INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI
800-521-0500
Programming Languages for Reusable Software Components

by

Matthew Flatt

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Matthias Felleisen
Professor of Computer Science

Robert S. Cartwright, Jr.
Professor of Computer Science

Keith D. Cooper
Associate Professor of Computer Science

David M. Lane
Associate Professor of Psychology and Statistics

Houston, Texas
June, 1999
Programming Languages for Reusable Software Components

Matthew Flatt

Abstract

Programming languages offer a variety of constructs to support code reuse. For example, functional languages provide function constructs for encapsulating expressions to be used in multiple contexts. Similarly, object-oriented languages provide class (or class-like) constructs for encapsulating sets of definitions that are easily adapted for new programs. Despite the variety and abundance of such programming constructs, however, existing languages are ill-equipped to support component programming with reusable software components.

Component programming differs from other forms of reuse in its emphasis on the independent development and deployment of software components. In its ideal form, component programming means building programs from off-the-shelf components that are supplied by a software-components industry. This model suggests a strict separation between the producer and consumer of a component. The separation, in turn, implies separate compilation for components, allowing a producer to test and distribute compiled components rather than proprietary source code. Since the consumer cannot modify a compiled software component, each component must be defined and compiled in a way that gives the consumer flexibility in linking components together.

This dissertation shows how a language for component programming can support both separate compilation and flexible linking. To that end, it expounds the principle of external connections:

A language should separate component definitions from component connections.

Neither conventional module constructs nor conventional object-oriented constructs follow the principle of external connections, which explains why neither provides an
effective language for component programming. We describe new language constructs for modules and classes—called *units* and *mixins*, respectively—that enable component programming in each domain.

The unit and mixin constructs modeled in this dissertation are based on constructs that we implemented for the MzScheme programming language, a dialect of the dynamically-typed language Scheme. To demonstrate that units and mixins work equally well for statically-typed languages, such as ML or Java, we provide typed models of the constructs as well as untyped models, and we formally prove the soundness of the typed models.
Acknowledgements

And you may ask yourself: Well, how did I get here?
—Talking Heads. “Once in a Lifetime”

I started life with wonderful parents who encouraged my academic and intellectual pursuits. I finished this dissertation with an exceptional advisor and many colleagues who enabled and encouraged my research.

Matthias Felleisen, my advisor, shaped this dissertation by recognizing the potential in bits and pieces of ideas. He taught me how to bring those pieces together, and how to fill in the gaps to form a coherent story. In doing so, Matthias also shaped me, bringing together bits and pieces of talent, then filling in the gaps to form a coherent researcher and teacher.

Corky Cartwright, my co-advisor, provided an inexhaustible fountain of knowledge and insightful criticism as my research developed. His approval of this work in its final form gives me more confidence in the results than one hundred program committees.

Shriram Krishnamurthi, Robby Findler, and Cormac Flanagan defined the cooperative, give-and-take environment from which this dissertation emerged. Together, we formulated and refined the notions of components, units, and mixins, and the significance of this dissertation depends crucially on the larger context defined by their own work.

Paul Graunke, John Clements, Paul Steckler, and Ian Barland provided invaluable feedback and support as part of the Programming Languages Team. Many other people provided helpful comments and criticisms along the way, especially Kent Dybvig, Dan Friedman, Bob Harper, Peter Lee, Didier Rémy, Scott Smith, Michael Sperber, and the anonymous reviewers for POPL’98, PLDI’98, and ICFP’98. I owe a further debt to the users of DrScheme, MzScheme, and MrEd for testing the implementation of many ideas.

This work was made possible by the generous support of Rice University, the National Science Foundation (with research grants and a graduate fellowship), the Texas Advanced Technology Program, and a Lodieska Stockbridge Vaughan Fellowship.
Dedication

for Wenyuan
# Contents

Abstract  
Acknowledgments  

1 Reusable Software Components  
1.1 Reuse without Source Code ........................................... 2  
1.2 Language Support for Reuse ........................................... 3  
1.2.1 Modules .............................................................. 3  
1.2.2 Classes .............................................................. 6  
1.3 Dissertation Overview ................................................. 7  

2 The Extensibility Problem  
2.1 Extensible Programming with Classes .................................. 9  
2.1.1 Shape Datatype ....................................................... 11  
2.1.2 Variant Extension .................................................... 12  
2.1.3 Operation Extension ................................................. 13  
2.2 Better Reuse through Units and Mixins ................................ 15  
2.2.1 Unitizing the Basic Shapes ......................................... 16  
2.2.2 Linking the Shape and Client Units ................................. 17  
2.2.3 Invoking Unit Programs .............................................. 19  
2.2.4 New Units for a New Variant ....................................... 21  
2.2.5 New Units and Mixins for a New Operation ....................... 21  
2.2.6 Units and Mixins at Work .......................................... 23  
2.3 Summary ................................................................. 25  

3 Units  
3.1 Existing Module Languages and Units .................................. 27  
3.2 Programming with Units ............................................... 28  
3.2.1 Defining Units ....................................................... 29
6 Related Work on Software Components

7 Limitations and Future Work
  7.1 Combining Typed Units and Mixins .................. 105
  7.2 Units and Mixins for Other Languages .............. 105

A MzScheme Class and Interface Syntax ................. 107
  A.1 Classes .......................................... 107
  A.2 Interfaces ........................................ 108
  A.3 Derived Classes ................................... 109

B UNITc Proofs ....................................... 111
  B.1 Proof of Subject Reduction ......................... 111
  B.2 Proof of Progress .................................. 118
  B.3 Supporting Lemmata ................................ 122

C CLASSICJAVA Proofs ................................ 127
  C.1 Proof of Subject Reduction ......................... 127
  C.2 Proof of Progress .................................. 129
  C.3 Supporting Lemmata ................................ 130

D MIXEDJAVA Proofs .................................. 133
  D.1 Proof of Subject Reduction ......................... 133
  D.2 Proof of Progress .................................. 136
  D.3 Supporting Lemmata ................................ 138

Bibliography ......................................... 141
Chapter 1

Reusable Software Components

To implement a reusable program component, a programmer must:

- define a component that is large enough to make its reuse worthwhile, but small enough to be widely applicable;

- design the component to export abstractions rather than implementation details, so that the component can be replaced by an improved version with a different implementation; and

- anticipate likely extensions to the component’s functionality, parameterizing the component’s behavior accordingly.

Programming language constructs can help programmers design reusable software components by making the natural expression of program parts conducive to reuse. Examples of helpful language constructs include higher-order functions, programmer-defined data types, and classes. Nevertheless, existing languages fail to support reuse in many ways. For example, existing module languages help a programmer to decompose a program into reusable pieces, but they typically force unnecessary context dependencies on the module that limit its reuse. Similarly, class-based object-oriented languages encourage the reuse of class definitions through extension, but they do not permit the reuse of a class extension in disjoint parts of a class hierarchy.

To explain why current languages fail, we must first define reuse more precisely. Section 1.1 explains that reuse for this dissertation means black-box reuse. Section 1.2 describes our thesis—a prescription for language designers who wish to support black-box reuse—and two novel language constructs that illustrate the thesis. Section 1.3 provides an overview of the rest of the dissertation.
1.1 Reuse without Source Code

At the 1968 NATO conference, McIlroy [59] described component reuse in its ideal form, a world where programmers construct software using off-the-shelf components that are supplied by a software-components industry. This off-the-shelf approach should also work within a development team, where each part of the team supplies components to other parts of the team. Indeed, this approach can even help an individual programmer to break up a large program into manageable pieces for separate development.

Component-based reuse depends crucially on separately-compiled components. In the case of a software-components industry, separate compilation permits a vendor to distribute components without exposing proprietary source code. In the case of a single team or single programmer, separate compilation allows type-checking and testing for individual components, and it allows a rapid modify-and-test cycle for the entire program.

A software component that is distributed in compiled form cannot be modified by a client (i.e., the component’s user). This restriction ensures that a vendor can create new versions of a component to replace the original one, either to improve the component or to correct a problem, without forcing clients to change their code. The separation also ensures that individual components can be tested independently via stubs and drivers that verify the component’s interface. Finally, because compilation applies only for components with a well-defined meaning, separate compilation enforces a semantic modularity for components; a programmer can therefore combine components by reasoning about their meanings rather than their implementations.

Since this dissertation concerns only those forms of reuse that conform to the model of a software-components industry, we define reuse to mean black-box reuse: reuse of a component without inspecting or modifying its source code. Black-box reuse requires programming language support. For example, while C++ templates allow programmers to recycle fragments of program syntax, they cannot implement components because templates cannot be compiled separately. In contrast, closed functor modules in ML\(^1\) can be compiled separately, so functors can implement components.

\(^1\)Throughout this dissertation, we use “ML” as an abbreviation for “SML or CAML”.
Separate compilation sometimes implies a loss of performance as compared to whole-program compilation, because it enforces abstraction boundaries and defeats optimization techniques such as inlining. In our view, however, the lack of reusable software components is a far more critical problem than a lack of high-performance software. We therefore investigate language constructs to maximize reuse, and we consider performance as a secondary (though important) constraint on the design space.

1.2 Language Support for Reuse

Separate compilation is a key prerequisite for constructing reusable software components, but component construction is only half of the story. Equally important to reuse is a language’s facility for connecting components to form a complete program.

This dissertation explores the principle of external connections:

A language should separate component definitions from component connections.

The dissertation demonstrates the application of this principle to two important areas: modules and classes. In the case of modules, the components are modules and the connections link modules. For classes, the components are class extensions and the connections derive concrete classes.

Figure 1.1 illustrates the difference between a language with external connections and a language where definitions and connections are specified together. The left-hand figure shows a component \( R \) that is defined separately from its connections; hence, a programmer can use \( R \) with either \( A \) or \( B \). In contrast, the right-hand figure shows a component \( R \) whose definition explicitly connects it to the component \( A \): in this case, using \( R \) with \( B \) requires modifying \( R \)'s definition. In the following subsections, we argue that most existing module and object constructs correspond to the right-hand figure. This dissertation describes alternatives that correspond to the left-hand figure.

1.2.1 Modules

Many languages (e.g., Ada 95 [38], Modula-3 [32], Haskell [37], and Java [31]) provide modules via packages. A package system delineates the boundaries of each mod-
Language with external connections

Component definitions:

Explicitly connect $R$ to $A$:

Explicitly connect $R$ to $B$:

Language w/o external connections

Component definitions:

$R$ is hard-wired to $A$:

Modify $R$ to work with $B$:

$R'$ is hard-wired to $B$:

Figure 1.1: Programming with and without external connections

ule and permits the separate compilation of packages. Package linking, however, is determined by import specifications within each package definition. Thus, packages correspond to the right-hand side of Figure 1.1; to use a package in different contexts with different import sources, a programmer must modify the package definition.

For example, the definition of a dictionary package in Java might include the following:

```java
import com.supersoft.splaytree.*;
```

This import specification hard-wires dictionary to the splay tree implementation from SuperSoft. A programmer using dictionary might want instead to use a compatible splay tree implementation from UltraSoft. Even if SuperSoft and UltraSoft export the same classes and methods for splay trees, the programmer must modify the definition of dictionary to use UltraSoft's implementation, replacing supersoft with ultrasoft.
A Java programmer might hack around the problem by defining a class loader to remap `com.supersoft.splaytree` to `com.ultrasoft.splaytree`. This name-remapping strategy, however, fails to scale to the general case. Suppose that the programmer wants to use both `dictionary` and `thesaurus`—which also imports `com.supersoft.splaytree`—and the programmer wants to preserve the SuperSoft import for `thesaurus` while switching `dictionary`'s import to UltraSoft. Because a class loader cannot map a single package path to multiple packages, the programmer is forced to modify either `dictionary` or `thesaurus`.

The underlying problem is that a package declares both the `shape` and `source` of its imports. The `shape` part of the declaration specifies that the imports include certain classes and methods; shape information is necessary for separate compilation. The `source` part of the declaration specifies that the `com.supersoft.splaytree` package provides those classes. Thus, the failure of `dictionary` and `thesaurus` is a failure to obey the principle of external connections, which indicates that the source of a module's imports should be specified external to the module. If the source of a package's imports were specified external to its definition, then a programmer could use `dictionary` and `thesaurus` in the same program, specifying different sources for each package's imports without modifying either package.

ML's module system follows the principle of external connections. An ML functor abstracts over a collection of definitions in the same way that a procedure abstracts over an expression. A functor imports other modules as formal arguments, describing the shape of each import using a `signature`. The signature does not specify the source of the imports; a given program may contain several modules that all implement the same interface. Instead, the programmer explicitly links the functor to the source of its imports via a functor application.

Unfortunately, although ML's module system satisfies the principle of external connections, its mechanism for connecting functors is overly restrictive. Functors cannot define mutually-recursive procedures, since functor application can combine only a single functor with other unparameterized modules. In addition, functor application conflates linking with instantiation, which prohibits a mixture of hierarchical linking and multiple instantiation.

This dissertation presents a new language of modules, called units. Units combine the benefits of external linking specifications, graph-based linking to support mutual recursion across modules, and hierarchical linking separated from instantiation.
1.2.2 Classes

Object-oriented programming languages offer classes, inheritance, and overriding to parameterize over program pieces for reuse. Using class extension (inheritance) and overriding, a programmer derives a new class to reuse an old one, specifying for the derived class only the elements that change from the base class. For example, given a Java Frame class that implements windows in a word-processing application, a programmer can derive SearchFrame to implement windows that allow searching within the document, or MultiFrame to implement windows for editing multiple documents at once:

In this example, the class extensions SearchFrame and MultiFrame are hard-wired to the superclass Frame. Consider implementing SearchMultiFrame, which supports both searching and multiple files in the window. The programmer cannot combine SearchFrame and MultiFrame to implement SearchMultiFrame. Instead, the extension that implements the difference between SearchFrame and Frame must be duplicated and modified to derive SearchMultiFrame from MultiFrame:

In short, while class-based programming supports the reuse of classes, it fails to support the reuse of class extensions.

Multiple inheritance gives the programmer a way to rearrange some hard-wired links in a class derivation. For example, if SearchFrame and MultiFrame were implemented as C++ classes, the programmer could combine them using multiple inheritance, effectively implementing the dotted line in the preceding figure with a single declaration. Multiple inheritance, however, does not follow the principle of external connections; it merely provides a restricted set of operations for rearranging hard-wired links. In practice, programmers find these link-rearranging operations difficult
to understand. For example, if \texttt{SearchMultiFrame} inherits from both \texttt{SearchFrame} and \texttt{MultiFrame}, does an instance of \texttt{SearchMultiFrame} contain two instances of \texttt{Frame}, or just one?\footnote{In \texttt{C++}, the answer depends on whether \texttt{SearchFrame} and \texttt{MultiFrame} are declared as "virtual" subclasses of \texttt{Frame}.}

A language that supports \texttt{mixins} follows the principle of external connections. A \texttt{mixin} is a pure class extension: it specifies the shape of its superclass (\textit{i.e.} the fields and methods that are expected from the superclass), but it does not indicate a specific source for the superclass. The programmer specifies separately the connection between a \texttt{mixin} and the superclass it extends, perhaps applying a single \texttt{mixin} to multiple superclasses. This dissertation presents a detailed model of \texttt{mixins}, and it demonstrates how to integrate \texttt{mixins} with a conventional object-oriented language, such as \texttt{Java}.

\section{Dissertation Overview}

Chapter 2 introduces units and \texttt{mixins} by showing how they help solve a particular component-programming problem. Chapter 3 presents units in depth, providing untyped and typed models of units, with a proof of soundness for the typed model. Chapter 4 presents \texttt{mixins} in depth, providing models of classes and \texttt{mixins} for a \texttt{Java}-like language, with a proof of soundness for each model. Chapter 5 relates our experience using units and \texttt{mixins} to implement a large system. Chapter 6 provides an overview of related work on reuse. Chapter 7 discusses the limitations of this work and future directions.
Chapter 2

The Extensibility Problem

Most programs evolve over time. A typical program develops around a core component that implements the program's essential functionality. While the programmer occasionally extends the core component to support a new feature in some part of the program, other parts of the program remain unchanged. Thus, different parts of a program may evolve at different rates, particularly if the parts are implemented by different people or by different groups. Support for such evolution is a key challenge for component programming.

In this chapter, we introduce units and mixins by illustrating how they address a particular instance of evolution that we call the extensibility problem [13, 68, 71]. The following table summarizes the problem:

<table>
<thead>
<tr>
<th>Original Variants</th>
<th>Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>□</td>
<td>draw(□)</td>
</tr>
<tr>
<td>○</td>
<td>draw(○)</td>
</tr>
<tr>
<td>~</td>
<td>draw(∼)</td>
</tr>
<tr>
<td>shrink(□)</td>
<td>shrink(□)</td>
</tr>
<tr>
<td>shrink(○)</td>
<td>shrink(○)</td>
</tr>
<tr>
<td>∼</td>
<td>shrink(∼)</td>
</tr>
</tbody>
</table>

The portion of the table contained in the dotted box represents a core program component that provides several operations (draw and shrink) on a collection of data (squares and circles). A programmer may wish to use such a component in three different contexts:

1. The programmer may wish to include the component as is.

2. The programmer may wish to extend the datatype with a variant (repositioned shapes, represented as ∼) and adapt the collection of operations accordingly. The table illustrates this case as a new column to the right of the dotted box.

3. The programmer may wish to add a new operation (rotate), shown in the table as a new row below the dotted box.
To avoid duplicate maintenance, or because the component is acquired in object form, the components of such a program must be organized so that programmers can add both new forms of data and new operations \textit{without modifying or recompiling}

- the original program component, or
- its existing clients.

Such a program organization dramatically increases the potential for software reuse and the seamless integration of proprietary modules.

Neither standard functional nor object-oriented strategies offer a satisfactory way to implement the component and its clients. In a functional language, the variants can be implemented as a programmer-defined type, with the operations as functions on the type. Using this approach, the set of operations is easily extended, but adding a new variant requires modifying the functions. In an object-oriented language, the variants can be implemented as a collection of classes, with each operation as a method that is common to all classes. Using this approach, the datatype is easily extended with a new variant, but adding a new operation is typically implemented by modifying the classes.

The existing literature provides three solutions to the problem. Kühne's [49] solution, which relies on generic procedures with double-dispatching, can interfere with the hierarchical structure of the program. Palsberg and Jay's [65] solution is based on reflection operators and incurs a substantial run-time penalty. Krishnamurthi, Felleisen, and Friedman [21, 48] propose an efficient solution that works with standard class mechanisms, but it requires the implementation (and maintenance) of a complex programming protocol. All of these solutions are partial because they do not address the reuse of clients. In contrast, the combination of units and mixins solves the problem simply and elegantly, and it addresses the reuse of both the original component and its clients.

2.1 Extensible Programming with Classes

Figure 2.1 outlines our solution to the extensibility problem:

- Diagram (a) represents the original component. The rhombus stands for the datatype, and the rectangles denote the datatype's variants. The oval is a client of the datatype component.
• Diagram (b) shows the datatype extended with a new variant. The extension is contained in the right inner dashed box. The solid box on the left represents the unmodified datatype code from (a). The original client is also preserved, and a new client of the datatype exploits the variant extension.

• Diagram (c) shows extension in the other direction: adding a new operation to the datatype. As before, the extension is implemented by the inner dashed box while the solid box represents the unmodified existing implementation from (b). The new squares in the extension represent the implementation of the operation for each variant. The existing clients have not been modified, but they are now linked to the extended variants.

---

(a) Original Datatype

(b) New Variant

(c) New Operation

Figure 2.1: Extensible programming on datatypes

The remainder of this section develops a concrete example, an evolving shape program [21, 48]. Since Figure 2.1 can be approximated using conventional classes, we first use only language features available in a typical object-oriented language. Classes are not enough, however; Section 2.2 introduces units and mixins to complete the solution.
(define Shape (interface () draw))

(define Rectangle
  (class* object% (Shape) (width height)
    (public
      [draw (lambda (window x y) ...)])))

(define Circle
  (class* object% (Shape) (radius)
    (public
      [draw (lambda (window x y) ...)])))

(define Translated
  (class* object% (Shape) (shape Δx Δy)
    (public
      [draw (lambda (window x y)
        (send shape draw
          window (+ x Δx) (+ y Δy)))])))

Figure 2.2: Shape classes

Throughout this chapter, we use the syntax for classes, units, and mixins of the MzScheme programming language [26], which is an extension of Scheme [11]. Where necessary, we explain the syntax of these forms, but we assume a passing familiarity with Scheme’s syntax and with common object-oriented constructs.

2.1.1 Shape Datatype

Initially, our shape datatype consists of three variants and one operation: rectangles, circles, and translated shapes for drawing. The rectangle and circle variants contain numbers that describe the dimensions of the shape. The translated variant consists of two numbers, Δx and Δy, and another shape. For all variants, the drawing operation consumes a destination window and two numbers describing a position to draw the shape.

The shape datatype is defined by the Shape interface and implemented by three classes: Rectangle, Circle and Translated. Each subclass declares a draw method, which is required to implement the Shape interface. Figure 2.2 shows the interface and class definitions using MzScheme’s class system. (MzScheme’s class system is similar to Java’s; see Appendix A for details.)
(define display-shape
  (lambda (shape)
    (if (not (is-a? shape Shape))
      (error "expected a Shape")
      (let ([window ...])
        (send shape draw window 0 0))))

(display-shape (make-object Translated
  (make-object Rectangle 50 100)
  30 30))

Figure 2.3 : Two shape clients

(define Union
  (class* object% (Shape) (left right)
    (public
     [draw (lambda (window x y)
              (send left draw window x y)
              (send right draw window x y)]))

(display-shape
  (make-object Union
    (make-object Rectangle 10 30)
    (make-object Translated
      (make-object Circle 20) 30 30)))

Figure 2.4 : Variant extension and a new client

Figure 2.3 contains two client expressions for the shape datatype. The first one defines display-shape, a function that consumes a shape and draws it in a new window. The second expression creates a shape and displays it. As the shape datatype is extended, we consider how these clients are affected.

2.1.2 Variant Extension

To create more interesting configurations of shapes, we extend the shape datatype with a new variant representing the union of two shapes. Following the strategy suggested in Figure 2.1 (b), we define a new Union class derived from Shape. Figure 2.4 defines the Union class, and shows an expression that uses the new class.
The simplicity of the variant extension reflects the natural expressiveness of object-oriented programming. The object-oriented approach also lets us add this variant without modifying the original code or the existing clients in Figure 2.3.

2.1.3 Operation Extension

Shapes look better when they are drawn centered in their windows. We can support centered shapes by adding the operation *bounding-box*, which computes the smallest rectangle enclosing a shape.

We add an operation to our shape datatype by defining four new classes, each derived from the variants of *Shape* in Section 2.1.2. Figure 2.5 defines the extended classes—BB-Circle, BB-Rectangle, BB-Translated, and BB-Union—that provide the *bounding-box* method. The figure also defines the BB-Shape interface, which describes the extended shape type for the bounding box classes, just as *Shape* described the type for the original shape classes.

The new *display-shape* client, shown in Figure 2.5, uses bounding box information to center its shape in a window. Unfortunately, to use the existing clients, we must modify each to create instances of the new bounding box classes instead of the original shape classes, even if the client does not use bounding box information directly. The standard object-oriented architecture thus does not satisfy our original goal: it does not support operation extensions to the shape datatype without modifying existing clients.

Since object-oriented programming constructs do not address this problem directly, we might resort to a programming protocol or pattern. In this case, the Abstract Factory pattern [29] and a mutable reference solves the problem. The Abstract Factory pattern relies on one object, called the *factory*, to create instances of the shape classes. The factory supplies one creation method for each variant of the shape, and clients create shapes by calling these methods instead of using *make-object* directly. To change the classes that are instantiated by clients, it is only necessary to change the factory, which is stored in the mutable reference. A revised client, using the Abstract Factory, is shown in Figure 2.6.

Although the Abstract Factory pattern solves the problem, the programmer is forced to maintain this pattern manually. The pattern actually implements a simple
(define BB-Shape (interface (Shape) bounding-box))

(define BB-Rectangle
  (class* Rectangle (BB-Shape) (width height)
    (public
      [bounding-box
        (lambda () (make-object BB 0 0 width height))])
    (sequence (super-init width height))))

(define BB-Circle
  (class* Circle (BB-Shape) (radius)
    (public
      [bounding-box
        (lambda () (make-object BB (- radius) (- radius) radius radius))])
    (sequence (super-init r))))

(define BB-Translated
  (class* Translated (BB-Shape) (shape Δx Δy)
    (public
      [bounding-box (lambda () ...)]
    (sequence (super-init shape Δx Δy)))))

(define BB-Union
  (class* Union (BB-Shape) (left right)
    (public
      [bounding-box (lambda () ...)]
    (sequence (super-init left right)))))

(define BB
  (class* object% () (left top right bottom)
  ...
))

(define display-shape
  (lambda (shape)
    (if (not (is-a? shape BB-Shape))
      (error "expected a BB-Shape")
    (let* ([bb (send shape bounding-box])
        [window ...] [x ...] [y ...]
      (send shape draw window x y))))))

Figure 2.5 : Operation extension
(define Factory
  (class* object% () ()
    (public
      [make-circle (lambda (r) (make-object Circle r))]
      ...
    )
  )
)(define factory (make-object Factory))

...(define BB-Factory
  (class* Factory () ()
    (override
      [make-circle (lambda (r) (make-object BB-Circle r))]
      ...
    )
  )
)(set! factory (make-object BB-Factory))

...(display-shape (send factory make-union
  (send factory make-rectangle 10 30)
  (send factory make-translated
    (send factory make-circle 20) 30 30)))

Figure 2.6: Revised clients using Abstract Factory

dynamic linker, where the set! expression installs the link.\(^\text{1}\) This technique successfully separates the definition of shapes and clients so that a specific shape implementation can be selected at a later time, rather than hard-wiring a reference to a particular implementation into the client. However, using a construct like set! for linking obscures this intent both to other programmers and to the compiler. A more robust solution is to improve the module language.

2.2 Better Reuse through Units and Mixins

In the previous section, we developed the Shape datatype and its collection of operations, and we showed how object-oriented programming supports new variants and operations in separately developed extensions. In this section, we make the separate development explicit using units, defining the basic definitions, the extensions, and the clients all in separate units. MzScheme supports separate compilation for units, and

\(^\text{1}\)Factory Method is a related pattern where an extra operation in the datatype is used to create instances instead of a separate factory object. Factory Method applies to an interesting special case: the datatype client and the datatype implementation are the same, thus making the datatype implementation extensible.
(define Basic-Shapes
  (unit (import))
    (export Shape Rectangle Circle Translated)
  (define Shape (interface ...)) : see Figure 2.2
  (define Rectangle (class* object% (Shape) ...))
  (define Circle (class* object% (Shape) ...))
  (define Translated (class* object% (Shape) ...)))

Figure 2.7 : Creating Units

provides a flexible language for linking them. Indeed, the linking implemented with an Abstract Factory in the previous section can be more naturally defined through unit linking. Finally, we show how MzScheme's class-unit combination supports mixins, which provides new opportunities for reuse that are not available in conventional object-oriented languages.

2.2.1 Unitizing the Basic Shapes

Figure 2.7 shows the basic shape classes encapsulated in a Basic-Shapes unit. This unit imports nothing and exports all of the basic shape classes. The body of the unit contains the class definitions exactly as they appear in Figure 2.2.

In general, the shape of a unit expression is

\[
\text{(unit (import variable ...) (export variable ...)} \\
\text{unit-body-expr ...)}
\]

(where centered ellipses indicate repeated syntactic patterns). The unit-body-exprs have the same form as top-level Scheme expressions, allowing a mixture of expressions and definitions, but define within a unit expression creates a unit-local variable instead of a top-level variable. The unit's imported variables are bound within the unit-body-exprs. Each exported variable must be defined by some unit-body-expr. Unexported variables that are defined in the unit-body-exprs are private to the unit.

Figure 2.8 defines two client units of Basic-Shapes: Gui and Picture. The Gui unit provides the function display-shape (the same as in Figure 2.3). Since it only depends on the functionality in the Shape type, not the specific variants, it only imports Shape. The Picture unit imports all of the shape variants—so it can
(define GUI
  (unit (import Shape)
    (export display-shape)
    (define display-shape . .)))

(define PICTURE
  (unit (import Rectangle Circle Translated display-shape)
    (export)
    (display-shape (make-object . .))))

: see Figure 2.3

Figure 2.8: Unitized shape clients

construct instances—as well as the display-shape function—and it exports nothing. When PICTURE is invoked as part of a program, it constructs a shape and displays it.

A unit is an unevaluated bundle of code, much like an object file created by compiling a traditional language. At the point where BASIC-SHAPES, GUI, and PICTURE are defined as units, no shape classes have been defined, no instances have been created, and no drawing window has been opened. Each unit encapsulates its definitions and expressions without evaluating them until the unit is invoked, just like a procedure encapsulates expressions until it is applied. However, none of the units in Figures 2.7 and 2.8 can be invoked directly because each unit requires imports. The units must first be linked together to form a program.

2.2.2 Linking the Shape and Client Units

Units are linked together with the compound-unit form. Figure 2.9 shows how to link the units of the previous sub-section into a complete program: BASIC-PROGRAM. The PICTURE unit's imports are not a priori associated with the classes in BASIC-SHAPES. This association is established only by the compound-unit expression, and it is established only in the context of BASIC-PROGRAM. The PICTURE unit can be reused with different Shape classes in other compound units.

The compound-unit form links several units, called constituent units, into one new compound unit. The linking process matches imported variables in each constituent unit with either variables exported by other constituents, or variables imported into the compound unit. The compound unit can then re-export some of
(define Basic-Program
  (compound-unit
    (import)
    (link [S (Basic-Shapes)]
      [G (GUI (S Shape))]
      [P (PICTURE (S Rectangle) (S Circle) (S Translated) (G display-shape))])
    (export)))

(involve-unit Basic-Program)

Figure 2.9: Linking basic shape program

the variables exported by the constituents. Thus, Basic-Program is a unit with imports and exports, just like Basic-Shapes or GUI, and no evaluation of the unit bodies has occurred. Unlike the GUI unit, Basic-Program is a complete program, since it has no imports.

Each compound-unit expression

\[
\text{(compound-unit (import variable \ldots)}
\text{(link [tag}_1\text{ (expr}_1\text{ linkspec}_1\text{ \ldots)]]}
\text{;}
\text{[tag}_n\text{ (expr}_n\text{ linkspec}_n\text{ \ldots))]}
\text{(export (tag variable) \ldots))}
\]

has three parts:

- The import clause lists variables that are imported into the compound unit. These imported variables can be linked to the constituent unit's imports.

- The link clause specifies the graph of connections among the constituent units. Each constituent unit is specified via an expr and identified with a unique tag. Following the expr, a link specification linkspec is provided for each of the constituent's imports. Each link specification has one of two forms:
  - A linkspec of the form variable links the constituent's import to an import of the compound unit.
  - A linkspec of the form (tag variable) links the constituent's import to variable as exported by the tag constituent.
• The **export** clause re-exports variables from the compound unit that are exported from the constituents. The **tag** indicates the constituent and **variable** is the variable exported by the constituent.

To evaluate a **compound-unit** expression, the **exprs** in the **link** clause are evaluated to determine the compound unit’s constituents. For each constituent, the number of variables it imports must match the number of **linkspecs** provided; otherwise, an exception is raised. Each **linkspec** is matched to an imported variable in the constituent unit by position.\(^2\) Each constituent must also export the variables that are referenced by **link** and **export** clauses using the constituent’s **tag**.

Once a compound unit’s constituents are linked, the compound unit is indistinguishable from an atomic unit. Conceptually, linking creates a new unit by merging the internal definitions and expressions from all the constituent units. During this merge, variables are renamed as necessary to implement linking between constituents and to avoid name collisions between unrelated variables. The merged **unit-body-exprs** are ordered to match the order of the constituents in the **compound-unit**’s **link** clause.\(^3\)

### 2.2.3 Invoking Unit Programs

The **BASIC-PROGRAM** unit from Figure 2.9 is a complete program, analogous to a conventional application, but the program still has not been executed. In most languages with module systems, a complete program is executed through commands outside the language. In MzScheme, a program unit is executed directly with the **invoke-unit** form:

\[
\text{(invoke-unit expr)}
\]

The value of **expr** must be a unit. Invocation evaluates the unit’s definitions and expressions, and the result of the last expression in the unit is the result of the **invoke-unit** expression. Hence, to run **BASIC-PROGRAM**, we would evaluate

---

\(^2\)In MzScheme’s extended unit language with signatures, linking matches variables by name rather than by position; see Section 5.1 for details. When the number of imports is small, linking by position is simpler because it avoids complex machinery for renaming variables.

\(^3\)The implementation of linking is equivalent to this reduction, but far more efficient. In particular, it is not necessary to extract expressions from the constituent units, which would break separate compilation.
(define UNION-SHAPE
  (unit (import Shape)
    (export Union)
    (define Union (class* object% (Shape) . .)))) : see Figure 2.4

(define BASIC+UNION-SHAPE
  (compound-unit
    (import)
    (link [S (BASIC-SHAPE)]
      [US (UNION-SHAPE (S Shape))])
    (export (S Shape)
             (S Rectangle)
             (S Circle)
             (S Translated)
             (US Union)))

Figure 2.10 : Variant extension in a unit

(define UNION-PICTURE
  (unit (import Rectangle Circle Translated Union
display-shape)
    (export)
    (display-shape (make-object . .))) : see Figure 2.4

(define UNION-PROGRAM
  (compound-unit
    (import)
    (link [S (BASIC+UNION-SHAPE)]
      [G (GUI (S Shape))]
      [P (PICTURE (S Rectangle) (S Circle) (S Translated) (G display-shape))]
      [UP (UNION-PICTURE (S Rectangle)
        (S Circle)
        (S Translated)
        (S Union)
        (G display-shape))])
    (export)))

(invoke-unit UNION-PROGRAM)

Figure 2.11 : New client and the extended program

(invoke-unit BASIC-PROGRAM)
2.2.4 New Units for a New Variant

To extend Shape with a Union variant, we define the extension in its own unit, UNION-SHAPE, as shown in Figure 2.10. The Shape class is imported into UNION-SHAPE, and the new Union class is exported. In terms of Figure 2.1 (b), UNION-SHAPE corresponds to the smaller dashed box, drawn around the new variant class. The solid box is the original unmodified BASIC-SHAPES unit, and the outer dashed box in Figure 2.1 (b) is BASIC+UNION-SHAPES, a compound unit linking UNION-SHAPE together with BASIC-SHAPES.

Since the BASIC+UNION-SHAPES unit exports the variants from both BASIC-SHAPES and UNION-SHAPE, it can serve as a replacement for the original BASIC-SHAPES unit, yet it also provides additional functionality for new clients. The UNION-PROGRAM unit in Figure 2.11 demonstrates both of these uses. In this new program, the GUI and PICTURE clients are reused intact from the original program, but they are now linked to BASIC+UNION-SHAPES instead of BASIC-SHAPES. An additional client unit, UNION-PICTURE, takes advantage of the shape extension to draw a superimposed rectangle and circle picture.

2.2.5 New Units and Mixins for a New Operation

To extend Shape with a bounding-box operation, we define the BB-SHAPES unit in Figure 2.12. This unit corresponds to the smaller dashed box in Figure 2.1 (c).

The BB-SHAPES unit is the first example to rely on mixins. The BB-Rectangle class is derived from an imported Rectangle class, which is not determined until the unit is linked—long after the unit is compiled. Thus, BB-Rectangle defines a class extension that is parameterized over its superclass.

The BASIC+UNION+BB-SHAPES unit links the BASIC+UNION-SHAPES unit from the previous section with the new bounding-box unit, then exports the bounding-box classes. As the bounding-box classes are exported, they are renamed to match the original class names, i.e., BB-Rectangle is renamed to Rectangle, and so on. This renaming does not affect the linking within BASIC+UNION+BB-SHAPES; it only affects the way that BASIC+UNION+BB-SHAPES is linked with other units.

\footnote{The simplified description of compound-unit in Section 2.2.2 did not cover the syntax for renaming exports. For a complete description of compound-unit, see the MzScheme manual [26].}
As before, the **Basic+Union+BB-Shapes** unit serves as a replacement for either **Basic-Shapes** or **Basic+Union-Shapes**, and also provides new functionality for new clients. One new client is **BB-GUI** (see Figure 2.13), which provides a *display-shape* that exploits bounding box information to center a shape in a window.

At this point in the class-based derivation of Section 2.1, we resorted to the Abstract Factory pattern to make the old clients reusable. With units, an Abstract Factory is unnecessary, because units already let us vary the connection between the shape-creating clients and the shape classes. The **BB-GUI** unit replaces **GUI**, but we can reuse **Picture** and **Union-Picture** without modifying them. Putting everything together produces the new program **BB-Program**, shown at the bottom of Figure 2.13.
(define BB-GUI
  (unit (import BB-Shape BB)
    (export display-shape)
    (define display-shape
      (lambda (shape)
        (if (not (is-a? shape BB-Shape))
          ... : see Figure 2.5
          ...))))

(define BB-PROGRAM
  (compound-unit
    (import)
    (link [S (Basic+Union+BB-Shapes)]
      [BG (BB-GUI (S BB-Shape) (S BB))]
      [P (Picture (S Rectangle) (S Circle) (S Translated) (BG display-shape))]
      [UP (Union-Picture (S Rectangle)
        (S Circle)
        (S Translated)
        (S Union)
        (BG display-shape))])
    (export))

(invoke-unit BB-PROGRAM)

Figure 2.13 : Program with the operation extension

2.2.6 Units and Mixins at Work

The shape example demonstrates the expressiveness of units and mixins. Units, by separating the definition and linking of modules, support the reuse of Picture and Union-Picture as the shape representation evolves. Mixins, by abstracting a class expression over an imported class, enable the encapsulation of each extension in its own unit. The combination of units and mixins thus enables a direct translation of the ideal program structure from Figure 2.1 into a working program.

We have achieved the complete reuse of existing code at every stage in the extension of Shape, but even more reuse is possible. The code in Figure 2.14 illustrates how units and mixins combine to allow the use of one extension multiple times. The Color-Shape unit imports a Shape class and extends it to handle colors. With this single unit containing a single mixin, we can extend all four of the shape variants: Rectangle, Circle, Translated, and Union. The compound unit Basic+Union+BB+Color-Shapes in Figure 2.14 uses the Color-Shape unit
(define Color-Shape
  (unit (import Shape)
    (export C-Shape)
    (define C-Shape
      (class* Shape () args
        (rename
          [super-draw draw])
        (public
          [color "black"]
          [change-color
            (lambda (c) (set! color c))])
        (override
          [draw
            (lambda (window x y)
              (send window set-color color)
              (super-draw window x y)])
          (sequence
            (apply super-init args)))]))

(define Basic+Union+BB+Color-Shapes
  (compound-unit
    (import)
    (link [S (Basic+Union+BB-Shapes)]
      [CR (Color-Shape (S Rectangle))]
      [CC (Color-Shape (S Circle))]
      [CT (Color-Shape (S Translated))]
      [CU (Color-Shape (S Union))])
    (export (S Shape)
      (S BB-Shape)
      (S BB)
      (CR (C-Shape Rectangle))
      (CC (C-Shape Circle))
      (CT (C-Shape Translated))
      (CU (C-Shape Union)))]))

Figure 2.14: Reusing a class extension

four times to obtain the set of color shape classes.

The code in Figure 2.14 uses a few features that are not described in this chapter: the rename and override clauses in a class* expression, and the use of args to stand for multiple arguments, passed on to super-init with apply. These details are covered in the MzScheme reference manual [26]. Independent of such details, the example shows how units and mixins open new avenues for reuse on a large scale.
2.3 Summary

We presented the extensibility problem because it highlights many of the advantages of units and mixins. In existing programming languages, the problem can be solved using conventional module and class systems and the Abstract Factory pattern, but the pattern is cumbersome and difficult to maintain. A straightforward datatype implementation using units and mixins is more immediately extensible. This implicit bias towards reuse and extension is the essential benefit of units and mixins.

The following chapters explore units and mixins in more detail. In particular, we show how the constructs can be integrated into a statically-typed language, such as ML or Java, while preserving type soundness.
Chapter 3

Units

Our solution to the extensibility problem demonstrates how component programming at the module level requires separate compilation for modules and an expressive linking language. Separate compilation allows programmers to develop and deploy software components independently. An expressive linking language gives programmers precise control over the assembly of components into a whole program.

In general terms, units support the following properties to enable the component-building side of component programming:

- **Encapsulation:** A unit encapsulates a program part, clearly delineating the interface between the unit and all other parts of the program.

- **Separate compilation:** A unit’s interface provides enough information for the separate compilation of the unit.

To support the linking process, the unit language provides the following mechanisms:

- **Individual reuse and replacement:** Individual units are reusable and replaceable. This implies that the connections between units are specified outside the units themselves rather than hard-wired within each unit. In addition, the language supports multiple instances of a unit in different contexts within a program.

- **Hierarchical structuring:** The unit language allows units to be linked together to create a single, larger unit, possibly hiding selected details of the component units in the process.

- **Dynamic linking:** Units support dynamic linking, connecting new and executing code through a well-defined and localized interface.

This chapter presents untyped and typed models of units that are suitable for Scheme-like and ML-like languages. For these core languages, scaling essential core features to the module level implies two final properties:
• **Types:** If the core programming language supports static type definitions, units import and export types as well as values.

• **Mutual dependencies:** In whatever manner the core language supports mutually recursive definitions (usually procedure and type definitions), the unit language allows definitions with mutual references across module boundaries.

In addition to the mechanisms for defining and linking modules, a practical implementation of modules must provide constructs for naming modules (to coordinate module definitions and uses) and for abstracting over linking specifications. The flexibility of the module system depends on the expressiveness of this module-level language. In our model, we integrate units as first-class values within the core language, so that a programmer writes program-linking programs within the core language. The only primitive operations on units are linking and invocation, which preserves separate compilation for individual units, but programmers can exploit the full flexibility of the core language to apply these operations.

Section 3.1 explains how our unit model relates to existing module languages. Section 3.2 provides an overview of programming with typed units. Section 3.3 briefly considers extensions to the typed unit model. Section 3.4 discusses pragmatic problems in building programs with units. Section 3.5 defines the precise syntax, type checking, and semantics of units.

### 3.1 Existing Module Languages and Units

The unit model synthesizes ideas from three popular existing module systems: `.o` files, packages, and ML modules. The first represents the traditional view of modules as compilation units. The second extends this view by moving the module language into the programming language. The last gives programmers greater control over how modules are combined into a program.

Traditional languages, such as C, rely on the filesystem for the language of modules. Programs (makefiles) manipulate `.o` files to select the modules that are linked into a program, and module files are partially linked to create new `.o` or library files. Modern linking systems, such as ELF [77], support dynamic linking, but even the most advanced linking systems rely on a global namespace of function names and module (i.e., file) names. As a result, modules can be linked and invoked only once in a program.
Many modern languages—such as Ada 95 [38], Modula-2 [84], Modula-3 [32], Haskell [37], and Java [31]—provide packages. A package system delineates the boundaries of each module and forces the specification of static dependencies between modules. Since module linking and invocation are clearly separated, packages allow mutually recursive function and type definitions across package boundaries.

The main weakness of a package system is its reliance on a global namespace of packages with hardwired connections among packages. Package systems do not permit the reuse of a single package for multiple invocations in a program or the external selection of connections between packages. Packages cannot be merged into a new package that hides parts of the constituent packages. In addition, among the languages with packages, only Java provides a mechanism for dynamic linking. This mechanism is expressed indirectly via the language of class loaders, and is not fully general due to the constraints of a global package namespace.\(^1\)

ML’s functor system [56, 62] is the most notable example of a language that lets a programmer describe abstractions over modules and gives a programmer direct control over assembling modules. In contrast to a package, an ML structure module is not a fragment of unevaluated code. Instead, a structure is a record with fields containing the module’s exported values and types. A module with dependencies is defined as a functor, a first-order function that consumes a structure and produces a new structure. Functors separate the specification of module dependencies from module linking. Unfortunately, linking by functor application prevents the definition of mutually recursive types or procedures across module boundaries. In addition, ML provides no mechanism for dynamic linking.

3.2 Programming with Units

Like a package in Java or Modula-3, a program unit is an unevaluated fragment of code, but there is no global namespace of units. Instead, like an ML functor, a unit describes its import requirements without specifying a particular unit that supplies those imports. The actual linking of the unit is specified externally at a later stage.

---

\(^1\)Ada and Modula-3’s generics permit such uses, but do not support separate compilation.

\(^2\)Java’s class system can also be viewed as a kind of module system or as a complement to the package system. Classes suffer the same drawbacks as packages: links, such as a superclass name, are hard-wired to a specific class [28].
Unlike in ML, unit linking is specified for groups of units with a graph of connections, which allows mutual recursion across unit boundaries. Furthermore, the result of linking a collection of units is a new (compound) unit that is available for further linking.

This section illustrates the basic design elements of our unit language using an informal, semi-graphical programming language. The examples assume a core language with lexical blocks and a sub-language of types. The syntax used for the core language mimics that of ML.

Figure 3.1: An atomic database unit

3.2.1 Defining Units

Figure 3.1 defines a unit called Database. In the graphical notation, a unit is drawn as a box with three sections:

- The top section lists the unit's imported types and values. The Database unit imports the type info (of kind \( \Omega \)) for data stored in the database, and the function error (of type str\( \rightarrow \)void) for error-handling.

---

The graphical language is currently being implemented for the DrScheme [23] programming environment. Programmers will define modules and linking by actually drawing boxes and arrows.

A kind is a type for a type. Most languages have only one kind, \( \Omega \), and do not ask programmers to specify the kind of a type. Some languages (such as ML, Haskell, and Miranda) also provide type constructors or functions on types, which have the kind \( \Omega^* \rightarrow \Omega \).
The middle section contains the unit's definitions and an initialization expression. The latter performs start-up actions for the unit at run time. The *Database* unit defines the type *db* and the functions *new, insert*, and *delete* (plus some other definitions that are not shown). Database entries are keyed by strings, so *Database* initializes a hash table for strings with the expression 

\[ strTable := makeStringHashTable() \]

The bottom section enumerates the unit's exported types and values. The *Database* unit exports the type *db* and the functions *new, insert*, and *delete*.

In a statically-typed language, all imported and exported variables have a type, and all imported and exported types have a kind.\(^4\) Imported and defined types can be used in the type expressions for imported and exported values. All exported variables must be defined within the unit, and the type expression for an exported value must use only imported and exported types. In *Database*, both the imported type *info* and the exported type *db* appear in the type expression for *insert*: \( db \times str \times info \to void \).

A unit is specifically *not* a record of values. It encapsulates unevaluated code, much like the *.o* file created by compiling a C++ module. Before a unit's definitions and initialization expression can be evaluated, it must first be linked with other units to resolve all of its imports.

### 3.2.2 Linking Units

In the graphical notation, a programmer links units together by drawing arrows to connect the exports of one box with the imports of another. Linking units together creates a compound unit, as illustrated in Figure 3.2 with the *PhoneBook* unit. This unit links *Database* with *NumberInfo*, a unit that implements the *info* type for phone numbers.

Figure 3.2 also shows how to link units in stages. The *error* function is not defined by either *Database* or *NumberInfo*, so *PhoneBook* imports *error* and passes the imported value on to *Database*. At the same time, *PhoneBook* hides the *delete* function, but re-exports all of the other values and types from *Database* and *NumberInfo*.

A complete program is a unit without imports. Figure 3.3 defines a complete interactive phone book program, *IPB* (Interactive Phone Book), which links *Phone-
Figure 3.2: Linking units to form a compound unit

Figure 3.3: Linking units to define a complete program
Book with a graphical interface implementation Gui. The Main\textsuperscript{5} unit contains an initialization expression that creates a database and an associated graphical user interface.

A program unit is analogous to an executable file: invoking the unit evaluates the definitions in all of the program's units and then executes their initialization expressions. Thus, invoking IPB executes Main's initialization expression, which creates a new phone book database and opens a phone book window. The variables exported by a program are ignored. Instead, the result of invoking a program is the value of its last initialization expression—a bool value in IPB (assuming Main's expression is evaluated last).\textsuperscript{6}

A compound unit's links must satisfy the type requirements of the constituent

\textsuperscript{5}The name Main is not special.

\textsuperscript{6}Our informal graphical notation does not specify the order of units in a compound unit, but the textual notation in Section 3.5 covers this aspect of the language.
units. For example, in IPB (see Figure 3.3). Main imports the type \texttt{db} from PhoneBook unit and also the function \texttt{openBook:db→bool} from Gui. The two occurrences of \texttt{db} must refer to the same type. A type checker can verify this constraint by proving that the two occurrences have the same source in the link graph, which is the \texttt{db} exported by PhoneBook. In contrast, Figure 3.4 defines a “program” \texttt{Bad} in which \texttt{Main} receives inconsistent imports. Specifically, \texttt{db} and \texttt{openBook:db→bool} refer to types named \texttt{db} that originate from different units. The type checker correctly rejects \texttt{Bad} due to this mismatch.

Linking can connect units in a mutually recursive manner, as illustrated in IPB (see Figure 3.3); links flow both from PhoneBook to Gui and from Gui to PhoneBook. Thus, the insert function in PhoneBook may call error in Gui, which might in turn call PhoneBook’s insert again to handle the error.

### 3.2.3 Programs that Link and Invoke Other Programs

The IPB program relies on a fixed set of constituent units, including a specific unit Gui to implement the graphical interface. In general, there may be multiple GUIs that work with the phone book. e.g., separate GUIs for novice and advanced users. Every GUI unit will have the same set of imports and exports, so the linking information required to produce the complete interactive phone book is independent of the specific GUI unit. In short, a programmer should abstract over IPB with respect to its GUI unit.

If the core evaluation language integrates the form for linking units, then a programmer can achieve the abstraction of IPB with a core function. Figure 3.5 defines \texttt{MakeIPB}, a function that accepts a GUI unit and returns an interactive phone book unit. The programmer draws a dashed box for aGui and \texttt{MakeIPB} to indicate that the actual GUI and interactive phone book units are not yet determined. The programmer can then apply \texttt{MakeIPB} to different GUI implementations to produce different interactive phone book programs.

The type associated with \texttt{MakeIPB}’s argument is a unit type, a signature, that contains all of the information needed to verify its linkage in \texttt{MakeIPB}. In the graphical notation, a signature corresponds to a box with imports, exports, and an initialization expression type, but no definitions or expressions. The signature for aGui is defined by its dotted box, with \texttt{void} indicating the type of the initialization expres-
fun MakeIPB(aGui) =

$$\text{PhoneBook}$$

$$\cdots$$

$$\cdots$$

$$\cdots$$

$$\text{Main}$$

$$\text{db}::\Omega$$ insert:db $\times$ str $\times$ info $\rightarrow$ void

$$\text{info}::\Omega$$ numInfo:int $\rightarrow$ info

$$\text{void}$$

$$\text{:openBook:db}$$ $\rightarrow$ bool

$$\text{error:str}$$ $\rightarrow$ void

Figure 3.5: Abstracting over constituent units

Starter

fun MakeIPB(aGui) =

val ExpertGui =

val NoviceGui =

invoke MakeIPB(if expertMode() ExpertGui else NoviceGui)

Figure 3.6: Linking and invoking other programs
sion. Using only this signature, the type system can completely verify the linking in MakeIPB and determine the signature of the resulting compound unit.

Figure 3.6 shows MakeIPB as part of a larger program. Starter, that selects a GUI unit and links together a complete interactive phone book program. Once MakeIPB returns a program unit, Starter launches the constructed program with the special invoke form, which takes a program unit and executes it.

3.2.4 Dynamic Linking

The invoke form also works on units that are not complete programs. In that case, the unit’s imports must be explicitly satisfied by types and values from the invoking program. This generalized form of invocation implements dynamic linking. For example, the phone book program can exploit dynamic linking to support third-party “plug-in” extensions that load phone numbers from a foreign source. A third-party implements each loader extension as a unit that is dynamically retrieved from an archive and then linked with the phone book program.7 With such plug-ins, the user of the phone book can install loader extensions at run-time via interactive dialogues.

Figure 3.7 defines a Gui unit that supports loader extensions. The function addLoader consumes a loader extension as a unit and dynamically links it into the program using invoke. The extension unit imports types and functions that enable it to modify the phone book database. These imports are satisfied in the invoke expression with types and variables that were originally imported into Gui, plus the error function defined within Gui. The result of invoking the extension unit is the value of the unit’s initialization expression, which is required (via signatures) to be a function of type \( \text{db} \times \text{file} \to \text{void} \). This function is then installed into the GUI’s table of loader functions.

---

7The core language must provide a syntactic form that retrieves a unit value from an archive, such as the Internet, and checks that the unit satisfies a particular signature. This type-checking must be performed in the correct context to ensure that dynamic linking is type-safe. Java’s dynamic class loading is broken because it checks types in a type environment that may differ from the environment where the class is used [74].
3.3 Possible Extensions to Units

Experience with other modules systems, particularly those of ML, suggests further extensions to UNITe, such as facilities for exposing the implementation of a type, or hiding the type (or parts of the type) of a value:

- **Exposing type information:** The ML module system allows signatures that reveal some information about an exported type [33, 53]. The partially exposed types (or translucent types) are used for propagating type dependencies in a way that allows type sharing, but they are also useful for assigning a name to a complex type that is exposed to clients. For example, consider the case of an Environment unit that exports values of type env while revealing to clients that env is a procedure type.

As shown in Figure 3.8, the translucent type env in this case may be viewed as a type abbreviation that is preserved within the signature. The unit Environment does not export the type env. Instead, the unit and its signature are extended with an extra section that defines the abbreviation env. The resulting unit and signature are equivalent to the unit and signature that expands env in all type expressions.
Figure 3.8: Exposing information for a type

Figure 3.9: Hiding type information for an exported value
Hiding type information: Large projects often have multiple levels of clients. Some of the clients are more trusted than others and are thus privy to more information about the implementation of certain abstractions. To support this situation, UNIt* could provide mechanisms for hiding a value's type information from untrusted clients after linking with trusted clients.

Consider the example in Figure 3.9. The Environment unit is linked with the Letrec unit, allowing the latter to exploit the implementation of environments as procedures. In contrast, other clients should not be allowed to exploit the implementation of environments. Hence, the type of environments should be opaque outside the compound unit RecEnv, which combines Environment and Letrec.

As shown in Figure 3.9, information about RecEnv's exports can be restricted via explicit signatures and an extended subtype relation. The extended relation allows a subtype signature to contain an extra exported type variable (e.g., env) in place of an abbreviation in the supertype signature. As a result, the information formerly exposed by the abbreviation becomes a hidden, opaque type.

3.4 Problems with Units

Language designers have often noted the tension between modules as constructs for separate compilation and modules as constructs for program organization. Compilation guarantees tend to limit abstractions for organizing a program, whereas powerful module abstractions tend to defeat separate compilation. By requiring modules to serve as components, we place strong demands on both the compilation and abstraction properties of our module language.

Units achieve this combination at the expense of programming convenience for small programs or widely-used library components. For programs with a flat module hierarchy (i.e., all modules are linked at once), programming with units resembles programming in a package language. Unfortunately, in addition to defining each individual module, the unit programmer must also define a final compound unit that explicitly links the modules together. This linking step is implicit and automatic in package languages.

For programs with a strict, tree-shaped linking hierarchy, programming with units
resembles programming with closed ML functors. Programmers, however, find this mode of programming cumbersome in practice, and ML programmers tend instead to write library components as package-like structures, relying on a compilation manager [53] to automate a functor-closing transformation on such libraries. MzScheme does not currently provide such a facility.

![Diagram](image)

Figure 3.10: The diamond import problem

For programs with more complex linking structures, programming with units differs from programming with either packages or functors. To illustrate the differences, consider the classic "diamond import" problem, as shown in Figure 3.10. A Symbol module exports the type sym to Lexer and Parser modules, which each supply a function to the Reader module. The Lexer and Parser modules use the type sym directly, but the Reader module uses sym only indirectly by composing the functions lex and parse.

A Java sketch of the program appears in Figure 3.11. The Lexer and Parser packages both import the Symbol packages, and the Reader package imports Lexer and Parser. Packages support diamond import transparently through hard-wired module connections and a global namespace of types; the Reader package need not refer to the Symbol package at all.

An SML sketch of the program appears in Figure 3.12. The first four blocks of code in the figure define the four modules as functors. The last block in the figure links the modules together, first instantiating the Symbol functor, then linking each of Lexer and Parser to the Symbol instance, and finally linking Reader to the Lexer and Parser instances. The application of the Reader functor succeeds only because both
`Lexer` and `Parser` explicitly reveal that `sym` originates from their `SYMBOL` argument: a first-order flow analysis proves that the `Lexer` and `Parser` functors are applied to the same `Symbol` instance. In general, diamond import with functors requires some work from the programmer, but the work is relatively localized due to the first-order nature of structures.

A unit sketch of the program appears in Figure 3.13, using a textual syntax defined in the next section.\(^8\) The first four blocks of code define the modules as units, and the last block links them together. Unlike packages, the linking specification is explicit and separate from the unit definitions. Unlike functors, all of the modules are linked together at once. Figure 3.14 shows the same program in our graphical notation.

The staged linking used in the functor-based program does not work with units. Figure 3.15 illustrates how separately linking `Lexer` with `Symbol` and `Parser` with `Symbol` causes a linking failure for `Reader`: `Reader` cannot import `lex` and `parse` because there is no single source for the type `sym`.

\(^8\)Although we have not yet defined a textual syntax for units, we use it to show a more direct comparison of units to packages and functors.
signature `SYMBOL`  
= sig type sym end

functor `Symbol` > `SYMBOL`  
= struct type sym = ... end

signature `LEXER` = sig  
  type sym
  val lex : str → sym
end

functor `Lexer(structure S : SYMBOL)`  
  > `LEXER` where type sym = S.sym  
= struct  
  type sym = S.sym
  fun lex(s : str) = ...
end

signature `PARSER` = sig  
  type sym
  val parse : sym → expr
end

functor `Parser(structure S : SYMBOL)`  
  > `PARSER` where type sym = S.sym  
= struct  
  type sym = S.sym
  fun parse(s : sym) = ...
end

functor `Reader(structure L : LEXER`  
  structure P : PARSER`  
  where type sym = L.sym)  
= struct  
  fun read(s : str) = P.parse(L.lex(s))
end

structure `S = Sym()`

structure `L = Lexer(structure S = S)`
structure `P = Parser(structure S = S)`
structure `R = Reader(structure L = L`  
  structure `P = P)`

Figure 3.12: Diamond import with SML functors

In general, a program's graph of unit instances can be partitioned into compound units, but these partitions must not overlap. Thus, unit linking tends to force library dependencies to the top of the linking hierarchy, which increases both the size of the top-level linking expression and the size of import interfaces for intermediate compound units. For component-based programming, propagating library dependencies to the top is beneficial; the programmer linking the final program gains the freedom to
select the library units. But for general-purpose programming, pushing dependencies to the top is often inconvenient and clumsy.

### 3.5 The Structure and Interpretation of Units

In this section, we develop a semantic and type-theoretic account of the unit language design in three stages. We start in Section 3.5.1 with units as an extension of a dynamically typed language (like Scheme) to introduce the basic syntax and semantics
Figure 3.14: Diamond import with units

Figure 3.15: Incorrect structure for diamond import with units
of units. In Section 3.5.2, we enrich this language with definitions for constructed types (like classes in Java or datatypes in ML). Finally, in Section 3.5.3 we consider arbitrary type definitions (like type equations in ML).

The rigorous description of the unit language, including its type structures and semantics, relies on well-known type checking and rewriting techniques for Scheme and ML [22, 34, 85]. In the rewriting model of evaluation, the set of program expressions is partitioned into a set of values and a set of non-values. Evaluation is the process of rewriting a non-value expression within a program to an equivalent expression, repeating this process until the whole program is rewritten to a value. An atomic unit expression—represented in the graphical language by a box containing text code—is a value, whereas a compound unit expression—a box containing linked boxes—is not a value. Thus, a compound unit expression must be re-written to obtain a value.

![Graphical reduction rule for a compound unit](image)

Figure 3.16: Graphical reduction rule for a compound unit

A compound unit expression with known constituents can be re-written to an equivalent unit expression by merging the text of its constituent units, as demon-
\[
\begin{align*}
v &= \mathit{unit-expr} \mid c \mid \mathbf{fn} \ x \Rightarrow e \\
e &= \mathit{compound-expr} \mid \mathit{invoke-expr} \mid \mathit{letrec-expr} \mid e ; e \mid x \mid e \ e \mid v \\
\mathit{unit-expr} &= \mathit{unit} \ \mathit{import} \ \mathit{variable-mapping}^* \\
&\quad \mathit{export} \ \mathit{variable-mapping}^* \\
&\quad \mathit{definitions} \ e \\
\mathit{compound-expr} &= \mathit{compound} \ \mathit{import} \ y^* \\
&\quad \mathit{export} \ y^* \\
&\quad \mathit{link} \ e \ \mathit{link} \ \mathit{and} \ e \ \mathit{link} \\
\mathit{invoke-expr} &= \mathit{invoke} \ e \ \mathit{with} \ \mathit{value-invoke-link}^* \\
\mathit{letrec-expr} &= \mathit{letrec} \ \mathit{definitions} \ \in \ e \\
\mathit{definitions} &= \mathit{value-defn}^* \\
\mathit{value-defn} &= \mathit{val} \ x \ = \ v \\
\mathit{link} &= \mathit{with} \ y^* \ \mathit{provides} \ y^* \\
\mathit{variable-mapping} &= \ y \ = \ x \\
\mathit{value-invoke-link} &= \ y \ = \ e \\
x &= \mathit{variable} \\
y &= \mathit{linking \ variable} \\
c &= \mathit{primitive \ constant}
\end{align*}
\]

Figure 3.17: Syntax for \textsc{Unit}_d (dynamically typed)

strated in Figure 3.16. Invocation for a unit is similar: an \textbf{invoke} expression is rewritten by extracting the invoked unit's definitions and initialization expression, and then replacing references to imported variables with values. Otherwise, the standard rules for functions, assignments, and exceptions apply.

### 3.5.1 Dynamically Typed Units

Figure 3.17 defines the syntax of \textsc{Unit}_d, an extension of a dynamically typed core language. The core language provides several standard forms: a procedure form, an application form, an expression sequence form (";") and a \textbf{letrec} form for lexical blocks containing mutually recursive definitions. \textsc{Unit}_d extends this core language with three unit-specific forms:

- a \textbf{unit} form for creating units,
- a \textbf{compound} form for linking units, and
• an **invoke** form for invoking units.

**The unit Form**

The **unit** form consists of a set of import and export declarations followed by internal definitions and an initialization expression:

\[
\text{unit import } y_i = x_i \cdots \text{ export } y_e = x_e \cdots \\
\text{val } r = v \cdots \\
\text{e}
\]

The imported variables \( y_i \) have internal names \( x_i \), which are bound in the definition and initialization expressions. The internal names \( x_e \) of the exported variables must be defined within the unit. The scope of each imported and defined variable includes all of the definition expressions \( v \) in the unit, as well as the initialization expression \( e \). The internal names \( x_i \) and \( x_e \) are subject to \( \alpha \)-renaming, but the external names \( y_i \) and \( y_e \) are not.

In each definition **val** \( r = v \). the right-hand side must be a value (a constant, function, or unit). This restriction simplifies the presentation of the formal semantics, since the definitions are in a mutually-recursive scope. The restriction can be lifted for an implementation, as in MzScheme, where accessing an undefined variable returns a default value or signals a run-time error.

A **unit** expression is a first-class value, just like a number or an object in Java. The language provides only two operations on units: linking and invoking. No operation can "look inside" a unit value to extract any information about its definitions or initialization expression. In particular, since a unit does not contain values, only unevaluated definition and expressions, there is no "dot notation" for externally accessing values from a unit (as for packages in Java) and there are no "instantiated units" (approximating ML structures) that contain the values of unit expressions.

**The compound Form**

The **compound** form links two constituent units together into a new unit:

\[
\text{compound import } y_i \cdots \text{ export } y_e \cdots \\
\text{link } e_1 \text{ with } y_{w1} \cdots \text{ provides } y_{p1} \cdots \\
\text{and } e_2 \text{ with } y_{w2} \cdots \text{ provides } y_{p2} \cdots
\]
Two subexpressions, $e_1$ and $e_2$, determine the constituent units. The **with** $y_w \cdots$ clause following each expression lists the variables that the corresponding unit is expected to import. Similarly, the **provides** $y_p \cdots$ clause lists the variables that the corresponding unit is expected to export.

The **compound** form links variables by name. Thus, the set of variables $y_{w1}$ linked into the first unit must be a subset of $y_i \cup y_{p2}$. Similarly, $y_{w2}$ must be a subset of $y_i \cup y_{w1}$. Finally, the set variables $y_e$ exported by the compound unit must be a subset of $y_{p1} \cup y_{p2}$.

A **compound** unit expression is not an immediate value, but it evaluates to a unit value that is indistinguishable from a unit created with **unit**. This unit's initialization expression is the sequence of the first constituent unit's initialization expression followed by the the second constituent unit’s.

We restrict **compound** so that it links only two units at a time to simplify our presentation. The linking construct implemented for MzScheme is less restrictive than **UNITd**'s. In MzScheme, the **compound** form links any number of units together at once (a simple generalization of **UNITd**'s two-unit form), and links imports and exports via source and destination name pairs, rather than requiring the same name at both ends of a linkage.

**The invoke Form**

The **invoke** form evaluates its first subexpression to a unit and invokes it:

\[
\text{invoke } e \text{ with } y_i \leftarrow e_i \cdots
\]

If the unit requires any imported values, they must be provided through $y_i \leftarrow e_i$ declarations, which associate values $e_i$ with names $y_i$ for the unit’s imports. An **invoke** expression evaluates to the invoked unit’s initialization expression.

**UNITd Context-sensitive Checking**

The rules in Figure 3.18 specify the context-sensitive properties that were informally described in the previous section. The checks ensure that a variable is not unbound or multiply defined, imported, or exported, that all exported variables are defined, and that the **link** clause of a **compound** expression is locally consistent.
The notation $\mathcal{E}$ denotes either a set or a sequence of variables $x$, depending on the context. The notation $\text{val } \mathcal{E} = e$ denotes a set or sequence of forms $\text{val } x = e$ where each $x$ is taken from the sequence $\mathcal{E}$ with a corresponding $e$ from the sequence $e$.

$$
\frac{\mathcal{E} \not\subset \text{dom}(\Gamma)}{\Gamma \vdash e} \quad \frac{\Gamma \vdash x \quad e \in \text{dom}(\Gamma)}{\Gamma \vdash e} \\
\frac{\mathcal{E} \not\subset \text{dom}(\Gamma) \quad \text{val } \mathcal{E} = e}{\Gamma \vdash \text{fn } x = e} \\
\frac{\mathcal{E} \not\subset \text{dom}(\Gamma) \quad \text{fun } x^e \quad \Gamma \vdash e_1 \quad \Gamma \vdash e_2}{\Gamma \vdash \text{app } x^e_1 e_2} \\
\frac{\mathcal{E} \not\subset \text{dom}(\Gamma) \quad \text{seq } \mathcal{E}_1 \quad \mathcal{E}_2}{\Gamma \vdash \text{seq } \mathcal{E}_1 \mathcal{E}_2} \\
\frac{\mathcal{E} \not\subset \text{dom}(\Gamma) \quad \text{letrec } \mathcal{E}_1 \quad \mathcal{E}_2}{\Gamma \vdash \text{letrec } \mathcal{E}_1 \mathcal{E}_2} \\
\frac{\mathcal{E} \not\subset \text{dom}(\Gamma) \quad \text{invoke } \mathcal{E}_1 \quad \mathcal{E}_2}{\Gamma \vdash \text{invoke } \mathcal{E}_1 \mathcal{E}_2} \\
\frac{\mathcal{E} \not\subset \text{dom}(\Gamma) \quad \text{compound } \mathcal{E}_1 \quad \mathcal{E}_2}{\Gamma \vdash \text{compound } \mathcal{E}_1 \mathcal{E}_2}
$$

The form of $\text{UNIT}_d$ expressions:

**UNIT\textsubscript{d} Evaluation**

Figure 3.19 contains the reduction rules for $\text{UNIT}_d$, which generalize the graphical example in Figure 3.16. The rules extend those for Scheme [22] and resemble equations in the higher-order module calculus of Harper, Mitchell, and Moggi [34]. The $\text{app}_d^+$, $\text{seq}_d^+$, and $\text{letrec}_d^+$ rules are standard.

The $\text{invoke}_d^+$ rule specifies that an $\text{invoke}$ expression reduces to a $\text{letrec}$ expression containing the invoked unit’s definitions and initialization expression. In this $\text{letrec}$ expression, imported variables are replaced by values. The set of variables supplied by $\text{invoke}$’s $\text{with}$ clause must cover the set of the imports required by the unit.

The $\text{compound}_d^+$ rule defines how a $\text{compound}$ expression combines two units: their definitions are merged and their initialization expressions are sequenced. The
\[
E = \begin{cases} 
[ | E | e | v : E | E : e \\
\text{invoke } E \ldots \\
\text{invoke } v \text{ with } ... \ y \leftarrow E \ldots \\
\text{compound } \ldots \text{link } E \ldots \text{and } e \\
\text{compound } \ldots \text{link } v \ldots \text{and } E 
\end{cases}
\]

\[E[e] \rightarrow E[e'] \quad \text{if } e \rightarrow e'\]

\[\text{app}_e^{+}:
\text{fn } x \Rightarrow e \quad \rightarrow \quad [v/x]e
\]

\[\text{seq}_e^{+}:
v : e \quad \rightarrow \quad e
\]

\[\text{letrec}_e^{+}:
\text{letrec } \text{val } x = v \text{ in } e_b \quad \rightarrow \quad \text{letrec } \text{val } x = v \text{ in } [v/x]e_b
\]

\[\text{invoke}_e^{+}:
\text{invoke (unit import } \bar{y}_1 = x_1, \text{ export } \bar{y}_e = x_e \quad \rightarrow \quad \text{[v}_w/x_w]\text{(letrec } \text{val } x = v \text{ in } e_b)
\]

\[\text{with } y_w \leftarrow x_w
\]

\[\text{if } y_1 = x_1 \subseteq y_w = x_w
\]

\[\text{compound}_e^{+}:
\text{compound import } \bar{y}_1 \text{ export } \bar{y}_e \quad \rightarrow \quad \text{unit import } \bar{y}_1 = x_1,
\]

\[\text{link (unit import } \bar{y}_{e1} = x_{e1} \text{ export } \bar{y}_{e1} = x_{e1} \quad \rightarrow \quad \text{export } \bar{y}_e = x_e
\]

\[\text{with } y_{e1} \text{ provides } \bar{y}_{p1}
\]

\[\text{and (unit import } \bar{y}_{e2} = x_{e2} \text{ export } \bar{y}_{e2} = x_{e2} \quad \rightarrow \quad \text{with } y_{e2} \text{ provides } \bar{y}_{p2}
\]

\[\text{if } x_1, x_2, x_e \text{ distinct,}
\]

\[y_{e1} = x_{e1} \subseteq y_{e1} = x_{e1}, \quad y_{e1} = x_{e1} \subseteq y_{e1} = x_{e1}, \quad y_{e2} = x_{e2} \subseteq y_{e2} = x_{e2}, \quad \text{and } y_{p2} = x_{p2} \subseteq y_{p2} = x_{p2}
\]

Figure 3.19 : Reducing UNIT_d expressions

side condition requires that the constituent units provide at least the expected exports (according to the provides clauses) and need no more than the expected imports (according to the with clauses). The side condition also ensures that bindings introduced by definitions in the two units are α-renamed to avoid collisions and to make the internal-external variable mappings match.
**unit import** even
**export** odd

**val** odd = **fn** 0 ⇒ false
| n ⇒ even (n-1)

odd 13

⇒

**fn** (even\textit{cell}, odd\textit{cell}) ⇒

\( \text{odd\textit{cell}} := (\text{fn} 0 ⇒ \text{false}) \)
| n ⇒ (\text{even\textit{cell}} (n-1)):

**fn** () ⇒ (!odd\textit{cell} 13)

---

**UNIT\textsubscript{d} Implementation**

In MzScheme’s implementation of UNIT\textsubscript{d}, units are compiled by elaborating them into functions. The unit’s imported and exported variables are implemented as first-class reference cells that are externally created and passed to the function when the unit is invoked. The function is responsible for filling the export cells with exported values and for remembering the import cells for accessing imports later. The return value of the function is a closure that evaluates the unit’s initialization expression. Figure 3.20 illustrates this transformation for an atomic unit.

Each compound unit is also compiled to a function. The function encapsulates a list of constituent units and a closure that propagates import and export cells to the constituent units, creating new cells to implement variables in the constituents that are hidden by the compound unit.

The transformed units have the same code-sharing properties as traditional shared libraries. The definition and initialization expressions of a unit are compiled in the body of the function produced by its transformation, and this one function is used for all instances of the unit. Thus, there exists a single copy of the definition and initialization code regardless of how many times the unit is linked or invoked.\(^9\)

\(^9\)Our native code compiler for MzScheme effectively transforms a **unit** expression to a shared
3.5.2 Units with Constructed Types

Figure 3.21 extends the language in Figure 3.17 for a statically typed language with programmer-defined constructed types, like ML datatypes. In the new language, \texttt{UNIT}_c, the imports and exports of a \texttt{unit} expression include type variables as well as value variables. All type variables have a kind\textsuperscript{10} and all value variables have a type. The type of the unit's initialization expression is also declared, following the \texttt{\triangleright} in the unit's header. The \texttt{compound} and \texttt{invoke} forms extend to imported and exported types as well, where each type has an internal name to be used in the type expressions for imported and exported values. The new \texttt{as} form permits explicit type generalization, casting an expression's type to a supertype.

The definition section of a \texttt{unit} or \texttt{letrec} expression contains both type and value definitions. Type definitions are similar to ML \texttt{datatype} definitions, but for simplicity, every type defined in \texttt{UNIT}_c has exactly two variants. Type definitions have the form \texttt{type} \texttt{l} = \texttt{x_{dl}.x_{di} \tau_i} \mid \texttt{x_{cr}.x_{dr} \tau_r \triangleleft x_t}. Instances of the first variant are constructed with the \texttt{x_{dl}} function, which takes a value of type \texttt{\tau_i} and constructs a value of type \texttt{l}. They are deconstructed with \texttt{x_{di}}. Instances of the second variant are constructed with \texttt{x_{cr}} given a value of type \texttt{\tau_r} and deconstructed with \texttt{x_{dr}}. Applying a deconstructor to the wrong variant signals an ML-style run-time error. To distinguish variants, the \texttt{x_t} function returns \texttt{true} for an instance of the first variant and \texttt{false} for an instance of the second. The \texttt{\tau_i} and \texttt{\tau_r} type expressions can refer to \texttt{l} or other type variables to form recursive or mutually recursive type definitions.

The type of a \texttt{unit} expression is a signature of the form \texttt{sig imports exports \triangleright \tau}, where \texttt{imports} specifies the kinds and types of a unit's imports, \texttt{exports} describes the kinds and types of its exports, and \texttt{\tau} is the type of the unit's initialization expression. In a \texttt{sig} form, as in a \texttt{unit} form, type expressions for variables in \texttt{imports} or \texttt{exports} can use imported and exported types declared within the signature. Type declarations in the signature consist of external–internal name pairs, where the internal name is used in type expressions within the signature and is subject to \texttt{\alpha}-renaming. For

\textsuperscript{10} Although the only kind in this language is \texttt{\Omega}, we declare kinds explicitly in anticipation of future work that handles type constructors.
\[ e = \ldots | e \text{ as } \tau \]

unit-expr = unit import type-mapping* variable-mapping*
export type-mapping* variable-mapping*
\[ \triangleright \tau \]
definitions e

compound-expr = compound import type-mapping* value-var-decl*
export type-mapping* value-var-decl*
\[ \triangleright \tau \]
link e link and e link

invoke-expr = invoke e with type-involve-link* value-involve-link*
definitions = datatype-defn* value-defn*
datatype-defn = type \( t = x, r \rightarrow \tau \mid r, x, r \circ r \)
value-defn = val \( x : \tau = v \)
link = with type-mapping* value-var-decl*
provides type-mapping* value-var-decl*

value-var-decl = y : \tau
type-mapping = s = t :: \kappa
variable-mapping = y = x : \tau
type-involve-link = s = \tau :: \kappa
value-involve-link = y = e : \tau

\( \tau, \sigma = t \mid s \mid \tau_{\text{prim}} \mid \tau \rightarrow \tau \mid \text{signature} \)
signature = sig import type-mapping* value-var-decl*
export type-mapping* value-var-decl*
\[ \triangleright \tau \]

\( t = \text{type variable} \)
\( s = \text{type linking variable} \)
\( \kappa = \text{type kind} \)

Figure 3.21: Syntax for UNITc (constructed types)

\textbf{example,}

sig import \( s = t :: \Omega \) y : t

export
\[ \triangleright t \]

is equivalent to

sig import \( s = t' :: \Omega \) y : t'

export
\[ \triangleright t' \]
because \( l \) is \( \alpha \)-renamed to \( l' \). In contrast, the signature

\[
\text{sig import } s' = l::\Omega \ y: t \\
\text{export} \ \\
\triangleright t
\]

differs from the previous signature, because the external type name \( s \) is not subject to \( \alpha \)-renaming.

\textbf{UNIT} \textsuperscript{e} \textbf{Type Checking}

For economy, we introduce the following unusual abbreviation, which summarizes the content of a signature via the indices on names:

\[
\text{sig}[i, e, b] \equiv \text{sig import } s_i = l_i::\kappa_i \ y_i: \tau_i \\
\text{export} \ s_e = l_e::\kappa_e \ y_e: \tau_e \\
\triangleright \tau_b
\]

Signatures have a subtype relation to allow the use of specialized units in place of more general units. As defined in Figure 3.22, a specific signature \( \tau_s \) is a subtype of a more general signature \( \tau_g \) (\( \tau_s \leq \tau_g \)) if there exists an \( \alpha \)-renaming for each signature such that:

1. the type of the initialization expression in \( \tau_s \) is a subtype of the one in \( \tau_g \);
2. \( \tau_s \) has fewer imports and more exports than \( \tau_g \);
3. for each imported variable in \( \tau_s \), its type in \( \tau_g \) is a subtype of its type in \( \tau_s \); and
4. for each exported variable in \( \tau_g \), its type in \( \tau_s \) is a subtype of its type in \( \tau_g \).

The typing rules for \textbf{UNIT} \textsuperscript{e} are shown in Figures 3.23 and 3.24. These rules are typed extensions of the rules from Section 3.5.1. The special judgement \( \vdash_s \) applies when subsumption is allowed on an expression's type. Subsumption is used carefully to ensure the existence of an algorithm for type checking. For example, subsumption is not allowed for the body of a function expression because the body's type determines the type of the function.

The \text{sig} \textsuperscript{e} typing rule checks the well-formedness of a signature. Each of the type expressions in a signature must be well-formed in an environment containing the
Figure 3.22: Subtyping and subsumption in UNITₜ signatures

\[ \forall y₁ : τ₁ \in y₂ : τ₂, \exists y₂ : τ₂ \quad \text{s.t.} \quad τ₂ \leq τ₁ \]
\[ y₁ : τ₁ \subseteq y₂ : τ₂ \]
\[ \tau_{b₁} \leq \tau_{b₂} \]
\[ \subseteq \]
\[ \text{sig}[i₁, e₁, b₁] \leq \text{sig}[i₂, e₂, b₂] \]
\[ \frac{\tau_1 \leq \tau'_1 \quad \tau'_2 \leq \tau_2}{\tau'_1 \rightarrow \tau'_2 \leq \tau_1 \rightarrow \tau_2} \]
\[ \frac{\Gamma \vdash c : \tau'}{\Gamma \vdash c : \tau} \]

Figure 3.23: Type checking for UNITₜ (Part I)

\[ \Gamma \vdash \tau_{prim} :: \Omega \]
\[ \Gamma \vdash \tau :: \Gamma(t) \]
\[ \Gamma \vdash \tau : \Omega \quad \Gamma \vdash \tau' : \Omega \]
\[ \Gamma \vdash \tau \rightarrow \tau' :: \Omega \]
\[ (\Gamma \cup \Gamma') \cap \text{dom}(\Gamma) = \emptyset \]
\[ \Gamma' = \Gamma, \text{let}_{e_{c₁}} \text{let}_{e_{c₂}} \rightarrow \text{FTV}(\tau_{bₜ}) \cap \tau = \emptyset \]
\[ \text{sig}^\vdash : \]
\[ \frac{\Gamma \vdash \tau_1 :: \kappa_1 \quad \Gamma' \vdash \tau_2 :: \kappa_2 \quad \Gamma' \vdash \tau_b :: \Omega}{\Gamma \vdash \text{sig}[i, e, b] :: \Omega} \]
\[ \Gamma \vdash c :: \text{TypeOf}(c) \]
\[ \Gamma \vdash x :: \Gamma(x) \]
\[ \text{func}^\vdash : \]
\[ \frac{\Gamma \vdash \tau_1 :: \Omega \quad \Gamma, \text{let}_{x_1} \vdash c :: \tau_2}{\Gamma \vdash \text{fun} \ x : \tau_1 \Rightarrow c :: \tau_2} \]
\[ \text{app}^\vdash : \]
\[ \frac{\Gamma \vdash e_1 :: \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash e_1 \ e_2 :: \tau_2} \]
\[ \text{seq}^\vdash : \]
\[ \frac{\Gamma \vdash e_1 :: \tau_1 \quad \Gamma \vdash e_2 :: \tau_2}{\Gamma \vdash e_1 \ e_2 :: \tau_2} \]
\[ \text{generalize}^\vdash : \]
\[ \frac{\Gamma \vdash e :: t}{\Gamma \vdash e :: t} \]
\[ \text{letrec}^\vdash : \]
\[ \frac{\Gamma \vdash \text{letrec} \ \text{type} \ t = x_{e₁} \ x_{e₂} \ τ_1 \ | \ x_{σ \cdot f_₁} \ τ_2 \ o \ x_1 \ \text{val} \ x_{τ} = ν \ \text{in} \ e_b \ \text{as} \ t : t}{\Gamma \vdash \text{letrec} \ \text{type} \ t = x_{e₁} \ x_{e₂} \ τ_1 \ | \ x_{σ \cdot f_₁} \ τ_2 \ o \ x_1 \ \text{val} \ x_{τ} = ν \ \text{in} \ e_b \ \text{as} \ t : τ_b} \]
signature's imported and exported type variables (the internal names), and the type expression for the initialization expression must not refer to any of the exported type variables (because exports are ignored when invoking a unit).

The \texttt{fun}_{\text{c}}^*, \texttt{app}_{\text{c}}^*, \texttt{seq}_{\text{c}}^*$ rules are standard. The \texttt{generalize}_{\text{c}}^* rule states that an expression's type can be generalized when the expression's actual type is a subtype of the target type (so the actual type subsumes the target type).

The \texttt{letrec}_{\text{c}}^* rule checks the local value definitions and body expression for a \texttt{letrec} expression in the context of local type definitions. The result type of the body expression must not depend on any locally-defined types (permitting local type names
\[ v = \ldots \mid \text{inj}(t) \mid \text{inj}(t) \mid \text{proj}(t) \mid \text{proj}(t) \mid \text{test}(t) \mid \text{inj}(t)v \mid \text{inj}(t)v \quad E = \ldots \mid E \text{ as } t \]

\[
\begin{align*}
\text{context}^-_c: \quad & \mathcal{T}. E[v] \rightarrow \mathcal{T}'. E'[v'] \quad \text{if } \mathcal{T}. e \rightarrow \mathcal{T}'. e' \\
\text{variant}^-_c: \quad & \mathcal{T}. E[v] \rightarrow \mathcal{T}' \cdot \text{variant error} \quad \text{if } \mathcal{T}. e \rightarrow \mathcal{T}'. \text{variant error} \\
\text{proj}^-_c: \quad & \mathcal{T}. \text{proj}(t)(\text{inj}(t)v) \rightarrow \mathcal{T}. v \\
\text{proj-fail}^-_c: \quad & \mathcal{T}. \text{proj}(t)(\text{inj}(t)v) \rightarrow \mathcal{T}. \text{variant error} \\
\text{test}^-_c: \quad & \mathcal{T}. \text{test}(t)(\text{inj}(t)v) \rightarrow \mathcal{T}. \text{true} \\
\text{test-fail}^-_c: \quad & \mathcal{T}. \text{test}(t)(\text{inj}(t)v) \rightarrow \mathcal{T}. \text{variant error} \\
\text{app}^-_c: \quad & \mathcal{T}.(\text{fn } z:t \Rightarrow e) v \rightarrow \mathcal{T}. [v/x]e \\
\text{seq}^-_c: \quad & \mathcal{T}. v ; e \rightarrow \mathcal{T}. e \\
\text{generalize}^-_c: \quad & \mathcal{T}. v \text{ as } t \rightarrow \mathcal{T}. v \\
\text{letrec}^-_c: \quad & \mathcal{T} \cdot \text{letrec } \text{val } z:t = v \text{ in } e \rightarrow \mathcal{T}. [\text{letrec } \text{val } z:t = v \text{ in } v/x]e \\
\text{letrec-types}^-_c: \quad & \mathcal{T} \cdot \text{letrec type } t \equiv x_1.x_2 \tau_1 \mid x_3.x_4 \tau_2 \circ x_5 \rightarrow \mathcal{T}[t \mapsto (\tau_1, \tau_2)] \cdot S(\text{letrec } \text{val } z:t = v \text{ in } e) \\
& \quad \text{val } z:t = v \\
& \quad \text{in } e \\
& \text{if } t \cap \text{dom}(\mathcal{T}) = \emptyset \\
& \text{where } S = [\text{inj}(t)/x_1, \text{inj}(t)/x_2, \text{proj}(t)/x_3, \text{proj}(t)/x_4, \text{test}(t)/x_5]
\end{align*}
\]

Figure 3.25: Reduction rules for \textsc{Unit}_c (Part I)

to escape the \texttt{letrec} expression), which means that the set of free type variables in
the expression's type must not intersect with the set of locally-defined type variables.

The \texttt{invoke}^-_c rule checks \texttt{invoke} expressions, first ensuring that the \texttt{with}
clause is well-formed. The first expression in an \texttt{invoke} form must have a signature
type whose imports match the \texttt{with} clause. The exports in the signature are ignored. The
type of the complete \texttt{invoke} expression is the initialization expression's type in the
unit's signature.

The \texttt{unit}^-_c rule determines the signature of a \texttt{unit} expression. The first two lines of
antecedents contain simple context-sensitive syntax checks as in \textsc{Unit}_d. In the third
line, all of the type expressions in the unit are checked in an environment that is
extended with the unit's imported and defined types. Once the type expressions are
validated, the environment is extended again, this time with the types for imported and
defined variables. Finally, the last line of antecedents verifies the types of all
definition expressions and the initialization expression. Subsumption is allowed for
all expressions in the unit, since every expression is explicitly typed. Similar to the
invoke_

\( T \cdot \text{invoke } \sigma_1 = t_1 : \text{unit} \ \ y_1 = x_1 \cdot \tau \)
\[
\begin{array}{c}
\text{export } \sigma_2 = t_2 : \text{unit} \ \ y_2 = x_2 \cdot \tau \\
\text{type } l = x_1 \cdot \text{unit} \ \ t_1 \ | \ x_2 \cdot \text{unit} \ \ t_2 \end{array}
\]
\( \Rightarrow \ T : (\sigma_1 / t_1 ; \sigma_2 / t_2) \]
\( \text{letrec type } l = x_1 ; x_2 ; t_1 | x_2 ; t_2 \ \ \text{val } x_1 ; t_1 = v \ \ \text{in } c_b \)
\( \text{if } \sigma_1 = t_1 ; \sigma_2 = t_2 \ \ \text{and } y_1 = x_1 \ \ \text{and } y_2 = x_2 \)

compound_

\( T \cdot \text{compound } \sigma_1 = t_1 : \text{unit} \ \ y_1 = x_1 \cdot \tau \)
\[
\begin{array}{c}
\text{export } \sigma_1 = t_1 : \text{unit} \ \ y_1 = x_1 \cdot \tau \\
\text{type } l_1 = x_1 \cdot \text{unit} \ \ t_1_1 | x_1 \cdot \text{unit} \ \ t_1_2 \end{array}
\]
body of a letrec expression, the initialization expression within a unit must have a
type that does not depend on any internal or exported types.

The compound₁ rule verifies the linking in a compound expression and determines
its signature. The first four lines of antecedents are simple context-sensitive syntax
checks. The fifth line obtains signatures from the constituent unit expressions. Each
of these signatures must approximate a signature derived from the with and provides
clauses in the corresponding linking line, as specified in the sixth and seventh lines of
antecedents. Finally, the signature of the compound unit is defined by the import
and export clauses and the declared type of the initialization expression.

UNITᵦ Evaluation

A reduction semantics for UNITᵦ must account for the local type definitions intro-
duced by a unit or letrec expression. The rules in Figures 3.25 and 3.26 model
such types through a type store T, where each reduction maps a store-expression
pair T · e to a new store-expression pair T' · e'. A type store T maps each type
l ∈ dom(T) to type expressions η and τ for the type's "left" and "right" variants,
respectively. Intermediate expressions in a reduction include pseudo-variables, such
as injl(t), which correspond to constructors and selectors for the type l. Type check-
ing treats pseudo-variables as type variables that are bound in the environment |T|,
which is the unloading of the type store T to a type environment (see Figure 3.27).

The letrec-typesᵦ⁺ rule reduces a letrec expression containing type definitions to
a letrec expression containing only value definitions. This reduction extends the
type store with the defined types and replaces xₐ with injl(t), etc., within the letrec
eexpression. The other reduction rules for UNITᵦ closely resemble the rules for UNITₐ in
Figure 3.19. Whereas the side conditions for invokeᵦ⁺ and compoundᵦ⁺ in the untyped
semantics enforce safety, the side conditions for invokeᵦ⁺ and compoundᵦ⁺ in the typed
semantics serve merely to require an appropriate $\alpha$-renaming of the units.

UNIT$_c$ Soundness

For a program of type $\tau$, the evaluation rules for UNIT$_c$ produce either a value that has a subtype of $\tau$ or variant error: an evaluation can never get stuck. This property can be formulated as a type soundness theorem.

Theorem 3.5.1 (Soundness) If $[] \vdash e : \tau_0$, then either:

1. $e \uparrow (e$ diverges$)$:

2. $[] \cdot e \longrightarrow^* \mathcal{T} \cdot$ variant error; or

3. $[] \cdot e \longrightarrow^* \mathcal{T} \cdot v, |\mathcal{T}| \vdash v : \tau'_0, \text{ and } \tau'_0 \leq \tau_0$.

Proof. Lemma 3.5.3 (Progress) shows that a non-value expression reduces either to variant error or to another expression. Thus, a reduction for $e$ either never ends, ends in variant error, or ends with a value. In the value case, Lemma 3.5.2 (Subject Reduction) establishes that each step in the evaluation of $e$ preserves the type of $e$. Induction on the number of reductions in $[] \cdot e \longrightarrow^* \mathcal{T} \cdot v$ therefore proves the theorem. $\square$

Lemma 3.5.2 (Subject Reduction) If $\mathcal{T} \cdot e \longrightarrow \mathcal{T}' \cdot e'$ and $|\mathcal{T}| \vdash e : \tau_0$, then $|\mathcal{T}'| \vdash e' : \tau'_0$ and $\tau'_0 \leq \tau_0$.

Proof. The proof is by induction on the structure of $e$. The lemma holds for the base case, $e = v$, since there is no $e'$ such that $\mathcal{T} \cdot e \longrightarrow \mathcal{T}' \cdot e'$. See Appendix B.1 for the complete proof. $\square$

Lemma 3.5.3 (Progress) If $|\mathcal{T}| \vdash e : \tau_0$, then either:

1. $e = v$ for some $v$;

2. $\mathcal{T} \cdot e \longrightarrow \mathcal{T} \cdot$ variant error; or

3. $\mathcal{T} \cdot e \longrightarrow \mathcal{T}' \cdot e'$ for some $\mathcal{T}'$ and $e'$.

Proof. The proof is by induction on the structure of $e$. The lemma holds for the base case where $e$ is a value. We consider all other expression forms and show that a reduction step exists. See Appendix B.2 for the complete proof. $\square$
Polymorphism with $\Lambda$: \[
\text{val } \text{apply} = \\
\Lambda \ a::\Omega \Rightarrow \\
f f : (a \rightarrow a) \Rightarrow (f\ n \ : a \Rightarrow (f\ x))
\]

\(\text{apply[bool] not true}\)

Polymorphism with \texttt{unit}: \[
\text{val } \text{apply} = \\
\texttt{unit\ import}\ a::\Omega \\
\text{export} \\
\triangleright (a \rightarrow a) \rightarrow a \rightarrow a \\
f f : (a \rightarrow a) \Rightarrow (f\ n \ : a \Rightarrow (f\ x))
\]

\((\text{invoke } \text{apply with } a::\Omega \leftarrow \text{bool}) \text{ not true}\)

Figure 3.28: Polymorphism with $\Lambda$ versus \texttt{unit}

\textbf{UNITc and Polymorphism}

Although our UNITc model does not support polymorphic functions directly, units can encode polymorphic functions in a straightforward way. Figure 3.28 shows how \texttt{unit} with \texttt{invoke} supports polymorphic functions in the same manner as $\Lambda$ with type application.

Our UNITc model also omits polymorphic type constructors, and they are not expressible using other constructs. We anticipate no problems in extending UNITc to handle type constructors. Glew and Morrisett [30] describe the extension of a closely-related module language with type constructors.

\textbf{UNITc Implementation}

Closed units in UNITc can be compiled separately in the same way as closed functions in ML. When compiling a unit, imported types are obviously not yet determined and thus have unknown representations. Hence, expressions involving imported types must be compiled like polymorphic functions in ML [52, 81], as suggested by the encoding of polymorphic functions in Section 3.28. Otherwise, the restrictions implied by a unit’s interface allow inter-procedural optimizations within the unit (such as
inlining, specialization, and dead-code elimination). Furthermore, since a compound unit is equivalent to a simple unit that merges its constituent units, intra-unit optimization techniques naturally extend to inter-unit optimizations when a compound expression has known constituent units.

3.5.3 Units with Type Dependencies and Equations

UNITe supports a core language where each type is associated with a distinct and independent constructor, but this view of types is too strict for many languages. For example, in Java, the constructor that instantiates a class depends on the constructor for the superclass. Other languages, such as ML, support type equations that introduce new types without explicit constructors: a type equation of the form \( \text{type } t = \tau \) defines the type variable \( t \) as an abbreviation for the type expression \( \tau \).

Naively mixing units with type dependencies and equations leads to problems. Since two units can contain mutually recursive definitions, linking units with type dependencies may result in cyclic definitions, which core languages like ML and Java do not support. To prevent these cycles, signatures must include information about dependencies between imported and exported types. The dependency information can be used to verify that cyclic definitions are not created in linking expressions.

UNITe extends UNITe with type dependencies and equations. Figure 3.29 defines syntax extensions for UNITe, including a new signature form that contains a \texttt{depends} clause. The dependency declaration \( t_e \sim t_i \) means that an exported type \( t_e \) depends on an imported type \( t_i \). When two units are linked with a \texttt{compound} expression, tracing the set of dependencies can ensure that linking does not create a cyclic type
definition. Also, the signature for a **compound** expression propagates dependency information for types imported into and exported from the compound unit.

**UNITₐ Type Checking**

The following abbreviation expresses a *UNITₐ* signature:

\[
\begin{align*}
\text{sig}[i, e, di, de, b] & \equiv \text{sig import } s_i = l_i : \kappa_i; \ y_i : \tau_i \\
\text{export } s_e = l_e : \kappa_e; \ y_e : \tau_e \\
\text{depends } s_{de} \rightsquigarrow s_{di} & \vdash \tau_b
\end{align*}
\]

The subtyping rule in Figure 3.30 accounts for the new dependency declarations. Specifically, a signature is more specific than another if it declares more dependencies.

The type checking rules for *UNITₐ* are defined in Figure 3.32. To calculate type dependencies, the type checking rules employ the “depends on” relation, \(\propto_D\). It associates a type expression with each of the type variables it references from the set of type equations \(D\):

\[
\tau \propto_D t \iff t \in FTV(\tau) \\
\text{or } (\exists (t' = \tau') \in D \text{ s.t. } t' \in FTV(\tau) \text{ and } \tau' \propto_D t)
\]

\(FTV(\tau)\) denotes the set of type variables in \(\tau\) that are not bound by the **import** or **export** clause of a **sig** type. Type abbreviations are eliminated from a type or expression with the \(|\bullet|_D\) operator, as sketched in Figure 3.31. The subscript is omitted from \(|\bullet|_D\) when \(D\) is apparent from context.
Figure 3.31: Expanding a set of type abbreviations in a type or expression

UNITe Evaluation

Given a type equation of the form `type t = τ`, the variable `t` can be replaced everywhere with `τ` once the complete program is known. Since the type system disallows cyclic type definitions, this expansion of types as abbreviations is guaranteed to terminate. Meanwhile, until the complete program is known, type equations are preserved as necessary. In the rewriting semantics for units, type equations are preserved by linking, and then expanded away by invocation. This semantics formalizes the intuition that type equations constrain how programs are linked, but they have no run-time effect when programs are executed.

The reduction rules for UNITe are nearly the same as the rules for UNITc (see Figures 3.25 and 3.26). As in UNITc, UNITe's invoke and compound reductions propagate type definitions as well as val definitions. In addition, the compound reduction propagates type abbreviations. but the invoke reduction immediately expands all type abbreviations in the invoked unit. A soundness proof for UNITe would also follow closely the proof for UNITc, based on a new lemma that validates the αD replacements.
3.6 Related Work

As already mentioned in Section 3.1, our unit model incorporates ideas from distinct language communities, particularly those using packages and ML-style modules. The Scheme and ML communities have produced a large body of work exploring variations on the standard module system, especially variations for higher-order modules [6, 15, 33, 39, 50, 53, 54, 57, 82]. Duggan and Sourelis [18] have investigated “mixin modules” for specifying recursive and extensible definitions across modules; their approach is
different from ours in its emphasis on extensible datatypes.

Crary, Harper, and Puri [14] model an extension of ML functors that allows mutually recursive procedure and type definitions across functor boundaries. Their work is based on the module calculus of Harper, Mitchell, and Moggi [34]. The calculus distinguishes core and module-level constructs, but also permits higher-order modules, such as functors that consume and produce other functors. Crary et al. thus provide a rigorous theoretical foundation for a form of “recursive modules.” But considerable work remains to determine whether these modules have the properties that are necessary to implement software components.

Glew and Morrisett [30] describe their implementation of MTAL, a linking language for a typed assembly language. MTAL closely resembles a first-order version of \textsc{Unit}_c, where modules are implicitly linked by matching names in a global namespace (like conventional .o linking). The typing issues in MTAL and \textsc{Unit}_c are nearly identical, though somewhat simpler to express in MTAL’s first-order environment.

The Mesa [63] programming language provides a module system that resembles units for a Pascal-like core language. Mesa’s module system includes notions equivalent to signatures, units, and compound units in a linking language that is distinct from the core language. Cardelli [10] anticipated the unit language’s emphasis on module linking as well as module definition. Our unit model is more concrete than his proposal and addresses many of his suggestions for future work. Kelsey’s proposed module system for Scheme [42] captures most of the organizational properties of units, but does not address static typing or dynamic linking.

### 3.7 Summary

Program units deliver both the traditional benefits of modules for separate compilation and the more recent advances of higher-order modules and programmer-controlled linking. Our unit model also addresses the often overlooked—but increasingly important—problem of dynamic linking.

Future work must focus on making units syntactically practical for typed languages. Our text-based model is far too verbose, and we do not address the design of a linking language. Instead, we provide a simple construct for linking units and rely on integration with the core language to build up linking expressions. This integration simplifies our presentation, and we believe it is an essential feature of units. Never-
theless, future research should explore more carefully the implications of integrating the core and module languages.
Chapter 4

Mixins

Class systems provide a simple and flexible mechanism for managing collections of highly parameterized program pieces. Using inheritance and overriding, a programmer derives a new class by specifying only the elements that change in the derived class. Nevertheless, a pure class-based approach suffers from a lack of abstractions that specify uniform extensions and modifications of classes. For example, the construction of a programming environment may require many kinds of text editor frames, including frames that can contain multiple text buffers and frames that support searching. In Java, for example, we cannot implement all combinations of multiple-buffer and searchable frames using derived classes. If we choose to define a class for all multiple-buffer frames, there can be no class that includes only searchable frames. Hence, we must repeat the code that connects a frame to the search engine in at least two branches of the class hierarchy: once for single-buffer searchable frames and again for multiple-buffer searchable frames. If we could instead specify a mapping from editor frame classes to searchable editor frame classes, then the code connecting a frame to the search engine could be abstracted and maintained separately.

Some class-based object-oriented programming languages provide multiple inheritance, which permits a programmer to create a class by extending more than one class at once. A programmer who also follows a particular protocol for such extensions can mimic the use of class-to-class functions. Common Lisp programmers refer to this protocol as mixin programming [43, 45], because it roughly corresponds to mixing in additional ingredients during class creation. Unfortunately, multiple inheritance and its cousins are semantically complex and difficult to understand for programmers.\footnote{Dan Friedman determined in an informal poll in 1996 that almost nobody who teaches C++ teaches multiple inheritance [pers. com.].} As a result, implementing a mixin protocol with multiple inheritance is error-prone and typically avoided.
In this chapter, we present a typed model of "class functors" for Java [31] that permits the direct expression of a mixin protocol to construct a single-inheritance class hierarchy. We refer to the functors as mixins due to their similarity to Common Lisp's multiple inheritance mechanism and Bracha's class operators [8]. Our proposal is superior in that it isolates the useful aspects of multiple inheritance yet retains the simple, intuitive nature of class-oriented programming. In Section 4.1, we develop a calculus of Java classes to serve as a foundation for the calculus of mixins. In Section 4.2, we motivate mixins as an extension of classes using a small but illuminating example, and Section 4.3 extends the type-theoretic model of Java to mixins.

4.1 A Model of Classes

CLASSIC.JAVA is a small but essential subset of sequential Java. To model its type structure and semantics, we use well-known type elaboration and rewriting techniques for Scheme and ML [22, 35, 85]. Figures 4.1 and 4.2 illustrate the essence of our strategy. Type elaboration verifies that a program defines a static tree of classes and a directed acyclic graph (DAG) of interfaces. A type is simply a node in the combined graph. Each type is annotated with its collection of fields and methods, including those inherited from its ancestors.

Evaluation is modeled as a reduction on expression-store pairs in the context of a static type graph. Figure 4.2 demonstrates reduction using a pictorial representation of the store as a graph of objects. Each object in the store is a tagged record of field values, where the tag indicates the class of the object and its field values are references to other objects. A single reduction step may extend the store with a new object, or it may modify a field for an existing object in the store. Dynamic method dispatch is accomplished by matching the class tag of an object in the store with a node in the static class tree; a simple relation on this tree selects an appropriate method for the dispatch.

The class model relies on as few implementation details as possible. For example, the model defines a mathematical relation, rather than a selection algorithm, to associate fields with classes for the purpose of type-checking and evaluation. Similarly, the reduction semantics only assumes that an expression can be partitioned into a proper redex and an (evaluation) context; it does not provide a partitioning algorithm. The model can easily be refined to expose more implementation details [20, 35].
interface Place'
interface Barrier'
interface Door'
    extends Place', Barrier'
...
class Door< extends Object
    implements Door'
        ...
        Room< Enter(Person< p) {... }
    ...
}
class LockedDoor< extends Door< ...
class ShortDoor< extends Door< ...

Figure 4.1: A program determines a static directed acyclic graph of types

Figure 4.2: Given a type graph, reductions map a store-expression pair to a new pair

4.1.1 CLASSICJAVA Programs

The syntax for CLASSICJAVA is shown in Figure 4.3. A program $P$ is a sequence of class and interface definitions followed by an expression. Each class definition consists of a sequence of field declarations and a sequence of method declarations, while an interface consists of methods only. A method body in a class can be abstract, indicating that the method must be overridden in a subclass before the class is instantiated. A method body in an interface must be abstract. As in Java, classes are instantiated with the new operator, but there are no class constructors in Clas-
\[ P = \text{defn}^* \ e \]
\[ \text{defn} = \text{class} \ c \ \text{extends} \ c \ \text{implements} \ i^* \{ \text{field}^* \ \text{meth}^* \} \]
\[ \quad \mid \text{interface} \ i \ \text{extends} \ i^* \{ \ \text{meth}^* \} \]
\[ \text{field} = t \ \text{fd} \]
\[ \text{meth} = t \ \text{md} \ (\ \text{arg}^* \} \{ \ \text{body} \} \]
\[ \text{arg} = t \ \text{var} \]
\[ \text{body} = e \mid \text{abstract} \]
\[ e = \text{new} \ c \mid \text{var} \mid \text{null} \mid e : c . \text{fd} \mid e :: c . \text{fd} = e \]
\[ \mid e . \text{md} (e^*) \mid \text{super} = \text{this} : c . \text{md} (e^*) \]
\[ \mid \text{view} \ t \ e \mid \text{let} \ \text{var} = e \ \text{in} \ e \]
\[ \text{var} = \text{a variable name or this} \]
\[ c = \text{a class name or Object} \]
\[ i = \text{interface name or Empty} \]
\[ \text{fd} = \text{a field name} \]
\[ \text{md} = \text{a method name} \]
\[ t = c \mid i \]

Figure 4.3: CLASSICJAVA syntax: underlined phrases are inserted by elaboration

CLASSICJAVA: instance variables are always initialized to null. Finally, the \textbf{view} and \textbf{let} forms represent Java's casting expressions and local variable bindings, respectively.

The evaluation rules for CLASSICJAVA are defined in terms of individual expressions, but certain rules require information about the context of the expression in the original program. For example, the evaluation rule for a field use depends on the syntactic type of the object position, which is determined by the expression's type environment in the original program. To remove such context dependencies before evaluation, the type-checker annotates field uses and \textbf{super} invocations with extra source-context information (see the underlined parts of the syntax).

A valid CLASSICJAVA program satisfies a number of simple predicates and relations: these are described in Figures 4.4 and 4.5. For example, the \textbf{ClassesOnce}(P) predicate states that each class name is defined at most once in the program \( P \). The relation \( \prec_p \) associates each class name in \( P \) to the class it extends, and the (overloaded) \( \in_p \) relations capture the field and method declarations of \( P \).

The syntax-summarizing relations induce a second set of relations and predicates that summarize the class structure of a program. The first of these is the subclass relation \( \leq_p \), which is a partial order if the \textbf{CompleteClasses}(P) predicate holds and the \textbf{WellFoundedClasses}(P) predicate holds. In this case, the classes declared in
The sets of names for variables, classes, interfaces, fields, and methods are assumed to be mutually distinct. The meta-variable $T$ is used for method signatures $(t \ldots \rightarrow t)$, $V$ for variable lists $(\mbox{var} \ldots)$, and $\Gamma$ for environments mapping variables to types. Ellipses on the baseline $\ldots$ indicate a repeated pattern or continued sequence, while centered ellipses $\cdots$ indicate arbitrary missing program text (not spanning a class or interface definition).

\textbf{ClassesOnce}(P)
Each class name is declared only once
\begin{align*}
\text{class } c & \cdots \text{class } c' \ldots \text{is in } P \Rightarrow c \neq c' \\
\end{align*}

\textbf{FieldOncePerClass}(P)
Field names in each class declaration are unique
\begin{align*}
\text{class } \cdots \{ \cdots \text{fd} \cdots \text{fd}' \cdots \} \text{ is in } P \Rightarrow \text{fd} \neq \text{fd}' \\
\end{align*}

\textbf{MethodOncePerClass}(P)
Method names in each class declaration are unique
\begin{align*}
\text{class } \cdots \{ \cdots \text{md} \{ \cdots \} \cdots \text{md}' \{ \cdots \} \cdots \} \text{ is in } P \Rightarrow \text{md} \neq \text{md}' \\
\end{align*}

\textbf{InterfacesOnce}(P)
Each interface name is declared only once
\begin{align*}
\text{interface } i & \cdots \text{interface } i' \cdots \text{is in } P \Rightarrow i \neq i' \\
\end{align*}

\textbf{InterfacesAbstract}(P)
Method declarations in an interface are abstract
\begin{align*}
\text{interface } \cdots \{ \cdots \text{md} \{ \cdots \} \{e\} \cdots \} \text{ is in } P \Rightarrow e \text{ is abstract} \\
\end{align*}

\textbf{MethodArgsDistinct}(P)
Each method argument name is unique
\begin{align*}
\text{Class is declared as an immediate subclass} \\
\text{c } \ni P \Rightarrow \text{c extends c' } \cdots \{ \cdots \} \text{ is in } P \\
\end{align*}

\textbf{FIELD}(P)
Field is declared in a class
\begin{align*}
(c, \text{fd}, t) \ni P \Rightarrow \text{c extends c } \cdots \{ \cdots \text{t fd} \cdots \} \text{ is in } P \\
\end{align*}

\textbf{INTERFACE}(P)
Interface is declared as an immediate subinterface
\begin{align*}
i \ni P \Rightarrow \text{i extends i' } \cdots \{ \cdots \} \text{ is in } P \\
\end{align*}

\textbf{IMPLEMENTATION}(P)
Class declares implementation of an interface
\begin{align*}
c \ni P \Rightarrow \text{c implements i } \cdots \{ \cdots \} \text{ is in } P \\
\end{align*}

\textbf{COMPLETEClasses}(P)
Classes that are extended are defined
\begin{align*}
rng(\ni P) \subseteq \text{dom}(\ni P) \cup \{\text{Object}\} \\
\end{align*}

\textbf{WELLFoundedClasses}(P)
Class hierarchy is an order
\begin{align*}
\ni P \text{ is antisymmetric} \\
\end{align*}

\textbf{CLASSMETHODSOK}(P)
Method overriding preserves the type
\begin{align*}
((\text{md}, T, V, e) \ni P \Rightarrow \text{c and } (\text{md}, T', V', e') \ni P \Rightarrow \text{c'}) \Rightarrow (T = T' \text{ or } c \ni P, c') \\
\end{align*}

\textbf{FIELD}(P)
Field is contained in a class
\begin{align*}
(c', \text{fd}, t) \ni P \Rightarrow \text{c' and e' = min\{e'' | c \ni P, c'' and 3t' s.t. (c'', fd, t') \ni P, c''\}} \\
\end{align*}

\textbf{IMPLEMENTATION}(P)
Class declares implementation of an interface
\begin{align*}
((\text{md}, T, V, e) \ni P \Rightarrow \text{c' and c' = min\{e'' | c \ni P, c'' and 3\exists e', V' s.t. (md, T, V', e') \ni P, c''\}}) \\
\end{align*}

Figure 4.4 : Predicates and relations in the model of CLASSICJAVA (Part I)
$\leq_P$ Interface is a subinterface

$\leq_P \equiv$ the transitive, reflexive closure of $\prec_P$

COMPLETEINTERFACES($P$) Extended/implemented interfaces are defined
rng($\prec_P$) $\cup$ rng($\prec_P$) $\subseteq$ dom($\prec_P$) $\cup$ {Empty}

WELLFOUNDEDINTERFACES($P$) Interface hierarchy is an order

$\preceq_P$ Class implements an interface $c \preceq_P i \iff \exists c', i' \text{ s.t. } c \preceq_P c' \text{ and } i' \preceq_P i$ and $c' \prec_P i'$

INTERFACEMETHODSOK($P$) Interface inheritance or redeclaration of methods is consistent
$(md, T', V', \text{abstract}) \in_P i$ and $(md, T'', V'', \text{abstract}) \in_P i''$

$\Rightarrow \ (T = T' \text{ or } \forall i''(i'' \in_P i \text{ or } i'' \in_P i''))$

Method is contained in an interface $(md, T, V, \text{abstract}) \in_P i \Rightarrow \exists i' \text{ s.t. } i \leq_P i'$ and $(md, T, V, \text{abstract}) \in_P i'$

$\in_P$ Classes supply methods to implement interfaces $c \prec_P i \Rightarrow (\forall md, T, V) (md, T, V, \text{abstract}) \in_P i \Rightarrow (\exists, V' \text{ s.t. } (md, T, V', e) \in_P c)$

NOABSTRACTMETHODS($P, c$) Class has no abstract methods (can be instantiated)
$(md, T, V, \text{abstract}) \not\in_P c \Rightarrow c \in_P c \Rightarrow c \not\in \text{abstract}$

$\leq_P$ Type is a subtype $\leq_P \equiv \leq_P \cup \prec_P$

$\in_P$ Field or method is in a type $\in_P \equiv \in_P \cup \in_P$

Figure 4.5: Predicates and relations in the model of CLASSICJAVA (Part II)

$P$ form a tree that has Object at its root.

If the program describes a tree of classes, we can "decorate" each class in the tree with the collection of fields and methods that it accumulates from local declarations and inheritance. The source declaration of any field or method in a class can be computed by finding the minimum (i.e., farthest from the root) superclass that declares the field or method. This algorithm is described precisely by the $\in_P$ relations. The $\in_P$ relation retains information about the source class of each field, but it does not retain the source class for a method. This reflects the property of Java classes that fields cannot be overridden (so instances of a subclass always contain the field), while methods can be overridden (and may become inaccessible).

Interfaces have a similar set of relations. The superinterface declaration relation $\prec_P$ induces a subinterface relation $\leq_P$. Unlike classes, a single interface can have multiple proper superinterfaces, so the subinterface order forms a DAG instead of a tree. The set methods of an interface, as described by $\in_P$, is the union of the interface’s declared methods and the methods of its superinterfaces.

Finally, classes and interfaces are related by implements declarations, as captured in the $\prec_P$ relation. This relation is a set of edges joining the class tree and
the interface graph, completing the subtype picture of a program. A type in the full
graph is a subtype of all of its ancestors.

4.1.2 CLASSICJAVA Type Elaboration

The type elaboration rules for CLASSICJAVA are defined by the following judgements:

\[
\begin{align*}
\Gamma &\vdash \Delta P \Rightarrow P' : t & \text{P elaborates to } P' \text{ with type } t \\
\Gamma &\vdash \Delta \text{defn} \Rightarrow \text{defn}' & \text{defn elaborates to } \text{defn}' \\
\Gamma &\vdash \Delta \text{meth} \Rightarrow \text{meth}' & \text{meth in } t \text{ elaborates to } \text{meth}' \\
\Gamma &\vdash \Delta \text{e} \Rightarrow \text{e}' : t & \text{e elaborates to } \text{e}' \text{ with type } t \text{ in } \Gamma \\
\Gamma &\vdash \Delta \text{e} \Rightarrow \text{e}' : t & \text{e has type } t \text{ using subsumption in } \Gamma \\
\Gamma &\vdash \Delta \text{t} & t \text{ exists}
\end{align*}
\]

The type elaboration rules translate expressions that access a field or call a super
method into annotated expressions (see the underlined parts of Figure 4.3). For
field uses, the annotated expression contains the compile-time type of the instance
expression, which determines the class containing the declaration of the accessed field.
For super method invocations, the annotated expression contains the compile-time
type of this, which determines the class that contains the declaration of the method
to be invoked.

The complete typing rules are shown in Figures 4.6 and 4.7. A program is well-
typed if its class definitions and final expression are well-typed. A definition, in turn,
is well-typed when its field and method declarations use legal types and the method
body expressions are well-typed. Finally, expressions are typed and elaborated in the
context of an environment that binds free variables to types. For example, the getc
and setc rules for fields first determine the type of the instance expression, and then
calculate a class-tagged field name using eP; this yields both the type of the field
and the class for the installed annotation. In the setc rule, the right-hand side of the
assignment must match the type of the field, but this match may exploit subsumption
to coerce the type of the value to a supertype. The other expression typing rules are
similarly intuitive.

4.1.3 CLASSICJAVA Evaluation

The operational semantics for CLASSICJAVA is defined as a contextual rewriting sys-
tem on pairs of expressions and stores. A store S is a mapping from objects to
class-tagged field records. A field record \( F \) is a mapping from elaborated field names to values. The evaluation rules are a straightforward modification of those for imperative Scheme [22].

The complete evaluation rules are in Figure 4.8. For example, the call rule invokes a method by rewriting the method call expression to the body of the invoked method, syntactically replacing argument variables in this expression with the supplied argument values. The dynamic aspect of method calls is implemented by selecting the method based on the run-time type of the object (in the store). In contrast, the super reduction performs super method selection using the class annotation that is statically determined by the type-checker.
Figure 4.7: Context-sensitive checks and type rules for CLASSICJAVA (Part II)

4.1.4 CLASSICJAVA Soundness

For a program of type $t$, the evaluation rules for CLASSICJAVA produce either a value that has a subtype of $t$, or one of two errors. Put differently, an evaluation cannot go wrong, which our model model models means getting stuck. This property can be formulated as a type soundness theorem.

**Theorem 4.1.1 (Type Soundness)** If $\vdash_p P \Rightarrow P': t$ and

- $P' \vdash (\epsilon, \emptyset) \leftrightarrow^* \langle \text{object}, S \rangle$ and $S(\text{object}) = \langle t', \mathcal{F} \rangle$ and $t' \leq_P t$: or

- $P' \vdash (\epsilon, \emptyset) \leftrightarrow^* \langle \text{null}, S \rangle$: or

- $P' \vdash (\epsilon, \emptyset) \leftrightarrow^* \langle \text{error: bad cast}, S \rangle$: or

- $P' \vdash (\epsilon, \emptyset) \leftrightarrow^* \langle \text{error: dereferenced null}, S \rangle$.

The main lemma in support of this theorem states that each step taken in the evaluation preserves the type correctness of the expression-store pair (relative to the
\[
E = \begin{cases} [ \text{object} ] & \text{if } \mathbf{e} = \text{object} \\ [ \text{null} ] & \text{if } \mathbf{v} = \text{null} \\ \mathbf{E} & \text{if } \mathbf{E} = \mathbf{E} \end{cases}
\]

\[
\begin{align*}
\mathbf{e} &= \ldots | \text{object} \\
\mathbf{v} &= \text{object} | \text{null}
\end{align*}
\]

\[
P \vdash (\mathbf{E[new, c]}, S) \leftrightarrow (\mathbf{E[object]}, S[\text{object} \rightarrow (c, F)]) \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \q
Since the rewriting rules reduce annotated terms, we derive new type judgements $\Gamma \vdash \lambda x$ and $\Gamma \vdash \lambda e$ that relate annotated terms to show that reductions preserve type correctness. Each of the new rules performs the same checks as the rule it is derived from without removing or adding annotation. Thus, $\Gamma \vdash \lambda x$ is derived from $\Gamma \vdash x$, and so forth.

The judgement on view expressions is altered slightly: we retain the view operation in all cases, and we collapse the $[\text{wcast}^c]$ and $[\text{ncast}^c]$ relations to a new $[\text{cast}^c]$ relation that permits any casting operation:

$$
\frac{P, \Gamma \vdash \lambda e : t'}{P, \Gamma \vdash \text{view} \ t \ e : t'}^{[\text{cast}^c]}
$$

The new $[\text{cast}^c]$ relation lets us prove that every intermediate expression in a reduction is well-typed, whereas $[\text{wcast}^c]$ and $[\text{ncast}^c]$ more closely approximate Java, which rejects certain expressions because they would certainly produce error: bad cast. For example, assuming that LockedDoor$^c$ and ShortDoor$^c$ extend Door$^c$ separately, a legal source program

```
let x = o.GetDoor() in (view LockedDoor$^c$ x)
```

might reduce to

```
view LockedDoor$^c$ shortDoorObject
```

where shortDoorObject is an instance of ShortDoor$^c$. Unlike the $[\text{wcast}^c]$ and $[\text{ncast}^c]$ rules in $\Gamma \vdash \lambda x$, the $[\text{cast}^c]$ rule in $\Gamma \vdash \lambda e$ assigns a type to the reduced expression.

The $\Gamma \vdash \lambda x$ relation also generalizes the $[\text{super}^c]$ judgement to allow an arbitrary expression within a super expression's annotation (in place of this). The generalized judgement permits replacement and substitution lemmas that treat super annotations in the same manner as other expression. Nevertheless, at each reduction step, every super expression's annotation contains either this or an object. This fact is crucial to proving the soundness of CLASSICJAVA, so we formalize it as a SUPEROK predicate.

$^{2}$In $\Sigma_5$, it would be wrong to write $\text{dom}(\Gamma) \subseteq \text{dom}(S)$ because $\Gamma$ may contain bindings for lexical variables.
Definition 4.1.3 (Well-Formed Super Calls)

\( \text{SUPEROK}(e) \Leftrightarrow \) For all \( \text{super} \equiv e_0 : c . \text{md}(e_1, \ldots, e_n) \) in \( e \).

either \( e_0 = \text{this} \) or \( e_0 = \text{object for some object} \).

Although \( \vdash_{\varphi} \) types more expressions than \( \vdash_{\sigma} \), we are only concerned with source expressions typed by \( \vdash_{\sigma} \). The following lemma establishes that the new typing judgements conserve the result of the original typing judgements.

Lemma 4.1.4 (Conserve) If \( \vdash_{\rho} P \Rightarrow P' : t \) and \( P' = \text{defn}_1 \ldots \text{defn}_n e. \) then \( P'.\emptyset \vdash_{\varphi} e' : t \).

Proof. The claim follows from a copy-and-patch argument. \( \square \)

Lemma 4.1.5 (Subject Reduction) If \( P, \Gamma \vdash_{\varphi} e : t, P, \Gamma \vdash_{\sigma} S, \) \( \text{SUPEROK}(e) \), and \( P \vdash (e,S) \Rightarrow (e',S') \), then \( e' \) is an error configuration or there exists a \( \Gamma' \) such that

1. \( P,\Gamma' \vdash_{\varphi} e' : t \).

2. \( P,\Gamma' \vdash_{\sigma} S' \), and

3. \( \text{SUPEROK}(e') \).

Proof. The proof examines reduction steps. For each case, if execution has not halted with an error configuration, we construct the new environment \( \Gamma' \) and show that the two consequents of the theorem are satisfied relative to the new expression, store, and environment. See Appendix C.1 for the complete proof, which is due to Shriram Krishnamurthi. \( \square \)

Lemma 4.1.6 (Progress) If \( P, \Gamma \vdash_{\varphi} e : t, P, \Gamma \vdash_{\sigma} S, \) and \( \text{SUPEROK}(e) \), then either \( e \) is a value or there exists an \( (e',S') \) such that \( P \vdash (e,S) \Rightarrow (e',S') \).

Proof. The proof is by analysis of the possible cases for the current redex in \( e \) (in the case that \( e \) is not a value). See Appendix C.2 for the complete proof. \( \square \)

By combining the Subject Reduction and Progress lemmas, we can prove that every non-value \text{CLASSICJAVA} program reduces while preserving its type, thus establishing the soundness of \text{CLASSICJAVA}. 
4.1.5 Related Work on Classes

Our model for class-based object-oriented languages is similar to two recently published semantics for Java [16, 78], but entirely motivated by prior work on Scheme and ML models [22, 35, 85]. The approach is fundamentally different from most of the previous work on the semantics of objects. Much of that work has focused on interpreting object systems and the underlying mechanisms via record extensions of lambda calculi [19, 41, 66, 58, 67] or as "native" object calculi (with a record flavor) [1, 2, 3]. In our semantics, types are simply the names of entities declared in the program: the collection of types forms a DAG, which is specified by the programmer. The collection of types is static during evaluation\(^3\) and is only used for field and method lookups and casts. The evaluation rules describe how to transform statements, formed over the given type context, into plain values. The rules work on plain program text such that each intermediate stage of the evaluation is a complete program. In short, the model is as simple and intuitive as that of first-order functional programming enriched with a language for expressing hierarchical relationships among data types.

4.2 From Classes to Mixins: An Example

Implementing a maze adventure game [29, page 81] illustrates the need for adding mixins to a class-based language. A player in the adventure game wanders through rooms and doors in a virtual world. All locations in the virtual world share some common behavior, but also differ in a wide variety of properties that make the game interesting. For example, there are many kinds of doors, including locked doors, magic doors, doors of varying heights, and doors that combine several varieties into one. The natural class-based approach for implementing different kinds of doors is to implement each variation with a new subclass of a basic door class, Door\(^c\). The left side of Figure 4.9 shows the Java definition for two simple Door\(^c\) subclasses, LockedDoor\(^c\) and ShortDoor\(^c\). An instance of LockedDoor\(^c\) requires a key to open the door, while an instance of ShortDoor\(^c\) requires the player to duck before walking through the door.

A subclassing approach to the implementation of doors seems natural at first, because the programmer declares only what is different in a particular door variation

---

\(^3\)Dynamic class loading could be expressed in this framework as an addition to the static context. Nevertheless, the context remains the same for most of the evaluation.
as compared to some other door variation. Unfortunately, since the superclass of each variation is fixed, door variations cannot be composed into more complex, and thus more interesting, variations. For example, the LockedDoor<sup>C</sup> and ShortDoor<sup>C</sup> classes cannot be combined to create a new LockedShortDoor<sup>C</sup> class for doors that are both locked and short.

```java
class LockedDoor<sup>C</sup> extends Door<sup>C</sup> {
    boolean canOpen(Person<sup>C</sup> p) {
        if (!p.hasItem(theKey)) {
            System.out.println("You don't have the Key");
            return false;
        }
        System.out.println("Using key...");
        return super.canOpen(p);
    }
}

class ShortDoor<sup>C</sup> extends Door<sup>C</sup> {
    boolean canPass(Person<sup>C</sup> p) {
        if (p.height() > l) {
            System.out.println("You are too tall");
            return false;
        }
        System.out.println("Ducking into door...");
        return super.canPass(p);
    }
}

/* Cannot merge for LockedShortDoor<sup>C</sup> */

interface Door<sup>I</sup> {
    boolean canOpen(Person<sup>I</sup> p);
    boolean canPass(Person<sup>I</sup> p);
}

mixin Locked<sup>I</sup> extends Door<sup>I</sup> {
    boolean canOpen(Person<sup>I</sup> p) {
        if (!p.hasItem(theKey)) {
            System.out.println("You don't have the Key");
            return false;
        }
        System.out.println("Using key...");
        return super.canOpen(p);
    }
}

mixin Short<sup>I</sup> extends Door<sup>I</sup> {
    boolean canPass(Person<sup>I</sup> p) {
        if (p.height() > l) {
            System.out.println("You are too tall");
            return false;
        }
        System.out.println("Ducking into door...");
        return super.canPass(p);
    }
}

class LockedDoor<sup>C</sup> = Locked<sup>I</sup>(Door<sup>C</sup>);
class ShortDoor<sup>C</sup> = Short<sup>I</sup>(Door<sup>C</sup>);
class LockedShortDoor<sup>C</sup> = Locked<sup>I</sup>(Short<sup>I</sup>(Door<sup>C</sup>));
```

Figure 4.9: Some class definitions and their translation to composable mixins

A mixin approach solves this problem. Using mixins, the programmer declares how a particular door variation differs from an arbitrary door variation. This creates a function from door classes to door classes, using an interface as the input type. Each basic door variation is defined as a separate mixin. These mixins are then functionally composed to create many different kinds of doors.

A programmer implements mixins in exactly the same way as a derived class, except that the programmer cannot rely on the implementation of the mixin’s superclass, only on its interface. We consider this an advantage of mixins because it enforces the maxim “program to an interface, not an implementation” [29, page 11].

The right side of Figure 4.9 shows how to define mixins for locked and short doors.
interface Secure extends Door {
    Object neededItem();
}

mixin Secure extends Door implements Secure {
    Object neededItem() { return null; }
    boolean canOpen(Person p) {
        Object item = neededItem();
        if (lp.hasItem(item)) {
            System.out.println("You don't have the " + item);
            return false;
        }
        System.out.println("Using " + item + ",...");
        return super.canOpen(p);
    }
}

mixin NeedsKey extends Secure {
    Object neededItem() {
        return theKey;
    }
}

mixin NeedsSpell extends Secure {
    Object neededItem() {
        return theSpellBook;
    }
}

mixin Locked = NeedsKey compose Secure;

mixin Magic = NeedsSpell compose Secure;

mixin LockedMagic = Locked compose Magic;

mixin LockedMagicDoor = LockedMagic compose Door;
class LockedDoor = Locked(Door); ...
This mixin has the expected behavior: to open an instance of LockedMagic, the player must have both the key and the book of spells. This combinational effect is achieved by a chain of super.canOpen() calls that use distinct, non-interfering versions of neededItem. The neededItem declarations of Locked and Magic do not interfere with each other because the interface extended by Locked is Door, which does not contain neededItem. In contrast, Door does contain canOpen, so the canOpen method in Locked overrides and chains to the canOpen in Magic.

4.3 Mixins for Java

MIXEDJAVA is an extension of CLASSICJAVA with mixins. In CLASSICJAVA, a class is assembled as a chain of class expressions. Specifically, the content of a class is defined by its immediate field and method declarations and by the declarations of its superclasses, up to Object. In MIXEDJAVA, a “class” is assembled by composing a chain of mixins. The content of the class is defined by the field and method declarations in the entire chain.

MIXEDJAVA provides two kinds of mixins:

- An atomic mixin declaration is similar to a class declaration. An atomic mixin declares a set of fields and methods that are extensions to some inherited set of fields and methods. In contrast to a class, an atomic mixin specifies its inheritance with an inheritance interface, not a static connection to an existing class. By abuse of terminology, we say that a mixin extends its inheritance interface.

A mixin’s inheritance interface determines how method declarations within the mixin are combined with inherited methods. If a mixin declares a method that is not contained in its inheritance interface, then that declaration never overrides another.

An atomic mixin implements one or more interfaces as specified in the mixin’s definition. In addition, a mixin always implements its inheritance interface.

---

4We use boldfaced class to refer to the content of a single class expression, as opposed to an actual class.
• A composite mixin does not declare any new fields or methods. Instead, it composes two existing mixins to create a new mixin. The new composite mixin has all of the fields and methods of its two constituent mixins. Method declarations in the left-hand mixin override declarations in the right-hand mixin according to the left-hand mixin’s inheritance interface. Composition is allowed only when the right-hand mixin implements the left-hand mixin’s inheritance interface.

A composite mixin extends the inheritance interface of its right-hand constituent, and it implements all of the interfaces that are implemented by its constituents. Composite mixins can be composed with other mixins, producing arbitrarily long chains of atomic mixin compositions.⁵

Figure 4.11 illustrates how the mixin LockedMagicDoorᵐ from the previous section corresponds to a chain of atomic mixins. The arrows connecting the tops of the boxes represent mixin compositions: in each composition, the inheritance interface for the left-hand side is noted above the arrow. The other arrows show how method declarations in each mixin override declarations in other mixins according to the composition interfaces. For example, there is no arrow from the first Secureᵐ’s neededItem to Magicᵐ’s method because neededItem is not included in the Door¹ interface. The canOpen method is in both Door¹ and Secure¹, so arrows connect all declarations of canOpen.

Mixins completely subsume the role of classes. A mixin can be instantiated with new when the mixin does not inherit any services. In MIXEDJAVA, this is indicated by declaring that the mixin extends the special interface Empty. Consequently, we omit classes from our model of mixins, even though a realistic language would include both mixins and classes.

The following subsections present a precise description of MIXEDJAVA. Section 4.3.1 describes the syntax and type structure of MIXEDJAVA programs, followed by the type elaboration rules in Section 4.3.2. Section 4.3.3 explains the operational

⁵Our composition operator is associative semantically, but not type-theoretically. The type system could be strengthened to make composition associative—giving MIXEDJAVA a categorical flavor—by letting each mixin declare a set of interfaces for inheritance, rather than a single interface. Each required interface must then either be satisfied or propagated by a composition. We have not encountered a practical use for the extended type system.
Figure 4.11: **LockedMagicDoor** mixin corresponds to a sequence of atomic mixins

semantics of MIXEDJAVA, which is significantly different from that of CLASSICJAVA. Section 4.3.4 presents a type soundness theorem, Section 4.3.5 briefly considers implementation issues, and Section 4.3.6 discusses related work.

### 4.3.1 MIXEDJAVA Programs

Figure 4.12 contains the syntax for MIXEDJAVA; the missing productions are inherited from the grammar of CLASSICJAVA in Figure 4.3. The primary change to the syntax is the replacement of class declarations with mixin declarations. Another change concerns the annotations added by type elaboration. First, view expressions are annotated with the syntactic type of the object expression. Second, a type is no longer included in the super annotation or the field use annotations. In addition, type elaboration inserts extra view expressions into a program to implement subsumption.

```
defn = mixin m extends i implements i* { field* meth* } 
    | mixin m = m compose m 
    | interface i extends i* { meth* } 
    | new m | var | null | e.fd | e.fd = e 
    | e.md (e*) | super == this . md (e*) 
    | view t as t e | let var = e in e 

e = mixin name 

m = mixin name 

Figure 4.12: Syntax extensions for MIXEDJAVA
```

The predicates and relations in Figures 4.13 and 4.14 (along with the interface-specific parts of Figures 4.4 and 4.5) summarize the syntactic content of a MIXEDJAVA
**Figure 4.13**: Predicates and relations in the model of MIXEDJAVA (Part I)
Figure 4.14: Predicates and relations in the model of MIXEDJAVA (Part II)

A well-formed program induces a subtype relation $\leq_P$ on its mixins such that a composite mixin is a subtype of each of its constituent mixins.

Since each composite mixin has two supertypes, the type graph for mixins is a DAG, rather than a tree as for classes. This DAG would result in ambiguities if subsumption were based on subtypes. For example, LockedMagic$^m$ is a subtype of Secure$^m$, but it contains two copies of Secure$^m$ (see Figure 4.11). Hence, interpreting an instance of LockedMagic$^m$ as an instance of Secure$^m$ is ambiguous. More concretely, the fragment...
LockedMagicDoor\textsuperscript{m} door = new LockedMagicDoor\textsuperscript{m}:
(view Secure\textsuperscript{m} door).neededItem();

is ill-formed because LockedMagic\textsuperscript{m} is not uniquely viewable as Secure\textsuperscript{m}. To eliminate such ambiguities, we introduce the "viewable as" relation $\leq_P$, which is a restriction on the subtype relation. Subsumption is thus based on $\leq_P$ rather than $\leq_P$. The relations $\in_P$, which collect the fields and methods contained in each mixin, similarly eliminate ambiguities.

### 4.3.2 MIXEDJAVA Type Elaboration

Despite the replacement of the subtype relation with the "viewable as" relation, CLASSICJAVA's type elaboration strategy applies equally well for typing MIXEDJAVA. The typing rules in Figure 4.15 are combined with the defn\textsuperscript{i}, meth, let, var, null, and abs rules from Figures 4.6 and 4.7.

Three of the new rules deserve special attention. First, the super\textsuperscript{m} rule allows a super call only when the method is declared in the current mixin's inheritance interface, where the current mixin is determined by looking at the type of this. Second, the wcast\textsuperscript{m} rule strips out the view part of the expression and delegates all work to the subsumption rules. Third, the sub\textsuperscript{m} rule for subsumption inserts a view operator to make subsumption coercions explicit.

### 4.3.3 MIXEDJAVA Evaluation

The operational semantics for MIXEDJAVA differs substantially from that of CLASSICJAVA. The rewriting semantics of the latter relies on the uniqueness of each method name in the chain of classes associated with an object. This uniqueness is not guaranteed for chains of mixins. Specifically, a composition $m_1$ compose $m_2$ contains two methods named $x$ if both $m_1$ and $m_2$ declare $x$ and $m_1$'s inheritance interface does not contain $x$. Both $x$ methods are accessible in an instance of the composite mixin since the object can be viewed specifically as an instance of either $m_1$ or $m_2$.

One strategy to avoid the duplication of $x$ is to rename it in $m_1$ and $m_2$. At best, this is a global transformation on the program, since $x$ is visible to the entire program as a public method. At worst, renaming triggers an exponential explosion in the size of the program, which occurs when $m_1$ and $m_2$ are actually the same mixin $m$. Since
the mixin \( m \) represents a type, renaming \( x \) in each use of \( m \) splits it into two different types, which requires type-splitting at every expression in the program involving \( m \).

Our MIXEDJAVA semantics handles the duplication of method names with runtime context information: the current view of an object.\(^6\) During evaluation, each reference to an object is bundled with its view of the object, so that values are of

\(^6\)A view is analogous to a "subobject" in languages with multiple inheritance, but without the complexity of shared superclasses [73].
the form \( \langle \text{object} \parallel \text{view} \rangle \). A reference's view can be changed by subsumption, method calls, or explicit casts.

Each view is represented as a chain of mixins. The chain is always a sub-chain of the object's full chain of mixins, i.e., the chain of mixins for the object's instantiation type. For example, when an instance of \texttt{LockedMagicDoor} is used as a \texttt{Magic} instance, the object's view corresponds to the boxed part of the following chain:

\[
[\texttt{NeedsKey}^{m} \texttt{Secure}^{m} [\texttt{NeedsSpell}^{m} \texttt{Secure}^{m}] \texttt{Door}^{m}]
\]

The full chain corresponds to \texttt{LockedMagicDoor} and the boxed part corresponds to \texttt{Magic}*. The view designates a specific point in the full mixin chain for selecting methods during dynamic dispatch. With the above view, a search for the \texttt{neededItem} method of the object begins in the \texttt{NeedsSpell} element of the chain.

Our notation for views exploits the fact that an object in \texttt{MIXEDJAVA} encodes its full chain of mixins (in the same way that an object in \texttt{CLASSICJAVA} encodes its class). Thus, the part of the chain before the box is not needed to describe the view:

\[
[\texttt{NeedsSpell}^{m} \texttt{Secure}^{m}] \texttt{Door}^{m}
\]

Furthermore, since the view is always at the start of the remaining chain, we can replace the box with the name of the type it represents, which provides a purely textual notation for views:

\[
[\texttt{NeedsSpell}^{m} \texttt{Secure}^{m} \texttt{Door}^{m}] / \texttt{Magic}^{n}.
\]

The view-based dispatching algorithm, described by the \( \in \beta \) relation, proceeds in two phases. The first phase of a search for method \( x \) locates the \textit{base declaration} of \( x \), which is the unique non-overriding declaration of \( x \) that is visible in the current view. This declaration is found by traversing the view from left to right, using the inheritance interface at each step as a guide for the next step (via the \( \prec \) and \( \triangleright \) relations). When the search reaches a mixin whose inheritance interface does not include \( x \), the base declaration of \( x \) has been found. But the base declaration is not the destination of the dispatch; the destination is determined by the second phase, which locates an overriding declaration of \( x \) that is contained in the object's

---

7 We could also use numeric position pairs to denote sub-chains, but the tail/type encoding works better for defining the operational semantics and soundness of \texttt{MIXEDJAVA}.
instantiated mixin. Among the declarations that override the base declaration, the leftmost declaration is selected as the destination, following customary overriding conventions. The location of the overriding declaration determines both the method definition that is invoked and the view of the object within the destination method body (i.e., the view for `this`).

The dispatching algorithm explains how `Secure^m`'s `canOpen` method calls the appropriate `neededItem` method in an instance of `LockedMagicDoor^m`, sometimes dispatching to the method in `NeedsKey^m` and sometimes to the one in `NeedsSpell^m`. The following example illustrates the essence of dispatching from `Secure^m`'s `canOpen`:

```java
Object canOpen(Secure^m o) { ... o.neededItem() ... }
```

```java
let door = new LockedMagicDoor^m
in canOpen(view Secure^m view Locked^m door) ...
            canOpen(view Secure^m view Magic^m door)
```

The `new LockedMagicDoor^m` expression produces `door` as an `{object||view}` pair, where `object` is a new object in the store and `view` is (recall Figure 4.11)

`[NeedsKey^m Secure^m NeedsSpell^m Secure^m Door^m]/LockedMagicDoor^m`.

The `view` expressions shift the view part of `door`. Thus, for the first call to `canOpen`, `o` is replaced by a reference with the view

`[Secure^m NeedsSpell^m Secure^m Door^m]/Secure^m`.

In this view, the base declaration of `neededItem` is in the leftmost `Secure^m` since `neededItem` is not in the interface extended by `Secure^m`. The overriding declaration is in `NeedsKey^m`, which appears to the left of `Secure^m` in the instantiated chain and extends an interface that contains `neededItem`.

In contrast, the second call to `canOpen` receives a reference with the view

`[Secure^m Door^m]/Secure^m`.

In this view, the base definition of `neededItem` is in the rightmost `Secure^m` of the full chain, and it is overridden in `NeedsSpell^m`. Neither the definition of `neededItem` in `NeedsKey^m` nor the one in the leftmost occurrence of `Secure^m` is a candidate relative to the given view, because `Secure^m` extends an interface that hides `neededItem`.
MIXEDJAVA not only differs from CLASSICJAVA with respect to method dispatching, but also in its treatment of super. In MIXEDJAVA, super dispatches are dynamic, since the "supermixin" for a super expression is not statically known. The super dispatch for mixins is implemented like regular dispatches with the εβ relation, but using a tail of the current view in place of both the instantiation and view chains: this ensures that a method is selected from the leftmost mixin that follows the current view.

Figure 4.16 contains the complete operational semantics for MIXEDJAVA as a rewriting system on expression-store pairs, similar to the class semantics described in Section 4.1.3. In the MIXEDJAVA semantics, an object in the store is tagged with a mixin instead of a class, and the values are null and (object||view) pairs.

\[
\begin{align*}
E & = \{ [] | E,fd \mid E,fd = \epsilon | v,fd = E \\
& \mid E,m,c(v) \mid v,m,\delta(v \cdots E \epsilon \cdots) \\
& \mid super \equiv v,m,\delta(v \cdots E \epsilon \cdots) \\
& \mid view \, t \rightarrow t, E \mid let \, var = E \, \epsilon
\end{align*}
\]

\[
P \vdash (E[new \, m], S) \iff (E, S) \\
\left( (E[object||M/m], S[object \rightarrow (m, [M_1, M_2, \ldots M_n \mapsto \text{null}])]) \right)
\] [new]

\[
P \vdash (E[object||M/m], S) \iff (E, S) \\
where S(object) = (m, F) and (m', f, t) ∈ \in M and M/m \triangleright M'/m'
\] [set]

\[
P \vdash (E[object||M/m], S) \iff (E, S) \\
where \delta(object) = (m, F) and (m', f, t) ∈ \in M and M/m \triangleright M'/m'
\] [set]

\[
P \vdash (E[object||M/m], S) \iff (E, S) \\
\left( (E[(object[m::M'/m')/this, v_1/var_1, \ldots v_n/var_n], S]) \right)
\] [call]

\[
P \vdash (E[super \equiv (object[m::M/m]), S]) \iff (E, S) \\
\left( (E[(object[m::M'/m')/this, v_1/var_1, \ldots v_n/var_n], S]) \right)
\] [super]

\[
P \vdash (E[view\, t\, \text{as}\, t \rightarrow (object[M'/t]), S]) \iff (E, S) \\
\left( (E[(object[M'/t]), S]) \right)
\] [view]

\[
P \vdash (E[view\, t\, \text{as}\, t \rightarrow (object[M'/t]), S]) \iff (E, S) \\
\left( (E[(object[M'/t]), S]) \right)
\] [cast]

\[
P \vdash (E[let\, \, var = v \, \epsilon \, in \, e, S]) \iff (E[e/v/var], S)
\] [let]

\[
P \vdash (E[view\, t\, \text{as}\, t \rightarrow (object[M'/t]), S]) \iff (E, S)
\] [xcast]

\[
P \vdash (E[view\, t\, \text{as}\, t \rightarrow (null), S]) \iff (E, S)
\] [ncast]

\[
P \vdash (E[null,fd, S]) \iff (E, S)
\] [nget]

\[
P \vdash (E[null,fd = v], S) \iff (E, S)
\] [nset]

\[
P \vdash (E[null, \delta(v_1, \ldots v_n), S]) \iff (E, S)
\] [ncall]

Figure 4.16 : Operational semantics for MIXEDJAVA
4.3.4 **MixedJava Soundness**

The type soundness theorem for **MixedJava** is *mutatis mutandis* the same as the soundness theorem for **ClassicJava** as described in Section 4.1.4.

**Theorem 4.3.1 (Type Soundness for MixedJava)** If \( \vdash_p P \Rightarrow P' : t \) and \( P' = \text{defn}_1 \ldots \text{defn}_n \ e \), then either

- \( P' \vdash (e, \emptyset) \rightarrow^* \langle \text{object}||M/t \rangle, S \rangle \) and \( S(\text{object}) = \langle t', F \rangle \) and \( t' \leq_P t \): or
- \( P' \vdash (e, \emptyset) \rightarrow^* \langle \text{null}, S \rangle \): or
- \( P' \vdash (e, \emptyset) \rightarrow^* \langle \text{error: bad cast}, S \rangle \): or
- \( P' \vdash (e, \emptyset) \rightarrow^* \langle \text{error: dereferenced null}, S \rangle \).

The proof of soundness for **MixedJava** is analogous to the proof for **ClassicJava**, but we must update the type of the environment and the environment-store consistency relation \( (\vdash_s) \) to reflect the differences between **ClassicJava** and **MixedJava**. In **MixedJava**, the environment \( \Gamma \) maps object-view pairs to the type part of the view. i.e., \( \Gamma(\langle \text{object}||M/t \rangle) = t \). The updated consistency relation is defined as follows:

**Definition 4.3.2 (Environment-Store Consistency for MixedJava)**

\[
P, \Gamma \vdash_s S \iff (S(\text{object}) = \langle m, F \rangle)
\]

\[
\Sigma_1: \quad (\Gamma(\langle \text{object}||M/t \rangle) = t' \Rightarrow (\text{WF}(M/t) \text{ and } t = t' \text{ and } m \leq_P t))
\]

\[
\Sigma_2: \quad \text{and } \text{dom}(F) = \{m':M'.fd | |m| \leq^M m':M' \text{ and } \exists t \text{ s.t. } (m'.fd,t) \in^P m'\}
\]

\[
\Sigma_3: \quad \text{and } \{\text{object} | \langle \text{object}||\_\_ \rangle \in \text{rng}(F)\} \subseteq \text{dom}(S) \cup \{\text{null}\}
\]

\[
\Sigma_4: \quad \text{and } (F(m':M'.fd) = \langle \text{object'}||M''/t' \rangle \text{ and } (m'.fd,t) \in^P m') \Rightarrow (t' = t))
\]

\[
\Sigma_5: \quad \text{and } \langle \text{object}||\_\_ \rangle \in \text{dom}(\Gamma) \Rightarrow \text{object} \in \text{dom}(S)
\]

\[
\Sigma_6: \quad \text{and } \text{object} \in \text{dom}(S) \Rightarrow \langle \text{object}||\_\_ \rangle \in \text{dom}(\Gamma)
\]
This definition of \( \vdash_{\sigma} \) relies on the WF predicate on views, which is true of well-formed views. A well-formed view combines 1) a chain that is a tail of some mixin’s chain and 2) a type, either a mixin whose chain is a prefix of the view’s chain or an interface implemented by the first mixin in the view’s chain. Formally, WF is defined as follows:

**Definition 4.3.3 (Well-Formed View)**

\[
WF(M/t) \Leftrightarrow \exists m_0. M_0 \text{ s.t. } m_0 \rightarrow_P M_0 \text{ and } M_0 \leq^M M \\
\quad \text{and } ((\exists M'.M'' \text{ s.t. } M = M' \bowtie M'' \text{ and } t \rightarrow_P M') \\
\quad \text{or } (\exists m.M' \text{ s.t. } M = m::M' \text{ and } m \leq^P t))
\]

The lemmata for proving \textsc{MixedJava} soundness are mostly the same as for \textsc{ClassicJava}, based on a revised typing relation \( \vdash_\zeta \). The annotations in \textsc{MixedJava} programs eliminate implicit subsumption by inserting explicit view expressions, so the \( \vdash_\zeta \) relation for proving \textsc{MixedJava} soundness is the same as \( \vdash_\xi \). The \( \vdash_\zeta \) relation is like \( \vdash_\xi \), except for the handling of view expressions:

\[
\frac{P, \Gamma \vdash_\xi e : t'}{P, \Gamma \vdash_\zeta \text{view } t' \text{ as } t e : t \quad \text{[cast}^m]}
\]

Also, as in \textsc{ClassicJava}, we define a \textsc{SuperOK} predicate for validating the shape of super calls:

**Definition 4.3.4 (Well-Formed Super Calls)**

\[
\text{SuperOK}(e) \Leftrightarrow \text{For all super } \equiv e_0 :: md(e_1, \ldots, e_n) \text{ in } e, \\
\quad \text{either } e_0 = \text{this} \\
\quad \text{or } e_0 = \langle \text{object}|m::M/m \rangle \text{ for some object, } m, \text{ and } M.
\]

**Lemma 4.3.5 (Conserve for MixedJava)** If \( \vdash_P P \Rightarrow P' : t \) and \( P' = \text{defn}_1 \ldots \text{defn}_n e \), then \( P' \cdot \emptyset \vdash_{\xi} e' : t \).

**Proof.** The claim follows from a copy-and-patch argument. \( \Box \)
Lemma 4.3.6 (Subject Reduction for MIXEDJAVA) If $P, \Gamma \vdash e : t, P, \Gamma \vdash S, \text{SUPEROK}(e), \text{ and } \langle e.S \rangle \leftrightarrow \langle e'.S' \rangle$, then $e'$ is an error configuration or there exists $\Gamma'$ such that

1. $P, \Gamma' \vdash e' : t$,
2. $P, \Gamma' \vdash S'$, and
3. $\text{SUPEROK}(e')$.

Proof. The proof examines reduction steps. For each case, if execution has not halted with an answer or in an error configuration, we construct the new environment $\Gamma'$ and show that the two consequents of the theorem are satisfied relative to the new expression, store, and environment. See Appendix D.1 for the complete proof. \qed

Lemma 4.3.7 (Progress for MIXEDJAVA) If $P, \Gamma \vdash e : t, P, \Gamma \vdash S, \text{ and } \text{SUPEROK}(e)$, then either $e$ is a value or there exists an $\langle e'.S' \rangle$ such that $\langle e.S \rangle \leftrightarrow \langle e'.S' \rangle$.

Proof. The proof is by analysis of the possible cases for the current redex in $e$ (in the case that $e$ is not a value). See Appendix D.2 for the complete proof. \qed

By combining the Subject Reduction and Progress lemmas, we can prove that every non-value MIXEDJAVA program reduces while preserving its type, thus establishing the soundness of MIXEDJAVA.

4.3.5 Implementation Considerations

The MIXEDJAVA semantics is formulated at a high level, leaving open the question of how to implement mixins efficiently. Common techniques for implementing classes can be applied to mixins, but two properties of mixins require new implementation strategies. First, each object reference must carry a view of the object. This can be implemented using double-wide references, one half for the object pointer and the other half for the current view. Second, method invocation depends on the current view as well as the instantiation mixin of an object, as reflected in the $\epsilon \beta$ relation. Although this relation depends on two inputs, it nevertheless determines a static, per-mixin method table that is analogous to the virtual method tables typically generated for classes.
The overall cost of using mixins instead of classes is equivalent to the cost of using interface-typed references instead of class-typed references. The justification for this cost is that mixins are used to implement parts of a program that cannot be easily expressed using classes. In a language that provides both classes and mixins, portions of the program that do not use mixins do not incur any extra overhead.

4.3.6 Related Work on Mixins

Mixins first appeared as a CLOS programming pattern [43, 45]. Unfortunately, the original linearization algorithm for CLOS's multiple inheritance breaks the encapsulation of class definitions [17], which makes it difficult to use CLOS for proper mixin programming. The CommonObjects [76] dialect of CLOS supports multiple inheritance without breaking encapsulation, but the language does not provide simple composition operators for mixins.

Bracha has investigated the use of "mixin modules" as a general language for expressing inheritance and overriding in objects [7, 8, 9]. His system is based on earlier work by Cook [12], and its underlying semantics was more recently reformulated in categorical terms by Ancona and Zucca [5]. Bracha's system gives the programmer a mechanism for defining modules (classes, in our sense) as a collection of attributes (methods). Modules can be combined into new modules through various merging operators. Roughly speaking, these operators provide an assembly language for expressing class-to-class functions and, as such, permit programmers to construct mixins. The language, however, forces the programmer to resolve attribute name conflicts manually and to specify attribute overriding explicitly at a mixin merge site. As a result, the programmer is faced with the same problem as in Common Lisp. i.e., the low-level management of details. In contrast, our system provides a language to specify both the content of a mixin and its interaction with other mixins for mixin compositions. The latter gives each mixin an explicit role in the construction of programs so that only sensible mixin compositions are allowed. It distinguishes method overriding from accidental name collisions and thus permits the system to resolve name collisions automatically in a natural manner.

Agesen et al. [4] suggest that a Java variant with type parameterization can support mixins. Their approach does indeed provide a form of separately-compiled mixins, but the resulting mixins are less powerful than MIXEJAVA. They do not resolve
name collisions, but instead signal a compile-time error for any name collision introduced by a mixin application.

4.4 Summary

We have presented a language of mixins that relies on the same programming intuition as single inheritance classes. Indeed, a mixin declaration in our language hardly differs from a class declaration since, from the programmer's local perspective, there is little difference between knowing the properties of a superclass as described by an interface and knowing the exact implementation of a superclass. From the programmer's global perspective, however, mixins free each collection of field and method extensions from the tyranny of a single superclass, enabling new abstractions and increasing the re-use potential of code.

Using mixins is inherently more expensive than using classes, but the additional cost is justified, reasonable, and offset by gains in code re-use. Future work on mixins must focus on exploring compilation strategies that lower the cost of mixins, and on studying how designers can exploit mixins to construct better design patterns.
Chapter 5

Experience with Units and Mixins

Most of our practical experience with units and mixins derives from implementing the DrScheme programming environment [23] using MzScheme. DrScheme provides students and programmers with a user-friendly environment for developing Scheme programs. Units define a mechanism for dividing DrScheme's implementation into components that are implemented by different members of the development team. Mixins simplify the implementation of DrScheme's graphical user interface by encapsulating behavioral extensions to graphical objects.

MzScheme's core unit and class constructs are described in Chapter 2. Section 5.1 of this chapter describes MzScheme's system for named import and export signatures, which makes units practical for large programs like DrScheme. Section 5.2 discusses MzScheme-style mixins, which approximate MIXEDJAVA mixins through classes as first-class values. Section 5.3 describes specific uses of units and mixins within DrScheme's implementation.

5.1 Units with Signatures in MzScheme

The MzScheme unit forms described in Chapter 2 provide no support for managing groups of exported variables, which makes those forms impractical for implementing realistic components. For example, a typical component in DrScheme exports ten to twenty variables; repeatedly listing all of the exports of a unit—at its definition, at every import site, and at every linking site—is too unwieldy.

To support practical programming with units, MzScheme provides the following additional constructs:

- a define-signature form for defining a signature, which is a named collection of variables,

- a unit/sig form for defining a unit with exports and imports that match specified signatures, and
• a **compound-unit/sig** form for linking together units with signature information.

MzScheme implements these forms by elaborating them to the basic unit forms. For example

```
(define-signature INFO (make-info info-num))
(define-signature ERROR (signal-error))
(define-signature DB (new insert delete))
(define db (unit/sig DB
    (import INFO ERROR)
    (define new ⋮)
    ⋮))
```

elaborates to

```
(define db (make-signed-unit
    (unit
        (import make-info info-num error)
        (export insert delete)
        (define new ⋮)
        ⋮)
    '((make-info info-num) (signal-error))
    '((new insert delete)))
```

The **make-signed-unit** primitive creates a record that encapsulates a unit along with signature information for its imports and exports. The **compound-unit/sig** form uses the signature information in a signed unit to validate linking.

### 5.2 Mixins in MzScheme

MzScheme provides mixins via first-class classes and a **class** form that may appear in any expression position. Thus, a **class** expression within a **lambda** or **unit** expression is effectively a mixin if its superclass is determined by the argument of a procedure or an imported variable of a unit. For example, the following expression defines a mixin
that extends any class by adding set-name and get-name methods:

\[
\text{(define name-mixin}
\begin{align*}
&\text{(lambda (superclass)} \\
&\text{(class superclass args)} \\
&\text{(private [name "no name"])} \\
&\text{(public)} \\
&\text{(set-name (lambda (n) (set! name n))]} \\
&\text{(get-name (lambda () name))] \\
&\text{(sequence (apply super-init args))])}
\end{align*}
\]

MzScheme's approach to mixins differs slightly from MIXEDJAVA:

- Unlike a mixin declaration in MIXEDJAVA, the formal argument superclass in the above expression has no associated interface. In MIXEDJAVA, the presence or absence of a method name in the formal argument's interface determines whether a method declared in the mixin is a new method or an overriding method. In MzScheme, the public and override clause keywords make this distinction.

- MIXEDJAVA permits mixins that extend a class with a new method having the same name as an existing method, because compile-time types and runtime views can disambiguate method calls as necessary. In contrast, MzScheme signals an error if a mixin declares a public instance variable that already exists in the superclass.

The second difference represents a significant compromise in our implementation of mixins. Nevertheless, the weaker form of mixins provided by MzScheme has proven powerful as a tool for implementing DrScheme.

### 5.3 Units and Mixins in DrScheme

The unit structure of DrScheme (version 53) is shown in Figure 5.1. Each empty box in the figure corresponds to a unit/sig instance, and each box-containing box corresponds to a compound-unit/sig instance. One box represents the graphical toolbox component, another implements the debugger component, etc. Each member of the DrScheme development team is responsible for a certain set of components.
The compound unit of the form

stands out in Figure 5.1, because it is instantiated four times. It represents a syntax-handling component, which DrScheme instantiates four times to implement four different programming languages (used for students at four different levels).

Figure 5.2 shows DrScheme's unit structure with linking lines. Each line represents an imported or exported signature. With all lines drawn at once, the linking specification is overwhelmingly complex. As illustrated in Figure 5.3, however, the linking specification at a particular point in the linking hierarchy is far easier to understand.

DrScheme relies on dynamically-linked units to support "third-party" extensions to the environment. For example, installing the optional MrSpidey [25] static debugger extends DrScheme's interface with an Analyze button. When the user clicks this button, DrScheme dynamically loads the MrSpidey implementation and links it into the running environment.

The implementation of DrScheme's graphical interface uses mixins extensively to encapsulate small behavioral extensions of graphical objects. For example, a search-frame mixin extends any editor frame with an interactive search control, and a scheme-text mixin adds parenthesis-highlighting to any text-editing buffer. By using mixins instead of classes, a DrScheme programmer can mix-and-match GUI extensions when defining graphical objects.
Figure 5.2: Unit structure of DrScheme with linking

Figure 5.3: Local unit structure in DrScheme
Chapter 6

Related Work on Software Components

McIlroy [59] first crystallized the idea of software components produced by a software-components industry. More recently, Weide et al. [83] and Szyperski [79, 80] substantiate the need to base reusable components on compiled code rather than source code. Szyperski [79] further points out that reuse of compiled components requires a "late linking mechanism" for connecting them, which is the thesis that we refine and explore in this dissertation.

Much of the existing literature on reuse fails to distinguish between the reuse of source code and the reuse of semantic abstractions that can be separately compiled. The distinction is crucial to our view of components, and Krishnamurthi and Felleisen [47] provide a foundation for formalizing the distinction. Some research in software engineering recognizes the distinction, but nevertheless relies on source-code reuse due to a lack of language support; see, for example, Hollingsworth's dissertation [36], which relies on uncompileable generics in Ada for implementing components.

Programming-languages research on reuse concentrates mostly on modules or object-oriented programming. We review work concerning modules in Section 3.6 and object-oriented programming in Sections 4.1.5 and 4.3.6. Much of the research bringing together modules and classes focuses on unifying the constructs within a single model. Lee and Friedman [50, 51] investigate languages that work directly on variables and bindings, which provides a theoretical foundation for implementing both modules and classes. Similarly, Jagannathan [39] and Miller and Rozas [61] propose first-class environments as a common mechanism. Bracha [7] explores mixins for both modular and object-oriented programming; Ancona and Zucca [5] provide a categorical treatment of this view. Our work is complementary to all of the above work, because we concentrate on the principles behind designing constructs for use by programmers, rather than the method used to implement those constructs.

Other research on programming language support for reuse includes the following:

- Design patterns [29] provide programmers with implementation techniques for
creating specific kinds of reusable components within existing programming languages. Patterns help an individual programmer to design a components, and they can help other programmers understand the resulting code. Since the patterns are not part of the language, however, each programmer is responsible for maintaining or understanding a particular coding discipline. Krishnamurthi et al. [46] explore technology for migrating patterns to language constructs.

- Smaragdakis and Batory [75] investigate the implementation of mixin layers for applying a family of cooperating mixins en masse to a family of classes. Mixin layers scale the mixin approach for components to larger systems. Smaragdakis and Batory rely on C++ templates to implement both mixins and mixin layers, but units provide a better framework for implementing mixins layers, because they support separate compilation and more flexible linking mechanisms.

- Mezini and Lieberherr's adaptive plug-and-play components [60] provide a more general alternative to mixin layers. Mezini and Lieberherr's language for components separates the specification of class connections from the definitions of the classes (or class extensions), which permits abstraction with respect to the structural details of the classes. Their language thus follows the principle of external connections.

- Kiczales's Aspect-Oriented programming [44] addresses the implementation of "cross-cutting functionality" that is not easily or efficiently expressed within a single module. An aspect represents a particular cross-cutting feature. Each aspect is combined with other aspects and a core program to implement a complete program. Aspect combination depends an aspect weaver, which operates on the source code of aspects. Since aspect weaving operates on source code, which can interfere with protection and interface control between modules, more work is necessary to determine how aspect-oriented programming integrates with component-based development.

In present software practice, COM [72], CORBA [64], and JavaBeans [40] define the standards for component programming. These standards, however, merely define low-level wiring conventions. They do not provide a language for specifying how components are linked together, and they do not support verification that components
are linked properly before executing the program. Our model of units as components addresses both of these problems.
Chapter 7

Limitations and Future Work

7.1 Combining Typed Units and Mixins

We defined typed models for both units and mixins, but only separately, whereas the example in Section 2 relies on both constructs in a single language. We anticipated a typed version of the example by including is-a? safety tests in the examples, and by showing how the Shape and BB-Shape interfaces are linked to clients to enable those tests. Nevertheless, certain challenges remain for bringing mixins and units together in a typed model. For mixins, the type rules in Chapter 4 assume a complete program and a single namespace for mixin names. For units, the typed language in Chapter 3 does not express the kind of type relationships necessary for importing and exporting interface types (e.g., importing types A and B where A must be a subtype of B).

Others have explored a similar combination of classes and modules in a typed setting. The module systems in Objective Caml [55, 69] and OML [70] support externally specified connections, and since a class can be defined within a module, these languages also provide a simplistic form of mixins. These modules and mixins do not allow the operation extension demonstrated in Section 2.2 because an imported class must match the expected type exactly—no extra methods are allowed. In our example, PICTURE is initially linked to the Rectangle class and later linked to BB-Rectangle; since the latter has more methods, neither Objective Caml nor OML would allow PICTURE to be reused in this way.

7.2 Units and Mixins for Other Languages

Since MzScheme is a dynamically typed language, we have no experience using or implementing typed versions of units or mixins. Glew and Morrisett [30] report on a typed language that resembles units in the way that it type-checks modules and linking, but the linking language relies on a flat namespace, and it implicitly links import to exports by matching names (similar to the linking of .o files).
As explained in Section 3.4, our current unit language trades programming convenience for reuse power. While this trade-off makes sense for a large application such as DrScheme, we need a more convenient language for expressing small programs, without losing the upgrade path that transforms small programs into reusable components. Such a language might take the form of a first-order language for defining and linking modules that elaborates to the more general unit language.

Finally, since COM, CORBA, and JavaBeans define components in practice, future work must explore how to define a unit language as an extension of these industry standards.
Appendix A

MzScheme Class and Interface Syntax

A.1 Classes

The shape of a MzScheme class declaration is:\footnote{Centered ellipses indicate repeated patterns.}

\begin{verbatim}
(class* superclass-expr (interface-expr ...) (init-variable ...) 
  instance-variable-clause ...)
\end{verbatim}

The expression `superclass-expr` determines the superclass for the new class, and the `interface-exprs` specify the interfaces implemented by the class. The `init-variables` receive instance-specific initialization values when the class is instantiated (like the arguments supplied with `new` in Java). Finally, the `instance-variable-clauses` define the instance variables of the class, plus expressions to be evaluated for each instance. For example, a `public` clause declares public instance variables and methods, and a `private` clause declares private instance variables and methods.

Consider the definition

\begin{verbatim}
(define Rectangle 
  (class* object% (Shape) (width height) 
  (public 
    [draw (lambda (window x y) ...)])))))
\end{verbatim}

It introduces the base class `Rectangle`, which is derived from the built-in primitive class `object%`. The `(Shape)` specification indicates that the class implements the `Shape` interface, and the `(width height)` part indicates that two initialization arguments are consumed for initializing an instance. There is one `instance-variable-clause` that defines a public method: `draw`.

MzScheme's object system does not distinguish between instance variables and methods. Instead, procedure-valued instance variables act like methods. The `draw`
declaration in Rectangle defines an instance variable, and (lambda (window x y) ...)
is its initial value expression, evaluated once per instance. When draw is called as the
method of some object, draw may refer to the object via this. In most object-oriented
languages, this is passed in as an implicit argument to a method; in MzScheme, this
is part of the environment for evaluating initialization expression, so each “method”
in an object is a closure containing the correct value of this.2

An instance of Rectangle is created using the make-object primitive. Along with the
class to instantiate, make-object takes any initialization arguments that are expected
for the class. In the case of Rectangle, two initialization arguments specify the size of
the shape:

(define rect (make-object Rectangle 50 100))

The value of an instance variable is extracted from an object using ivar. The following
expression calls the draw “method” of rect by extracting the value of draw and
applying it as a procedure:

((ivar rect draw) window 0 0)

Since method calls of this form are common, MzScheme provides a send macro.
The following send expression is equivalent to the above ivar expression:

(send rect draw window 0 0)

A.2 Interfaces

An interface is declared in MzScheme using the interface form:

(interface (superinterface-expr ...) 
  variable ...)
The superinterface-exprs specify all of the superinterfaces for the new interface, and the variables are the instance variables required by the interface (in addition to variables declared by the superinterfaces). For example, the definition

```
(define Shape (interface () draw))
```

creates an interface named Shape with one variable: draw. Every class that implements Shape must declare a draw instance variable. The definition

```
(define BB-Shape (interface (Shape) bounding-box))
```

creates an interface named BB-Shape with two variables: draw and bounding-box. Since Shape is the superinterface of BB-Shape, every class that implements BB-Shape also implements Shape.

A class implements an interface only when it specifically declares the implementation by “name” (as in Java).³ Thus, the Rectangle class in the previous section implements only the Shape interface.

### A.3 Derived Classes

The definition

```
(define BB-Rectangle
  (class* Rectangle (BB-Shape) (width height)
    (public [bounding-box ...])
    (sequence (super-init width height))))
```

derives a BB-Rectangle class that implements BB-Shape. The draw method, required to implement BB-Shape, is inherited from Rectangle.

The BB-Rectangle class declares the new bounding-box method. It also includes a sequence clause that calls super-init. A sequence clause specifies expressions to be evaluated for a newly-created instance of the class. The sequence clause is commonly used to call the special super-init procedure, which initializes the part of the instance defined by the superclass (like calling super in a Java constructor); a derived class must call super-init exactly once for every instance. In the case of BB-Rectangle, calling super-init performs Rectangle's initialization for the instance. BB-Rectangle

---

³Since interfaces are first-class values in MzScheme, classes implement interfaces by value.
consumes two arguments and supplies them to `super-init`. because the `Rectangle` class consumes two initialization arguments.
Appendix B

UNIT\textsubscript{C} Proofs

B.1 Proof of Subject Reduction

Lemma 3.5.2 (Subject Reduction) If $\mathcal{T} \cdot e \rightarrow \mathcal{T}' \cdot e'$ and $|\mathcal{T}| \vdash e : \tau_0$, then $|\mathcal{T}'| \vdash e' : \tau'_0$ and $\tau'_0 \leq \tau_0$.

Proof. The proof is by induction on the structure of $e$. The lemma holds for the base case, $e = v$, since there is no $e'$ such that $\mathcal{T} \cdot e \rightarrow \mathcal{T}' \cdot e'$.

Case $e = e_1 : e_2$.

By seq\textsubscript{c}. $|\mathcal{T}| \vdash e_1 : \tau$ for some $\tau$ and $|\mathcal{T}| \vdash e_2 : \tau_0$. There are two subcases:

Case $e = v : e_2$.

By seq\textsuperscript{c}. $e' = e_2$ and $\mathcal{T}' = \mathcal{T}$. Since $e' = e_2$, $|\mathcal{T}'| \vdash e' : \tau_0$.

Case $e = e_1 : e_2$ where $\mathcal{T} \cdot e_1 \rightarrow \mathcal{T}' \cdot e'_1$, so $e' = e'_1 : e_2$.

By induction. $|\mathcal{T}'| \vdash e'_1 : \tau'$ for some $\tau'$. By Lemma B.3.7 (Store Growth) and Lemma B.3.4 (Environment Extension), $|\mathcal{T}'| \vdash e_2 : \tau_0$. Therefore, by seq\textsubscript{c} again, $|\mathcal{T}'| \vdash e' : \tau_0$.

Case $e = e_1$ as $\tau$.

By generalize\textsubscript{c}. $\tau = \tau_0$ and $|\mathcal{T}| \vdash e_1 : \tau'_0$ for some $\tau'_0 \leq \tau_0$. There are two subcases:

Case $e = v$ as $\tau$.

By generalize\textsuperscript{c}. $e' = v$ and $\mathcal{T}' = \mathcal{T}$. Thus, $|\mathcal{T}'| \vdash e' : \tau'_0$.

Case $e = e_1$ as $\tau$ where $\mathcal{T} \cdot e_1 \rightarrow \mathcal{T}' \cdot e'_1$, so $e' = e'_1$ as $\tau$.

By induction. $|\mathcal{T}'| \vdash e'_1 : \tau'$ for some $\tau' \leq \tau'_0$. By the transitivity of $\leq$, $\tau' \leq \tau_0$ (where $\tau_0 = \tau$). Therefore, by generalize\textsuperscript{c} again, $|\mathcal{T}'| \vdash e' : \tau_0$. 
Case $\varepsilon = \varepsilon_1 \varepsilon_2$.

By $\text{app}^+_{\varepsilon}$. $|T| \vdash \varepsilon_1 : \tau \rightarrow \tau_0$ and $|T| \vdash \varepsilon_2 : \tau_2$ for some $\tau_2 \leq \tau$.
There are two possibilities for $\varepsilon_1$:

Case $\varepsilon_1 = v_1$.

We must consider the two cases for $\varepsilon_2$:

Case $\varepsilon_2 = v_2$.

We must consider the possible shapes for $v_1$. By Lemma B.3.8 (Value Types), $v_1$ can be a primitive operation or a procedure:

Case $v_1 = \text{projl}(t)$ where $T(t) = \langle \eta, \tau_\ell \rangle$, so $\tau = t$ and $\tau_0 = \eta$.
Since $v_2$’s type is a subtype of $t$, it must be exactly $t$. According to Lemma B.3.8 (Value Types), a value of type $t$ has one of the following shapes:

Case $v_2 = \text{injl}(t)v_3$ where $|T| \vdash v_3 : \tau'_0$ and $\tau'_0 \leq \tau_0$.

By proj$_c^+$, $T' = T$ and $e' = v_3$, so $|T'| \vdash e' : \tau'_0$.

Case $v_2 = \text{injr}(t)v_3$.

By proj-fail$_c^+$, $T \cdot \varepsilon \rightarrow \rightarrow T \cdot \text{variant error}$: there is no $e'$ such that $T \cdot \varepsilon \rightarrow \rightarrow T' \cdot e'$.

Case $v_1 = \text{projr}(t)$ where $T(t) = \langle \eta, \tau_r \rangle$, so $\tau = t$ and $\tau_0 = \tau_r$.

Analogous to previous case.

Case $v_1 = \text{test}(t)$ where $T(t) = \langle \eta, \tau_r \rangle$, so $\tau = t$ and $\tau_0 = \text{bool}$.
Since $v_2$’s type is a subtype of $t$, it must be exactly $t$. A value of type $t$ has one of the following shapes:

Case $v_2 = \text{injl}(t)v_3$ where $|T| \vdash v_3 : \eta$.

By test-true$_c^+$, $T' = T$ and $e' = \text{true}$, which has type bool.

Case $v_2 = \text{injr}(t)v_3$ where $|T| \vdash v_3 : \eta$.

By test-false$_c^+$, $T' = T$ and $e' = \text{true}$, which has type bool.

Case $v_1 = \text{fn} \ x : \tau \Rightarrow e_0$ where $|T|[x:\tau] \vdash e_0 : \tau_0$.

By app$_c^+$, $T' = T$ and $e' = [v_2/x]e_0$. By Lemma B.3.3 (Substitution), $|T| \vdash [v_2/x]e_0 : \tau'_0$ where $\tau'_0 \leq \tau_0$.

Case $T \cdot e_2 \rightarrow \rightarrow T' \cdot e'_2$ and $e' = v_1 e'_2$. 
By induction. \(|\mathcal{T}'| \vdash e' : \tau'_2 \text{ where } \tau'_2 \leq \tau_2\). By Lemma B.3.7 (Store Growth) and Lemma B.3.4 (Environment Extension). \(|\mathcal{T}'| \vdash e_1 : \tau \rightarrow \tau_0\). Since \(\tau'_2 \leq \tau_2\), \(\tau'_2 \leq \tau\) by the transitivity of \(\leq\). Thus, by \(\text{app}_c^+\).
\(|\mathcal{T}'| \vdash e' : \tau_0\).

Case \(\mathcal{T} \cdot e_1 \longrightarrow \mathcal{T}' \cdot e'_1\) and \(e' = e'_1 e_2\).

By induction. \(|\mathcal{T}'| \vdash e'_1 : \tau'_1 \text{ where } \tau'_1 \leq \tau \rightarrow \tau_0\). By Lemma B.3.7 (Store Growth) and Lemma B.3.4 (Environment Extension). \(|\mathcal{T}'| \vdash e_2 : \tau_2\). Since \(\tau'_1\) is a subtype of \(\tau \rightarrow \tau_0\), \(\tau'_1\) must be a function type \(\tau' \rightarrow \tau'_0\) where \(\tau \leq \tau'\) and \(\tau'_0 \leq \tau_0\). By \(\tau_2 \leq \tau\) and the transitivity of \(\leq\), \(\tau_2 \leq \tau'\). Thus, by \(\text{app}_c^+\).
\(|\mathcal{T}'| \vdash e' : \tau'_0\) (where \(\tau'_0 \leq \tau_0\)).

Case \(e = \text{letrec type } l = x_{cl}, x_{dl} \eta | x_{cr}, x_{dr} \tau_r \diamond x_t.\)
\(\text{val } x_v : \tau = v\)
\(\text{in } e_b\)

The \(\text{letrec}_c^+\) rule ensures that \(l \cap \text{dom}(\mathcal{T}) = \emptyset\). By \(\text{letrec-types}_c^+\), \(\mathcal{T}' = \mathcal{T}[l \mapsto (\eta, \tau_r)]\) and \(e' = S(\text{letrec val } x_v : \tau = v \text{ in } e_b)\) where \(S\) is a replacement on \(x_{cl}, x_{dl}, x_{cr}, x_{dr}, \) and \(x_t\).

Let \(\Gamma = |\mathcal{T}|.\) As in \(\text{letrec}_c^+\), let \(\Gamma' = \Gamma[l::\Omega]\) and let \(\Gamma''\) be the extension of \(\Gamma'\) with types for \(x_{cl}, \cdots x_t\). Since \(\Gamma \vdash e : \tau_0\), by \(\text{letrec}_c^+\) we have \(\Gamma'' \vdash \overline{v} : \overline{\tau}\) and \(\Gamma'' \vdash e_b : \tau_0\).

Since \(S\) replaces only variables. \(S(\Gamma'') \vdash S(\overline{v}) : \overline{\tau}\) and \(S(\Gamma'') \vdash S(e_b) : \tau_0.\) Furthermore, \(S(\Gamma'') = |\mathcal{T}'|:\)

- The difference between \(\mathcal{T}'\) and \(\mathcal{T}\) is \(l\), which means that \(|\mathcal{T}'|\) extends \(|\mathcal{T}|\) with type bindings \(l::\Omega\) and \(\text{injl}(l) : \eta \rightarrow l, \cdots \text{test}(l) : l \rightarrow \text{bool}\).
- By construction, \(\Gamma''\) adds \(l::\Omega\) and \(x_{cl} : \eta \rightarrow l, \cdots x_t : l \rightarrow \text{bool}\) to \(\Gamma\).
- \(S\) replaces the variables \(x_{cl}, \cdots x_t\) with the variables \(\text{injl}(l), \cdots \text{test}(l)\).

By using the \(\Gamma'' = |\mathcal{T}'|\) equivalence and combining judgements with \(\text{letrec}_c^+\), we obtain \(|\mathcal{T}'| \vdash S(\text{letrec val } x_v : \tau = v \text{ in } e_b) : \tau_0\). Thus, \(|\mathcal{T}'| \vdash e' : \tau_0\).

Case \(e = \text{letrec val } x_v : \tau = v \text{ in } e_b.\)

By \(\text{letrec}_c^+\), \(\mathcal{T}' = \mathcal{T}\) and \(e' = [\text{letrec val } x_v : \tau = v \text{ in } v/x_{v}]e_b.\) The derivation for \(|\mathcal{T}| \vdash \text{letrec val } x_v : \tau = v \text{ in } e_b : \tau_0\) proves in intermediate steps
that $\sem{\mathcal{T}}[x_v : \tau] \vdash s \overline{v} : \tau$. Then, using $\text{letrec}^c_\mathcal{C}$, we can synthesize the judgement $\sem{\mathcal{T}}[\mathcal{T}] \vdash \text{letrec} \, \text{val} \, x_v : \tau = v \text{ in } v : \tau$. Thus, according to Lemma B.3.3 (Substitution), $\sem{\mathcal{T}'} \vdash e' : \tau_0'$ where $\tau_0' \leq \tau_0$.

Case $e = \text{invoke}_{\mathcal{U}}$ with $s = t::\Omega \leftarrow \sigma \ y : \tau \leftarrow e_i$.

The interesting case is where $e_u = v_u$ and $e_i = \overline{v}$, so we consider that case first.

By $\text{invoke}^+_\mathcal{V}$, $\sem{\mathcal{T}} \vdash e_u : \tau_u$, $\sem{\mathcal{T}} \vdash e_i : \tau$, and $\tau_u$ is a signature such that $\tau_u \leq \text{sig import } s = t::\Omega \ y : \tau \text{ export } \emptyset \sigma \ b$, where $[\sigma/t] \tau_b = \tau_0$.

Since $v_u$'s type is the signature $\tau_u$, by Lemma B.3.8 (Value Types) $v_u$ must be a unit of the form

\[
\text{unit import } s_i = t_i::\Omega \ y_i = x_i : \tau_i \text{ export } s_e = t_e::\Omega \ y_e = x_e : \tau_e \\
\sigma \ b \prec \tau_b \\
\text{type } t_d = x_{cl} \cdot x_{dl} \cdot \tau_1 \cdot x_{cr} \cdot x_{dr} \cdot \tau_r \cdot x_t \\
\text{val } x_v : \tau_v = v_v \\
\epsilon_b
\]

where $s_i = t_i \subseteq s = t$, $y_i = x_i \subseteq y = x$, $\tau \leq \tau_i$ for each $y_i$, and the type assigned to $e_b$ within the unit is $\tau_b'$ where $\tau_b' \leq \tau_b$.

By $\text{invoke}^+_\mathcal{V}$, $\mathcal{T} \cdot e \mapsto \mathcal{T} \cdot [\sigma/t, v/x]e_i$ where

$e_l = \text{letrec type } t_d = x_{cl} \cdot x_{dl} \cdot \tau_1 \cdot x_{cr} \cdot x_{dr} \cdot \tau_r \cdot x_t \\
\text{val } x_v : \tau_v = v_v \\
\text{in } e_b$ as $\tau_b$.

The expression $e_l$ can be typed in an environment $\sem{\mathcal{T}}[\mathcal{T}]$ because $v_u$ types in $\sem{\mathcal{T}}$. To verify this, we check each antecedent in the $\text{letrec}^c_\mathcal{C}$ rule applied to $e_l$ with $\sem{\mathcal{T}}[\mathcal{T}]$ and show that it corresponds to an antecedent in the $\text{unit}^c_\mathcal{C}$ rule applied to $v_u$ with $\sem{\mathcal{T}}$ (which is already proved, by assumption):

- Distinct variable names, and not in $\text{dom}(\Gamma$): The $\text{unit}^c_\mathcal{C}$ antecedents include all of the names in the $\text{letrec}^c_\mathcal{C}$ antecedents.

- $\tau_i$, $\tau_r$, and $\tau_v$ validations: The $\text{unit}^c_\mathcal{C}$ definition of $\Gamma'$ adds both $\overline{t_i}$ and $\overline{t_r}$ to $\Gamma$, while the $\text{letrec}^c_\mathcal{C}$ definition of $\Gamma'$ adds only $\overline{t_r}$. But the $\text{letrec}^c_\mathcal{C}$ rule is applied with $\sem{\mathcal{T}'}$ instead of $\sem{\mathcal{T}}$, so the final $\Gamma'$ in $\text{letrec}^c_\mathcal{C}$ and $\text{unit}^c_\mathcal{C}$ have the same type variable bindings. The extra value variable bindings
\([x_i; \tau_i]\) in \(\Gamma'\) for \(\text{letrec}_c^+\) cannot affect the validation of type expressions by Lemma B.3.4 (Environment Extension).

- \(\epsilon_c^+\) typings: As above, \(\Gamma''\) in both \(\unit_c^+\) and \(\text{letrec}_c^+\) contain the same bindings, because \(\text{letrec}_c^+\) is applied with \(\mathcal{T}[t_i]; \Omega, \overline{x_i}; \tau_i]\). Thus, the antecedents in \(\unit_c^+\) imply the ones in \(\text{letrec}_c^+\).

- \(\epsilon_b\) as \(\tau_b\) typing: In \(\unit_c^+\), \(\Gamma'' \vdash \epsilon_b : \tau_b\) for some \(\tau_b' \leq \tau_b\). Since \(\Gamma''\) is the same in \(\unit_c^+\) and \(\text{letrec}_c^+\), \(\Gamma'' \vdash \epsilon_b\) as \(\tau_b\) in \(\text{letrec}_c^+\).

- \(FTV(\tau_b) \cap \overline{l_d} = \emptyset\) check: The \(FTV(\tau_b) \cap \overline{l_d} = \emptyset\) check in \(\unit_c^+\) implies the equivalent check in \(\text{letrec}_c^+\).

Thus, \(\mathcal{T}[t_i]; \Omega, \overline{x_i}; \tau_i] \vdash e_i : \tau_b\). By Lemma B.3.5 (Stable Typing), we have \(\mathcal{T}[t_i]_{\epsilon_i} : \mathcal{T}[x_i]; \tau_i] \vdash e_i : \mathcal{T}[t_i]_{\epsilon_i} : \mathcal{T}[x_i]; \tau_i]. Then, using Lemma B.3.3 (Substitution), \(\mathcal{T}[t_i]_{\epsilon_i} : \mathcal{T}] \vdash e_i : \mathcal{T}[x_i]; \tau_i]\), which is equivalent to the judgement \(\mathcal{T}[t_i]_{\epsilon_i} : \mathcal{T}] \vdash e_i : \mathcal{T}[x_i]; \tau_i]\). Since \(\unit_c^+\) ensures \(\overline{l_i} \cap \text{dom}(\mathcal{T}) = \emptyset\), \(\mathcal{T}] \vdash e_i : \mathcal{T}[x_i]; \tau_i]\).

Finally, since \(\tau_0 = \mathcal{T}[t_i]_{\epsilon_i} : \mathcal{T}[x_i]; \tau_i]\), \(\mathcal{T} \cdot e \longrightarrow \mathcal{T}' \cdot e'\) for \(\mathcal{T}' = \mathcal{T}\) and \(\mathcal{T}' \vdash e' : \tau_0'\) where \(\tau_0' \leq \tau_0\).

To complete the \textbf{invoke} case for \(e\), we must consider all possibilities for \(\epsilon_u\):

Case \(\epsilon_u = v\).

We must consider the three cases for each \(\epsilon_i\):

Case \(\epsilon_i\) is a value for each \(i\).

Covered above.

Case \(\mathcal{T} \cdot e_i \longrightarrow \mathcal{T}' \cdot e_i'\) for the first \(e_i\) that is not a \(v\).

By induction, \(\mathcal{T}' \vdash e_i' : \tau'\) where \(\tau' \leq \tau\). Substituting \(e_i'\) for \(e_i\) in \(e\) produces \(e'\). Using Lemma B.3.1 (Replacement), Lemma B.3.7 (Store Growth), and Lemma B.3.4 (Environment Extension), \(\mathcal{T}' \vdash e' : \tau_0'\) and \(\tau_0' \leq \tau_0\).

Case \(\mathcal{T} \cdot e_u \longrightarrow \mathcal{T}' \cdot e_u'\) where .

By induction, \(\mathcal{T}' \vdash e_u' : \tau_u'\) where \(\tau_u' \leq \tau_u\). Substituting \(e_u'\) for \(e_u\) in \(e\) produces \(e'\). Using Lemma B.3.1 (Replacement), Lemma B.3.7 (Store Growth), and Lemma B.3.4 (Environment Extension), \(\mathcal{T}' \vdash e' : \tau_0'\) and \(\tau_0 \leq \tau_0\).
Case $e = \text{compound}_c$

\[
\begin{align*}
\text{import } s_i & = l_i::\Omega \ y_i::\tau_i \\
\text{export } s_e & = l_e::\Omega \ y_e::\tau_e \\
\text{link } e_1 \text{ with } s_{w1} & = l_{w1}::\Omega \ y_{w1}::\tau_{w1} \\
\text{provides } s_{p1} & = l_{p1}::\Omega \ y_{p1}::\tau_{p1} \\
\text{and } e_2 \text{ with } s_{w2} & = l_{w2}::\Omega \ y_{w2}::\tau_{w2} \\
\text{provides } s_{p2} & = l_{p2}::\Omega \ y_{p2}::\tau_{p2}
\end{align*}
\]

The interesting case is where $e_1 = v_1$ and $e_2 = v_2$ and \text{compound}_c^+ applies, so we consider that case first.

By \text{compound}_c^+, $v_1$ has a signature type, so by Lemma B.3.8 (Value Types) it must be a unit of the form

\[
\begin{align*}
\text{unit import } s_{11} & = l_{11}::\Omega \ y_{11} = x_{11}::\tau_{11} \\
\text{export } s_{e1} & = l_{e1}::\Omega \ y_{e1} = x_{e1}::\tau_{e1} \\
\text{type } l_{d1} & = x_{d1}, \ x_{d1} \ \tau_{11} | x_{e1}, \ x_{e1} \ \tau_{e1} \ \circ \ x_{t1} \\
\text{val } x_{v1} : \tau_{v1} & = v_{v1}
\end{align*}
\]

and $v_2$ is similarly a unit. The definitions and initialization expressions of $v_1$ and $v_2$ are merged to form a new unit, $v_3$, of the form

\[
\begin{align*}
\text{unit import } s_i & = l_i::\Omega \ y_i = x_i::\tau_i \\
\text{export } s_e & = l_e::\Omega \ y_e = x_e::\tau_e \\
\text{type } l_d & = x_{d1}, \ x_{d1} \ \tau_1 | x_{e1}, \ x_{e1} \ \tau_e \ \circ \ x_t \\
\text{val } x_v : \tau_v & = v_v
\end{align*}
\]

We show that the new unit has the signature type $\tau_0$ by inspecting the type proofs for $v_1$, $v_2$ and $e$, matching antecedents in those proofs to the needed antecedents for $v_3$. Let $\Gamma = |\mathcal{T}| = |\mathcal{T}'|$.

- Distinct variable names: The antecedents in the proofs for $v_1$ and $v_2$ combined with the distinctness requirement for applying \text{compound}_c^+ imply the distinctness requirement for $v_3$.

- Exports are a subset of definitions: The exports of $v_3$ are the same as the exports in the \text{compound} expression $e$. The proof for $e$ requires that the exports are a subset of the exports from $v_1$ and $v_2$, and the proofs for $v_1$ and $v_2$ require that a definition is provided for each export. Since all of
the definitions from \(v_1\) and \(v_2\) are in \(v_3\), every exported variable for \(v_3\) is defined in \(v_3\). Furthermore, the subtyping requirements in \(\text{compound}_e\) (for propagating exports) and \(\text{unit}_e\) (for exports) ensure that the declaration type for each exported value is a subtype of the export type, using the transitivity of \(\leq\).

- **Signature type is valid:** The signature for \(v_3\) is the same as the signature of \(e\), so the signature is valid by assumption.

- **Type expressions \(\bar{\tau}_i\) are valid:** The \(\bar{\tau}_i\) expressions in \(v_3\) are the combined type expressions \(\bar{\tau}_{i1}\) and \(\bar{\tau}_{i2}\). For \(v_1\), the type expressions \(\bar{\tau}_{i1}\) are validated in an environment \(\Gamma'_1 = [\ell_{i1} : \Omega, t_{d1} : \Omega]\). For \(v_2\), the same type expressions must be validated in the environment \(\Gamma' = [\ell_{i1} : \Omega, t_{d1} : \Omega, t_{d2} : \Omega]\). But \(\Gamma'\) includes all of the type bindings of \(\Gamma'_1\), since \(\ell_{i1}\) must be a subset of \(\ell_i \cup t_{d2}\) by \(\text{compound}_e\). Although \(\Gamma'\) may contain additional type variables, they cannot affect the validation of \(\bar{\tau}_{i1}\). By Lemma B.3.4 (Environment Extension). By a similar argument, the type expressions \(\bar{\tau}_{i2}\) can be validated in \(v_3\).

- **Expression types, \(\Gamma'' \vdash \bar{v}_e : \bar{\tau}_e\):** The \(\bar{v}_e\) expressions in \(v_3\) are the combined expressions \(\bar{v}_{e1}\) and \(\bar{v}_{e2}\). The argument for typing \(\bar{v}_e\) in \(v_3\) is similar to the argument above for validating the type expressions. But the environment \(\Gamma''\) is not merely a superset of \(\Gamma'_1\) or \(\Gamma'_2\). \(\Gamma''\) contains at least as many variable bindings as \(\Gamma'_1\), but for each variable in \(\Gamma'_2\), its type binding in \(\Gamma''\) is a subtype of the one in \(\Gamma'_1\). The subtyping is possible according to the \(\text{compound}_e\) rule, which allows a subtype relationship between the types \(\bar{\tau}_{i1}\) expected by \(v_1\) and the types \(\bar{\tau}_{e2}\) provided by \(v_2\) (or the types \(\bar{\tau}_i\) imported by \(e\)). As a result, by Lemma B.3.2 (Environment Replacement), if \(\Gamma'' \vdash \bar{v}_{e1} : \bar{\tau}_{e1}\) for \(v_1\), \(\Gamma'' \vdash \bar{v}_{e1} : \bar{\tau}_e\) in \(v_3\) where \(\tau'_v \leq \tau'_{e1}\). This subtyping relationship is sufficient for validating \(v_3\), which requires the subsumption relation \(\Gamma'' \vdash \bar{v}_e : \bar{\tau}_e\).

- **Initialization expression type, \(\Gamma'' \vdash e_b : \tau_b\):** By \(\text{compound}_e\), \(e_b = e_{b1} : e_{b2}\). The type of \(e_b\) is thus the type of \(e_{b2}\) in the environment \(\Gamma''\). By the same argument as for the \(\bar{v}_e\) expressions, if \(e_{b2}\) has type \(\tau'_{b2}\) in \(\Gamma'_2\), it has some type \(\tau'_b\) in \(\Gamma''\) such that \(\tau'_{b2} \leq \tau'_{b2}\). By assumption, \(\tau'_{b2} \leq \tau_{b2}\), and by \(\text{compound}_e\) \(\tau_{b2} \leq \tau_b\), so \(\tau'_b \leq \tau_b\) by transitivity.
• \( FTV(\tau_b) \cap \overline{\ell_d} = \emptyset \) check: By \( \text{compound}_{c}^{-} \), \( \tau_b \) must be well-formed in \( \Gamma \) extended with \( \overline{\ell_i} \), so \( FTV(\tau_b) \subseteq (\text{dom}(\Gamma) \cup \overline{\ell_i}) \). Type-checking for \( \nu_1 \) and \( \nu_2 \) ensures that \( (\overline{\ell_{d_1}} \cup \overline{\ell_{d_2}}) \cap \text{dom}(\mathcal{T}) = \emptyset \), and \( \text{compound}_{c}^{+} \) applies only when \( (\overline{\ell_{d_1}} \cup \overline{\ell_{d_2}}) \cap \overline{\ell_i} = \emptyset \). Since \( \overline{\ell_d} = \overline{\ell_{d_1}} \cup \overline{\ell_{d_2}} \), we have \( FTV(\tau_b) \cap \overline{\ell_d} = \emptyset \).

Thus, all of the antecedents hold: \( \nu_3 \) must have the signature \( \tau_0 \) because it has the same imports, exports, and declared initialization type as \( \epsilon \).

To complete the \( \text{compound} \) case for \( \epsilon \), we must consider the remaining possibilities for \( \epsilon_1 \):

**Case** \( \epsilon_1 = \nu_1 \).

**Case** \( \epsilon_2 = \nu_2 \).

Covered above.

**Case** \( \mathcal{T} \cdot \epsilon_2 \rightarrow \mathcal{T'} \cdot \epsilon'_2 \).

By induction, \( |\mathcal{T'}| \vdash \epsilon'_2 : \tau'_2 \) where \( \tau'_2 \leq \tau_2 \). Substituting \( \epsilon'_2 \) for \( \epsilon_2 \) in \( \epsilon \) produces \( \epsilon' \). Using Lemma B.3.1 (Replacement), Lemma B.3.7 (Store Growth), and Lemma B.3.4 (Environment Extension), \( |\mathcal{T'}| \vdash \epsilon' : \tau'_0 \) and \( \tau_0 \leq \tau_0 \).

**Case** \( \mathcal{T} \cdot \epsilon_1 \rightarrow \mathcal{T'} \cdot \epsilon'_1 \).

By induction, \( |\mathcal{T'}| \vdash \epsilon'_1 : \tau'_1 \) where \( \tau'_1 \leq \tau_1 \). Substituting \( \epsilon'_1 \) for \( \epsilon_1 \) in \( \epsilon \) produces \( \epsilon' \). Using Lemma B.3.1 (Replacement), Lemma B.3.7 (Store Growth), and Lemma B.3.4 (Environment Extension), \( |\mathcal{T'}| \vdash \epsilon' : \tau'_0 \) and \( \tau_0 \leq \tau_0 \). \( \square \)

### B.2 Proof of Progress

**Lemma 3.5.3 (Progress)** If \( |\mathcal{T}| \vdash \epsilon : \tau_0 \), then either:

1. \( \epsilon = \nu \) for some \( \nu \);

2. \( \mathcal{T} \cdot \epsilon \rightarrow \mathcal{T} \cdot \text{variant error} \); or

3. \( \mathcal{T} \cdot \epsilon \rightarrow \mathcal{T'} \cdot \epsilon' \) for some \( \mathcal{T'} \) and \( \epsilon' \).
Proof. The proof is by induction on the structure of $e$. The lemma holds for the base case where $e$ is a value. We consider all other expression forms and show that a reduction step exists.

Case $e = e_1 : e_2$.

By induction, there are three possibilities for $e_1$:

Case $e_1 = v$ for some $v$.

By seq$^+_c$, $T' = T$ and $e' = e_2$.

Case $T \cdot e_1 \rightarrow T \cdot \text{variant error}$.

By variant$^+_c$, a variant error replaces the entire context of the erroneous subexpression. Thus, $T \cdot e \rightarrow T \cdot \text{variant error}$.

Case $T \cdot e_1 \rightarrow T' \cdot e'_1$.

$e' = e'_1 : e_2$.

Case $e = e_1$ as $\tau$.

By induction, there are three possibilities for $e_1$:

Case $e_1 = v$ for some $v$.

By generalize$^+_c$, $T' = T$ and $e' = e_1$.

Case $T \cdot e_1 \rightarrow T \cdot \text{variant error}$.

By variant$^+_c$, $T \cdot e \rightarrow T \cdot \text{variant error}$.

Case $T \cdot e_1 \rightarrow T' \cdot e'_1$.

$e' = e'_1 \text{ as } \tau$.

Case $e = e_1 e_2$.

By app$^+_c$, $|T| \vdash e_1 : \tau \rightarrow \tau_0$ and $|T| \vdash e_2 : \tau_2$ for some $\tau_2$ where $\tau_2 \leq \tau$.

By induction, there are three possibilities for $e_1$:

Case $e_1 = v_1$.

We must consider the three cases for $e_2$:
Case $e_2 = v_2$.

The shape analysis proceeds as for Lemma 3.5.2 (Subject Reduction). For certain cases, such as $v_1 = \text{proj}(\tau_0, \tau_r)$ and $v_2 = \text{injr}(\tau_0, \tau_r)v_3$. \( \mathcal{T} \cdot e \mapsto \mathcal{T} \cdot \text{variant error} \), and for other cases, \( \mathcal{T} \cdot e \mapsto \mathcal{T}' \cdot e' \) for some $\mathcal{T}'$ and $e'$.

Case $\mathcal{T} \cdot e_2 \mapsto \mathcal{T} \cdot \text{variant error}$.

By variant$_c^{+}$. \( \mathcal{T} \cdot e \mapsto \mathcal{T} \cdot \text{variant error} \).

Case $\mathcal{T} \cdot e_2 \mapsto \mathcal{T}' \cdot e'_2$.

\[ e' = v_1 e'_2. \]

Case $\mathcal{T} \cdot e_1 \mapsto \mathcal{T} \cdot \text{variant error}$.

By variant$_c^{+}$. \( \mathcal{T} \cdot e \mapsto \mathcal{T} \cdot \text{variant error} \).

Case $\mathcal{T} \cdot e_1 \mapsto \mathcal{T}' \cdot e'_1$.

\[ e' = e'_1 e_2. \]

Case $e = \text{letrec type } \ell = x_{\text{cl}}, x_{\text{dl}} \mid x_{\text{cr}}, x_{\text{dr}} \circ x_t \cdot $

\[ \text{val } x_v : \tau = v_v \]

\[ \text{in } e_b \]

By letrec$_c^{+}$, $\ell \cap \text{dom}(|\mathcal{T}|) = \emptyset$. Since $\text{dom}(|\mathcal{T}|)$ and $\text{dom}(\mathcal{T})$ contain the same types, $\ell \cap \text{dom}(\mathcal{T}) = \emptyset$. Thus, letrec-types$_c^{+}$ applies, and $\mathcal{T} \cdot e \mapsto \mathcal{T}' \cdot S($letrec $\text{val } x_v : \tau = v_v \text{ in } e_b$).

Case $e = \text{letrec } \text{val } x_v : \tau = v_v \text{ in } e_b$.

By letrec$_c^{+}$, $\mathcal{T} \cdot e \mapsto \mathcal{T} \cdot [\text{letrec } \text{val } x_v : \tau = v_v \text{ in } v_v/x_v]e_b$.

Case $e = \text{invoke } e_u$ with $s :: \Omega = \sigma \ y : \tau = e_i$.

If $e_u = v_u$ and $e_i = \overline{v}$, then the invoke$_c^{+}$ rule defines $e'$: by invoke$_c^{+}$, $v_u$ has a signature type, so by Lemma B.3.8 (Value Types) $v_u$ must be a unit of the form

\[ \text{unit import } s_i = l_i :: \Omega \ y_i = x_{i \cdot \tau_i} \]

\[ \text{export } s_e = l_e :: \Omega \ y_e = x_{e \cdot \tau_e} \]

\[ \text{type } l_d = x_{\text{cl}}, x_{\text{dl}} \mid x_{\text{cr}}, x_{\text{dr}} \circ x_t \]

\[ \text{val } x_v : \tau_v = v_v \]

\[ e_b \]
where $\overline{s_i} = \overline{l_i} \subseteq \overline{s} = \overline{l}$, and $\overline{y_i} \subseteq \overline{y}$. We can choose $\overline{x}$ so that $\overline{y_i} = \overline{x_i} \subseteq \overline{y} = \overline{x}$.

Thus, $\text{invoke}^{\downarrow}_c$ defines $\mathcal{T}'$ and $e'$ when $e_u = v_u$ and $\overline{e_i} = \overline{e_i}$.

To complete the $\text{invoke}$ case for $e$, we must consider all possibilities for $e_u$:

**Case** $e_u = v$.

We must consider the three cases for each $e_i$:

**Case** $e_i = v$ for each $e_i$.

Covered above.

**Case** $\mathcal{T} \cdot e_i \rightarrow \mathcal{T} \cdot \text{variant error}$ for the first $e_i$ that is not a $v$.

By $\text{variant}^{\downarrow}_c$, $\mathcal{T} \cdot e \rightarrow \mathcal{T} \cdot \text{variant error}$.

**Case** $\mathcal{T} \cdot e_i \rightarrow \mathcal{T}' \cdot e_i'$.

Substituting $e_i'$ for $e_i$ in $e$ produces an $e'$ such that $\mathcal{T} \cdot e \rightarrow \mathcal{T}' \cdot e'$.

**Case** $\mathcal{T} \cdot e_u \rightarrow \mathcal{T} \cdot \text{variant error}$.

By $\text{variant}^{\downarrow}_c$, $\mathcal{T} \cdot e \rightarrow \mathcal{T} \cdot \text{variant error}$.

**Case** $\mathcal{T}' \cdot e_u \rightarrow \mathcal{T}' \cdot e_u'$ and $|\mathcal{T}'| \vdash e_u' : \tau_u'$ where $\tau_u' \leq \tau_u$.

Substituting $e_u'$ for $e_u$ in $e$ produces an $e'$ such that $\mathcal{T} \cdot e \rightarrow \mathcal{T}' \cdot e'$.

**Case** $e = \text{compound}$

- $\text{import } \overline{s_i} = \overline{l_i} : \Omega \overline{y_i} = \overline{x_i} : \tau_i$
- $\text{export } \overline{s_e} = \overline{l_e} : \Omega \overline{y_e} = \overline{x_e} : \tau_e$
- $\triangleright \tau_b$
- $\text{link } e_1 \text{ with } \overline{l_{w1}} : \Omega \overline{x_{w1}} : \tau_{w1}$
  - provides $\overline{l_{p1}} : \Omega \overline{x_{p1}} : \tau_{p1}$
  - and $e_2 \text{ with } \overline{l_{w2}} : \Omega \overline{x_{w2}} : \tau_{w2}$
  - provides $\overline{l_{p2}} : \Omega \overline{x_{p2}} : \tau_{p2}$

If $e_1 = v_1$ and $e_2 = v_2$, then the $\text{compound}^{\downarrow}_c$ rule applies: by $\text{compound}^{\downarrow}_c$ and Lemma B.3.8 (Value Types), $v_1$ and $v_2$ must be units with import and export specifications that match the $\text{compound}$ expression's with and provides clauses. We can $\alpha$-rename $v_1$ and $v_2$ to avoid name clashes as required for $\text{compound}^{\downarrow}_c$. Thus, $\text{compound}^{\downarrow}_c$ defines $\mathcal{T}'$ and $e'$ when $e_1 = v_1$ and $e_2 = v_2$.

To complete the $\text{compound}$ case for $e$, we must consider all possibilities for $e_1$:

**Case** $e_1 = v_1$. 
Case \( e_2 = v_2 \).

Covered above.

Case \( \mathcal{T} \cdot e_2 \rightarrow \mathcal{T} \cdot \text{variant error} \).

By \( \text{variant}_c^+ \cdot \mathcal{T} \cdot e \rightarrow \mathcal{T} \cdot \text{variant error} \).

Case \( \mathcal{T} \cdot e_2 \rightarrow \mathcal{T}' \cdot e'_2 \) and \( |\mathcal{T}'| \vdash e'_2 : \tau'_2 \) where \( \tau'_2 \leq \tau_2 \).

Substituting \( e'_2 \) for \( e_2 \) in \( e \) produces an \( e' \) such that \( \mathcal{T} \cdot e \rightarrow \mathcal{T}' \cdot e' \).

Case \( \mathcal{T} \cdot e_1 \rightarrow \mathcal{T} \cdot \text{variant error} \).

By \( \text{variant}_c^+ \cdot \mathcal{T} \cdot e \rightarrow \mathcal{T} \cdot \text{variant error} \).

Case \( \mathcal{T} \cdot e_1 \rightarrow \mathcal{T}' \cdot e'_1 \) and \( |\mathcal{T}'| \vdash e'_1 : \tau'_1 \) where \( \tau'_1 \leq \tau_1 \).

Substituting \( e'_1 \) for \( e_1 \) in \( e \) produces an \( e' \) such that \( \mathcal{T} \cdot e \rightarrow \mathcal{T}' \cdot e' \). \( \square \)

### B.3 Supporting Lemmata

**Lemma B.3.1 (Replacement)** If \( \Gamma \vdash C[e] : \tau \) using \( \Gamma' \vdash e : \tau_c \) and if \( \Gamma' \vdash e' : \tau'_c \) where \( \tau'_c \leq \tau_c \), then \( \Gamma \vdash C[e'] : \tau' \) and \( \tau' \leq \tau \).

**Proof.** The proof is by induction on the size of the context \( C \). We partition \( C \) into \( C_1[C_2] \), where \( C_2 \) is a context of depth one. Since \( \Gamma \vdash C[e] : \tau \), \( \Gamma \vdash C_1[C_2[e]] : \tau \), and therefore \( \Gamma'' \vdash C_2[e] : \tau_0 \) for some \( \Gamma'' \) and \( \tau_0 \). We consider the possible shapes of \( C_2 \) to show that \( \Gamma'' \vdash C_2[e'] : \tau'_0 \) where \( \tau'_0 \leq \tau_0 \), which implies the lemma by induction.

Case \( C_2 = \text{fn } x : \tau_1 \Rightarrow [] \).

By \( \text{lambda}_c^+ \cdot \tau_0 = \tau_1 \rightarrow \tau_c \) and \( \tau'_0 = \tau_1 \rightarrow \tau'_c \). By the definition of \( \leq \) on function types, \( \tau'_0 \leq \tau_0 \).

Case \( C_2 = [] e_2 \).

By \( \text{app}_c^+ \cdot \Gamma'' \vdash e_2 : \tau_2 \) and \( \tau_e = \tau'_2 \rightarrow \tau_0 \) where \( \tau_2 \leq \tau'_2 \). Similarly, \( \tau'_e = \tau''_2 \rightarrow \tau'_0 \) for some \( \tau''_2 \) and \( \tau'_0 \). By assumption, \( \tau'_2 \leq \tau_e \), so \( \tau'_2 \leq \tau'_2 \). The transitivity of \( \leq \) means that \( \tau_2 \leq \tau''_2 \) so \( \Gamma'' \vdash C_2[e_2] : \tau'_0 \) by \( \text{app}_c^+ \). Furthermore, since \( \tau'_e \leq \tau_e \), \( \tau'_0 \leq \tau_0 \).

Case \( C_2 = e_2 [] \).
The \texttt{app}_
^+ \text{c} \text{ rule explicitly allows subsumption for the operand expression, so the type of } C_2[e] \text{ and } C_2[e'] \text{ is the same in } \Gamma''. \text{ In other words, } \tau'_0 = \tau_0.

\textbf{Case } C_2 = [] \text{ as } \tau.

The \texttt{generalize}_
^+ \text{c} \text{ rule explicitly allows subsumption the expression, so } \tau'_0 = \tau_0.

\textbf{Case } C_2 = [] : e_2.

The type of } C_2[e] \text{ is determined only by } e_2, \text{ which is the same in } C_2[e] \text{ and } C_2[e']. \text{ Thus, } \tau'_0 = \tau_0.

\textbf{Case } C_2 = e_1 : [].

The type of } C_2[e_0] \text{ is the type of } e_0, \text{ so } \tau_0 = \tau_0 \text{ and } \tau'_0 = \tau'_0. \text{ By assumption, } \tau'_0 \leq \tau_0, \text{ so } \tau'_0 \leq \tau_0.

\textbf{Case } C_2 = \texttt{letrec} ... \texttt{val} \ x_v : \tau_v = [] ... \texttt{in} \ e_b.

The \texttt{letrec}_
^+ \text{c} \text{ rule explicitly allows subsumption for } \texttt{val} \text{ expressions, so } \tau'_0 = \tau_0.

\textbf{Case } C_2 = \texttt{letrec} ... \texttt{in} [].

The type of } C_2[e_0] \text{ is the type of } e_0, \text{ so } \tau'_0 \leq \tau_0 \text{ (analogous to the sequence case).

\textbf{Case } C_2 = \texttt{invoke} [] ....

By \texttt{invoke}_
^+ \text{c}, \tau_0 \text{ must be a signature containing an initialization expression type } \tau_b. \text{ Since } \tau'_0 \leq \tau_0, \tau'_0 \text{ must also be a signature with fewer (or the same) imports and an initialization expression type } \tau'_0 \text{ where } \tau'_0 \leq \tau_b. \text{ By } \texttt{invoke}_
^+ \text{c}, \tau_0 = [\sigma/\ell] \tau_b \text{ and } \tau'_0 = [\sigma/\ell] \tau'_b, \text{ and by Lemma B.3.6 (Type Substitution), } \tau'_0 \leq \tau_0.

\textbf{Case } C_2 = \texttt{invoke} e_u ... y : \tau_i \leftarrow [] ....

The \texttt{invoke}_
^+ \text{c} \text{ rule explicitly allows subsumption for imported expressions, so } \tau'_0 = \tau_0.

\textbf{Case } \texttt{unit} ... \texttt{val} \ x_v : \tau_v = [] ... e_b.

The \texttt{unit}_
^+ \text{c} \text{ rule explicitly allows subsumption for } \texttt{val} \text{ expressions, so } \tau'_0 = \tau_0.

\textbf{Case } \texttt{unit} ... [].

By unit$c$. $\tau_0$ is a signature type containing an initialization expression type $\tau_e$. Similarly, $\tau'_0$ is a signature type with the same imports and exports, but the initialization expression type $\tau'_e$. Since $\tau'_e \leq \tau_e$, $\tau'_0 \leq \tau_0$ by $\leq$ for signatures.

Case compound ... link [] ... and $e_2$ .

The compound$c$ rule effectively allows subsumption for the first unit expression in compound, so $\tau'_0 = \tau_0$.

Case compound ... link $e_1$ ... and [] .

Same as he previous case. □

Lemma B.3.2 (Environment Replacement) If $\Gamma[\overline{x} : \tau] \vdash e : \tau_0$ and $\overline{\tau'} \leq \overline{\tau}$. then $\Gamma[\overline{x} : \tau'] \vdash e : \tau'_0$ and $\tau'_0 \leq \tau_0$.

Proof. Instead of replacing $\overline{\tau}$ by $\overline{\tau'}$ in $\Gamma[\overline{x} : \tau]$, we could extend the environment with bindings for fresh variables $\overline{\overline{x}_n : \tau'}$, then replace $\overline{x}$ with $\overline{x}_n$. By Lemma B.3.1 (Replacement), $\Gamma[\overline{x} : \tau, \overline{x}_n : \tau'] \vdash [\overline{x}_n / \overline{x}]e : \tau'_0$ where $\tau'_0 \leq \tau_0$. Because no $x$ is free in $[\overline{x}_n / \overline{x}]e$. $\Gamma[\overline{x}_n : \tau'] \vdash [\overline{x}_n / \overline{x}]e : \tau'_0$. We can then rename $\overline{x}_n$ to $\overline{x}$ everywhere to obtain $\Gamma[\overline{x} : \tau'] \vdash e : \tau'_0$. □

Lemma B.3.3 (Substitution) If $\Gamma[\overline{x} : \tau] \vdash e : \tau_0$ and $\Gamma \vdash \overline{v} : \overline{\tau'}$ where $\overline{\tau'} \leq \overline{\tau}$. then $\Gamma \vdash [\overline{v} / \overline{x}]e : \tau'_0$ where $\tau'_0 \leq \tau_0$.

Proof. For the one-variable case $\overline{x} = \{x\}$. apply Lemma B.3.1 (Replacement) and induction on number of replacements of $x$ in $e$. In applying Lemma B.3.1 (Replacement), we rely on Lemma B.3.4 (Environment Extension) and the implicit renaming of lexical variables by substitution. since each occurrence of $x$ appears in a potentially extended environment $\Gamma'$. Having proved the lemma for one variable, apply induction on the number of variables in $\overline{x}$ to prove the lemma for an arbitrary $\overline{x}$. □

Lemma B.3.4 (Environment Extension) If $\Gamma \vdash e : \tau_0$ and $(\overline{l} \cup \overline{x}) \cap \text{dom}(\Gamma) = \emptyset$, then $\Gamma[\overline{l} : \overline{\Omega}, \overline{x} : \tau] \vdash e : \tau_0$ and $\Gamma[\overline{l} : \overline{\Omega}, \overline{x} : \tau] \vdash \sigma : \Omega$.

Proof. Although some type rules depend on $\text{dom}(\Gamma)$, the expression being typed may always be lexically renamed to avoid any conflicts with $\overline{l}$ or $\overline{x}$ without affecting the type of the expression:
Lemma B.3.5 (Stable Typing) \( \text{If } \Gamma[l] : \Omega \vdash e : \tau_0, S = [\sigma/l], \text{ and } \Gamma \vdash \sigma :: \Omega, \text{ then } S(\Gamma) \vdash S(e) : S(\tau_0). \)

Proof. In the proof tree showing \( \Gamma \vdash e : \tau_0 \), we can replace each \( t \) with its \( \sigma \). Each proof tree’s leaf of the form \( \Gamma' \vdash t :: \Omega \) is replaced with \( \Gamma' \vdash \sigma \), which is provable by assumption (\( \Gamma' \) may have more type bindings than \( \Gamma \), but the extra bindings cannot affect the validation of \( \sigma \) by Lemma B.3.4 (Environment Extension)). Type equivalence judgements in the proof tree are clearly unaffected by the substitution, and Lemma B.3.6 (Type Substitution) shows that subtyping relations are also preserved by the substitution. Thus, the new proof tree must be a valid proof of \( S(\Gamma) \vdash S(e) : S(\tau_0). \)

Lemma B.3.6 (Type Substitution) \( \text{If } \tau \leq \tau' \text{ and } S = [\sigma/l], \text{ then } S(\tau) \leq S(\tau'). \)

Proof. No typing or subtyping rule allows any comparison between disjoint type variables, and a type variable is known only to be equivalent to itself. Thus, in the proof tree showing \( \tau \leq \tau' \), replacing each \( t \) with its \( \sigma \) produces a valid proof of \( S(\tau) \leq S(\tau') \), relying only on the equivalence of each \( \sigma \) to itself.

Lemma B.3.7 (Store Growth) \( \text{If } T \cdot e \mapsto T' \cdot e', \text{ then } \forall t \in \text{dom}(T), T'(t) = T(t) \text{ and } |T'|(|T|(t) = |T|(t)). \)

Proof. The letrec-types\( \tau \) reduction extends the type store, and no reduction contracts the store, so each reduction step must preserve or extend the type store. The environment \( |T| \) includes type and variable bindings for each type in \( T \), so extending the type store also extends the corresponding environment.

Lemma B.3.8 (Value Types) \( \text{For all } \Gamma \text{ and } v. \)
1. If $\Gamma \vdash v : t$, then $v$ is either $\text{inj}(t)v'$ or $\text{injr}(t)v'$ for some $v'$.

2. If $\Gamma \vdash v : \tau \rightarrow \tau'$, then $v$ is either $\text{fn } x: \tau \Rightarrow e$ for some $e$ or $\text{inj}(t)$. $\text{inj}(t)$, $\text{proj}(t)$. $\text{proj}(t)$. or $\text{test}(t)$ for some $t$.

3. If $\Gamma \vdash v : \tau$ and $\tau$ is a signature, then $v$ is a $\textbf{unit}$ expression.

Proof. The claim follows from inspecting the possible shapes of values and matching each shape to the applicable type rules.
Appendix C

CLASSICJAVA Proofs

C.1 Proof of Subject Reduction

The subject reduction proof is due to Shriram Krishnamurthi.

Lemma 4.1.5 (Subject Reduction) If \( P, \Gamma \vdash_{\Sigma} e : t \), \( P, \Gamma \vdash \sigma S \), \( \text{SUPEROK}(e) \), and \( P \vdash (e.S) \leftrightarrow (e'.S') \), then \( e' \) is an error configuration or there exists a \( \Gamma' \) such that

1. \( P, \Gamma' \vdash_{\Sigma} e' : t \).
2. \( P, \Gamma' \vdash_{\sigma} S' \), and
3. \( \text{SUPEROK}(e') \).

Proof. The proof examines reduction steps. For each case, if execution has not halted with an error configuration, we construct the new environment \( \Gamma' \) and show that the two consequents of the theorem are satisfied relative to the new expression, store, and environment.

Case \([\text{new}]\). Set \( \Gamma' = \Gamma \{ \text{object} : c \} \).

1. We have \( P, \Gamma \vdash_{\Sigma} E[\text{new} c] : t \). From \([\text{new}]\), \( \text{object} \notin \text{dom}(S) \). Then, by \( \Sigma_8 \), \( \text{object} \notin \text{dom}(\Gamma) \). Thus \( P, \Gamma' \vdash_{\Sigma} E[\text{new} c] : t \) by Lemma C.3.1. Since \( P, \Gamma' \vdash_{\Sigma} \text{new} c : c \) and \( P, \Gamma' \vdash_{\Sigma} \text{object} : c \), Lemma C.3.2 implies \( P, \Gamma' \vdash_{\Sigma} E[\text{object}] : t \).
2. Let \( S'(\text{object}) = (c,F) \). \( \text{object} \) is the only new element in \( \text{dom}(S') \) and \( \text{dom}(\Gamma') \).

\( \Sigma_1 \): \( \Gamma'(\text{object}) = c \).
\( \Sigma_2 \): \( \text{dom}(F) \) is correct by construction.
\( \Sigma_3 \): \( \text{rng}(F) = \{ \text{null} \} \).
\( \Sigma_4 \): Since \( \text{rng}(F) = \{ \text{null} \} \), this property is unaffected.
\( \Sigma_5 \) and \( \Sigma_6 \): The only change to \( \Gamma \) and \( S \) is \textit{object}.

3. Since \( E[\textit{object}] \) contains the same \textbf{super} expressions as \( E[\textbf{new} \, c] \), and no instance of \textit{this} or \textit{object} is replaced in the new expression. \textbf{SUPEROK}(e') holds.

\textbf{Case} \( \{\text{get}\} \).

1. Let \( t' \) be such that \( P, \Gamma \vdash \texttt{object} : c' \cdot fd : t' \). If \( v \) is \texttt{null}, it can be typed as \( t' \), so \( P, \Gamma' \vdash E[v] : t \) by Lemma C.3.2. If \( v \) is not \texttt{null}, then by \( \Sigma_4 \), \( S(v) = \langle c', \_ \rangle \) where \( c' \leq_P t' \). By Lemma C.3.4. \( P, \Gamma' \vdash E[v] : t \).

2. \( S \) and \( \Gamma \) are unchanged.

3. \textbf{SUPEROK}(e') holds because no \textbf{super} expression is changed.

\textbf{Case} \( \{\text{set}\} \).

1. The proof is by a straightforward extension of the proof for \( \{\text{get}\} \).

2. The only change to the store is a field update; thus only \( \Sigma_3 \) and \( \Sigma_4 \) are affected. Let \( v \) be the assigned value, and assume that \( v \) is not \texttt{null}.

\( \Sigma_3 \): Since \( v \) is typable, it must be in \( \text{dom}(\Gamma) \). By \( \Sigma_5 \), it is therefore in \( \text{dom}(S) \).

\( \Sigma_4 \): The typing of the assignment expression demands that the type of \( v \) can be treated as the type of the field \( fd \) by subsumption. Combining this with \( \Sigma_4 \) indicates that the type tag of \( v \) will preserve \( \Sigma_4 \).

3. \textbf{SUPEROK}(e') holds because no \textbf{super} expression is changed.

\textbf{Case} \( \{\text{call}\} \).

1. From \( P, \Gamma \vdash \texttt{object} \cdot \text{md}(v_1, \ldots \, v_n) : t \) we know \( P, \Gamma \vdash \texttt{object} : t' \). \( P, \Gamma \vdash v_i : t_i \) for \( i \) in \([1, n]\), and \( \langle \text{md}, (t_1 \ldots \, t_n \to t), \langle \text{var}_1, \ldots, \text{var}_n \rangle, e_m \rangle \in_P t' \). The type-checking of \( P \) proves that \( P, t_0 \vdash m \, t \, \text{md} \, (t_1 \, \text{var}_1, \ldots \, t_n \, \text{var}_n) \, \{e_m\} \), which implies that \( P, \langle \textbf{this} : t_0, \text{var}_1 : t_1, \ldots \, \text{var}_n : t_n \rangle \vdash e_m : t \) where \( t_0 \) is the defining class of \( \text{md} \). Further, we know that \( t' \leq_P t_0 \) from \( \in_P \) for methods and \textbf{CLASSMETHODSOK}(P). Thus, Lemma C.3.3 shows that \( P, \Gamma \vdash e_m[\texttt{object/this}, v_1/\text{var}_1, \ldots \, v_n/\text{var}_n] : t \).
2. $S$ and $\Gamma$ are unchanged.

3. The reduction may introduce new `super` expressions into the complete expression, but each new `super` must originate directly from $P$, which contains `super` expressions with `this` annotations only. The `[object/this]` part of the substitution may replace `this` in a `super` annotation with `object`, but no other part of the substitution can affect `super` annotations. Thus, $\text{SUPEROK}(\epsilon')$ holds.

**Case** `[super]`. The proof is essentially the same as the proof for `[call]`.

**Case** `[cast]`.

1. By assumption, $S(\text{object}) = (c, \_)$ where $c \leq_P t$. Since $c \leq_P t$, $P, \Gamma \vdash_S \text{object} : t$.

2. $S$ and $\Gamma$ are unchanged.

3. $\text{SUPEROK}(\epsilon')$ holds because no `super` expression is changed.

**Case** `[let]`.

1. $P, \Gamma \vdash_S \text{let} \; \text{var} = v \; \text{in} \; e : t$ implies $P, \Gamma \vdash_S v : t'$ for some type $t'$ and $P, \Gamma \vdash_S \text{var} : t' \vdash_S e : t$. By Lemma C.3.3, $P, \Gamma' \vdash_S \epsilon \ [v/var] : t$.

2. $S$ and $\Gamma$ are unchanged.

3. $\text{SUPEROK}(\epsilon')$ holds because no `super` expression is changed.

**Case** `[xcast]`, `[ncast]`, `[nget]`, `[nset]`, and `[ncall]`. $\epsilon'$ is an error configuration. □

### C.2 Proof of Progress

**Lemma 4.1.6 (Progress)** If $P, \Gamma \vdash_S e : t$, $P, \Gamma \vdash_S S$, and $\text{SUPEROK}(\epsilon)$, then either $\epsilon$ is a value or there exists an $\langle \epsilon', S' \rangle$ such that $P \vdash (\epsilon, S) \leftrightarrow \langle \epsilon', S' \rangle$.

**Proof.** The proof is by analysis of the possible cases for the current redex in $\epsilon$ (in the case that $\epsilon$ is not a value).

**Case** `new` $c$. The `[new]` reduction rules constructs the appropriate $\epsilon'$ and $S'$.
Case $v : c_{fd}$. If $v$ is null, then the [nget] reduction rule applies. Otherwise, $v = object$, and we show that [get] applies.

Type-checking combined with $\Sigma_5$ implies $S(object) = \langle c, F \rangle$ for some $c$ and $F$. Type-checking also implies that $\langle c', fd.t \rangle \in_P c$ for some $c'$ by [get]. By the definition of $\in_P (\in_P^c$ in this case), we have $c \leq_P c'$. Finally, by $\Sigma_2$, $c'.fd \in$ dom($F$).

Case $v : c_{fd} = v'$. Similar to $v : c_{fd}$, either [nset] or [set] applies.

Case $v.md(v_1, \ldots, v_n)$. If $v$ is null, then the [ncall] reduction rule applies. Otherwise, $v = object$, and [call] applies: type-checking combined with $\Sigma_5$ implies $S(object) = \langle c, F \rangle$ for some $c$ and $F$, and type-checking also implies $(md.T.V.e_m) \in_P c$.

Case super $v : c(v_1, \ldots, v_n)$. By SUPEROK(e), $v$ must be of the form object. Type-checking ensures $(md.T.V.e_m) \in_P c$.

Case view $t v$. If $v$ is null, then [ncast] applies. Otherwise, $v = object$, and by $\Sigma_5$, $S(object) = \langle c, F \rangle$ for some $c$ and $F$. Either [cast] or [xcast] applies, depending on whether $c \leq_P t$.

Case let $var = v \in e$. The [let] reduction always applies, constructing an $e'$ and $S' (= S)$. □

C.3 Supporting Lemmata

The supporting lemmata are due to Shriram Krishnamurthi.

Lemma C.3.1 (Free) If $P, \Gamma \vdash e : t$ and $a \not\in$ dom($\Gamma$), then $P, \Gamma [a : t'] \vdash e : t$.

Proof. The claim follows by reasoning about the shape of the derivation. □

Lemma C.3.2 (Replacement) If $P, \Gamma \vdash E[e] : t$, $P, \Gamma \vdash e : t'$, $P, \Gamma \vdash e' : t'$, then $P, \Gamma \vdash E[e'] : t$.

Proof. The proof is a replacement argument in the derivation tree. □

Lemma C.3.3 (Substitution) If $P, \Gamma [var_1 : t_1, \ldots, var_n : t_n] \vdash e : t$ and \{var_1, \ldots, var_n\} \cap dom($\Gamma$) = \emptyset and $P, \Gamma \vdash v_i : t_i$ for $i \in [1, n]$, then $P, \Gamma \vdash e [v_1/var_1, \ldots, v_n/var_n] : t$. 


Proof. Let $\sigma$ denote the substitution $[v_1/var_1, \ldots, v_n/var_n]$, let $\gamma$ denote the type environment $[var_1 : t_1, \ldots, var_n : t_n]$, and let $e' = \sigma(e)$. The proof proceeds by induction on the structure of the derivation showing that $P, \Gamma \vdash e : t$. We perform a case analysis on the last step in the derivation.

Case $e = \text{new } c$. Since $e' = \text{new } c$ and its type does not depend on $\Gamma$, then $P, \Gamma \vdash e' : c$.

Case $e = \text{var}$. If $\text{var} \notin \text{dom}(\sigma)$, then $e' = \text{var}$ and $\Gamma(\text{var}) = t$, so $P, \Gamma \vdash e' : t$. Otherwise, $\text{var} = \text{var}_i$ for some $i \in [1..n]$, and $e' = \sigma(\text{var}_i) = v_i$. By assumption, $P, \Gamma \vdash v_i : t_i$, so $P, \Gamma \vdash e' : t_i$.

Case $e = \text{null}$. By $[\text{null}]$, any type is derivable for null.

Case $e = e_1 : \text{c.fd}$. By $[\text{get}]$, $P, \Gamma \vdash e_1 : t'$ and $(\text{c.fd}, t) \in_P t'$ from some $t'$. By induction. $P, \Gamma \vdash \sigma(e_1) : t''$ where $t''$ is a sub-type of $t'$. Since $\in_P$ for fields is closed over subtypes on the right-hand side, $(\text{c.fd}, t) \in_P t''$. Thus, by $[\text{get}]$ on $e' = \sigma(e_1) : \text{c.fd}$, $P, \Gamma \vdash e' : t$.

Case $e = e_1 : \text{c.fd} = e_2$. This case is similar to the previous case, relying on subsumption for the right-hand side of an assignment as allowed by $[\text{set}]$.

Case $e = \text{view } t \ e_1$. By $[\text{cast}]$, $P, \Gamma \vdash e_1 : t'$ for some $t'$. By induction. $P, \Gamma \vdash \sigma(e_1) : t'$. Since $e' = \sigma(\text{view } t \ e_1) = \text{view } t \ \sigma(e_1)$, $[\text{cast}]$ gives $P, \Gamma \vdash e' : t$.

Case $e = \text{let } \text{var} = e_1 \ \text{in } e_2$. Let $\sigma_1 = \sigma$ and $\gamma_1 = \gamma$, and lexically rename $\text{var}$ in $e$ so that $\text{var} \notin \text{dom}(\gamma_1)$. By $[\text{let}]$, $P, \Gamma \gamma_1 \vdash e_1 : t_1$. By induction. $P, \Gamma \vdash \sigma_1(e_1) : t_1$. Let $\gamma_2 = [\text{var} : t_1]$, so that $P, \Gamma \gamma_2 \vdash e_2 : t$. Since $\text{var} \notin \text{dom}(\gamma_1)$, we can reverse the order of the $\gamma_1$ and $\gamma_2$ extensions to $\Gamma$, so $P, \Gamma \gamma_2 \gamma_1 \vdash e_2 : t$. By induction, $P, \Gamma \gamma_2 \gamma_1 \vdash \sigma_1(e_2) : t$. Finally, by $[\text{let}]$ on $e' = \sigma_1(\text{let } \text{var} = e_1 \ \text{in } e_2)$, $P, \Gamma \vdash e' : t$.

Case $e = e_0.\text{md } (e_1, \ldots, e_n)$. By $[\text{call}]$, $P, \Gamma \vdash e_i : t_i$ for $i \in [1..n]$ and $P, \Gamma \vdash e_0 : t_0$ such that $(\text{md}, (t_1 \ldots, t_n \rightarrow t), (\text{var}_1, \ldots, \text{var}_n), (e_m)) \in_P t_0$. By induction, $P, \Gamma \vdash \sigma(e_i) : t_i$ for each $e_i$, and $P, \Gamma \vdash \sigma(e_0) : t_0'$ where $t_0' \leq_P t_0$. Since $\in_P$ must preserve the type of methods over subtypes on the right-hand side (by $\text{CLASSMETHODSOK}$), $(\text{md}, (t_1 \ldots, t_n \rightarrow t), (\text{var}_1', \ldots, \text{var}_n'), (e_m')) \in_P t_0'$. Thus, by $[\text{call}]$ on $e' = \sigma(e_0.\text{md } (e_1, \ldots, e_n))$, $P, \Gamma \vdash e' : t$. 
Case \( \epsilon = \text{super} \equiv \text{this} : c.md \ (e_1, \ldots, e_n) \). This case is similar to the previous case. \( \square \)

**Lemma C.3.4 (Replacement with Subtyping)** If \( P, \Gamma \vdash_a E[\epsilon] : t \), \( P, \Gamma \vdash_a \epsilon : t' \), and \( P, \Gamma \vdash_a \epsilon' : t'' \) where \( t'' \leq_P t' \), then \( P, \Gamma \vdash_a E[\epsilon'] : t \).

*Proof.* The proof is by induction on the depth of the evaluation context \( E \). If \( E \) is the empty context \([\ ]\) we are done. Otherwise, partition \( E[\epsilon] = E_1[E_2[\epsilon]] \) where \( E_2 \) is a singular evaluation context, i.e., a context whose depth is one. Consider the shape of \( E_2[\bullet] \), which must be one of:

Case \( \bullet : c.fd \). Since \( c \) is fixed, \( \bullet \)'s type does not matter; the expression's type is the type of the field.

Case \( \bullet : c.fd = \epsilon \). Same as the previous case.

Case \( v : c.fd = \bullet \). Since \([\text{set}^c]\) allows subsumption on the right-hand side of the assignment, the type of the expression is the same replacing \( \bullet \) with \( \epsilon \) or \( \epsilon' \).

Case \( \bullet.md(\epsilon \ldots) \). Since \( t'' \leq_P t' \) and methods in an inheritance chain preserve the return type, the type of the expression is the same replacing \( \bullet \) with \( \epsilon \) or \( \epsilon' \).

Case \( v.md(v \ldots \bullet \ldots) \). Since \([\text{call}^c]\) allows subsumption on method arguments, the type of the expression is the same replacing \( \bullet \) with \( \epsilon \) or \( \epsilon' \).

Case \( \text{super} \equiv v : c.md (v \ldots \bullet \ldots) \). Same as the previous case.

Case \text{view} \( t \bullet \). Since \( t \) is fixed, \( \bullet \)'s type does not matter in \([\text{cast}^c]\) (our less restrictive typing rule); the expression’s type is \( t \).

Case \text{let} \( \text{var} = \bullet \) \text{in} \( e_2 \). By \([\text{let}]\) with \( P, \Gamma \vdash a e : t' \), \( P, \Gamma \gamma_1 \vdash a e_2 : t_1 \) for some type \( t_1 \) where \( \gamma_1 \) is \([\text{var} : t']\). We must show that \( P, \Gamma \gamma_2 \vdash a e_2 : t_1 \) where \( \gamma_2 = [\text{var} : t''] \). which follows from Lemma C.3.6. \( \square \)

**Definition C.3.5** \( \Gamma \leq_P \Gamma' \) if \( \text{dom}(\Gamma) = \text{dom}(\Gamma') \) and \( \forall v \in \text{dom}(\Gamma), \Gamma'(v) \leq_P \Gamma(v) \).

**Lemma C.3.6** If \( P, \Gamma \vdash a e : t \) and \( \Gamma \leq_P \Gamma' \), then \( P, \Gamma' \vdash a e : t \).

*Proof.* The proof is a straightforward adaptation of the proof for Lemma C.3.3. \( \square \)
Appendix D

**MixedJava Proofs**

### D.1 Proof of Subject Reduction

**Lemma 4.3.6 (Subject Reduction for MixedJava)** If \( P, \Gamma \vdash_\delta e : t \), \( P, \Gamma \vdash_\sigma S \), \( \text{SuperOk}(e) \), and \( \langle e, S \rangle \rightarrow \langle e', S' \rangle \), then \( e' \) is an error configuration or there exists \( \Gamma' \) such that

1. \( P, \Gamma' \vdash_\delta e' : t \).

2. \( P, \Gamma' \vdash_\sigma S' \) and

3. \( \text{SuperOk}(e') \).

**Proof.** The proof examines reduction steps. For each case, if execution has not halted with an answer or in an error configuration, we construct the new environment \( \Gamma' \) and show that the two consequents of the theorem are satisfied relative to the new expression, store, and environment.

**Case** \([\text{new}]\). Set \( \Gamma' = \Gamma [\langle \text{object} || M / m \rangle : m] \).

1. We have \( P, \Gamma \vdash_\delta \text{E}[\text{new } m] : t \). From \( \Sigma_\delta \), \( \text{object} \notin \text{dom}(S) \Rightarrow \langle \text{object} || \_ \_ \rangle \notin \text{dom}(\Gamma) \). Thus \( P, \Gamma' \vdash_\delta \text{E}[\text{new } m] : t \) by Lemma D.3.1. Since \( P, \Gamma' \vdash_\delta \text{new } m : m \) and \( P, \Gamma' \vdash_\delta \langle \text{object} || M / m \rangle : m \), Lemma D.3.2 implies \( P, \Gamma' \vdash_\delta \text{E}[\langle \text{object} || M / m \rangle] : t \).

2. Let \( S'(\text{object}) = \langle m, \mathcal{F} \rangle \), so \( \text{object} \) is the only new element in \( \text{dom}(S') \) and \( \langle \text{object} || M / m \rangle \) is the only new element in \( \text{dom}(\Gamma') \).

   \( \Sigma_1 : \Gamma'(\langle \text{object} || M / m \rangle) = m \) and \( m \leq_p m \). Since \( m \rightarrow_p M \), WF(M/m).

   \( \Sigma_2 : \text{dom}(\mathcal{F}) \) is correct by construction.

   \( \Sigma_3 : \text{rng}(\mathcal{F}) = \{\text{null}\} \).

   \( \Sigma_4 : \) Since \( \text{rng}(\mathcal{F}) = \{\text{null}\} \), this property is unaffected.
\( \Sigma_5 \) and \( \Sigma_6 \): The only addition to the domains of \( \Gamma \) and \( S \) is \text{object}.

3. Since \( E[(\text{object}|M/m)] \) contains the same \text{super} expressions as \( E[\text{new} \ m] \), and no instance of \text{this} or \text{object} is replaced in the new expression. \( \text{SUPEROK}(e') \) holds.

Case \([\text{get}]\). Set \( \Gamma' = \Gamma \).

1. If \( v \) is \text{null}, it can be typed as \( t \), so \( P, \Gamma' \vdash_E E[v] : t \) by Lemma D.3.2. If \( v \) is not \text{null}, then by \( \Sigma_4 \), \( v = (\text{object}|M/t) \) for some \text{object} and \( M \). By \( \Sigma_4 \), \( \Gamma(v) = t \), so by Lemma D.3.2, \( P, \Gamma' \vdash_E E[v] : t \).

2. \( S \) and \( \Gamma \) are unchanged.

3. \( \text{SUPEROK}(e') \) holds because no \text{super} expression is changed.

Case \([\text{set}]\). Set \( \Gamma' = \Gamma \).

1. The proof is by a straightforward extension of the proof for \([\text{get}]\).

2. The only change to the store is a field update; thus only \( \Sigma_3 \) and \( \Sigma_4 \) are affected. Let \( v \) be the assigned value, and assume that \( v \) is not \text{null}.

\( \Sigma_3 \): Since \( v \) is typable, it must be in \( \text{dom}(\Gamma) \). By \( \Sigma_5 \), its object part is therefore in \( \text{dom}(S) \).

\( \Sigma_4 \): The typing of the assignment expression indicates that the type of \( v \) is \( t \), so \( v \) must be of the form \( (\text{object}'|M''/t) \).

3. \( \text{SUPEROK}(e') \) holds because no \text{super} expression is changed.

Case \([\text{call}]\). Set \( \Gamma' = \Gamma [((\text{object})\mid m'::M'/m') : m'] \).

1. We are given \( (md,(t_1 \ldots t_n \rightarrow t),(\text{var}_1,\ldots,\text{var}_n),e_m,m'::M'/m') \in_p M_v \) in \( M_0 \), which implies \( (md,T.V.e_m) \in_p m' \) by the definition of \( \in_p \), where \( T = (t_1 \ldots t_n \rightarrow t) \) and \( V = (\text{var}_1 \ldots \text{var}_n) \).

We are also given \( P, \Gamma \vdash_E (\text{object}|M_v/t') \cdot md(v_1,\ldots,v_n) : t \), which implies \( (md,T') \in_p t' \). By Lemma D.3.6, \( T' = T \). Since the method call type-checks and \( T' = T \), \( P, \Gamma \vdash_E v_i : t_i \) for \( i \) in \([1,n]\).

Type-checking for the program \( P \) ensures that \( P,m' \vdash_m t \cdot md(t_1 \text{var}_1,\ldots,t_n \text{var}_n) \{e_m\} \), and thus \( P,[\text{this} : m', \text{var}_1 : t_1, \ldots, \text{var}_n : t_n] \vdash_E e : t \).
Hence, by Lemma D.3.3. $P.\Gamma \vdash_\varepsilon \epsilon_m[(\text{object}|m':M'/m')/\text{this. } v_1/\text{var}_1. \ldots v_n/\text{var}_n] : t$.

2. $S' = S$. If $\Gamma'$ contained a mapping for $(\text{object}|m':M'/m')$, it was $m'$ by $\Sigma_1$, so $\Gamma' = \Gamma$. Otherwise, $(\text{object}|m':M'/m')$ is new in $\Gamma'$, which might affect $\Sigma_1$, $\Sigma_5$, and $\Sigma_6$:

$\Sigma_1$: $\Gamma'((\text{object}|m':M'/m')) = m'$. The $\in_P^\mathcal{N}$ relation ensures that $m \leq_P m'$ because $\mathcal{O}_\mathcal{N} \leq^\mathcal{M} m'::M'$. $WF(m':M'/m')$ is immediate.

$\Sigma_5$ and $\Sigma_6$: Since $\Gamma((\text{object}|M_i/t')) = t'$. object $\in \text{dom}(\mathcal{S})$ by $\Sigma_5$ on $\mathcal{S}$ and $\Gamma$. Thus, adding $(\text{object}|m':M'/m')$ to $\Gamma'$ does not require any new elements in $S'$.

3. The reduction may introduce new super expressions into the complete expression, but each new super expression must originate directly from $P$, which contains super expressions with this annotations only. The $[(\text{object}|m':M'/m')/\text{this}]$ part of the substitution may replace this in a super annotation with $(\text{object}|m':M'/m')$, but no other part of the substitution can affect super annotations. Thus, $\text{SUPEROK}(e')$ holds.

Case [super]. Set $\Gamma' = \Gamma[(\text{object}|m':M'/m') : m']$.

1. Similar to [call]. The object for dispatching is $(\text{object}|m::M/m)$, and we are given $m \prec_P i$ and $(md.T) \in_P i$. (The $i$ in [super$^m$] and the $i$ in [call] are the same, since a mixin extends only one interface.) To apply Lemma D.3.6, we need $WF(M'/i)$, where $M/i \triangleright M''/i$. Lemma D.3.4 is not strong enough to guarantee $WF(M''/i)$, since $M/i$ is not necessarily well-formed. However, $\triangleright$ with an interface always produces a well-formed view on the right-hand side by construction, so $WF(M''/i)$. Thus, we can apply Lemma D.3.6 as for [call].

2. Similar to [call]. If $(\text{object}|m':M'/m')$ is new:

$\Sigma_1$: $\Gamma'((\text{object}|m':M'/m')) = m'$. If $\mathcal{S}(\text{object}) = (m_\alpha, \ldots, m_\alpha \leq_P m'$ because $\Sigma_1$ on $\Gamma$ ensures that the original view $m::M$ is part of $m_\alpha$, $\triangleright$ selects a sub-view of $M$ as $M''$, and $\in_P^\mathcal{N}$ selects $m'::M'$ within $M''$.

$\Sigma_5$ and $\Sigma_6$: Same as [call].

3. Same as [call].
Case [view]. Set $\Gamma' = \Gamma[\langle object || M'/t \rangle : t]$.

1. Since $\Gamma(\langle object || M'/t \rangle) = t$, by Lemma D.3.2, $P, \Gamma' \vdash_{E} E[\langle object || M'/t \rangle] : t$.

2. Similar to [call]. If $\langle object || M'/t \rangle \notin \Gamma$:

   \[ \Sigma_1: \Gamma'(\langle object || M'/t \rangle) = t. \] The side condition for [view] requires $t' \leq_P t$, which implies $t' \leq_P t$. \( \Sigma_1 \) on $\Gamma$ ensures $m \leq_P t'$ when $S(object) = (m, \_)$, so $m \leq_P t$ by transitivity.

   \[ \Sigma_5 \text{ and } \Sigma_6: \text{ Same as [call].} \]

3. SUPEROK($e'$) holds because no super expression is changed.

Case [cast]. Set $\Gamma' = \Gamma[\langle object || M''/t \rangle : t]$.

1. Same as [view].

2. Similar to [call]. If $\langle object || M''/t \rangle$ is new:

   \[ \Sigma_1: \Gamma'(\langle object || M''/t \rangle) = t. \] The side condition for [cast] requires $m \leq_P t$, which implies $m \leq_P t$.

   \[ \Sigma_5 \text{ and } \Sigma_6: \text{ Same as [call].} \]

3. SUPEROK($e'$) holds because no super expression is changed.

Case [let]. $P, \Gamma \vdash_{E} \text{let } \text{var} = v \text{ in } e : t$ implies $P, \Gamma \vdash_{E} v : t'$ for some type $t'$ and $P, \Gamma \vdash_{E} \text{var : t'} \vdash_{E} e : t$. Set $\Gamma' = \Gamma$.

1. By Lemma D.3.3, $P, \Gamma' \vdash_{E} e[v/var] : t$.

2. $S$ and $\Gamma$ are unchanged.

3. SUPEROK($e'$) holds because no super expression is changed.

Case [xcast], [ncast], [nget], [nsel] and [ncall]. $e'$ is an error configuration. \( \Box \)

D.2 Proof of Progress

Lemma 4.3.7 (Progress for MixedJava) If $P, \Gamma \vdash_{E} e : t$. $P, \Gamma \vdash_{E} S$. and SUPEROK($e$), then either $e$ is a value or there exists an $\langle e'.S' \rangle$ such that $\langle e.S \rangle \rightarrow \langle e'.S' \rangle$. 
Proof. The proof is by analysis of the possible cases for the current redex in e (in the case that e is not a value).

Case new m. The [new] reduction rules constructs the appropriate e' and S'.

Case v.fd. If v is null, then the [nget] reduction rule applies. Otherwise, v = \langle object \mid M/t \rangle. and we show that [get] applies.

Type-checking combined with \Sigma_5 implies S(object) = \langle m_o.F \rangle for some m_o and F. Type-checking also implies that t = m for some mixin m. and \langle m'.fd,t \rangle \in_P m for some m' by [get^m]. By the definition of \in_P (\in_P^m in this case), we have m \leq_P m'. so there is a unique m' such that M/m \triangleright M'/m', and M \leq^M M'.

Environment-store consistency implies M_o \leq^M M, where m_o \longrightarrow_P M_o. By transitivity, M_o \leq^M M'. Finally, by \Sigma_2, M'.fd \in \text{dom}(F).

Case v.fd = v'. Similar to v.fd; either [nset] or [set] applies.

Case v.md(v_1, \ldots, v_n). If v is null, then the [ncall] reduction rule applies. Otherwise, v = \langle object \mid M/t' \rangle, and we show that [call] applies.

Type-checking combined with \Sigma_5 implies S(object) = \langle m_o.F \rangle for some m_o and F. Define M_o as m_o \longrightarrow_P M_o.

By [call^m]. \langle md.T \rangle \in_P t'. The \in_P^F relation necessarily selects some m':::M'/m' and e:

1. m_x::M_x exists because WF(M/t') and \langle md.T \rangle \in_P t' implies that some atomic mixin in M contains md.

2. M_b exists because \alpha can at least relate m_x::M_x.md to itself. More specifically, we know that M_b starts with an atomic mixin m_b where \langle md.T, V, e \rangle \in_F^m m_b:

   • If there is no i such that m_x \prec^F_i i and \langle md.T \rangle \in_P i, then \langle md.T, V, e \rangle \in_F^m m_x by the definition of \in_F^m.

   • Otherwise, there must be some m_y::M_y such that M_x/i \triangleright m_y::M_y/i, or else m_o would not be a proper mixin composition. Thus, m_x::M_x.md \propto m_y::M_y.md where m_x::M_x \leq^M m_y::M_y. This argument on m_x::M_x applies inductively to m_y::M_y, showing that \langle md.T, V, e \rangle \in_F^m m_b.
3. By \( \text{WF}(M/t') \) and \( M \leq^M M_b \), then \( M_o \leq^M M_b \), so the final phase of the \( \in_P^\beta \) calculation must succeed (although the selected view is not necessarily \( M_b \)).

**Case** super \( \equiv v.(v_1, \ldots, v_n) \). By \( \text{SUPEROK} \), \( v \) must be a reference of the form \( \langle \text{object} || mL::M/m \rangle \). Thus, we show that the \([\text{super}]\) reduction applies.

Since \( m \not\prec_P i \), there must be some atomic mixin in \( M \) that implements \( i \), otherwise the object's instantiation mixin would not be a proper composition. Thus, there is some \( M' \) such that \( M/i \triangleright M'/i \). Since type-checking ensures that \( \langle md.T \rangle \in_P i \), we can apply the same reasoning as for the \( v.md(v_1, \ldots, v_n) \) case, showing that the \( \in_P^\beta \) relation necessarily selects some \( m''::M'/m' \) and \( e \).

**Case** view \( t' \text{ as } t \ v \). If \( v \) is null, then \([\text{ncast}]\) applies. Otherwise, \( v = \langle \text{object} || M/t' \rangle \) for some \( M \), and by \( S(o) = \langle m, F \rangle \) for some \( m_o \) and \( F \).

If \( t' \not\prec_P t \), then \([\text{view}]\) applies since \( M'/t \) clearly exists for \( M/t \triangleright M'/t \). Assume that \( t' \not\in_P t \). If \( m_o \not\in_P t' \), then \([\text{xcast}]\) applies. Otherwise, \([\text{cast}]\) applies since \( m_o \not\in_P t \) means that \( M''/t \) clearly exists for \( M'/m \triangleright M''/t \), where \( m \rightarrow_P M' \).

**Case** let \( var = v \in e \). The \([\text{let}]\) reduction always applies, constructing an \( e' \) and \( S' \)

\( (=S) \). \( \square \)

By combining the Subject Reduction and Progress lemmas, we can prove that every non-value MIXEDJAVA program reduces while preserving its type, thus establishing the soundness of MIXEDJAVA.

### D.3 Supporting Lemmata

**Lemma D.3.1 (Free for MIXEDJAVA)** If \( P, \Gamma \vdash e : t \) and \( a \not\in \text{dom}(\Gamma) \), then \( P, \Gamma [a : t'] \vdash e : t \).

**Proof.** This follows by reasoning about the shape of the derivation. \( \square \)

**Lemma D.3.2 (Replacement for MIXEDJAVA)** If \( P, \Gamma \vdash E[e] : t \), \( P, \Gamma \vdash e : t' \), and \( P, \Gamma \vdash e' : t' \), then \( P, \Gamma \vdash E[e'] : t \).

**Proof.** The proof is a replacement argument in the derivation tree. \( \square \)
Lemma D.3.3 (Substitution for MIXEDJAVA) If \( P, \Gamma \vdash \{ \var_1 : t_1, \ldots, \var_n : t_n \} \vdash_{\varepsilon} c : t \) and \( \{ \var_1, \ldots, \var_n \} \cap \text{dom}(\Gamma) = \emptyset \) and \( P, \Gamma \vdash \var_i : t_i \) for \( i \in \{1..n\} \), then \( P, \Gamma \vdash_{\varepsilon} \{ \var_i/\var_1, \ldots, \var_n/\var_n \} : t \).

Proof. Unlike the Substitution lemma for CLASSICJAVA, the proof of this lemma follows simply from reasoning about the shape of the derivation, since it makes no claims about subsumption. The proof uses nested induction over the number of variables \( \var_1, \ldots, \var_n \) and over the number of replacements for each variable. □

Lemma D.3.4 (\( \bullet/\bullet \triangleright \bullet/\bullet \) Preserves Well-Formedness) If \( \text{WF}(M/t) \) and \( M/t \triangleright M'/t' \), then \( \text{WF}(M'/t') \).

Proof. There are two cases, depending on whether \( t' \) is a mixin or an interface:

Case \( t \) and \( t' \) are mixins, \( m \) and \( m' \). The proof is by lexicographic induction on the length of \( M \) and size of \( m \) (i.e., the number atomic mixins composed to define \( m \)). If \( M = [m] \) (the base case), then \( t' = m \), \( M' = M \), and \( \text{WF}([m]/m') \). Also, if \( m = m' \) and \( M = M' \), then \( \text{WF}(M/m) = \text{WF}(M'/m') \). Otherwise, \( m' = m'' \circ m''' \), and there are two sub-cases:

* If \( m'' \preceq_P m' \) and \( M/m'' \triangleright M'/m' \), then \( \text{WF}(M'/m') \) by induction: \( m'' \) is smaller than \( m \) and \( \text{WF}(M/m') \) since \( m'' \) is a prefix of \( m \).

* If \( m''' \preceq_P m \) and \( M_r/m''' \triangleright M'/m' \), then \( \text{WF}(M'/m') \) by induction: \( M_r \) is smaller than \( M \), and \( \text{WF}(M_r/m''' \) because \( m''' \) is a prefix of \( M_r \).

Case \( t' \) is an interface, \( i \). \( M' \) is constructed as \( m::M'' \) where \( m \preceq_P i \). Thus, \( \text{WF}(M'/i) \) because \( M' = m::M'' \) and \( m \preceq_P i \). □

Lemma D.3.5 (Consistency of \( \bullet/\bullet \preceq \bullet/\bullet \)) If \( m::M-md \preceq m':M':md \) and \( \langle md,T \rangle \in_P m \), then \( \langle md,T \rangle \in_P m' \).

Proof. The proof is by induction on the length of \( M \). If \( M = [] \), then \( m' = m \) and \( M' = [] \), because \( \triangleright \) (used in the definition of \( \preceq \)) cannot select any chain other than \([m]\).

Otherwise, \( m \preceq_P i \) for some \( i \) where \( \langle md,T \rangle \in_P i \). Since \( M/i \triangleright M''/i \) and \( \triangleright \) preserves well-formedness of views, \( M'' \) is of the form \( m''::M''' \) for some \( m'' \) and \( M''' \).
where $m'' \prec_P i$. By MIXINSIMPLEMENTALL. MIXINMETHODSOK. $m'' \prec_P m$ and 
$\langle md.T \rangle \in_P i$. we have $\langle md.T \rangle \in_P m''$. Finally, $m''::M''md \propto m'::M'$, so $\langle md.T \rangle \in_P m'$ by induction. □

**Lemma D.3.6 (Soundness of $\bullet \in_P \bullet$ in $\bullet$)** If $\langle md.T.\_,\_,m::M/m \rangle \in_P M_v$ in $M_v$. WF($M_v/t_v$), and $\langle md.T' \rangle \in_P t_v$, then $T = T'$.

**Proof.** To get $\langle md.T.\_,\_,m::M/m \rangle$. the $\in_P$ relation first finds $m_z::M_z$ such that
$\langle md,T'' \rangle \in_P m_z$. Since WF($M_v/t_v$) and $\langle md.T' \rangle \in_P t_v$, then $T'' = T'$ by reasoning about the possible forms of $t$:

**Case** $t_v$ is an interface $i$. Then. WF($M_v/t_v$) implies $m_z::M_z = M_v$ and $m_z \subseteq_P t$. Since $m_z \subseteq_P t$, $T'' = T'$ by MIXINMETHODSOK and MIXINSIMPLEMENTALL.

**Case** $t_v$ is an atomic mixin $m'$. Then. WF($M_v/t_v$) implies $m_z::M_z = M_v$ and $m_z = t$, so $T'' = T'$ by METHODONCEPERMIXIN.

**Case** $t_v$ is a composite mixin $m'$. Then. $\langle md.T' \rangle \in_P t_v$ implies $\langle md,T'' \rangle \in_P m''$ for some $m''$ where $m' \subseteq_P m''$. which means that some candidate for $m_z$ exists within $m'$. Furthermore, $\langle md.T'' \rangle \in_P m''$ where $m' \subseteq_P m''$ implies $T'' = T''$ by the definition of $\in_P$ (because $\in_P$ eliminates ambiguities), so every candidate for $m_z$ with $m'$ gives the same type to method $md$. Since $M_v$ starts with the chain of atomic mixins of $m'$. then $m_z$ must be part of $m'$. Finally, $T'' = T'$ because $m' \subseteq_P m_z$.

Next. $\in_P$ finds an $M_b$ such that $m_z::M_z md \propto M_b md$. From WF($m_z::M_z/m_z$), $\langle md.T' \rangle \in_P m_z$, and Lemma D.3.5, the first element of $M_b$ must be an atomic mixin $m_b$ such that $\langle md,T' \rangle \in_P m_b$. Finally, the $\in_P$ relation selects a $m::M$ such that $\langle md.T.\_,\_\rangle \in_P m$ and $m::M md \propto M_b md$. Thus, by Lemma D.3.5 again, $\langle md.T \rangle \in_P m_b$. Since $m_b$ is an atomic mixin, $\in_P$ implies $\in_P$, so we have both $\langle md.T.\_,\_\rangle \in_P m_b$ and $\langle md,T'.\_,\_\rangle \in_P m_b$. By METHODONCEPERMIXIN, those must be the same method in $m_b$, so $T = T'$. □
Bibliography


www.cs.rice.edu/~matthias/411web/mo no.ps.


[29] Gamma, E., R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Massachusetts, 1994.


