listing 6.2: Decorator Pattern-Mixin Composition Solution

```
class Component { Operation(); }

class ConcreteComponent implements
    Component
  Operation() { ... };
  ...

class DecoratorMixA extends
    Component
  Operation() { ... Component.
      Operation() };
  ...

class DecoratorMixB extends
    Component
  Operation() { ... Component.
      Operation() };
  ...

begin
  ConcreteComponent cc = new
      ConcreteComponent
  extend cc with DecoratorMixA;
  extend cc with DecoratorMixB;
  ...
  cc.Operation()
end
```

Listing 6.3: Proxy Pattern-Object Composition Solution

```
class Subject
   Request() ;
   ...

class RealSubject implements Subject
   Request();
   ...

class RemoteProxy implements
    Subject
   Subject rs;
   Request()
      ...
      rs.Request() // send request to
          remote object

class Client
   main()
      Subject rs = new RealSubject();
      rs.Request();

class NewClient
   main()
      Subject rp = new RemoteProxy();
      rp.Request();
```

Listing 6.4: Proxy Pattern-Mixin Composition Solution

```
class Subject
  Request();

class RealSubject implements Subject
  Request();
   ...
class RemoteProxy implements
    Subject
  Request()
  ...
  // send request to remote object

class Controller
  switchToRemote(Subject s)
    implement s with RemoteProxy
  switchToLocal(Subject s)
    implement s with RealSubject
    ...

begin
  RealSubject rs = new RealSubject()
  rs.Request()
  Controller sc = new Controller()
  sc.switchToRemote(rs as Subject)
end
```

security, concurrent access or expensive object creation time. In the following example, a remote proxy (used to provide access to an object in a different address space) is presented.

The object composition approach in Listing 6.3 allows a *RemoteProxy* and a *RealSubject* object to collaborate. The *RemoteProxy* object contains a field that holds a reference to the actual remote object targeted. The *NewClient* creates the proxy object and is insulated from interacting with the actual *Subject* object being used. Since the *RemoteProxy* and *Subject* both implement the same interface, the proxy can simply forward any client requests to the remote subject. Either the *RemoteProxy* and *RealSubject* mixin is combined with an object upon instantiation when mixin composition is used to implement this pattern as in Listing 6.4. The object can morph into a proxy object by atomically removing the *RealSubject* mixin and adding

the *RemoteProxy* one via the **implement** statement in the *Controller* class.

The key difference between the two approaches is the fact that when object composition is used the client code must be changed. Listing 6.3 shows that the *Client* code must be changed to instantiate the *RemoteProxy* object instead of the *RealSubject* or the remote access feature must have been anticipated during the original design of this program. Using mixin composition, *RealSubject* does not have to be aware of the remote feature and the *Client* objects do not have to change to support it. In the example, the external object *Controller* allows the remote access feature to be added or removed transparently.

6.3 Chain of Responsibility Pattern

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

A linked list of objects is created by having each object store the reference of its successor. When a request is received by the object in the list, the object either handles the request and returns to the caller or forwards the request to its defined successor. The last object in the chain has no successor and must handle the request if no one else does.

The pattern is traditionally implemented with a set of objects that implement a particular *Handler* interface as shown in Listing 6.5. All objects except for the last one, initialize their successor field to be a reference to the next object in the list. The last one does not have to initialize its successor because it is never used. The last object will be a *Base* object. Each object implements its *HandleRequest()* method in a manner such that it can determine whether it can handle the request or not. If it can, its custom handling code is executed, otherwise the same call is made on its successor. The *Base* object has no successor so it provides a default handler.

When mixins are used, as in Listing 6.6, successor references are not required, because the list is implicitly created by the mixin linearization scheme. Each handler is implemented as a mixin and instead of calling a successor's handler method, the *Base.HandleRequest()* statement is called. This will call the method of the same name higher in the linearization that implements the interface defined by the *Base* class. This implementation allows new handlers to be added by simply extending the object. Replacing a handler with another will ensure that the added handler will take the place of the original one in the linearization order.

In the Listing 6.5, the object composition design approach takes into account that the base object will be extended by handler. If the base object was originally designed to handle all requests, the Base implementation would not have the *Handler* class or the *successor* field. These would be added after the system required the feature. The mixin composition approach does not require such prior knowledge. The mixin class

Listing 6.5: CofR Pattern-Object Composition Solution

```
class Handler
    Handler successor;
    HandleRequest();

class ConcreteHandlerA implements
     Handler
    ConcreteHandlerA(Handler h) {
         successor = h ...}
    HandleRequest()
        if ("can handle request")
            ... // code to handle
        else
            successor.HandleRequest();

class ConcreteHandlerB implements
     Handler {
    ConcreteHandlerB(Handler h) {
         successor = h ...}
    HandleRequest()
        if ("can handle request")
            ... // code to handle
        else
            successor.HandleRequest();

class Base implements Handler
    HandleRequest() { ... }

class Client
    main()
        ConcreteHandlerB chB = new
             ConcreteHandlerB(new
             ConcreteHandlerA(new Base
             (null)));
        chB.HandleRequest();
        ...
```

Listing 6.6: CofR Pattern-Mixin Composition Solution

```
class ConcreteHandlerA extends Base

    HandleRequest() {
        if ("can handle request") then
            ... // code to handle
        else
            Base.HandleRequest();

class ConcreteHandlerB extends Base

    HandleRequest() {
        if ("can handle request") then
            ... // code to handle”
        else
            Base.HandleRequest();

class Base
    HandleRequest() { ... }

class Client
    main()
        Base b = new Base();
        extend b with
             ConcreteHandlerA;
        extend b with
             ConcreteHandlerB;
        b.HandleRequest()
        ...
```

Listing 6.7: Strategy Pattern-Object Composition Solution

```
class Context
    Strategy s;
    ContextInterface() { s.
        AlgorithmInterface() };

class Strategy
    AlgorithmInterface();

class ConcreteStrategyA implements
     Strategy
    AlgorithmInterface() { ... }

class ConcreteStrategyB implements
     Strategy
    AlgorithmInterface() { ... }

class Client
    main()
        Context c = new Context();
        c.s = new ConcreteStrategyA();
        c.ContextInterface();
        ...
        c.s = new ConcreteStrategyB();
```

Listing 6.8: Strategy Pattern-Mixin Composition Solution

```
class Strategy
    AlgorithmInterface();

class ConcreteStrategyA implements
     Strategy
    AlgorithmInterface() { ... }

class ConcreteStrategyB implements
     Strategy
    AlgorithmInterface() { ... }

class Client
    main()
        Strategy s = new
            ConcreteStrategyA;
        s.AlgorithmInterface()
        ...
        implement ConcreteStrategyA
            with ConcreteStrategyB();
        s.AlgorithmInterface()
        ...
```

extends the *Base* class without its knowledge by implementing extensions conforming to the *Base* interface.

6.4 Strategy Pattern

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.

The object that allows a client to initiate an algorithm and retrieve its results is static. Algorithm execution requests are delegated to a secondary object which does the actual computation.

Here we use object composition to create an aggregate relationship between the *Context* and *Strategy* objects. The client interface is decoupled from the algorithm as shown in Listing 6.7. This allows the code implementing the algorithm (the *Strategy*

object) to be changed at runtime with minimal changes to the client. In the example, the client would only have to change code at the point of the instantiation. The *Context* code exposes access to the embedded *Strategy* object so that the algorithm can be changed and the developer is responsible for ensuring that the *ConcreteStrategy* object is available.

Using mixins, both the client interface and algorithm services can be combined into one object without losing the ability to change the algorithm implementation at runtime. In Listing 6.8, the algorithm is mixed into client interface's *s* object via the *ConcreteStrategyA* mixin when it is created. Since its *Strategy* interface implementation is needed, *ConcreteStrategyA* cannot be removed dynamically from the object unless it is replaced by another one, such as *ConcreteStrategyB*, that implements it. This can be done using the **implement** construct that can be used to switch the algorithm's implementation at runtime. The mixin language typing rules ensure that the new algorithm mixin actually implements the *Strategy* class, that such a mixin is always available to the object and a call to the *AlgorithmInterface()* is safe throughout a *Strategy* typed object's lifetime. If object composition is used to implement this pattern, safety checking code should be added to ensure that some *Strategy* implementation is always available.

6.5 Patterns to Support Object Extension

Patterns have been documented to address the need for object extension at runtime. Delegation is commonly used to accomplish this in statically typed languages like Java and C++. In this section, we identify patterns in the literature relevant to supporting object extension via delegation and discuss their relevance and usage relative to mixin composition.

6.5.1 Dynamic Object Model Pattern

Allow a system to have new and changing object types without having to reprogram the system. Representing the object types as objects means that they can be changed at configuration time or at runtime, which makes it easy to change and adapt the system to new requirements. (Dirk Riehle and Johnson 2006)

This pattern encodes an object's type and its attributes into objects separate from itself. An object instance aggregates these objects and delegates calls to them when received from clients.

Listing 6.9: Dynamic Object Model Pattern-Type Code

```
class ObjectId
    String objName
    Integer objId
    ObjectId(String str, Integer i) {
        objName = str, objId = i }

class Type
    String typeName;
    Integer typeId;
    List<String, PropertyType>
        typeProperties;

class PropertyType
    String name;
    Type type;

class Property
    PropertyType propType;
    Value val;
```

Listing 6.10: Dynamic Object Model Pattern-Object/Client Code

```
class GenericObject
    Type type;
    ObjectId oId;
    List<String, Property> properties;

    GenericObject(Type t, ObjectId id) {
        type = t; oId = id}

    Type getType() { return type; }
    Property getProperty(String name)
        return properties.get(name);
    void setProperty(String name,
        Property p)
        properties.put(name, p);

class Client
    main ()
        GenericObject obj =
            new GenericObject( new
                Type() , 20032);
        obj.setProperty("some field",
            new Property() );
        ...
```

This pattern was proposed to allow objects to change their type at runtime and is an alternative to mixin composition and delegation for object extension. Types are defined as objects themselves as shown in Listing 6.9. Each type object defines a collection of *PropertyType* objects which define and place restrictions on the object's fields. State is stored as a collection of *Property* objects. Here, an object's identity (*GenericObject*) is distinct from its type and state. Method definitions are not included in the example but would be contained in the *Type* objects. This approach allows an object to morph into any type but checks to ensure that the *Value* objects stored in the *Property* objects are of the correct type. Furthermore, the object must be checked to ensure all required *Property* objects are present. The client code must refer to fields by identifier which is prone to error. With mixin composition, these safety checks are not required because the type system ensures that the object contains all fields defined in its base class's definition and the fields are of the correct type.

An object can be an instance of one type at a time but can have optional fields

bound to it. However, these fields aren't grouped via relationship so adding features as a unit is not easy. Since mixin composition adds object extensions in a modular fashion, features are easily identified in the code. Furthermore, extending an object with a mixin ensures that all fields defined in that mixin class are added. A developer must only ensure that a particular mixin has been added but need not check for the presence and type of individual fields.

6.5.2 Extension Objects Pattern

Anticipate that an object's interface needs to be extended in the future. Additional interfaces are defined by extension objects (Gamma 1997)

When using delegation for object extension, each extension object gives a client a different view to an object. The availability of an extension interface object is embedded in the base object. At runtime, clients can query to check for an object extension's availability and receives a reference to the object implementing that extension.

Listing 6.11: Extension Object Pattern

```
class SpecificExtension implements Extension
      { ... }

class Subject
   Extension GetExtension() { return NULL;}

class ConcreteSubject extends Subject
   SpecificExtension specificExtension;
   GetExtension(String name)
      if ("SpecificExtension" == name)
            return specificExtension;
      else
         Subject::GetExtension(name);

class Client
   main()
      SpecificExtension ext;
      Subject sub = new ConcreteSubject();
      ext = (SpecificExtension)
         (sub.GetExtension("SpecificExtTypo
              ") );

class MixClient
   main()
      SpecificExtension ext;
      Subject sub = new ConcreteSubject();
      ...
      if (sub has SpecificExtension)
         ext = sub as SpecificExtension
```

Listing 6.12: Role Object Pattern

```
class Component
   getRole(String); addRole(String);
   removeRole(String); hasRole(String);

class ComponentCore implements
     Component
   Map<String, ComponentRole> role;
   getRole(String roleName) { return role[
        roleName]; }
   addRole(String roleName)
      if (roleName == "ConcreteRoleA")
         role[roleName] = new ConcreteRoleA
              ()
   ...

class ComponentRole implements Component
   ComponentCore core;
   Operation() { core.Operation() };
   getRole(String roleName) { core.getRole(
        roleName); };
   addRole(String roleName)
      return core.addRole(roleName);
   ...

class ConcreteRoleA extends ComponentRole
     { ... }

class Client {
   main () {
      Component cs = new ComponentCore();
      cs.addRole("ConcreteRoleA");
      ConcreteRoleA cr =
         (ConcreteRoleA)cs.getRole("
              ConcreteRoleA");
      ...
      cs.removeRole("ConcreteRoleA") ;

class MixClient {
   main () {
      ComponentCore cs = new
           ComponentCore();
      extend cs with ConcreteRoleA;
      ConcreteRoleA cr = cs as
           ConcreteRoleA;
      ...
      remove ConcreteRoleA from cs;
```

In Listing 6.11, the *ConcreteSubject* has been extended by an object of type *SpecificExtension*. The *GetExtension()* is provided, so that clients can query for the availability of this extension using an identifier. Note that any mistake in the identifier (like the *sub.GetExtension*(`"SpecificExtTypo"`) statement in *Client.main()*) will not be detected by the compiler. Any future extensions require modification to the if statement in the *GetExtension()* method and the addition of a new field to store the extension's reference.

If the solution is designed using mixin composition, the code currently found in the *ConcreteSubject* object is not needed. The query can be done using the **has** and **as** keywords as shown in the *MixClient*. The query is type checked at compile time. Also, since the composition results in a single object, no reference fields need to be added if the object type supports future extensions.

6.5.3 Role Object Pattern

Adapt an object to different client's needs through transparently attached role objects, each one representing a role the object has to play in that client's context. The object manages its role set dynamically. By representing roles as individual objects, different contexts are kept separate and system configuration is simplified (Bäumer, Riehle, Siberski, and Wulf 1997).

This has been presented as a systematic way of structuring objects based on the role they play in pattern collaborations. Here, roles can be acquired and released by an object at runtime. It has been developed because it is often desirable to have an object participate in multiple pattern collaborations concurrently (Fowler 1997). This approach provides an interface for managing associated role objects. All role objects implement the role management interface functions, so the clients have control over the roles that are available for each object.

This pattern is similar to the *Extension Object* but emphasizes that different clients may require a subject object to take on different roles depending on the current context. Clients have the ability to add and remove roles to the object. The example in Listing 6.12 shows that the *ComponentCore* object has a reference to a container of extensions indexed by an identifier. This allows the pattern instance to accept future extensions without new fields. The roles are independent and of type *ComponentRole*. This allows the roles such as objects of type *ConcreteRoleA* to obtain a reference to its core object (of type *ComponentCore*) but not to the other roles in the pattern. This makes method combination difficult to implement using this approach. Clients use identifiers to index roles and access them via the *ComponentCore* class. These identifiers can not be checked by the compiler so run-time errors can occur if the wrong identifiers are used. By allowing the clients to manage the mixin class directly as in *MixClient*, object management functionality such as adding and removing roles can be type checked by the compiler.

6.6 Discussion

The examples presented show that mixin composition is a viable alternative to object composition in many situations. While object composition results in multiple objects requiring management, mixins place the whole pattern implementation into a single object, yet entering or leaving the collaboration remains simple.

The patterns presented in Section 6 all benefit when they are implemented using dynamic mixins for implementation. They eliminate the need to manage the memory of extension objects. In the *Strategy* pattern, the core object must ensure that the strategy extension is always available when a client invokes it. The mixin approach guarantees this if the core object is defined to need the extension.

Object composition requires that a core object contains fields explicitly defined to store references to its extensions. This must be done at design time so if object evolution involves future extensions, the core object code must be modified to support them. In the *Decorator* and *Chain of Command* pattern implementations, method combinations are formed using these references. When mixin composition is used, these references are not required so the core object's class specification need not change if the original design was not intended to support method combination. This issue is of particular interest when implementing the *Proxy* pattern. The object composition implementation of the *Proxy* pattern requires clients of a core object to change their code when proxy functionality was unexpectedly added. Using mixins, proxy functionality can be added with the modifications being transparent to client objects.

Patterns are found in the literature that specifically address the problems associated with extension via object composition. The *Dynamic Object Model* pattern is an alternative to mixin composition but most type checking is left to the developer and treating extensions as distinct features is difficult. Using the *Extension Object* pattern requires that its use is anticipated in the original design and the *Role Object* pattern can not rely on the type system to verify that client lookup identifiers are valid. Mixin composition does not suffer from these shortcomings.

Mixins are a useful approach for composition when runtime object evolution is desired. They are also useful in patterns like the *Decorator* that may create many objects. While some patterns benefit from mixin implementation, static patterns like the *Facade* or the creational design patterns in (Gamma, Helm, Johnson, and Vlissides 1995) do not exploit the advantages that mixins offer. Also, class based patterns such as the *Template* pattern do not benefit from mixin composition because they are explicitly designed to use subclassing and inheritance. Another known issue with mixin composition is that mixins cannot retain state after they are removed from an object. In situations where persistent state is required, object composition is a better option. Since mixins in a collaboration are embedded in a single object, they cannot be shared among pattern implementations like objects can. Although this usually complicates an application, there may be some cases where this possibility is useful

and object composition would be a more appropriate choice.

We view mixins in the proposed language as a means of providing required services for an object to be instantiated. They can be swapped out for compatible ones at runtime in a safe, systematic fashion. Alternatively, they can be used to add optional features to an object. While safety must be explicitly handled by the programmer, this allows objects to evolve when modifications were not anticipated.

Chapter 7

Conclusion

The goal of this work is to explore the viability of dynamic mixins as a useful software design element. In particular, the focus was placed on the benefit of reusing mixin code in different applications.

Mixins allow object extension at runtime without requiring previous design to support them while ensuring that the object has a single identity. Shortcomings with dynamic mixins currently include type-safety, performance and unexpected behaviour changes after mixin composition.

This work takes a multi-pronged approach to addressing these concerns. First of all, the type safety concern is addressed by proposing a new statically typed language mix. The language requires the user to explicitly specify the interface implementation required when a mixin is bound to a dependent object. This removes the possibility of run-time typing errors.

Since objects can be extended by an arbitrary number of mixins, a fixed memory layout is not suitable. Current implementations store extensions in an abstract data structure which limits the efficiency of method calls. This issue is addressed with an object model that represents an object with a linked list of memory segments. This list is managed by the language implementor not the developer and is designed to optimize method combination required when mixin composition modifies current behaviour. A compiler which parses mix code was built as part of the project. It uses the proposed object model when generating executable code. Tests conducted using the compiler's generated code show that method calls and field access approach the performance of statically typed language implementations.

Finally, a criticism of mixins is that linearization can cause unintended method overriding and that method chaining can be ambiguous. The correct use of interfaces, however, reduces the chance of accidental overriding and ambiguous method chaining gives mixins the power to support unexpected object extensions. The rules proposed in this work give programmers the ability to determine when mixin composition can be done without compromising program correctness.

The thesis doesn't address mixin extensions occurring in a concurrent environment. In a multithreaded environment, it is possible that a thread relying on a objects' mixin has been removed by another one during a task switch. Furthermore, security implications associated with object extension are not addressed. In the model proposed, it is possible that an object is extended with malicious code.

Initializing mixins upon object extension is required in some applications. Implementation of this has been left as future work. Safe object extension is guaranteed by following the rules mentioned in this work. Automating the enforcement of such rules so that they can be checked at compile-time is also left as future work.

Bibliography

Abrial, J.-R. (1996). *The B Book: Assigning Programs to Meanings*. Cambridge University Press.

Allen, E., J. Bannet, and R. Cartwright (2003, October). A first-class approach to genericity. *SIGPLAN Not. 38*(11), 96–114.

America, P. (1990). Designing an object-oriented programming language with behavioural subtyping. In J. W. d. Bakker, W. P. d. Roever, and G. Rozenberg (Eds.), *Foundations of Object-Oriented Languages*, Volume 489 of *Lecture Notes in Computer Science*, pp. 60–90. Springer Berlin Heidelberg.

Ancona, D., G. Lagorio, and E. Zucca (2003, September). Jam—designing a java extension with mixins. *ACM Trans. Program. Lang. Syst. 25*, 641–712.

Apel, S., T. Leich, and G. Saake (2006). Aspectual mixin layers: aspects and features in concert. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, New York, NY, USA, pp. 122–131. ACM.

Back, R.-J. and J. v. Wright (1998). *Refinement Calculus: A Systematic Introduction*. Springer-Verlag.

Barnett, M., B.-Y. Chang, R. DeLine, B. Jacobs, and K. Leino (2006). Boogie: A modular reusable verifier for object-oriented programs. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever (Eds.), *Formal Methods for Components and Objects*, Volume 4111 of *Lecture Notes in Computer Science*, pp. 364–387. Springer Berlin / Heidelberg.

Barnett, M., R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte (2004). Verification of object-oriented programs with invariants. *Journal of Object Technology 3*(6), 27–56.

Bäumer, D., D. Riehle, W. Siberski, and M. Wulf (1997). The role object pattern. In *Proceedings of Pattern Languages of Programs '97*, Number Technical Report WUCS-97-34 in PLoP '97. PLoP: Washington University Dept. of Computer Science.

Bolz, C. F. (2011, November). Efficiently implementing python object with maps. http://morepypy.blogspot.ca/2010/11/efficiently-implementing-python-objects.html.

Bracha, G. and W. Cook (1990). Mixin-based inheritance. In *European Conference on Object-oriented Programming / Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pp. 303–311. ACM.

Burton, E. and E. Sekerinski (2013). Correctness of intrusive data structures using mixins. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '13, New York, NY, USA, pp. 53–58. ACM.

Burton, E. and E. Sekerinski (2014). Using dynamic mixins to implement design patterns. In *Proceedings of the 19th European Conference on Pattern Languages of Programs*, EuroPLoP '14, New York, NY, USA, pp. 14:1–14:19. ACM.

Burton, E. and E. Sekerinski (2015). The safety of dynamic mixin composition. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, New York, NY, USA, pp. 1992–1999. ACM.

Burton, E. and E. Sekerinski (2016). An object model for a dynamic mixin based language. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, New York, NY, USA, pp. 1986–1992. ACM.

Burton, E. and E. Sekerinski (2017). An object model for dynamic mixins (in print). *Computer Languages, Systems & Structures 2*(12), 1–12.

Ceri, S., F. Daniel, M. Matera, and F. M. Facca (2007, February). Model-driven development of context-aware web applications. *ACM Transactions on Internet Technology 7*(1), 343–355.

Chalin, P., J. Kiniry, G. Leavens, and E. Poll (2006). Beyond assertions: Advanced specification and verification with jml and esc/java2. In F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever (Eds.), *Formal Methods for Components and Objects*, Volume 4111 of *Lecture Notes in Computer Science*, pp. 342–363. Springer Berlin Heidelberg.

Chambers, C., D. Ungar, and E. Lee (1989, September). An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not. 24*(10), 49–70.

Chen, W. and J. T. Udding (1989). Towards a calculus of data refinement. In J. L. A. v. d. Snepscheut (Ed.), *Mathematics of Program Construction, 375th Anniversary of the Groningen University*, Lecture Notes in Computer Science 375, Groningen, The Netherlands, pp. 197–218. Springer-Verlag.

CodeHaus (2014, February). Groovy - runtime mixins. http://groovy-lang.org/.

Dhara, K. K. and G. T. Leavens (1996). Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, Washington, DC, USA, pp. 258–267. IEEE Computer Society.

Dirk Riehle, M. T. and R. Johnson (2006). *Pattern Languages of Program Design 5*. Addison-Wesley.

Dixon, R., T. McKee, M. Vaughan, and P. Schweizer (1989). A fast method dispatcher for compiled languages with multiple inheritance. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '89, New York, NY, USA, pp. 211–214. ACM.

Ducasse, S., O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black (2006, March). Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst. 28*(2), 331–388.

Ducournau, R. (2011, May). Coloring, a versatile technique for implementing object-oriented languages. *Softw. Pract. Exper. 41*(6), 627–659.

Fitzgerald, M. (2007). *Learning Ruby*. O'Reilly Media.

Flatt, M., S. Krishnamurthi, and M. Felleisen (1998a). Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, New York, NY, USA, pp. 171–183. ACM.

Flatt, M., S. Krishnamurthi, and M. Felleisen (1998b). Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, New York, NY, USA, pp. 171–183. ACM.

Fowler, M. (1997). Dealing with roles. In *Proceedings of PLoP '97*, Number Technical Report WUCS-97-34 in PLoP '97. Washington University Dept. of Computer Science.

Gamma, E. (1997). The extension objects pattern. In R. Martin, D. Riehle, and F. Buschmann (Eds.), *PLoP'96. 3rd Conference on Pattern Languages of Programs*. Addison-Wesley.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Goldberg, A. and D. Robson (1983). *Smalltalk-80: the language and its implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Google (2015, December). Design elements - chrome v8. https://developers.google.com/v8/design.

Harmes, R. and D. Diaz (2007). *Pro JavaScript Design Patterns* (1 ed.). Apress.

Hirschfeld, R., P. Costanza, and O. Nierstrasz (2008). Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich 7*(3), 125–151.

Hoare, C. A. R. (1972). Proof of correctness of data representation. *Acta Informatica 1*(4), 271–281.

Ishizaki, K., T. Ogasawara, J. Castanos, P. Nagpurkar, D. Edelsohn, and T. Nakatani (2012, March). Adding dynamically-typed language support to a statically-typed language compiler: performance evaluation, analysis, and tradeoffs. *SIGPLAN Not. 47*(7), 169–180.

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold (2001). An overview of aspectj. In J. Knudsen (Ed.), *ECOOP 2001 — Object-Oriented Programming*, Volume 2072 of *Lecture Notes in Computer Science*, pp. 327–354. Springer Berlin / Heidelberg.

Kristensen, B. B. and K. Osterbye (1996, December). Roles: conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems 2*(3), 143–160.

Leino, K. R. M. (2008). This is boogie 2.

Leino, K. R. M. and P. Müller (2010). Advanced lectures on software engineering. In P. Müller (Ed.), *Advanced Lectures on Software Engineering: LASER Summer School 2007/2008*, Chapter Using the Spec# Language, Methodology, and Tools to Write Bug-free Programs, pp. 91–139. Berlin, Heidelberg: Springer-Verlag.

Liskov, B. H. and J. M. Wing (1994). A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems 16*(6), 1811–1841.

Lutz, M. (2008). Learning python.

Meyer, B. (1997). *Object-Oriented Software Construction* (2nd ed.). Prentice-Hall, Inc.

Mikhajlov, L. and E. Sekerinski (1998). A study of the fragile base class problem. In E. Jul (Ed.), *ECOOP'98 — 12th European Conference on Object-Oriented Programming*, Volume 1445 of *Lecture Notes in Computer Science*, pp. 355–382. Springer-Verlag. July 20-24, 1998.

Mikhajlova, A. and E. Sekerinski (1997). Class refinement and interface refinement in object-oriented programs. In J. Fitzgerald, C. Jones, and P. Lucas (Eds.), *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, Volume 1313 of *Lecture Notes in Computer Science*, Graz, Austria, pp. 82–101. Springer-Verlag.

Moon, D. A. (1986, June). Object-oriented programming with Flavors. *SIGPLAN Notices 21*, 1–8.

Morgan, C. C. (1998). *Programming from Specifications* (2nd ed.). Prentice Hall.

Myers, A. C. (1995). Bidirectional object layout for separate compilation. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '95, New York, NY, USA, pp. 124–139. ACM.

Naumann, D. A. (1995, October). Predicate transformers and higher-order programs. *Theoretical Computer Science 150*(1), 111–159.

Neumann, G. and U. Zdun (1999). Enhancing object-based system composition through per-object mixins. In *IN PROCEEDINGS OF ASIAPACIFIC SOFTWARE ENGINEERING CONFERENCE (APSEC*.

Ossher, H., M. Kaplan, A. Katz, W. H. Harrison, and V. J. Kruskal (1996). Specifying subject-oriented composition. *TAPOS 2*(3), 179–202.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*, 1053–1058.

Prehofer, C. (2001). Feature-oriented programming: A new way of object composition. *Concurrency and Computation: Practice and Experience 13*(6), 465–501.

Pugh, W. and G. Weddell (1990, June). Two-directional record layout for multiple inheritance. *SIGPLAN Not. 25*(6), 85–91.

Ressia, J., T. Gîrba, O. Nierstrasz, F. Perin, and L. Renggli (2014, April). Talents: An environment for dynamically composing units of reuse. *Softw. Pract. Exper. 44*(4), 413–432.

Ruby, C. and G. T. Leavens (2000, October). Safely creating correct subclasses without seeing superclass code. *SIGPLAN Notices 35*(10), 208–228.

Salehie, M. and L. Tahvildari (2009, May). Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems 4*(2), 14:1–14:42.

Simons, A. J. H. (2004). The theory of classification part 15: Mixins and the superclass interface. *Journal of Object Technology 3*(10), 7–18.

Stroustrup, B. (1999). Multiple inheritance for c++.

Subramaniam, V. (2008). *Programming Groovy: Dynamic Productivity for the Java Developer*. Pragmatic Bookshelf.

Szyperski, C. A. (1992). Import is not inheritance - why we need both: Modules and classes. In *Proceedings of the European Conference on Object-Oriented Programming*, London, UK, pp. 19–32. Springer-Verlag.

Templ, J. (1993, April). A systematic approach to multiple inheritance implementation. *SIGPLAN Not. 28*(4), 61–66.

Ungar, D. and R. B. Smith (1991). Self: The power of simplicity. *Lisp and Symbolic Computation 4*(3), 187–205.

Van Limberghen, M. and T. Mens (1996). Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems 3*, 1–30.

VanHilst, M. and D. Notkin (1996, October). Using role components in implement collaboration-based designs. *SIGPLAN Notices 31*(10), 359–369.

Zibin, Y. and J. Gil (2003). Two-dimensional bi-directional object layout. In L. Cardelli (Ed.), *ECOOP 2003 – Object-Oriented Programming*, Volume 2743 of *Lecture Notes in Computer Science*, pp. 329–350. Springer Berlin Heidelberg.

Appendix A

mix Concrete Syntax

This section describes the concrete grammar of mix. It adds tokens to the abstract syntax found in Section 2.2 needed to correctly parse program input. Non-executable statements are removed and standard result parameter passing is done. Levels of indentation can be used as delimiters and to group statements as in Python. Here, INDENT stands for having more spaces then the previous line. This sets a new level of indentation. NL stands for a new line and retaining the same level of indentation as the previous line. DEDENT stands for setting the level of indentation to the previous level. Here token sequences enclosed in curly braces, { } , may occur zero to many times. Tabs are not supported. Tokens in bold are keywords in the language.

⟨*compilation_unit*⟩ ::= ⟨*package*⟩ | ⟨*program*⟩

⟨*package*⟩ ::= **package** ⟨*id*⟩ *INDENT* ⟨*class*⟩ {*NL* ⟨*class*⟩} *DEDENT*

⟨*program*⟩ ::= **program** ⟨*id*⟩ *INDENT* ⟨*class*⟩ {*NL* ⟨*class*⟩} **begin** ⟨*statement_suite*⟩ **end** *DEDENT*

⟨*class*⟩ ::= **class** ⟨*id*⟩ [**extends** *idList*] [**implements** *idList*] *INDENT* ⟨*member*⟩ {*NL* ⟨*member*⟩ } *DEDENT*

⟨*member*⟩ ::= ⟨*constant*⟩ | ⟨*variable*⟩ | ⟨*method*⟩ | ⟨*init*⟩

⟨*constant*⟩ ::= **const** ⟨*id*⟩ [":" ⟨*type*⟩] "=" ⟨*expression*⟩

⟨*variable*⟩ ::= **var** ⟨*idList*⟩ ":" ⟨*type*⟩

⟨*local_variable*⟩ ::= ⟨*constant*⟩ | ⟨*variable*⟩

⟨*init*⟩ ::= **initialization** [⟨*formals*⟩] ⟨*statement_suite*⟩

⟨*method*⟩ ::= **method** ⟨*id*⟩ [⟨*formals*⟩] [":" ⟨*type*⟩] [⟨*statement_suite*⟩]

⟨*statement_suite*⟩ ::= ⟨*simple_statement_list*⟩ | *INDENT* ⟨*statement*⟩ {*NL* ⟨*statement*⟩ } *DEDENT*

```
⟨statement⟩ ::= ⟨simple_statement_list⟩ | ⟨compound_statement⟩

⟨simple_statement_list⟩ ::= ⟨simple_statement⟩ {";" ⟨simple_statement⟩}

⟨simple_statement⟩ ::= abort
                     | ⟨designator⟩ [":=" ⟨expression⟩ ]
                     | return [⟨expression⟩]
                     | extend ⟨designator⟩ with ⟨id⟩
                     | implement ⟨id⟩ with ⟨id⟩ in ⟨designator⟩

⟨compound_statement⟩ ::= if ⟨expression⟩ then ⟨statement_suite⟩ [else ⟨statement_suite⟩]
                       | while ⟨expression⟩ do ⟨statement_suite⟩
                       | ⟨local_variable⟩ {NL ⟨local_variable⟩} ⟨statement_suite⟩

⟨formals⟩ ::= "(" ⟨idList⟩ ":" ⟨type⟩ {"," ⟨idList⟩ ":" ⟨type⟩ } ")"

⟨idList⟩ ::= ⟨id⟩ {"," ⟨id⟩}

⟨type⟩ ::= {array of} ( integer | boolean | ⟨type⟩ | ⟨id⟩ )

⟨digit⟩ ::= 0 | 1 | ...
⟨char⟩ ::= "a" | "b" | ...
⟨integer⟩ ::= ⟨digit⟩ { ⟨digit⟩ }
⟨id⟩ ::= ⟨char⟩ {⟨char⟩ | ⟨digit⟩ }
⟨boolean⟩ ::= true | false

⟨expression⟩ ::= ⟨conjunction⟩ {or ⟨conjunction⟩ }
⟨conjunction⟩ ::= ⟨relational⟩ {and ⟨relational⟩ }
⟨relational⟩ ::= ⟨additive⟩ [("<" | ">" | "≤" | "≥" | "=" | "≠") ⟨additive⟩ ]
⟨additive⟩ ::= ⟨multiplicative⟩ {("+" | "−") ⟨multiplicative⟩}
⟨multiplicative⟩ ::= ⟨unary⟩ {("*" | "/" | "div" | "mod") ⟨unary⟩}
⟨unary⟩ ::= ["-" | "¬ | "+"] ⟨primary⟩

⟨primary⟩ ::= nil | ⟨boolean⟩ | ⟨integer⟩
            | new ⟨id⟩ [⟨expression⟩ | ⟨actuals⟩]
            | ⟨designator⟩ [⟨actuals⟩] [(as | has) ⟨id⟩]
            | ⟨designator⟩ "(" ⟨expression⟩ ")"

⟨designator⟩ ::= ⟨id⟩ {"." ⟨id⟩ | "[" ⟨expression⟩ {, ⟨expression⟩} "]" }

⟨actuals⟩ ::= "(" ⟨expression⟩ { ["," ] ⟨expression⟩ } ")"
⟨intList⟩ ::= ⟨integer⟩ {"," ⟨integer⟩ }
```

Appendix B

mix Program Generated Code

This section contains code generated by the compiler that was built for this work. Note that indentation and comments were added for improved readability.

Listing B.1: C Code Generated by Compiler of *Point* Class Example in Listing 3.1

```
#include <stdlib.h>
#include <stdio.h>
void* object_MethodTable;
struct Object_Interface;
typedef struct {
        void (*print)(struct Object_Interface*);
} Object_Methods;

typedef struct Object_Interface {
        Object_Methods* methods;
        struct Object_Interface* Object_cycle;
} Object_Interface;


void Object_print(Object_Interface* self) {
        printf("%p", self);
}

Object_Interface* findBottom(Object_Interface* p, void* t) {
         short type_found = 0;
         Object_Interface* q = p;
         Object_Interface* bottom = NULL;
         do {
                 if (q->methods == t) type_found = 1;
                 if (q->Object_cycle->methods == object_MethodTable) bottom = q;
                 q = q->Object_cycle;
} while(p != q);
        if (type_found) return bottom;
        return 0;
}
```

```
void* castObject(void* m, void* t) {
        Object_Interface* temp = ((Object_Interface*) m)->Object_cycle;

        while ((temp->methods != object_MethodTable) && (temp->methods != t))
                temp = temp->Object_cycle;

        if ((temp->methods == object_MethodTable) && (t != object_MethodTable)) {
                printf("exception");
                return NULL;
        }
        return temp;
}

void Object_init() {

        object_MethodTable = malloc(sizeof(Object_Methods));
        ((Object_Methods*) object_MethodTable)->print = &Object_print;
}

Object_Interface* Object_new() {
        Object_Interface* x = (Object_Interface *) malloc(sizeof(Object_Interface));
        x->methods = (Object_Methods*) object_MethodTable;
        x->Object_cycle = (Object_Interface*) x;
        return (Object_Interface*) x;
}



void* point_MethodTable;
void* multipoint_MethodTable;
void* arraypoint_MethodTable;
void* member_MethodTable;

// structure holds pointers to methods for class Point
// along with type of class that it extends, implements or inherits
// also used identify type of segment
typedef struct Point_Methods {
        void* extends;
        void* implements;
        void* inherits;
        void (*construct)(void*, int y0 , int x0);
        void (*move)(void*, int y0 , int x0);
} Point_Methods;

typedef struct Point_Impl {
        Point_Methods* methods;
        Object_Interface* Object_cycle;
        struct Point_Impl* Point_bottom;
        int x;
        int y;
} Point_Impl;
```

```
void Point_construct ( Point_Impl* self, int y0 , int x0){
        self->x = x0;
        self->y = y0;
}
void Point_move ( Point_Impl* self, int y0 , int x0){
        self->x = self->x+x0;
        self->y = self->y+y0;
}

void Point_init() {
        point_MethodTable = malloc(sizeof(Point_Methods));
        ((Point_Methods*)point_MethodTable)->construct = (void(*)(void *, int y0 , int x0))
            &Point_construct;
        ((Point_Methods*)point_MethodTable)->move = (void(*)(void *, int y0 , int x0)) &
            Point_move;
}

void Point_bottomdateSelfPointers(Object_Interface* p, void* e) {
}

Point_Impl* Point_extend(Object_Interface* object_arg) {
        Point_Impl* x = (Point_Impl*) malloc(sizeof(Point_Impl));
        Object_Interface* bottom = findBottom(object_arg, object_MethodTable);
        Object_Interface* root = bottom->Object_cycle;
        x->methods = point_MethodTable;

        x->Point_bottom = (struct Point_Impl*) x;
        bottom->Object_cycle = (Object_Interface*) x;
        x->Object_cycle = root;

        return ( Point_Impl* ) x;
}

Point_Impl* Point_implement(Object_Interface* object_arg) {

        //allocate memory for new segment, set segment type
        Point_Impl* x = (Point_Impl*) malloc(sizeof(Point_Impl));
        Object_Interface* bottom = findBottom(object_arg, point_MethodTable);
        Object_Interface* root = bottom->Object_cycle;
        x->methods = point_MethodTable;


        Object_Interface* r = object_arg;
        // if found, update segment's cycle, bottom, and implements attribute
        if (r != root) {
                x->Object_cycle = r->Object_cycle;
                x->Point_bottom = ( (Point_Impl* ) r)->Point_bottom;
        }

        // if this is not the bottom of the chain,
```

```
        // update up pointer of next one in the extends chain
        do {
                r = r->Object_cycle;
        }
        while ((r != object_arg) && ((( (Point_Methods*) (r->methods))->extends) ==
              point_MethodTable) );
        if (r != object_arg) {
                x->Point_bottom = ( (Point_Impl*)r)->Point_bottom;
        };

        // call convert routine with r and x has parameters ,
        // then finally dispose r
        free(r);
        return ( Point_Impl* ) x;
}

// create new object of type Point
// allocate root object segment, call extend to add Point segment
Point_Impl* Point_new() {
        Object_Interface* o = Object_new();
        return (Point_Impl*) Point_extend(o);
}

int Point_has(Object_Interface* object_arg) {
        Object_Interface* r = object_arg;
        do {
        // inspect each object segment type
        do {
                if (
                        p->methods==arraypoint_MethodTable
                )
                return true;

                r = r->Object_cycle;
        } while (r != object_arg);
        return false;
}


// structure holds pointers to methods for class MultiPoint
// along with type of class that it extends, implements or inherits
// also used identify type of segment
typedef struct MultiPoint_Methods {
        void* extends;
        void* implements;
        void* inherits;
        void (*construct)(void*,int sz2);
        void (*setMultiplier)(void*,int sm);
        void (*move)(void*, int y0 , int x0);
} MultiPoint_Methods;
```

```
typedef struct MultiPoint_Impl {
    MultiPoint_Methods* methods;
    Object_Interface* Object_cycle;  // *possibly* a pointer to a needed object

    Point_Impl* Point_bottom;
    Point_Impl* Point_up;
    struct MultiPoint_Impl* MultiPoint_bottom;   // pointer to latest mixed methods

    int m;
} MultiPoint_Impl;

void MultiPoint_construct ( MultiPoint_Impl* self,int sz2){
    self->m = sz2;
}
void MultiPoint_setMultiplier ( MultiPoint_Impl* self,int sm){
    self->m = sm;
}
void MultiPoint_move ( MultiPoint_Impl* self, int y0 , int x0){
    int x1;
    int y1;
    x1 = x0*self->m;
    y1 = y0*self->m;
    (((Point_Methods*)(self->Point_up)->methods)->move(self->Point_up,x1 , y1));
}

void MultiPoint_init() {
    multipoint_MethodTable = malloc(sizeof(MultiPoint_Methods));
    ((MultiPoint_Methods*)multipoint_MethodTable)->construct = (void(*)(void *,int sz2))
        &MultiPoint_construct;
    ((MultiPoint_Methods*)multipoint_MethodTable)->setMultiplier = (void(*)(void *,int
        sm)) &MultiPoint_setMultiplier;
    ((MultiPoint_Methods*)multipoint_MethodTable)->move = (void(*)(void *, int y0 , int
        x0)) &MultiPoint_move;
}


void MultiPoint_bottomdateSelfPointers(Object_Interface* p, void* e) {
    if (

            p->methods==multipoint_MethodTable ||
            p->methods==arraypoint_MethodTable ||
            p->methods==point_MethodTable
      )
            ((MultiPoint_Impl*) p)->Point_bottom=e;
    Point_bottomdateSelfPointers(p,e);
}


MultiPoint_Impl* MultiPoint_extend(Object_Interface* object_arg) {
    MultiPoint_Impl* x = (MultiPoint_Impl*) malloc(sizeof(MultiPoint_Impl));
```

```
        Object_Interface* bottom = findBottom(object_arg, point_MethodTable);
        Object_Interface* root = bottom->Object_cycle;
        x->methods = multipoint_MethodTable;


        x->Point_bottom = (Point_Impl*) x;
        x->MultiPoint_bottom = (struct MultiPoint_Impl*) x;

        Object_Interface* r = root->Object_cycle;
        do {
                MultiPoint_bottomdateSelfPointers(r, x);

                if (0||
                r->methods==point_MethodTable||
                r->methods==multipoint_MethodTable||
                r->methods==arraypoint_MethodTable
                ) {
                        ( (MultiPoint_Impl*) x) -> Point_up = (Point_Impl*) r;
                }
                r = r->Object_cycle;
        } while (r != root);

        bottom->Object_cycle = (Object_Interface*) x;
        x->Object_cycle = root;

        return ( MultiPoint_Impl* ) x;
}


MultiPoint_Impl* MultiPoint_implement(Object_Interface* object_arg) {

        //allocate memory for new segment, set segment type
        MultiPoint_Impl* x = (MultiPoint_Impl*) malloc(sizeof(MultiPoint_Impl));
        Object_Interface* bottom = findBottom(object_arg, multipoint_MethodTable);
        Object_Interface* root = bottom->Object_cycle;
        x->methods = multipoint_MethodTable;


        Object_Interface* r = object_arg;
        // if found, update segment's cycle, bottom, and implements attribute
        if (r != root) {
                x->Object_cycle = r->Object_cycle;
                x->MultiPoint_bottom = ( (MultiPoint_Impl* ) r)->MultiPoint_bottom;
        }

        // if this is not the bottom of the chain,
        // update up pointer of next one in the extends chain
        do {
                r = r->Object_cycle;
        }
```

```
        while ((r != object_arg) && (( ( (MultiPoint_Methods*) (r->methods))->extends) ==
            multipoint_MethodTable) );
        if (r != object_arg) {
                x->MultiPoint_bottom = ( (MultiPoint_Impl*)r)->MultiPoint_bottom;
        };

        // call convert routine with r and x has parameters ,
        // then finally dispose r
        free(r);
        return ( MultiPoint_Impl* ) x;
}


int MultiPoint_has(Object_Interface* object_arg) {
        Object_Interface* r = object_arg;
        do {
                if (r->methods==multipoint_MethodTable) return true;
                r = r->Object_cycle;
        } while (r != object_arg);
        return false;
}


// structure holds pointers to methods for class ArrayPoint
// along with type of class that it extends, implements or inherits
// also used identify type of segment
typedef struct ArrayPoint_Methods {
        void* extends;
        void* implements;
        void* inherits;
        void (*construct)(void*, int y0 , int x0);
        void (*move)(void*, int y0 , int x0);
} ArrayPoint_Methods;


typedef struct ArrayPoint_Impl {
         ArrayPoint_Methods* methods;
         Object_Interface* Object_cycle;  // *possibly* a pointer to a needed object

         struct Point_Impl* Point_bottom;   // pointer to latest mixed methods


        int* ar;
} ArrayPoint_Impl;

void ArrayPoint_construct ( ArrayPoint_Impl* self, int y0 , int x0){
        self->ar = (void*)calloc(2,sizeof(void*));
        self->ar[1] = y0;
        self->ar[0] = x0;
}
void ArrayPoint_move ( ArrayPoint_Impl* self, int y0 , int x0){
```

```
        self->ar[0] = self->ar[0]+x0;
        self->ar[1] = self->ar[1]+y0;
}

void ArrayPoint_init() {
        arraypoint_MethodTable = malloc(sizeof(ArrayPoint_Methods));
        ((ArrayPoint_Methods*)arraypoint_MethodTable)->construct = (void(*)(void *, int y0 ,
            int x0)) &ArrayPoint_construct;
        ((ArrayPoint_Methods*)arraypoint_MethodTable)->move = (void(*)(void *, int y0 , int
            x0)) &ArrayPoint_move;
}


void ArrayPoint_bottomdateSelfPointers(Object_Interface* p, void* e) {
}


ArrayPoint_Impl* ArrayPoint_extend(Object_Interface* object_arg) {
        ArrayPoint_Impl* x = (ArrayPoint_Impl*) malloc(sizeof(ArrayPoint_Impl));
        Object_Interface* bottom = findBottom(object_arg, object_MethodTable);
        Object_Interface* root = bottom->Object_cycle;
        x->methods = arraypoint_MethodTable;


        x->Point_bottom = (struct Point_Impl*) x;
        bottom->Object_cycle = (Object_Interface*) x;
        x->Object_cycle = root;

        return ( ArrayPoint_Impl* ) x;
}


ArrayPoint_Impl* ArrayPoint_implement(Object_Interface* object_arg) {

        //allocate memory for new segment, set segment type
        ArrayPoint_Impl* x = (ArrayPoint_Impl*) malloc(sizeof(ArrayPoint_Impl));
        Object_Interface* bottom = findBottom(object_arg, arraypoint_MethodTable);
        Object_Interface* root = bottom->Object_cycle;
        x->methods = arraypoint_MethodTable;


        Object_Interface* r = object_arg;
        // if found, update segment's cycle, bottom, and implements attribute
        if (r != root) {
                x->Object_cycle = r->Object_cycle;
                x->Point_bottom = ( (Point_Impl* ) r)->Point_bottom;
        }

        // if this is not the bottom of the chain,
        // update up pointer of next one in the extends chain
        do {
```

```
                r = r->Object_cycle;
        }
        while ((r != object_arg) && ((  ( (ArrayPoint_Methods*) (r->methods))->extends) ==
              point_MethodTable) );
        if (r != object_arg) {
                x->Point_bottom = ( (Point_Impl*)r)->Point_bottom;
        };

        // call convert routine with r and x has parameters ,
        // then finally dispose r
        free(r);
        return ( ArrayPoint_Impl* ) x;
}

// create new object of type ArrayPoint
// allocate root object segment, call extend to add ArrayPoint segment
ArrayPoint_Impl* ArrayPoint_new() {
        Object_Interface* o = Object_new();
        return (ArrayPoint_Impl*) ArrayPoint_extend(o);
}


int ArrayPoint_has(Object_Interface* object_arg) {
        Object_Interface* r = object_arg;
        do {
                if (r->methods==arraypoint_MethodTable) return true;
                r = r->Object_cycle;
        } while (r != object_arg);
        return false;
}


// structure holds pointers to methods for class Member
// along with type of class that it extends, implements or inherits
// also used identify type of segment
typedef struct Member_Methods {
        void* extends;
        void* implements;
        void* inherits;
        void (*setg)(void*,short x);
} Member_Methods;


typedef struct Member_Impl {
         Member_Methods* methods;
         Object_Interface* Object_cycle;  // *possibly* a pointer to a needed object

         struct Member_Impl* Member_bottom;   // pointer to latest mixed methods


        short g;
```

```
} Member_Impl;

void Member_setg ( Member_Impl* self,short x){
        self->g = x;
}

void Member_init() {
        member_MethodTable = malloc(sizeof(Member_Methods));
        ((Member_Methods*)member_MethodTable)->setg = (void(*)(void *,short x)) &
            Member_setg;
}


void Member_bottomdateSelfPointers(Object_Interface* p, void* e) {
}


Member_Impl* Member_extend(Object_Interface* object_arg) {
        Member_Impl* x = (Member_Impl*) malloc(sizeof(Member_Impl));
        Object_Interface* bottom = findBottom(object_arg, object_MethodTable);
        Object_Interface* root = bottom->Object_cycle;
        x->methods = member_MethodTable;


        x->Member_bottom = (struct Member_Impl*) x;
        bottom->Object_cycle = (Object_Interface*) x;
        x->Object_cycle = root;

        return ( Member_Impl* ) x;
}


Member_Impl* Member_implement(Object_Interface* object_arg) {

        //allocate memory for new segment, set segment type
        Member_Impl* x = (Member_Impl*) malloc(sizeof(Member_Impl));
        Object_Interface* bottom = findBottom(object_arg, member_MethodTable);
        Object_Interface* root = bottom->Object_cycle;
        x->methods = member_MethodTable;


        Object_Interface* r = object_arg;
        // if found, update segment's cycle, bottom, and implements attribute
        if (r != root) {
                x->Object_cycle = r->Object_cycle;
                x->Member_bottom = ( (Member_Impl* ) r)->Member_bottom;
        }

        // if this is not the bottom of the chain,
        // update up pointer of next one in the extends chain
        do {
```

```
            r = r->Object_cycle;
        }
        while ((r != object_arg) && (( ( (Member_Methods*) (r->methods))->extends) ==
            member_MethodTable) );
        if (r != object_arg) {
            x->Member_bottom = ( (Member_Impl*)r)->Member_bottom;
        };

        // call convert routine with r and x has parameters ,
        // then finally dispose r
        free(r);
        return ( Member_Impl* ) x;
}

// create new object of type Member
// allocate root object segment, call extend to add Member segment
Member_Impl* Member_new() {
        Object_Interface* o = Object_new();
        return (Member_Impl*) Member_extend(o);
}


int Member_has(Object_Interface* object_arg) {
        Object_Interface* r = object_arg;
        do {
            if (r->methods==member_MethodTable) return true;
            r = r->Object_cycle;
        } while (r != object_arg);
        return false;
}

int main() {
        Object_init();
        Point_init();
        MultiPoint_init();
        ArrayPoint_init();
        Member_init();
        Point_Impl* p;
        Member_Impl* m;
        p = Point_new();
        Member_extend((Object_Interface *)p);
        MultiPoint_extend((Object_Interface *)p);
        ArrayPoint_implement((Object_Interface *)p);
        (((Point_Methods*)(p->Point_bottom)->methods)->move(p->Point_bottom,5 , 7));
        m = (Member_Impl *) castObject( (void *) p,member_MethodTable);
        (((Member_Methods*)(m->Member_bottom)->methods)->setg(m->Member_bottom,1)
            );
}
```

Appendix C

Code Used to Gather Timing Results

Enclosed is the source code used to obtain the timing results. The code is for a method call length of 2, but can easily be extrapolated to the lengths of 4 and 8 shown as shown in the work.

Listing C.1: C++ Sample

```
#include <iostream>
#include <time.h>

using namespace std;

class C0 {
public:
  int c0;
  C0() {c0 = 1;}
  virtual void m(int d) {c0 = (c0 + d) % 99;}
};

class C1: public C0 {
public:
  int c1;
  C1() {c1 = 1;}
  virtual void m(int d) {C0::m(d); c1 = (c1 + d) % 99;}
};

class C2: public C1 {
public:
  int c2;
  C2() {c2 = 1;}
  virtual void m(int d) {C1::m(d); c2 = (c2 + d) % 99;}
};

int main(){
  clock_t start, end;
  double cpu_time_used;

  start = clock();
  C2* x = new C2();
  for (int i = 0; i < 10000000; i++) {
    x->m(i);
  }
  end = clock();
  cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
  cout << cpu_time_used;
}
```

Listing C.2: Java Sample

```
class C0 {
  int c0;
  C0() {c0 = 1;}
  void m(int d) {c0 = (c0 + d) % 99;}
};

class C1 extends C0 {
  int c1;
  C1() {super(); c1 = 1;}
  void m(int d) {super.m(d); c1 = (c1 + d) % 99;}
};

class C2 extends C1 {
  int c2;
  C2() {super(); c2 = 1;}
  void m(int d) {super.m(d); c2 = (c2 + d) % 99;}
};

class chaininheritance {
  public static void main(String[] args) {
    final long start = System.currentTimeMillis();

    C2 x = new C2();
    for (int i = 0; i < 10000000; i++) {
      x.m(i);
    }

    final long end = System.currentTimeMillis();
    System.out.println((end - start) + " milliseconds");
  }
}
```

Listing C.3: Python Inheritance Sample

```
class C0:
    def __init__(self):
        self.c0 = 1
    def m(self, d):
        self.c0 = (self.c0 + d) % 99

class C1(C0):
    def __init__(self):
        C0.__init__(self)
        self.c1 = 1
    def m(self, d):
        C0.m(self, d)
        self.c1 = (self.c1 + d) % 99

class C2(C1):
    def __init__(self):
        C1.__init__(self)
        self.c2 = 1
    def m(self, d):
        C1.m(self, d)
        self.c2 = (self.c2 + d) % 99

def main():
    x = C2()
    for i in range(10000000):
        x.m(i)

import timeit
print(timeit.timeit("main()", setup="from __main__ import main", number=1))
```

Listing C.4: Python Mixin Sample

```
from types import MethodType

class C0:
    def __init__(self):
        self.c0 = 1
    def m(self, d):
        self.c0 = (self.c0 + d) % 99

def C1_m(self, d):
    self.C1_m0(d)
    self.c1 = (self.c1 + d) % 99

def C1_extend(x):
    x.c1 = 1
    x.C1_m0 = x.m
    x.m = MethodType(C1_m, x)

def C2_m(self, d):
    self.C2_m0(d)
    self.c2 = (self.c2 + d) % 99

def C2_extend(x):
    x.c2 = 1
    x.C2_m0 = x.m
    x.m = MethodType(C2_m, x)

def main():
    global x
    x = C0()
    C1_extend(x)
    C2_extend(x)
    for i in range(10000000):
        x.m(i)

import timeit
print(timeit.timeit("main()", setup="from __main__ import main", number=1))
```

Appendix D

Proofs

This section outlines proofs of the lemmata used in Section 4.

Lemma 1 (Preservation Over Structure)**.** *For statements* S, T *and predicates* b, p, q*:*

(a) $\textbf{skip}$ preserves p
(b) $(b \Rightarrow S$ preserves $p) \;\equiv\; \textbf{assume}\, b\, ; S$ preserves p
(c) $(b \Rightarrow S$ preserves $p) \;\equiv\; \textbf{assert}\, b\, ; S$ preserves p
(d) $(p \Rightarrow p[x \backslash e]) \;\equiv\; x := e$ preserves p
(e) $(p \Rightarrow (\forall x \in e \bullet p)) \;\equiv\; x :\in e$ preserves p
(f) $(S$ preserves $p) \wedge (T$ preserves $p) \;\Rightarrow\; (S \,[\!]\, T$ preserves $p)$
(g) $(S$ preserves $p) \wedge (T$ preserves $p) \;\Rightarrow\; (S\, ; T$ preserves $p)$
(h) $(T[x \backslash e]$ preserves $p) \;\equiv\; (\textbf{const}\, x = e\, \textbf{in}\, T$ preserves $p)$
(i) $(\forall \overline{x} : \overline{X} \bullet b \Rightarrow T$ preserves $p) \;\Rightarrow\; (\textbf{var}\, \overline{x} : \overline{X} \mid b\, \textbf{in}\, T$ preserves $p)$

Proof. For (a), we observe that by definition, **skip** preserves any predicate. For (b) we have:

$\textbf{assume}\, b\, ; S$ preserves p
$\equiv\; p \wedge (b \Rightarrow wp(S, true)) \Rightarrow (b \Rightarrow wp(S, p))$ — def. of preserves, wp of ;, **assume**
$\equiv\; b \Rightarrow S$ preserves p — logic, def. of preserves

For (c) we have:

$\textbf{assert}\, b\, ; S$ preserves p
$\equiv\; p \wedge b \wedge wp(S, true) \Rightarrow b \wedge wp(S, p)$ — def. of preserves, wp of ;, **assert**
$\equiv\; b \Rightarrow S$ preserves p — logic, def. of preserves

For (d) we have:

$$\begin{array}{llr} & x := e \;\text{ preserves }\; p & \\ \equiv & p \wedge wp(x := e, true) \Rightarrow wp(x := e, p) & \text{def. of preserves, } wp \text{ of } := \\ \equiv & p \Rightarrow p[x \backslash e] & wp \text{ of } := \end{array}$$

For (e) we have:

$$\begin{array}{llr} & x :\in e \;\text{ preserves }\; p & \\ \equiv & p \wedge wp(x :\in e, true) \Rightarrow wp(x :\in e, p) & \text{def. of preserves} \\ \equiv & p \wedge (\forall x \in e \bullet true) \Rightarrow (\forall x \in e \bullet p) & wp \text{ of } :\in \\ \equiv & p \Rightarrow (\forall x \in e \bullet p) & \text{logic} \end{array}$$

For (f) we have:

$$\begin{array}{llr} & S \,[\!]\, T \text{ preserves } p & \\ \equiv & p \wedge wp(S, true) \wedge wp(T, true) \Rightarrow wp(S, p) \wedge wp(T, p) & \text{definitions} \\ \Leftarrow & (p \wedge wp(S, true) \Rightarrow wp(S, p)) \wedge (p \wedge wp(T, true) \Rightarrow wp(T, p)) & \text{logic} \\ \equiv & (S \text{ preserves } p) \wedge (T \text{ preserves } p) & \text{definitions} \end{array}$$

For (g), assuming S preserves p and T preserves p, we have:

$$\begin{array}{llr} & S \,;\, T \text{ preserves } p & \\ \equiv & p \wedge wp(S, wp(T, true)) \Rightarrow wp(S, wp(T, p)) & \text{definitions} \\ \Leftarrow & p \wedge wp(S, wp(T, true)) \Rightarrow wp(S, p \wedge wp(T, true)) & \text{as } T \text{ preserves } p \\ \equiv & p \wedge wp(S, wp(T, true)) \Rightarrow wp(S, p) \wedge wp(S, wp(T, true)) & \text{conjunctivity} \\ \equiv & p \wedge wp(S, wp(T, true)) \Rightarrow wp(S, p) & \text{logic} \\ \Leftarrow & p \wedge wp(S, true) \Rightarrow wp(S, p) & \text{monotonicity} \\ \Leftarrow & true & \text{as } S \text{ preserves } p \end{array}$$

For (h), we note that by definition of equality of statements, $\mathbf{const}\, x = e \,\mathbf{in}\, T = T[x \backslash e]$; hence (h) follows immediately. For (i) we have:

$$\begin{array}{llr} & \mathbf{var}\, \overline{x} : \overline{X} \mid b \,\mathbf{in}\, T \text{ preserves } p & \\ \equiv & p \wedge wp(\mathbf{var}\, \overline{x} : \overline{X} \mid b \,\mathbf{in}\, T, true) \Rightarrow wp(\mathbf{var}\, \overline{x} : \overline{X} \mid b \,\mathbf{in}\, T, p) & \text{definition} \\ \equiv & p \Rightarrow ((\forall \overline{x} : \overline{X} \bullet b \Rightarrow wp(T, true)) \Rightarrow (\forall \overline{x} : \overline{X} \bullet b \Rightarrow wp(T, p))) & \text{logic, definition} \\ \Leftarrow & p \Rightarrow (\forall \overline{x} : \overline{X} \bullet (b \Rightarrow wp(T, true)) \Rightarrow (b \Rightarrow wp(T, p))) & \text{logic} \\ \equiv & (\forall \overline{x} : \overline{X} \bullet p \Rightarrow (b \Rightarrow (wp(T, true) \Rightarrow wp(T, p)))) & \text{logic} \\ \equiv & (\forall \overline{x} : \overline{X} \bullet b \Rightarrow T \text{ preserves } p) & \text{logic, definition} \end{array}$$

$\square$

Lemma 2 (Piecewise Preservation)**.** *For statements S, T and predicates p, q:*

(a) $(S \text{ preserves } p) \wedge (S \text{ preserves } q) \Rightarrow (S \text{ preserves } p \wedge q)$
(b) $(q \Rightarrow S \text{ preserves } p) \wedge (p \Rightarrow S \text{ preserves } q) \Rightarrow (S \text{ preserves } p \wedge q)$

Proof. Implication (a) follows from (b), as the antecedent of (a) is weaker. For (b), we continue:

$$
\begin{array}{ll}
 & (S \text{ preserves } p) \wedge (p \Rightarrow S \text{ preserves } q) \\
\equiv & (q \wedge p \wedge wp(S, true) \Rightarrow wp(S,p)) \wedge (p \wedge q \wedge wp(S, true) \Rightarrow wp(S,q)) \\
 & \hfill \text{def. of preserves, logic} \\
\Rightarrow & p \wedge q \wedge wp(S, true) \Rightarrow wp(S,p) \wedge wp(S,q) \hfill \text{logic, conjunctivity} \\
\equiv & S \text{ preserves } p \wedge q \hfill \text{def. of preserves}
\end{array}
$$

□

Lemma 4 (Statement Updating a Function)**.** *For statement S, variable f of function type, and boolean function p, assume that S modifies f only at d, that S does not modify s, and that S establishes $p(f(d))$:*

$$(\forall i \in s - \{d\} \bullet p(f(i))) \wedge wp(S, true) \;\Rightarrow\; wp(S, \forall i \in s \bullet p(f(i)))$$

Proof.

$$
\begin{array}{ll}
 & wp(S, \forall i \in s \bullet p(f(i))) \\
\equiv & wp(S, (\forall i \in s - \{d\} \bullet p(f(i))) \wedge p(f(d))) \hfill \text{case analysis} \\
\Leftarrow & wp(S, \forall i \in s - \{d\} \bullet p(f(i))) \wedge wp(S, true) \\
 & \hfill \text{conjunctivity, } S \text{ establishes } p(f(r)) \\
\Leftarrow & (\forall i \in s - \{d\} \bullet p(f(i))) \wedge wp(S, true) \hfill \text{as } S \text{ modifies } f \text{ only at } d
\end{array}
$$

□

Lemma 5 (Refinement Laws)**.** *Let S be a statement over variables that include x, the abstract variables, let T be a statement over variables that include y, the concrete variables, let $R(x, y)$ relate x and y, and let z be among the global variables:*

$$
\begin{array}{rcll}
(R(x,y) \Rightarrow R(e,f)) & \equiv & x := e \sqsubseteq_R y := f & \text{(a)} \\
(R(x,y) \Rightarrow e = f) & \equiv & z := e \sqsubseteq_R z := f & \text{(b)} \\
(R(x,y) \wedge x \in e \Rightarrow (\forall y \in f \bullet R(x,y)) & \equiv & x :\in e \sqsubseteq_R y :\in f & \text{(c)} \\
(R(x,y) \Rightarrow e \supseteq f) & \equiv & x :\in e \sqsubseteq_R y :\in f & \text{(d)} \\
(S_1 \sqsubseteq_R T_1) \wedge (S_2 \sqsubseteq_R T_2) & \Rightarrow & S_1 \mathbin{[\!]} S_2 \sqsubseteq_R T_1 \mathbin{[\!]} T_2 & \text{(e)} \\
(S_1 \sqsubseteq_R T_1) \wedge (S_2 \sqsubseteq_R T_2) & \Rightarrow & S_1 \,;\, S_2 \sqsubseteq_R T_1 \,;\, T_2 & \text{(f)}
\end{array}
$$

Proof. For (a):

$$\begin{array}{lll} & x := e \sqsubseteq_R y := f & \\ \equiv & \forall q \bullet R(x,y) \wedge wp(x := e, q) \Rightarrow wp(y := f, \exists y \bullet R(x,y) \wedge q) & \text{def. of } \sqsubseteq_R \\ \equiv & \forall q \bullet R(x,y) \wedge q[x \backslash e] \Rightarrow (\exists x \bullet R(x,f) \wedge q) & wp \text{ of } :=, y \text{ not in } q \\ \equiv & \forall q \bullet R(x,y) \wedge q[x \backslash e] \Rightarrow R(e,f) \wedge q[x \backslash e] & \text{logic} \\ \equiv & R(x,y) \Rightarrow R(e,f) & \text{logic} \end{array}$$

For (b):

$$\begin{array}{lll} & z := e \sqsubseteq_R z := f & \\ \equiv & \forall q \bullet R(x,y) \wedge wp(z := e, q) \Rightarrow wp(z := f, \exists y \bullet R(x,y) \wedge q) & \text{def. of } \sqsubseteq_R \\ \equiv & \forall q \bullet R(x,y) \wedge q[z \backslash e] \Rightarrow (\exists x \bullet R(x,y) \wedge q[z \backslash f]) & wp \text{ of } := \\ \equiv & R(x,y) \Rightarrow e = f & \text{mutual implication} \end{array}$$

For (c):

$$\begin{array}{lll} & x :\in e \sqsubseteq_R y :\in f & \\ \equiv & \forall q \bullet R(x,y) \wedge wp(x :\in e, q) \Rightarrow wp(y :\in f, \exists x \bullet R(x,y) \wedge q) & \text{def. of } \sqsubseteq_R \\ \equiv & \forall q \bullet R(x,y) \wedge (\forall x \in e \bullet q) \Rightarrow (\forall y \in f \bullet \exists x \bullet R(x,y) \wedge q) & \text{def. of } :\in \\ \equiv & R(x,y) \wedge x \in e \Rightarrow (\forall y \in f \bullet R(x,y)) & \text{mutual implication} \end{array}$$

For (d):

$$\begin{array}{lll} & z :\in e \sqsubseteq_R z :\in f & \\ \equiv & \forall q \bullet R(x,y) \wedge wp(z :\in e, q) \Rightarrow wp(z :\in f, \exists x \bullet R(x,y) \wedge q) & \text{def. of } \sqsubseteq_R \\ \equiv & \forall q \bullet R(x,y) \wedge (\forall z \in e \bullet q) \Rightarrow (\forall z \in f \bullet \exists x \bullet R(x,y) \wedge q) & \text{def. of } :\in \\ \equiv & R(x,y) \Rightarrow e \supseteq f & \text{mutual implication} \end{array}$$

For (e):

$$\begin{array}{lll} & S_1 \,[\!]\, S_2 \sqsubseteq_R T_1 \,[\!]\, T_2 & \\ \equiv & \forall q \bullet R(x,y) \wedge wp(S_1, q) \wedge wp(S_2, q) \Rightarrow & \\ & \quad wp(T_1, \exists x \bullet R(x,y) \wedge q) \wedge wp(T_2, \exists x \bullet R(x,y) \wedge q) & \text{def. of } \sqsubseteq_R, [\!] \\ \Leftarrow & (\forall q \bullet R(x,y) \wedge wp(S_1, q) \Rightarrow wp(T_1, \exists x \bullet R(x,y) \wedge q)) \wedge & \\ & (\forall q \bullet R(x,y) \wedge wp(S_2, q) \Rightarrow wp(T_2, \exists x \bullet R(x,y) \wedge q)) & \text{logic} \\ \equiv & (S_1 \sqsubseteq_R T_1) \wedge (S_2 \sqsubseteq_R T_2) & \text{def. of } \sqsubseteq_R \end{array}$$

For (f), we start with:

$$\begin{array}{lll} & S_1 \,;\, S_2 \sqsubseteq_R T_1 \,;\, T_2 & \\ \equiv & \forall q \bullet R(x,y) \wedge wp(S_1, wp(S_2, q)) \Rightarrow & \\ & \quad wp(T_1, wp(T_2, \exists x \bullet R(x,y) \wedge q)) & \text{def. of } \sqsubseteq_R, ; \end{array}$$

For the hypotheses, we observe that x does not occur in $wp(T_2, \exists x \bullet R(x,y) \wedge q)$, as T by assumption is not over x. This allows to move the implicit universal quantification

of x as an existential quantification into the antecedent and hence we have:

$$\begin{array}{lcl} S_1 \sqsubseteq_R T_1 & \equiv & \forall q \bullet R(x,y) \wedge wp(S_1, q) \Rightarrow wp(T_1, \exists x \bullet R(x,y) \wedge q) \\ S_2 \sqsubseteq_R T_2 & \equiv & \forall q \bullet (\exists x \bullet R(x,y) \wedge wp(S_2, q)) \Rightarrow wp(T_2, \exists x \bullet R(x,y) \wedge q) \end{array}$$

We continue for any q:

$$\begin{array}{lll} & wp(T_1, wp(T_2, \exists x \bullet R(x,y) \wedge q)) & \\ \Leftarrow & wp(T_1, \exists x \bullet R(x,y) \wedge wp(S_2, q)) & \text{as } S_2 \sqsubseteq_R T_2 \\ \Leftarrow & R(x,y) \wedge wp(S_1, wp(S_2, q)) & \text{as } S_1 \sqsubseteq_R T_1 \end{array}$$

Hence $S_1 \,;\, S_2 \sqsubseteq_R T_1 \,;\, T_2$ follows. □