RICE UNIVERSITY

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 small bits of ideas. He taught me how to bring those bits together, and how to fill in the gaps to form a coherent story. In doing so, Matthias also shaped me, bringing together small bits 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.

Keith Cooper supported me at every stage in my graduate career, from writing recommendation letters for me as first-year student to serving on my dissertation committee. He also taught me that not all well-known problems were solved in 1980—not even the ones that seem easy, such as register allocation or programming languages for software components.

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 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

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-

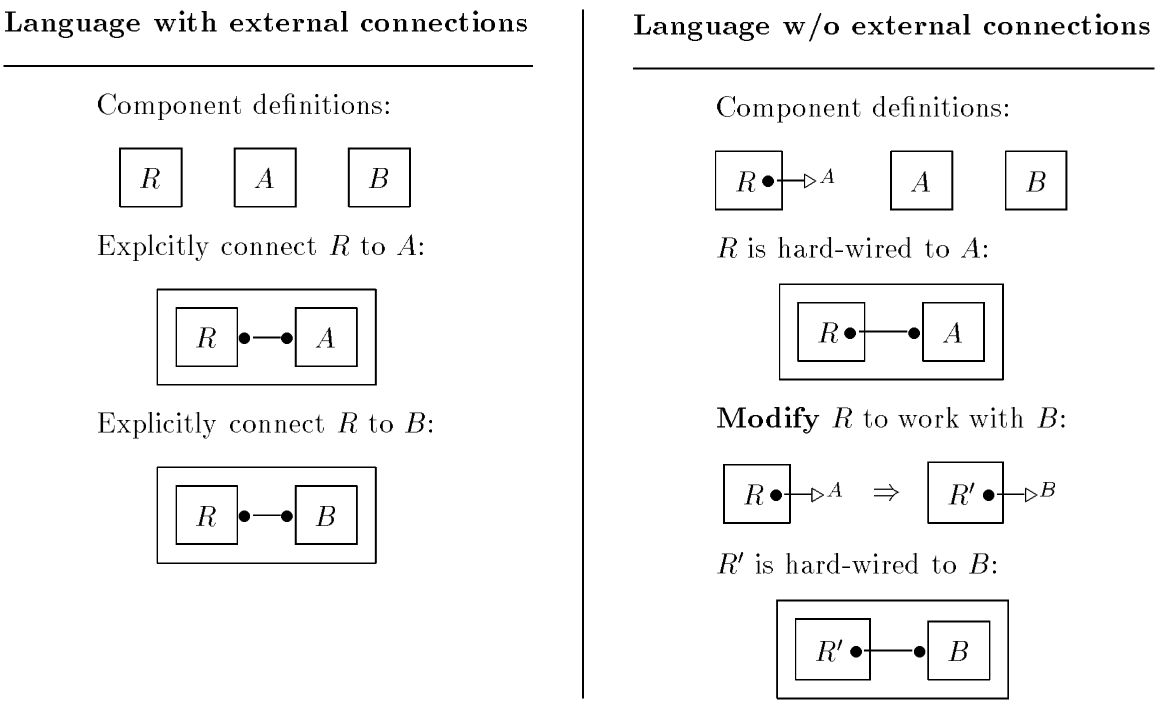


Figure 1.1 : Programming with and without external connections

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:

```
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 packages imports without modifying either package.

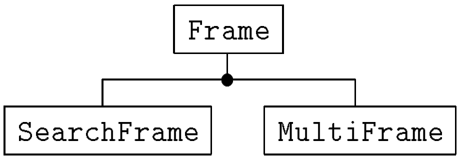
ML's module system follows the principle of external connections. An ML *functor* abstracts over a collection of definitions in the same way that a procedure abstracts over an expression. A functor imports other modules as formal arguments, describing the shape of each import using a *signature*. The 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.

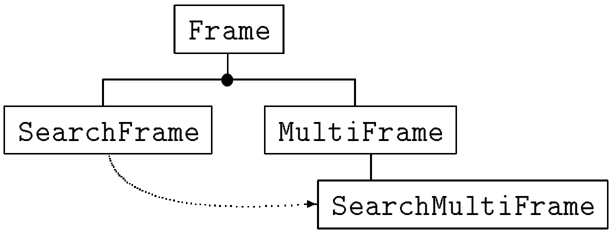
This dissertation presents a new language of modules, called *units*. Units combine the benefits of external linking specifications, graph-based linking to support mutual recursion across modules, and hierarchical linking separated from instantiation.

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 `SearchMultiFrame` inherits from both `SearchFrame` and `MultiFrame`, does an instance of `SearchMultiFrame` contain two instances of `Frame`, or just one?[2]

A language that supports *mixins* follows the principle of external connections. A *mixin* is a pure class extension; it specifies the shape of its superclass (*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 mixin and the superclass it extends, perhaps applying a single mixin to multiple superclasses. This dissertation presents a detailed model of mixins, and it demonstrates how to integrate mixins with a conventional object-oriented language, such as Java.

1.3 Dissertation Overview

Chapter 2 introduces units and 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 mixins in depth, providing models of classes and mixins for a Java-like language, with a proof of soundness for each model. Chapter 5 relates our experience using units and 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.

[2] In C++, the answer depends on whether `SearchFrame` and `MultiFrame` are declared as "virtual" subclasses of `Frame`.

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:

		original variants		extension
		$\square$	$\bigcirc$	$\leadsto$
original operations	`draw`	`draw(`$\square$`)`	`draw(`$\bigcirc$`)`	`draw(`$\leadsto$`)`
	`shrink`	`shrink(`$\square$`)`	`shrink(`$\bigcirc$`)`	`shrink(`$\leadsto$`)`
extension	`rotate`	`rotate(`$\square$`)`	`rotate(`$\bigcirc$`)`	`rotate(`$\leadsto$`)`

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 $\leadsto$) 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 *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.

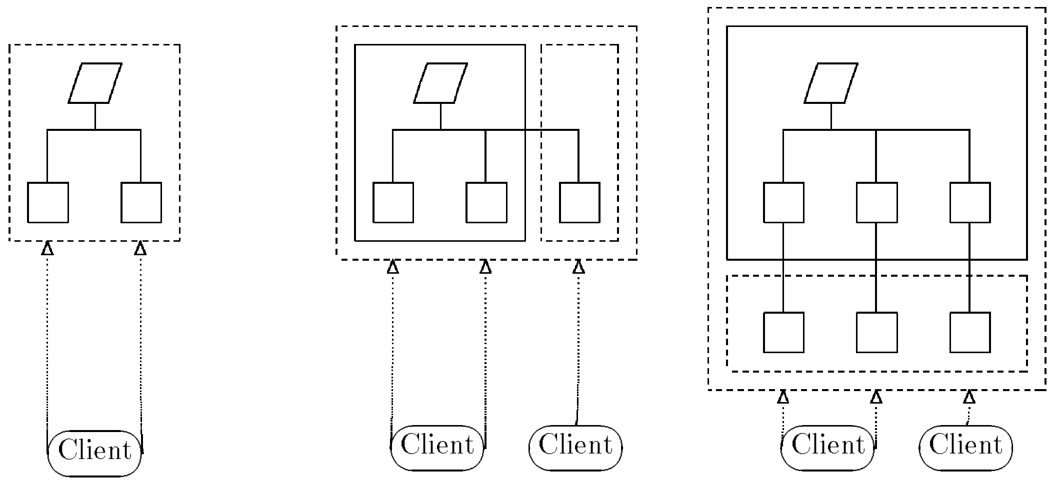
2.1 Extensible Programming with Classes

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, Δ_x and Δ_y, 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.[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

[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

$$\begin{array}{l}(\textbf{unit}\ (\textbf{import}\ \textit{variable}\ \cdots)\\ \qquad\quad (\textbf{export}\ \textit{variable}\ \cdots)\\ \quad\ \textit{unit-body-expr}\ \cdots)\end{array}$$

(where centered ellipses indicate repeated syntactic patterns). The *unit-body-expr*s 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-expr*s. Each exported variable must be defined by some *unit-body-expr*. Unexported variables that are defined in the *unit-body-expr*s 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 ...)))                    ; see Figure 2.3

(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)))

(invoke-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

$$
\begin{array}{ll}
(\textbf{compound-unit} & (\textbf{import}\ \textit{variable}\ \cdots) \\
 & (\textbf{link}\ [\textit{tag}_1\ (\textit{expr}_1\ \textit{linkspec}_1\ \cdots)] \\
 & \qquad \vdots \\
 & \qquad [\textit{tag}_n\ (\textit{expr}_n\ \textit{linkspec}_n\ \cdots)]) \\
 & (\textbf{export}\ (\textit{tag}\ \textit{variable})\ \cdots))
\end{array}
$$

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 *expr*s 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 *linkspec*s 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-expr*s 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:

(**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-SHAPES
  (compound-unit
    (import)
    (link [S (BASIC-SHAPES)]
          [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-SHAPES)]
          [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,[4] *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.

[4]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].

```
(define BB-SHAPES
  (unit (import Shape Rectangle Circle Translated Union)
        (export BB-Shape BB-Rectangle BB-Circle
                BB-Translated BB-Union BB)
    (define BB-Shape (interface (Shape) ...))           ; see Figure 2.5
    (define BB-Rectangle (class* Rectangle ...))
    (define BB-Circle (class* Circle ...))
    (define BB-Translated (class* Translated ...))
    (define BB-Union (class* Union ...))
    (define BB ...)))

(define BASIC+UNION+BB-SHAPES
  (compound-unit
   (import)
   (link [S (BASIC+UNION-SHAPES)]
         [BS (BB-SHAPES (S Shape)
                        (S Rectangle)
                        (S Circle)
                        (S Translated)
                        (S Union))]])
   (export (S Shape)
           (BS BB-Shape) (BS BB)
           ; rename BS's BB-Rectangle to Rectangle, etc.:
           (BS (BB-Rectangle Rectangle))
           (BS (BB-Circle Circle))
           (BS (BB-Translated Translated))
           (BS (BB-Union Union)))))
```

Figure 2.12 : Operation extension in a unit

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.[1] 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.[2]

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.[3] 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.

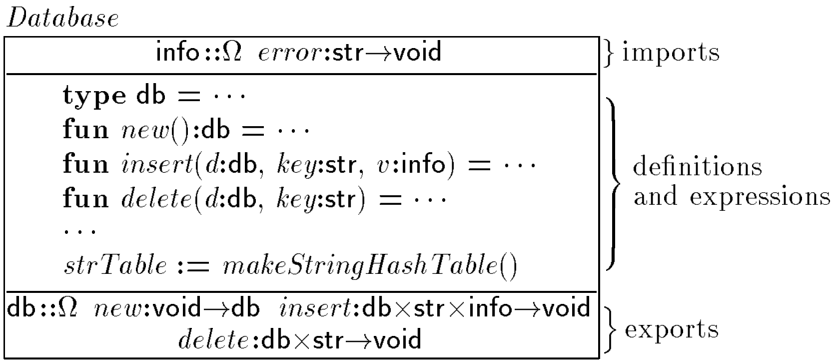


Figure 3.1 : An atomic database unit

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[4] Ω) for data stored in the database, and the function *error* (of type str$\rightarrow$void) for error-handling.

[3]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.

[4]A kind is a type for a type. Most languages have only one kind, Ω, 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*: $\mathsf{db}\times\mathsf{str}\times\mathsf{info}\rightarrow\mathsf{void}$.

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-*

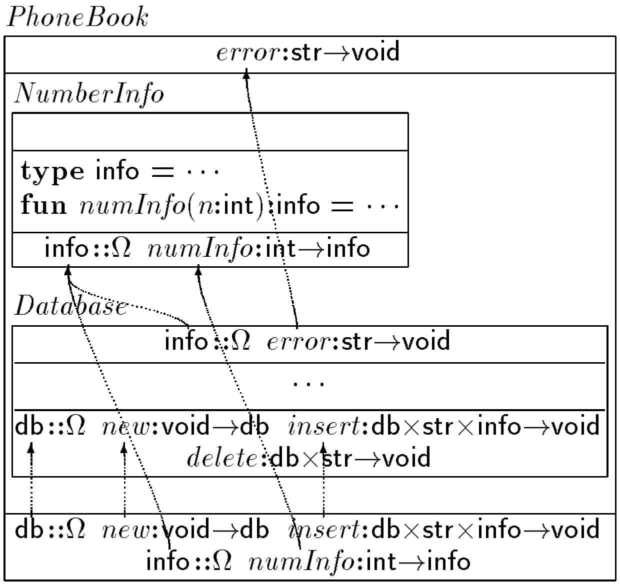


Figure 3.2 : Linking units to form a compound unit

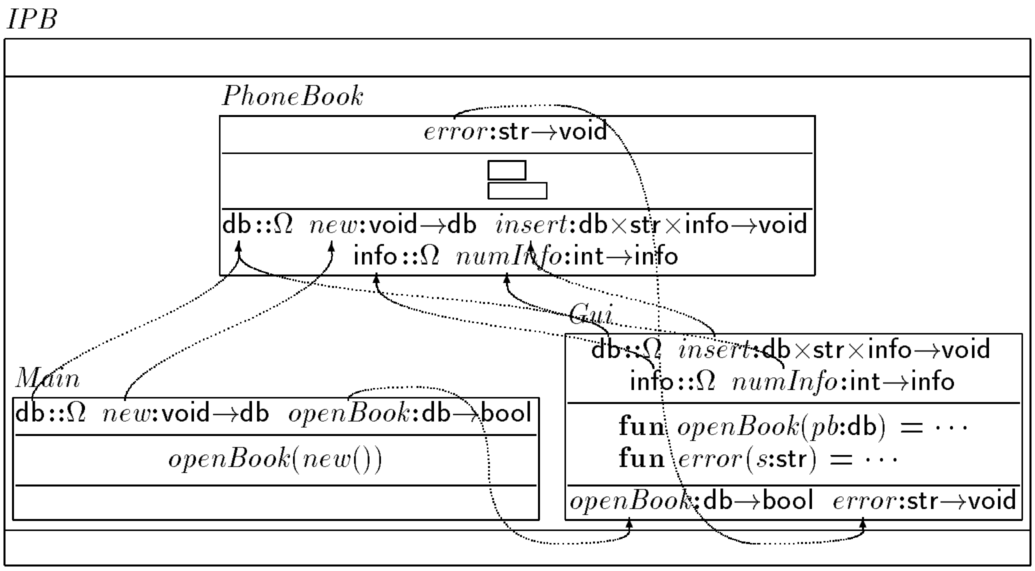


Figure 3.3 : Linking units to define a complete program

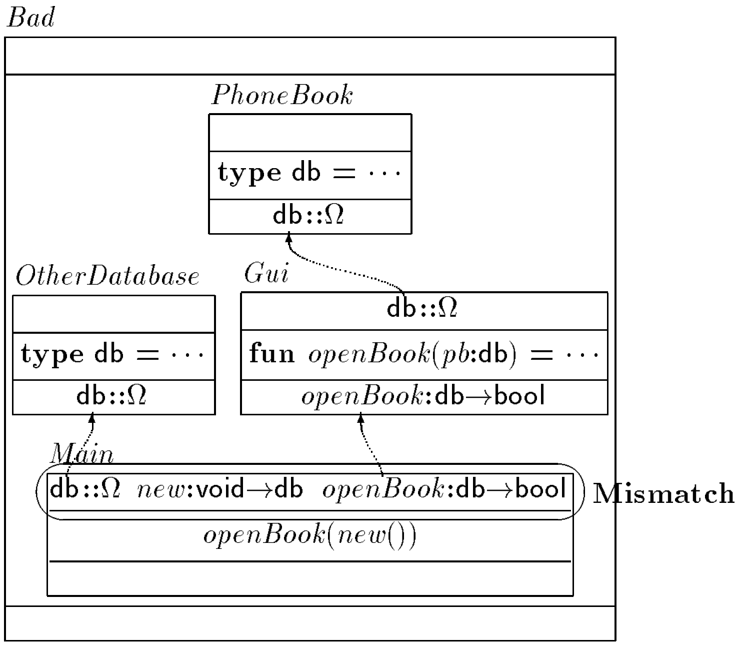


Figure 3.4 : Illegal linking due to a type mismatch

Book with a graphical interface implementation *Gui*. The *Main*[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).[6]

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

[5] The name *Main* is not special.

[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 db from *PhoneBook* unit and also the function *openBook*:db→bool from *Gui*. The two occurrences of 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 db exported by *PhoneBook*. In contrast, Figure 3.4 defines a "program" *Bad* in which *Main* receives inconsistent imports. Specifically, db and *openBook*:db→bool refer to types named db that originate from different units. The type checker correctly rejects *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 *MakeIPB*, a function that accepts a GUI unit and returns an interactive phone book unit. The programmer draws a dashed box for *aGui* and *MakeIPB* to indicate that the actual GUI and interactive phone book units are not yet determined. The programmer can then apply *MakeIPB* to different GUI implementations to produce different interactive phone book programs.

The type associated with *MakeIPB*'s argument is a unit type, a *signature*, that contains all of the information needed to verify its linkage in *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 :void indicating the type of the initialization expres-

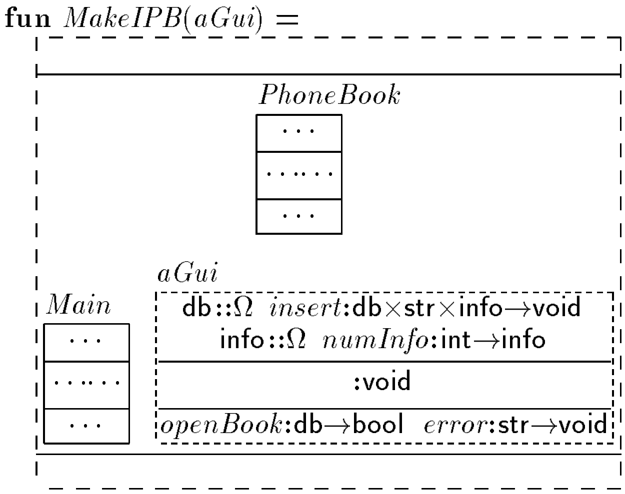


Figure 3.5 : Abstracting over constituent units

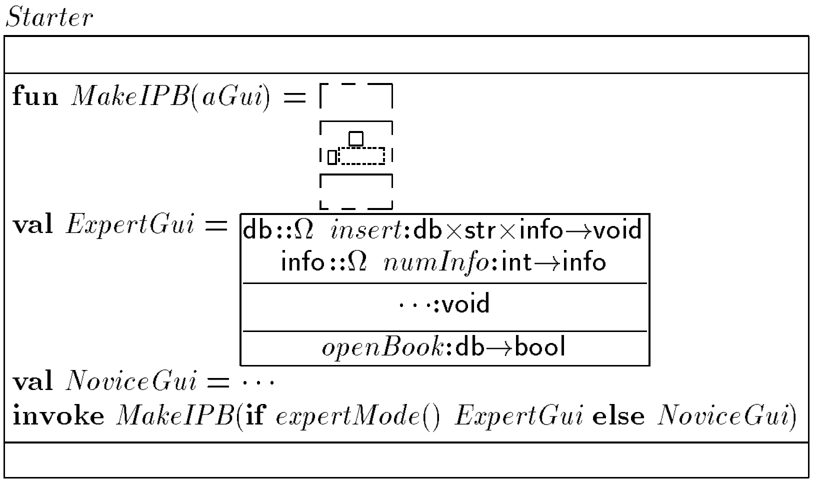


Figure 3.6 : Linking and invoking other programs

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 $\mathsf{db}\times\mathsf{file}\rightarrow\mathsf{void}$. This function is then installed into the GUI's table of loader functions.

[7] The 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].

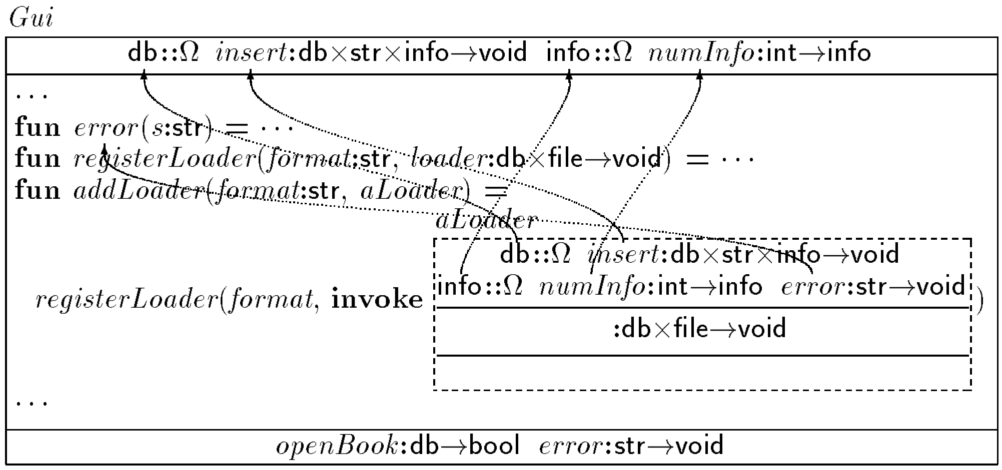


Figure 3.7 : Dynamic linking with **invoke**

3.3 Possible Extensions to Units

Experience with other modules systems, particularly those of ML, suggests further extensions to UNIT_{e}, 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.

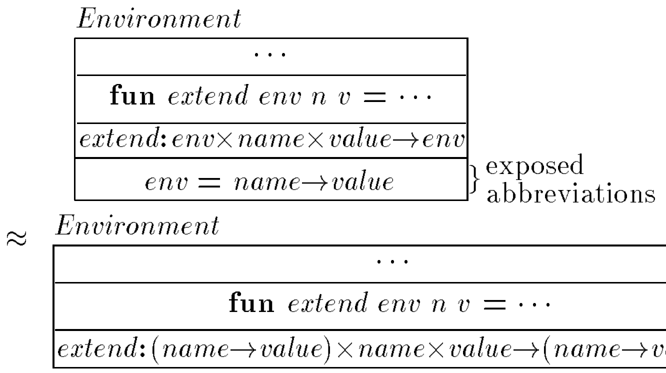


Figure 3.8 : Exposing information for a type

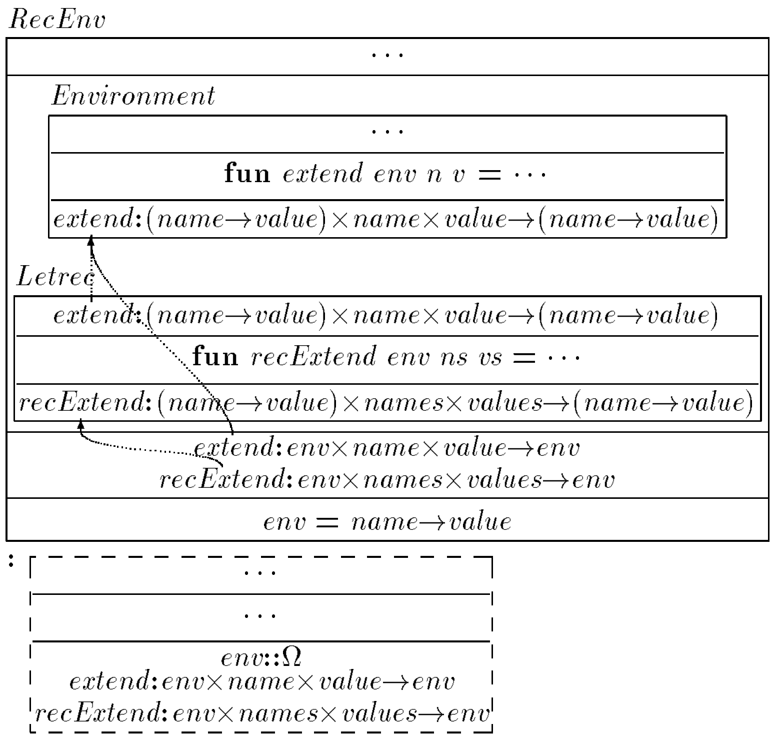


Figure 3.9 : Hiding type information for an exported value

- **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_{e} 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.

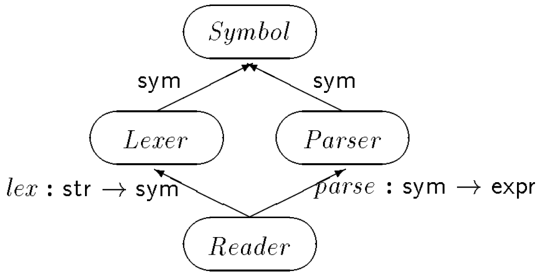


Figure 3.10 : The diamond import problem

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

```
package Symbol;
class Symbol ···
```

```
package Lexer;
import Symbol.Symbol;
class Lexer {
    Symbol lex(String s) { ··· }
}
```

```
package Parser;
import Symbol.Symbol;
class Parser {
    Expr parse(Symbol s) { ··· }
}
```

```
package Reader;
import Lexer.Lexer;
import Parser.Parser;
class Reader {
    Expr read(String s) {
        return Parser.parse(Lexer.lex(s));
    }
}
```

Figure 3.11 : Diamond import with Java packages

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

```
val Symbol = unit import
                  export sym::Ω
                  ▷ void
                  ...
```

```
val Lexer = unit import sym::Ω
                 export lex : str → sym
                 ▷ void
             fun lex(s : str) : sym = ···
```

```
val Parser = unit import sym::Ω
                  export parse : sym → expr
                  ▷ void
              fun parse(s : sym) : expr = ···
```

```
val Reader = unit import sym::Ω
                         lex : str → sym
                         parse : sym → expr
                  export read : str → expr
                  ▷ void
              fun read(s : str) : expr = parse(lex(s))
```

```
val LinkedReader = compound import
                            export read : str → expr
                            link Symbol with
                                        provides sym::Ω
                                 Lexer with sym::Ω
                                       provides lex : str → sym
                                 Parser with sym::Ω
                                        provides parse : sym → expr
                                 Reader with sym::Ω
                                             lex : str → sym
                                             parse : sym → expr
                                        provides read : str → expr
```

Figure 3.13 : Diamond import in the textual unit language

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

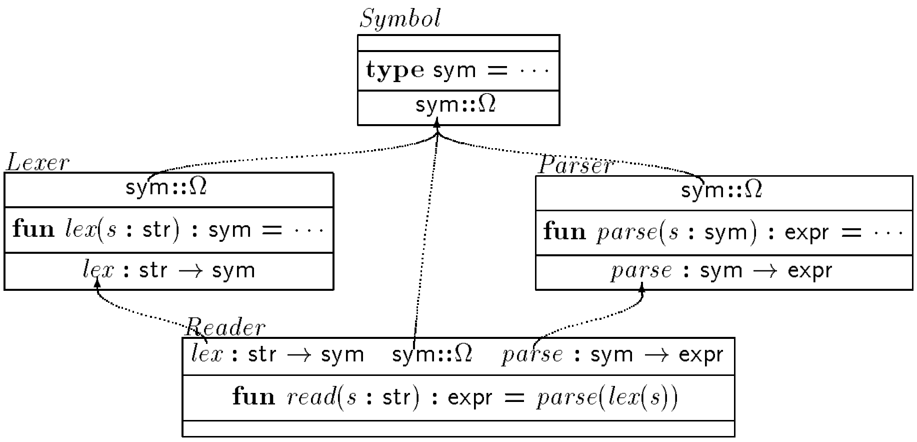


Figure 3.14 : Diamond import with units

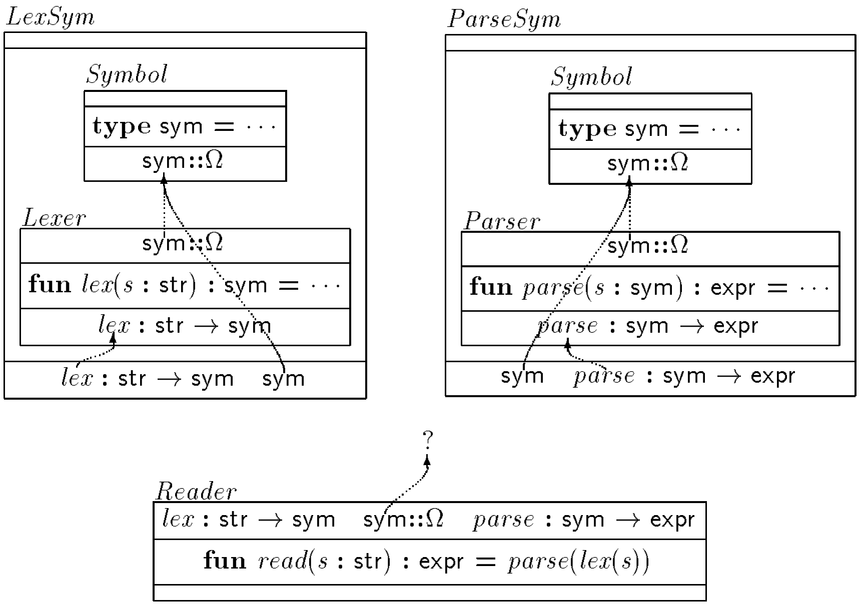


Figure 3.15 : Incorrect structure for diamond import with units

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.

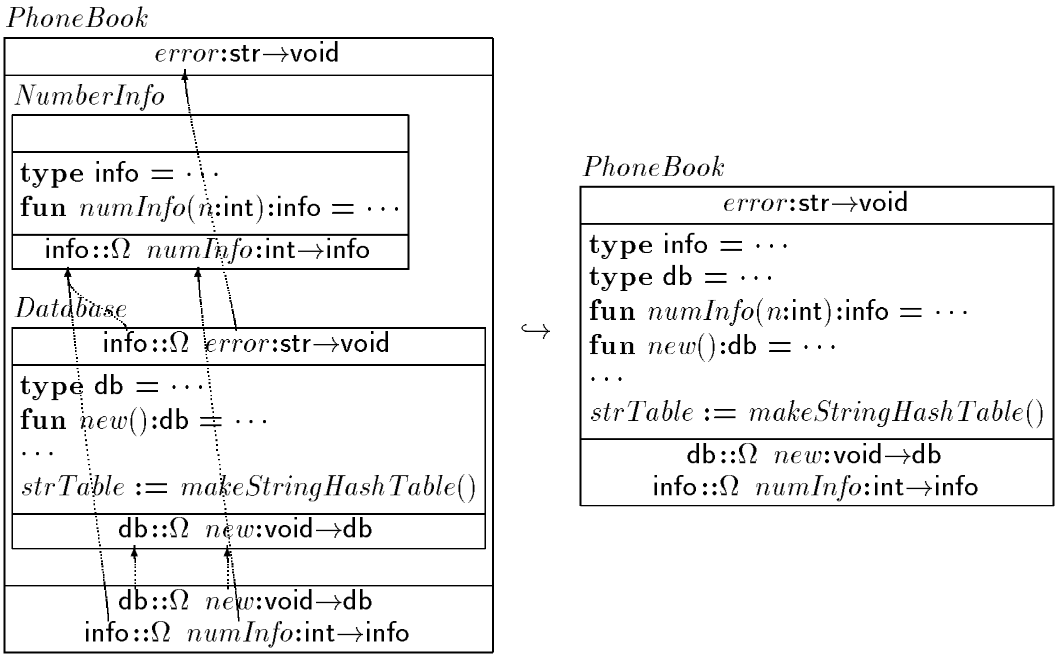


Figure 3.16 : Graphical reduction rule for a compound unit

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-

$$
\begin{array}{rcl}
v & = & \textit{unit-expr} \mid c \mid \textbf{fn}\ x \Rightarrow e \\
e & = & \textit{compound-expr} \mid \textit{invoke-expr} \mid \textit{letrec-expr} \mid e\ ;\ e \mid x \mid e\ e \mid v \\
\textit{unit-expr} & = & \textbf{unit import}\ \textit{variable-mapping}^* \\
 & & \quad \textbf{export}\ \textit{variable-mapping}^* \\
 & & \quad \textit{definitions}\ e \\
\textit{compound-expr} & = & \textbf{compound import}\ y^* \\
 & & \qquad \textbf{export}\ y^* \\
 & & \qquad \textbf{link}\ e\ \textit{link}\ \textbf{and}\ e\ \textit{link} \\
\textit{invoke-expr} & = & \textbf{invoke}\ e\ \textbf{with}\ \textit{value-invoke-link}^* \\
\textit{letrec-expr} & = & \textbf{letrec}\ \textit{definitions}\ \textbf{in}\ e \\
\textit{definitions} & = & \textit{value-defn}^* \\
\textit{value-defn} & = & \textbf{val}\ x = v \\
\textit{link} & = & \textbf{with}\ y^*\ \textbf{provides}\ y^* \\
\textit{variable-mapping} & = & y = x \\
\textit{value-invoke-link} & = & y = e \\
x & = & \text{variable} \\
y & = & \text{linking variable} \\
c & = & \text{primitive constant}
\end{array}
$$

Figure 3.17 : Syntax for $\textsc{Unit}_{\mathsf{d}}$ (dynamically typed)

strated in Figure 3.16. Invocation for a unit is similar: an **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 $\textsc{Unit}_{\mathsf{d}}$, 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 **letrec** form for lexical blocks containing mutually recursive definitions. $\textsc{Unit}_{\mathsf{d}}$ extends this core language with three unit-specific forms:

- a **unit** form for creating units,
- a **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:

$$\begin{array}{l} \textbf{unit import } y_i = x_i \cdots \textbf{ export } y_e = x_e \cdots \\ \quad \textbf{val } x = v \quad \cdots \\ \quad e \end{array}$$

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 α-renaming, but the external names y_i and y_e are not.

In each definition **val** $x = v$, 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:

$$\begin{array}{l} \textbf{compound import } y_i \cdots \textbf{ export } y_e \cdots \\ \quad \textbf{link } e_1 \textbf{ with } y_{w1} \cdots \textbf{ provides } y_{p1} \cdots \\ \quad \textbf{and } e_2 \textbf{ with } y_{w2} \cdots \textbf{ provides } y_{p2} \cdots \end{array}$$

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 UNIT$_\mathsf{d}$'s. In MzScheme, the **compound** form links any number of units together at once (a simple generalization of UNIT$_\mathsf{d}$'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:

$$\textbf{invoke}\ e\ \textbf{with}\ y_i \leftarrow e_i\ \cdots$$

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.

UNIT$_\mathsf{d}$ 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 $\overline{x}$ denotes either a set or a sequence of variables x, depending on the context. The notation $\overline{\textbf{val}\ x = e}$ denotes a set or sequence of forms $\textbf{val}\ x = e$ where each x is taken from the sequence $\overline{x}$ with a corresponding e from the sequence $\overline{e}$.

$$\Gamma \vdash c \qquad \Gamma \vdash x \text{ if } x \in \mathrm{dom}(\Gamma) \qquad \mathsf{fun}_{\mathsf{d}}^{\vdash}: \frac{\Gamma, x \vdash e}{\Gamma \vdash \textbf{fn}\ x \Rightarrow e} \qquad \mathsf{app}_{\mathsf{d}}^{\vdash}: \frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2}{\Gamma \vdash e_1\ e_2}$$

$$\mathsf{seq}_{\mathsf{d}}^{\vdash}: \frac{\Gamma \vdash e_1 \quad \Gamma \vdash e_2}{\Gamma \vdash e_1\ ;\ e_2} \qquad \mathsf{letrec}_{\mathsf{d}}^{\vdash}: \frac{\overline{x} \text{ distinct} \quad \Gamma, \overline{x} \vdash \overline{v} \quad \Gamma, \overline{x} \vdash e_b}{\Gamma \vdash \textbf{letrec}\ \overline{\textbf{val}\ x = v}\ \textbf{in}\ e_b}$$

$$\mathsf{invoke}_{\mathsf{d}}^{\vdash}: \frac{\overline{y} \text{ distinct} \quad \Gamma \vdash e_u \quad \Gamma \vdash \overline{e}}{\Gamma \vdash \textbf{invoke}\ e_u\ \textbf{with}\ \overline{y \leftarrow e}} \qquad \mathsf{unit}_{\mathsf{d}}^{\vdash}: \frac{\begin{array}{c}\overline{x_i}, \overline{x} \text{ distinct} \quad \overline{y_i}, \overline{y_e} \text{ distinct} \quad \overline{x_e} \subseteq \overline{x} \\ \Gamma, \overline{x}, \overline{x_i} \vdash \overline{v} \quad \Gamma, \overline{x}, \overline{x_i} \vdash e_b\end{array}}{\begin{array}{c}\Gamma \vdash \textbf{unit import}\ \overline{y_i = x_i}\ \textbf{export}\ \overline{y_e = x_e} \\ \overline{\textbf{val}\ x = v}\ \textbf{in}\ e_b\end{array}}$$

$$\mathsf{compound}_{\mathsf{d}}^{\vdash}: \frac{\begin{array}{c}\overline{y_i}, \overline{y_{p1}}, \overline{y_{p2}} \text{ distinct} \\ \overline{y_{w1}} \subseteq \overline{y_i} \cup \overline{y_{p2}} \quad \overline{y_{w2}} \subseteq \overline{y_i} \cup \overline{y_{p1}} \quad \overline{y_e} \subseteq \overline{y_{p1}} \cup \overline{y_{p2}} \\ \Gamma \vdash e_1 \quad \Gamma \vdash e_2\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{compound import}\ \overline{y_i}\ \textbf{export}\ \overline{y_e} \\ \qquad\qquad \textbf{link}\ e_1\ \textbf{with}\ \overline{y_{w1}}\ \textbf{provides}\ \overline{y_{p1}} \\ \qquad\qquad \textbf{and}\ e_2\ \textbf{with}\ \overline{y_{w2}}\ \textbf{provides}\ \overline{y_{p2}}\end{array}}$$

Figure 3.18 : Checking the form of UNIT_{d} expressions

UNIT_{d} Evaluation

Figure 3.19 contains the reduction rules for 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 $\mathsf{app}_{\mathsf{d}}^{\rightarrow}$, $\mathsf{seq}_{\mathsf{d}}^{\rightarrow}$, and $\mathsf{letrec}_{\mathsf{d}}^{\rightarrow}$ rules are standard.

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

The $\mathsf{compound}_{\mathsf{d}}^{\rightarrow}$ rule defines how a **compound** expression combines two units: their definitions are merged and their initialization expressions are sequenced. The

$$
\begin{array}{rcl}
E & = & [\,] \mid E\ e \mid v\ E \mid E\ ;\ e \\
 & \mid & \textbf{invoke}\ E\ \ldots \\
 & \mid & \textbf{invoke}\ v\ \textbf{with}\ \ldots\ y \leftarrow E\ \ldots \\
 & \mid & \textbf{compound}\ \ldots\ \textbf{link}\ E\ \ldots\ \textbf{and}\ e \\
 & \mid & \textbf{compound}\ \ldots\ \textbf{link}\ v\ \ldots\ \textbf{and}\ E
\end{array}
$$

$$E[e] \longmapsto E[e'] \ \text{ if } e \longrightarrow e'$$

$\mathsf{app}_{\mathsf{d}}^{\rightarrow}$:

$$(\textbf{fn}\ x \Rightarrow e)\ v \quad \longrightarrow \quad [v/x]e$$

$\mathsf{seq}_{\mathsf{d}}^{\rightarrow}$:

$$v\ ;\ e \quad \longrightarrow \quad e$$

$\mathsf{letrec}_{\mathsf{d}}^{\rightarrow}$:

$$\textbf{letrec}\ \overline{\textbf{val}\ x = v}\ \textbf{in}\ e_b \quad \longrightarrow \quad [\overline{\textbf{letrec}\ \overline{\textbf{val}\ x = v}\ \textbf{in}\ v/x}]e_b$$

$\mathsf{invoke}_{\mathsf{d}}^{\rightarrow}$:

$$
\begin{array}{l}
\textbf{invoke}\ (\textbf{unit import}\ \overline{y_i = x_i}\ \textbf{export}\ \overline{y_e = x_e} \quad \longrightarrow \quad [\overline{v_w/x_w}](\textbf{letrec}\ \overline{\textbf{val}\ x = v}\ \textbf{in}\ e_b) \\
\qquad\qquad\qquad \overline{\textbf{val}\ x = v}\ \textbf{in}\ e_b\,) \\
\qquad \textbf{with}\ \overline{y_w \ \leftarrow\ v_w}
\end{array}
$$

$$\text{if } \overline{y_i = x_i} \subseteq \overline{y_w = x_w}$$

$\mathsf{compound}_{\mathsf{d}}^{\rightarrow}$:

$$
\begin{array}{lcl}
\textbf{compound import}\ \overline{y_i}\ \textbf{export}\ \overline{y_e} & \longrightarrow & \textbf{unit import}\ \overline{y_i = x_i} \\
\qquad \textbf{link}\ (\textbf{unit import}\ \overline{y_{i1} = x_{i1}}\ \textbf{export}\ \overline{y_{e1} = x_{e1}} & & \qquad \textbf{export}\ \overline{y_e = x_e} \\
\qquad\qquad\qquad \overline{\textbf{val}\ x_1 = v_1}\ \textbf{in}\ e_{b1}\,) & & \qquad \overline{\textbf{val}\ x_1 = v_1} \\
\qquad\qquad \textbf{with}\ \overline{y_{w1}} & & \qquad \overline{\textbf{val}\ x_2 = v_2} \\
\qquad\qquad \textbf{provides}\ \overline{y_{p1}} & & \qquad \textbf{in}\ e_{b1}\ ;\ e_{b2} \\
\qquad \textbf{and}\ (\textbf{unit import}\ \overline{y_{i2} = x_{i2}}\ \textbf{export}\ \overline{y_{e2} = x_{e2}} & & \\
\qquad\qquad\qquad \overline{\textbf{val}\ x_2 = v_2}\ \textbf{in}\ e_{b2}\,) & & \\
\qquad\qquad \textbf{with}\ \overline{y_{w2}} & & \\
\qquad\qquad \textbf{provides}\ \overline{y_{p2}} & &
\end{array}
$$

if $\overline{x_1}, \overline{x_2}, \overline{x_i}$ distinct,

$\overline{y_{w1} = x_{w1}} \subseteq \overline{y_i = x_i} \cup \overline{y_{p2} = x_{p2}}$, $\overline{y_{w2} = x_{w2}} \subseteq \overline{y_i = x_i} \cup \overline{y_{p1} = x_{p1}}$, $\overline{y_e = x_e} \subseteq \overline{y_{p1} = x_{p1}} \cup \overline{y_{p2} = x_{p2}}$,
$\overline{y_{i1} = x_{i1}} \subseteq \overline{y_{w1} = x_{w1}}$, $\overline{y_{p1} = x_{p1}} \subseteq \overline{y_{e1} = x_{e1}}$, $\overline{y_{i2} = x_{i2}} \subseteq \overline{y_{w2} = x_{w2}}$, and $\overline{y_{p2} = x_{p2}} \subseteq \overline{y_{e2} = x_{e2}}$

Figure 3.19 : Reducing $\textsc{Unit}_{\mathsf{d}}$ 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.

$$
\begin{array}{l}
\textbf{unit import } \mathit{even} \\
\quad\quad \textbf{export } \mathit{odd} \\
\quad \textbf{val } \mathit{odd} = \textbf{fn } \mathit{0} \Rightarrow \textsf{false} \\
\quad\quad\quad\quad\quad\quad\quad \mid n \Rightarrow \mathit{even}\ (\mathit{n\text{-}1}) \\
\quad \mathit{odd}\ \textsf{13}
\end{array}
$$

$$\Longrightarrow$$

$$
\begin{array}{l}
\textbf{fn } (\mathit{evencell}, \mathit{oddcell}) \Rightarrow \\
\ (\mathit{oddcell} := (\textbf{fn } \mathit{0} \Rightarrow \textsf{false} \\
\quad\quad\quad\quad\quad\quad\quad \mid n \Rightarrow (!\mathit{evencell})\ (\mathit{n\text{-}1})); \\
\quad \textbf{fn } () \Rightarrow (!\mathit{oddcell})\ \textsf{13})
\end{array}
$$

Figure 3.20 : An example of UNIT_d compilation

UNIT_d Implementation

In MzScheme's implementation of UNIT_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, $\text{U}\textsc{nit}_{\mathsf{c}}$, the imports and exports of a **unit** expression include type variables as well as value variables. All type variables have a kind[10] and all value variables have a type. the type of the unit's initialization expression is also declared, following the $\triangleright$ in the unit's header. The **compound** and **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 **as** form permits explicit type generalization, casting an expression's type to a supertype.

The definition section of a **unit** or **letrec** expression contains both type and value definitions. Type definitions are similar to ML **datatype** definitions, but for simplicity, every type defined in $\text{U}\textsc{nit}_{\mathsf{c}}$ has exactly two variants. Type definitions have the form **type** $t = x_{\mathsf{cl}}, x_{\mathsf{dl}}\ \tau_{\mathsf{l}} \mid\ x_{\mathsf{cr}}, x_{\mathsf{dr}}\ \tau_{\mathsf{r}} \diamond x_{\mathsf{t}}$. Instances of the first variant are constructed with the x_{cl} function, which takes a value of type τ_{l} and constructs a value of type t. They are deconstructed with x_{dl}. Instances of the second variant are constructed with x_{cr} given a value of type τ_{r} and deconstructed with x_{dr}. Applying a deconstructor to the wrong variant signals an ML-style run-time error. To distinguish variants, the x_{t} function returns true for an instance of the first variant and false for an instance of the second. The τ_{l} and τ_{r} type expressions can refer to t or other type variables to form recursive or mutually recursive type definitions.

The type of a **unit** expression is a signature of the form **sig** *imports exports* $\triangleright\ \tau$, where *imports* specifies the kinds and types of a unit's imports, *exports* describes the kinds and types of its exports, and τ is the type of the unit's initialization expression. In a **sig** form, as in a **unit** form, type expressions for variables in *imports* or *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 α-renaming. For

---

library that is managed by the operating system.

[10] Although the only kind in this language is Ω, we declare kinds explicitly in anticipation of future work that handles type constructors.

$$
\begin{array}{rcl}
e & = & \ldots \mid e \ \textbf{as} \ \tau \\
\textit{unit-expr} & = & \textbf{unit import} \ \textit{type-mapping}* \ \textit{variable-mapping}* \\
 & & \quad \textbf{export} \ \textit{type-mapping}* \ \textit{variable-mapping}* \\
 & & \quad \rhd \ \tau \\
 & & \quad \textit{definitions} \ e \\
\textit{compound-expr} & = & \textbf{compound import} \ \textit{type-mapping}* \ \textit{value-var-decl}* \\
 & & \qquad\qquad \textbf{export} \ \textit{type-mapping}* \ \textit{value-var-decl}* \\
 & & \qquad\qquad \rhd \ \tau \\
 & & \qquad\qquad \textbf{link} \ e \ \textit{link} \ \textbf{and} \ e \ \textit{link} \\
\textit{invoke-expr} & = & \textbf{invoke} \ e \ \textbf{with} \ \textit{type-invoke-link}* \ \textit{value-invoke-link}* \\
\textit{definitions} & = & \textit{datatype-defn}* \ \textit{value-defn}* \\
\textit{datatype-defn} & = & \textbf{type} \ t = x{,}x \ \tau \mid x{,}x \ \tau \diamond x \\
\textit{value-defn} & = & \textbf{val} \ x : \tau = v \\
\textit{link} & = & \textbf{with} \ \textit{type-mapping}* \ \textit{value-var-decl}* \\
 & & \textbf{provides} \ \textit{type-mapping}* \ \textit{value-var-decl}* \\
\textit{value-var-decl} & = & y : \tau \\
\textit{type-mapping} & = & s = t :: \kappa \\
\textit{variable-mapping} & = & y = x : \tau \\
\textit{type-invoke-link} & = & s = \tau :: \kappa \\
\textit{value-invoke-link} & = & y = e : \tau \\
 & & \\
\tau, \sigma & = & t \mid s \mid \tau_{prim} \mid \tau \rightarrow \tau \mid \textit{signature} \\
\textit{signature} & = & \textbf{sig import} \ \textit{type-mapping}* \ \textit{value-var-decl}* \\
 & & \quad \textbf{export} \ \textit{type-mapping}* \ \textit{value-var-decl}* \\
 & & \quad \rhd \ \tau \\
 & & \\
t & = & \text{type variable} \\
s & = & \text{type linking variable} \\
\kappa & = & \text{type kind}
\end{array}
$$

Figure 3.21 : Syntax for $\textsc{Unit}_{\mathsf{c}}$ (constructed types)

example,

$$
\begin{array}{l}
\textbf{sig import} \ s{=}t{::}\Omega \ \ y{:}t \\
\quad \textbf{export} \\
\quad \rhd \ t
\end{array}
$$

is equivalent to

$$
\begin{array}{l}
\textbf{sig import} \ s{=}t'{::}\Omega \ \ y{:}t' \\
\quad \textbf{export} \\
\quad \rhd \ t'
\end{array}
$$

because t is α-renamed to t'. In contrast, the signature

$$\begin{array}{l}\textbf{sig import } s'{=}t{::}\Omega\ \ y{:}t \\ \quad\ \textbf{export} \\ \quad\ \triangleright\ t\end{array}$$

differs from the previous signature, because the external type name s is not subject to α-renaming.

UNIT$_{\mathsf{c}}$ Type Checking

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

$$\mathsf{sig}[i,e,b] \equiv \begin{array}[t]{l}\textbf{sig import } \overline{s_i = t_i{::}\kappa_i}\ \overline{y_i{:}\tau_i} \\ \quad\ \ \textbf{export } \overline{s_e = t_e{::}\kappa_e}\ \overline{y_e{:}\tau_e} \\ \quad\ \ \triangleright\, \tau_b\end{array}$$

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 τ_{s} is a subtype of a more general signature τ_{g} ($\tau_{\mathsf{s}} \leq \tau_{\mathsf{g}}$) if there exists an α-renaming for each signature such that:

1. the type of the initialization expression in τ_{s} is a subtype of the one in τ_{g};
2. τ_{s} has fewer imports and more exports than τ_{g};
3. for each imported variable in τ_{s}, its type in τ_{g} is a subtype of its type in τ_{s}; and
4. for each exported variable in τ_{g}, its type in τ_{s} is a subtype of its type in τ_{g}.

The typing rules for UNIT$_{\mathsf{c}}$ 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 $\vdash_{\mathsf{s}}$ 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 $\mathsf{sig}^{\vdash}_{\mathsf{c}}$ 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

$$\frac{\forall y_1{:}\tau_1 \in \overline{y_1{:}\tau_1}, \exists y_1{:}\tau_2 \in \overline{y_2{:}\tau_2} \text{ s.t. } \tau_2 \leq \tau_1}{\overline{y_1{:}\tau_1} \sqsubseteq \overline{y_2{:}\tau_2}}$$

$$\frac{\tau_{b1} \leq \tau_{b2} \quad \begin{array}{c}\overline{s_{i1} = t_{i1}{::}\kappa_{i1}} \subseteq \overline{s_{i2} = t_{i2}{::}\kappa_{i2}} \quad \overline{s_{e2} = t_{e2}{::}\kappa_{e2}} \subseteq \overline{s_{e1} = t_{e2}{::}\kappa_{e1}} \\ \overline{y_{i1}{:}\tau_{i1}} \sqsubseteq \overline{y_{i2}{:}\tau_{i2}} \quad \overline{y_{e2}{:}\tau_{e2}} \sqsubseteq \overline{y_{e1}{:}\tau_{e1}}\end{array}}{\mathsf{sig}[i1, e1, b1] \leq \mathsf{sig}[i2, e2, b2]}$$

$$\frac{\tau_1 \leq \tau_1' \quad \tau_2' \leq \tau_2}{\tau_1' \to \tau_2' \leq \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash_{\mathsf{s}} e : \tau}$$

Figure 3.22 : Subtyping and subsumption in $\text{U}_{\text{NIT}}{}_{\mathsf{c}}$ signatures

$$\Gamma \vdash \tau_{prim} :: \Omega \qquad \Gamma \vdash t :: \Gamma(t) \qquad \frac{\Gamma \vdash \tau :: \Omega \quad \Gamma \vdash \tau' :: \Omega}{\Gamma \vdash \tau \to \tau' :: \Omega}$$

$$\mathsf{sig}_{\mathsf{c}}^{\vdash}: \frac{\begin{array}{c}(\overline{t_i} \cup \overline{t_e}) \cap \mathrm{dom}(\Gamma) = \emptyset \\ \Gamma' = \Gamma, \overline{t_i{::}\kappa_i}, \overline{t_e{::}\kappa_e} \quad FTV(\tau_b) \cap \overline{t_e} = \emptyset \\ \Gamma' \vdash \tau_i :: \kappa_i \quad \Gamma' \vdash \tau_e :: \kappa_e \quad \Gamma' \vdash \tau_b :: \Omega\end{array}}{\Gamma \vdash \mathsf{sig}[i, e, b] :: \Omega}$$

$$\Gamma \vdash c : \mathit{TypeOf}(c) \qquad \Gamma \vdash x : \Gamma(x)$$

$$\mathsf{fun}_{\mathsf{c}}^{\vdash}: \frac{\Gamma \vdash \tau_1 :: \Omega \quad \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathbf{fn}\ x{:}\tau_1 \Rightarrow e : \tau_1 \to \tau_2} \qquad \mathsf{app}_{\mathsf{c}}^{\vdash}: \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash_{\mathsf{s}} e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\mathsf{seq}_{\mathsf{c}}^{\vdash}: \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \,;\, e_2 : \tau_2} \qquad \mathsf{generalize}_{\mathsf{d}}^{\vdash}: \frac{\Gamma \vdash_{\mathsf{s}} e : t}{\Gamma \vdash e\ \mathbf{as}\ t : t}$$

$$\mathsf{letrec}_{\mathsf{c}}^{\vdash}: \frac{\begin{array}{c}\overline{t}, \overline{x_{\mathsf{cl}}}, \overline{x_{\mathsf{dl}}}, \overline{x_{\mathsf{cr}}}, \overline{x_{\mathsf{dr}}}, \overline{x_{\mathsf{t}}}, \overline{x} \text{ distinct} \quad \overline{t} \cap \mathrm{dom}(\Gamma) = \emptyset \\ \Gamma' = \Gamma, \overline{t{::}\Omega} \quad \Gamma' \vdash \overline{\tau_{\mathsf{l}}} :: \Omega \quad \Gamma' \vdash \overline{\tau_{\mathsf{r}}} :: \Omega \quad \Gamma' \vdash \overline{\tau} :: \Omega \\ \Gamma'' = \Gamma', \overline{x{:}\tau}, \overline{x_{\mathsf{cl}}{:}\tau_{\mathsf{l}} \to t}, \overline{x_{\mathsf{dl}}{:}t \to \tau_{\mathsf{l}}}, \overline{x_{\mathsf{cr}}{:}\tau_{\mathsf{r}} \to t}, \overline{x_{\mathsf{dr}}{:}t \to \tau_{\mathsf{r}}}, \overline{x_{\mathsf{t}}{:}t \to \mathsf{bool}} \\ \Gamma'' \vdash_{\mathsf{s}} \overline{v} : \overline{\tau} \quad \Gamma'' \vdash e_b : \tau_b \quad FTV(\tau_b) \cap \overline{t} = \emptyset\end{array}}{\begin{array}{l}\Gamma \vdash \mathbf{letrec}\ \overline{\mathbf{type}\ t = x_{\mathsf{cl}}, x_{\mathsf{dl}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{cr}}, x_{\mathsf{dr}}\ \tau_{\mathsf{r}} \diamond x_{\mathsf{t}}} \\ \qquad\qquad \overline{\mathbf{val}\ x{:}\tau = v} \\ \qquad\qquad \mathbf{in}\ e_b \\ \quad : \tau_b\end{array}}$$

Figure 3.23 : Type checking for $\text{U}_{\text{NIT}}{}_{\mathsf{c}}$ (Part I)

$$\mathsf{invoke}^{\vdash}_{\mathsf{c}} : \frac{\begin{array}{c} \overrightarrow{s}, \overrightarrow{y} \text{ distinct} \quad \Gamma \vdash \overrightarrow{\sigma} :: \overrightarrow{\kappa} \\ \Gamma \vdash_{\mathsf{s}} \overrightarrow{e} : [\overrightarrow{\sigma/t}]\overrightarrow{\tau} \quad \Gamma \vdash e_u : \mathsf{sig}[i,e,b] \\ \mathsf{sig}[i,e,b] \leq \textbf{sig} \quad \textbf{import}\; \overrightarrow{s = t{::}\kappa}\; \overrightarrow{y{:}\tau}\; \textbf{export}\; \emptyset \rhd \tau_b \end{array}}{\Gamma \vdash \textbf{invoke}\; e_u \;\textbf{with}\; \overrightarrow{s = t{::}\kappa \;\leftarrow\; \sigma}\; \overrightarrow{y{:}\tau \;\leftarrow\; e} : [\overrightarrow{\sigma/t}]\tau_b}$$

$$\mathsf{unit}^{\vdash}_{\mathsf{c}} : \frac{\begin{array}{c} \overrightarrow{t_i}, \overrightarrow{t}, \overrightarrow{x_i}, \overrightarrow{x_{\mathsf{cl}}}, \overrightarrow{x_{\mathsf{dl}}}, \overrightarrow{x_{\mathsf{cr}}}, \overrightarrow{x_{\mathsf{dr}}}, \overrightarrow{x_{\mathsf{t}}}, \overrightarrow{x} \text{ distinct} \quad \overrightarrow{s_i}, \overrightarrow{s_e}, \overrightarrow{y_i}, \overrightarrow{y_e} \text{ distinct} \quad (\overrightarrow{t_i} \cup \overrightarrow{t}) \cap \mathrm{dom}(\Gamma) = \emptyset \\ \overrightarrow{t_e{::}\kappa_e} \subseteq \overrightarrow{t{::}\Omega} \quad \overrightarrow{x_e{:}\tau_e} \sqsubseteq \overrightarrow{x{:}\tau} \cup \overrightarrow{x_{\mathsf{cl}}{:}\tau_{\mathsf{l}} \rightarrow t} \cup \overrightarrow{x_{\mathsf{dl}}{:}t \rightarrow \tau_{\mathsf{l}}} \cup \overrightarrow{x_{\mathsf{cr}}{:}\tau_{\mathsf{r}} \rightarrow t} \cup \overrightarrow{x_{\mathsf{dr}}{:}t \rightarrow \tau_{\mathsf{r}}} \cup \overrightarrow{x_{\mathsf{t}}{:}t \rightarrow \mathsf{bool}} \\ \Gamma \vdash \mathsf{sig}[i,e,b] :: \Omega \qquad \Gamma' = \Gamma, \overrightarrow{t_i{::}\kappa_i}, \overrightarrow{t{::}\Omega} \qquad \Gamma' \vdash \overrightarrow{\tau_{\mathsf{l}}} :: \Omega \quad \Gamma' \vdash \overrightarrow{\tau_{\mathsf{r}}} :: \Omega \quad \Gamma' \vdash \overrightarrow{\tau} :: \Omega \\ \Gamma'' = \Gamma', \overrightarrow{x_i{:}\tau_i}, \overrightarrow{x{:}\tau}, \overrightarrow{x_{\mathsf{cl}}{:}\tau_{\mathsf{l}} \rightarrow t}, \overrightarrow{x_{\mathsf{dl}}{:}t \rightarrow \tau_{\mathsf{l}}}, \overrightarrow{x_{\mathsf{cr}}{:}\tau_{\mathsf{r}} \rightarrow t}, \overrightarrow{x_{\mathsf{dr}}{:}t \rightarrow \tau_{\mathsf{r}}}, \overrightarrow{x_{\mathsf{t}}{:}t \rightarrow \mathsf{bool}} \\ \Gamma'' \vdash_{\mathsf{s}} \overrightarrow{v} : \overrightarrow{\tau} \quad \Gamma'' \vdash_{\mathsf{s}} e_b : \tau_b \quad FTV(\tau_b) \cap \overrightarrow{t} = \emptyset \end{array}}{\begin{array}{l} \Gamma \vdash \textbf{unit import}\; \overrightarrow{s_i = t_i{::}\kappa_i}\; \overrightarrow{y_i = x_i{:}\tau_i}\; \textbf{export}\; \overrightarrow{s_e = t_e{::}\kappa_e}\; \overrightarrow{y_e = x_e{:}\tau_e} \rhd \tau_b \\ \qquad \overrightarrow{\textbf{type}\; t = x_{\mathsf{cl}}, x_{\mathsf{dl}}\; \tau_{\mathsf{l}} \mid x_{\mathsf{cr}}, x_{\mathsf{dr}}\; \tau_{\mathsf{r}} \diamond x_{\mathsf{t}}} \\ \qquad \overrightarrow{\textbf{val}\; x{:}\tau = v}\; \textbf{in}\; e_b \\ \quad : \mathsf{sig}[i,e,b] \end{array}}$$

$$\mathsf{compound}^{\vdash}_{\mathsf{c}} : \frac{\begin{array}{c} \overrightarrow{s_i}, \overrightarrow{s_{p1}}, \overrightarrow{s_{p2}}, \overrightarrow{y_i}, \overrightarrow{y_{p1}}, \overrightarrow{y_{p2}} \text{ distinct} \\ \overrightarrow{s_{w1} = t_{w1}{::}\kappa_{w1}} \subseteq \overrightarrow{s_i = t_i{::}\kappa_i} \cup \overrightarrow{s_{p2} = t_{p2}{::}\kappa_{p2}} \quad \overrightarrow{y_{w1}{:}\tau_{w1}} \sqsubseteq \overrightarrow{y_i{:}\tau_i} \cup \overrightarrow{y_{p2}{:}\tau_{p2}} \\ \overrightarrow{s_{w2} = t_{w2}{::}\kappa_{w2}} \subseteq \overrightarrow{s_i = t_i{::}\kappa_i} \cup \overrightarrow{s_{p1} = t_{p1}{::}\kappa_{p1}} \quad \overrightarrow{y_{w2}{:}\tau_{w2}} \sqsubseteq \overrightarrow{y_i{:}\tau_i} \cup \overrightarrow{y_{p1}{:}\tau_{p1}} \\ \overrightarrow{s_e = t_e{::}\kappa_e} \subseteq \overrightarrow{s_{p1} = t_{p1}{::}\kappa_{p1}} \cup \overrightarrow{s_{p2} = t_{p2}{::}\kappa_{p2}} \quad \overrightarrow{y_e{:}\tau_e} \sqsubseteq \overrightarrow{y_{p1}{:}\tau_{p1}} \cup \overrightarrow{y_{p2}{:}\tau_{p2}} \\ \Gamma \vdash e_1 : \mathsf{sig}[i1, e1, b1] \quad \Gamma \vdash e_2 : \mathsf{sig}[i2, e2, b2] \\ \Gamma \vdash \mathsf{sig}[w1, p1, b1] :: \Omega \quad \Gamma \vdash \mathsf{sig}[w2, p2, b] :: \Omega \\ \mathsf{sig}[i1, e1, b1] \leq \mathsf{sig}[w1, p1, b1] \quad \mathsf{sig}[i2, e2, b2] \leq \mathsf{sig}[w2, p2, b] \\ \Gamma \vdash \mathsf{sig}[i,e,b] :: \Omega \quad FTV(\tau_b) \cap (\overrightarrow{s_{p1}} \cup \overrightarrow{s_{p2}}) = \emptyset \end{array}}{\begin{array}{l} \Gamma \vdash \textbf{compound import}\; \overrightarrow{s_i = t_i{::}\kappa_i}\; \overrightarrow{y_i{:}\tau_i}\; \textbf{export}\; \overrightarrow{s_e = t_e{::}\kappa_e}\; \overrightarrow{y_e{:}\tau_e} \rhd \tau_b \\ \qquad \textbf{link}\; e_1\; \textbf{with}\; \overrightarrow{s_{w1} = t_{w1}{::}\kappa_{w1}}\; \overrightarrow{y_{w1}{:}\tau_{w1}}\; \textbf{provides}\; \overrightarrow{s_{p1} = t_{p1}{::}\kappa_{p1}}\; \overrightarrow{y_{p1}{:}\tau_{p1}} \\ \qquad \textbf{and}\; e_2\; \textbf{with}\; \overrightarrow{s_{w2} = t_{w2}{::}\kappa_{w2}}\; \overrightarrow{y_{w2}{:}\tau_{w2}}\; \textbf{provides}\; \overrightarrow{s_{p2} = t_{p2}{::}\kappa_{p2}}\; \overrightarrow{y_{p2}{:}\tau_{p2}} \\ \quad : \mathsf{sig}[i,e,b] \end{array}}$$

Figure 3.24 : Type checking for UNIT$_{\mathsf{c}}$ (Part II)

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 $\mathsf{fun}^{\vdash}_{\mathsf{c}}$, $\mathsf{app}^{\vdash}_{\mathsf{c}}$, and $\mathsf{seq}^{\vdash}_{\mathsf{c}}$ rules are standard. The $\mathsf{generalize}^{\vdash}_{\mathsf{c}}$ 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 $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$ rule checks the local value definitions and body expression for a **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

$v = \ldots \mid \mathsf{injl}\langle t\rangle \mid \mathsf{injr}\langle t\rangle \mid \mathsf{projl}\langle t\rangle \mid \mathsf{projr}\langle t\rangle \mid \mathsf{test}\langle t\rangle \mid \mathsf{injl}\langle t\rangle v \mid \mathsf{injr}\langle t\rangle v$ $\qquad \mathsf{E} = \ldots \mid \mathsf{E}\ \textbf{as}\ t$

$\mathsf{context}_{\mathsf{c}}^{\rightarrow}$: $\mathcal{T}\cdot E[e] \longmapsto \mathcal{T}'\cdot E[e']$ if $\mathcal{T}\cdot e \longrightarrow \mathcal{T}'\cdot e'$
$\mathsf{variant}_{\mathsf{c}}^{\rightarrow}$: $\mathcal{T}\cdot E[e] \longmapsto \mathcal{T}\cdot \mathsf{variant\ error}$ if $\mathcal{T}\cdot e \longrightarrow \mathcal{T}\cdot \mathsf{variant\ error}$

$\mathsf{projl}_{\mathsf{c}}^{\rightarrow}$: $\mathcal{T}\cdot \mathsf{projl}\langle t\rangle(\mathsf{injl}\langle t\rangle v) \longrightarrow \mathcal{T}\cdot v$
$\mathsf{projr}_{\mathsf{c}}^{\rightarrow}$: $\mathcal{T}\cdot \mathsf{projr}\langle t\rangle(\mathsf{injr}\langle t\rangle v) \longrightarrow \mathcal{T}\cdot v$
$\mathsf{testl}_{\mathsf{c}}^{\rightarrow}$: $\mathcal{T}\cdot \mathsf{test}\langle t\rangle(\mathsf{injl}\langle t\rangle v) \longrightarrow \mathcal{T}\cdot \mathsf{true}$

$\mathsf{projl\text{-}fail}_{\mathsf{c}}^{\rightarrow}$: $\mathcal{T}\cdot \mathsf{projl}\langle t\rangle(\mathsf{injr}\langle t\rangle v) \longrightarrow \mathcal{T}\cdot \mathsf{variant\ error}$
$\mathsf{projr\text{-}fail}_{\mathsf{c}}^{\rightarrow}$: $\mathcal{T}\cdot \mathsf{projr}\langle t\rangle(\mathsf{injl}\langle t\rangle v) \longrightarrow \mathcal{T}\cdot \mathsf{variant\ error}$
$\mathsf{testr}_{\mathsf{c}}^{\rightarrow}$: $\mathcal{T}\cdot \mathsf{test}\langle t\rangle(\mathsf{injr}\langle t\rangle v) \longrightarrow \mathcal{T}\cdot \mathsf{false}$

$\mathsf{app}_{\mathsf{c}}^{\rightarrow}$:
$\mathcal{T}\cdot (\textbf{fn}\ x{:}\tau \Rightarrow e)\ v \longrightarrow \mathcal{T}\cdot [v/x]e$

$\mathsf{seq}_{\mathsf{c}}^{\rightarrow}$:
$\mathcal{T}\cdot v\ ;\ e \longrightarrow \mathcal{T}\cdot e$

$\mathsf{generalize}_{\mathsf{c}}^{\rightarrow}$:
$\mathcal{T}\cdot v\ \textbf{as}\ t \longrightarrow \mathcal{T}\cdot v$

$\mathsf{letrec}_{\mathsf{c}}^{\rightarrow}$:
$\mathcal{T}\cdot \textbf{letrec}\ \overline{\textbf{val}\ x{:}\tau = v}\ \textbf{in}\ e_b \longrightarrow \mathcal{T}\cdot [\overline{\textbf{letrec}\ \overline{\textbf{val}\ x{:}\tau = v}\ \textbf{in}\ v/x}]e_b$

$\mathsf{letrec\text{-}types}_{\mathsf{c}}^{\rightarrow}$:

$$\begin{array}{ll} \mathcal{T}\cdot \textbf{letrec} & \overline{\textbf{type}\ t = x_{\mathsf{cl}}, x_{\mathsf{dl}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{cr}}, x_{\mathsf{dr}}\ \tau_{\mathsf{r}} \diamond x_{\mathsf{t}}} \quad \longrightarrow \quad \mathcal{T}[\overline{t \mapsto \langle \tau_{\mathsf{l}}, \tau_{\mathsf{r}}\rangle}]\cdot S(\textbf{letrec}\ \overline{\textbf{val}\ x{:}\tau = v}\ \textbf{in}\ e_b) \\ & \overline{\textbf{val}\ x{:}\tau = v} \\ & \textbf{in}\ e_b \end{array}$$

if $\overline{t} \cap \mathrm{dom}(\mathcal{T}) = \emptyset$
where $S = [\overline{\mathsf{injl}\langle t\rangle/x_{\mathsf{cl}}}, \overline{\mathsf{injr}\langle t\rangle/x_{\mathsf{cr}}}, \overline{\mathsf{projl}\langle t\rangle/x_{\mathsf{dl}}}, \overline{\mathsf{projr}\langle t\rangle/x_{\mathsf{dr}}}, \overline{\mathsf{test}\langle t\rangle/x_{\mathsf{t}}}]$

Figure 3.25 : Reduction rules for $\textsc{Unit}_{\mathsf{c}}$ (Part I)

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

The $\mathsf{unit}_{\mathsf{c}}^{\vdash}$ rule determines the signature of a **unit** expression. The first two lines of antecedents contain simple context-sensitive syntax checks as in $\textsc{Unit}_{\mathsf{d}}$. 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

$\mathsf{invoke}_{\mathsf{c}}^{\rightarrow}$:

$$
\begin{array}{l}
\mathcal{T}\cdot \textbf{invoke}\ (\textbf{unit import}\ \overline{s_i = t_i{::}\kappa_i}\ \overline{y_i = x_i{:}\tau_i} \\
\qquad\qquad \textbf{export}\ \overline{s_e = t_e{::}\kappa_e}\ \overline{y_e = x_e{:}\tau_e} \\
\qquad\qquad \triangleright\ \tau_b \\
\qquad\qquad \overline{\textbf{type}\ t = x_{\mathsf{cl}},x_{\mathsf{dl}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{cr}},x_{\mathsf{dr}}\ \tau_{\mathsf{r}} \diamond x_{\mathsf{t}}} \\
\qquad\qquad \overline{\textbf{val}\ x{:}\tau = v}\ \textbf{in}\ e_b) \\
\qquad \textbf{with}\ \overline{s_w = t_w{::}\kappa_w\ \leftarrow\ \sigma_w}\ \overline{y_w{:}\tau_w\ \leftarrow\ v_w} \\
\longrightarrow\quad \mathcal{T}\cdot [\overline{\sigma_w/t_w}, \overline{v_w/x_w}](\textbf{letrec}\ \overline{\textbf{type}\ t = x_{\mathsf{cl}},x_{\mathsf{dl}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{cr}},x_{\mathsf{dr}}\ \tau_{\mathsf{r}} \diamond x_{\mathsf{t}}} \\
\qquad\qquad\qquad\qquad\qquad \overline{\textbf{val}\ x{:}\tau = v}\ \textbf{in}\ e_b\ \textbf{as}\ \tau_b)
\end{array}
$$

if $\overline{s_i = t_i} \subseteq \overline{s_w = t_w}$ and $\overline{y_i = x_i} \subseteq \overline{y_w = x_w}$

$\mathsf{compound}_{\mathsf{c}}^{\rightarrow}$:

$$
\begin{array}{l}
\mathcal{T}\cdot \textbf{compound import}\ \overline{s_i = t_i{::}\kappa_i}\ \overline{y_i{:}\tau_i}\ \textbf{export}\ \overline{s_e = t_e{::}\kappa_e}\ \overline{y_e{:}\tau_e} \triangleright \tau_b \\
\qquad \textbf{link}\ (\textbf{unit import}\ \overline{s_{i1} = t_{i1}{::}\kappa_{i1}}\ \overline{y_{i1} = x_{i1}{:}\tau_{i1}} \\
\qquad\qquad \textbf{export}\ \overline{s_{e1} = t_{e1}{::}\kappa_{e1}}\ \overline{y_{e1} = x_{e1}{:}\tau_{e1}} \\
\qquad\qquad \triangleright\ \tau_{b1} \\
\qquad\qquad \overline{\textbf{type}\ t_1 = x_{\mathsf{cl1}},x_{\mathsf{dl1}}\ \tau_{\mathsf{l1}} \mid x_{\mathsf{cr1}},x_{\mathsf{dr1}}\ \tau_{\mathsf{r1}} \diamond x_{\mathsf{t1}}} \\
\qquad\qquad \overline{\textbf{val}\ x_1{:}\tau_1 = v_1} \\
\qquad\qquad \textbf{in}\ e_{b1}) \\
\qquad\quad \textbf{with}\ \overline{s_{w1} = t_{w1}{::}\kappa_{w1}}\ \overline{y_{w1}{:}\tau_{w1}} \\
\qquad\quad \textbf{provides}\ \overline{s_{p1} = t_{p1}{::}\kappa_{p1}}\ \overline{y_{p1}{:}\tau_{p1}} \\
\qquad \textbf{and}\ (\textbf{unit import}\ \overline{s_{i2} = t_{i2}{::}\kappa_{i2}}\ \overline{y_{i2} = x_{i2}{:}\tau_{i2}} \\
\qquad\qquad \textbf{export}\ \overline{s_{e2} = t_{e2}{::}\kappa_{e2}}\ \overline{y_{e2} = x_{e2}{:}\tau_{e2}} \\
\qquad\qquad \triangleright\ \tau_{b2} \\
\qquad\qquad \overline{\textbf{type}\ t_2 = x_{\mathsf{cl2}},x_{\mathsf{dl2}}\ \tau_{\mathsf{l2}} \mid x_{\mathsf{cr2}},x_{\mathsf{dr2}}\ \tau_{\mathsf{r2}} \diamond x_{\mathsf{t2}}} \\
\qquad\qquad \overline{\textbf{val}\ x_2{:}\tau_2 = v_2} \\
\qquad\qquad \textbf{in}\ e_{b2}) \\
\qquad\quad \textbf{with}\ \overline{s_{w2} = t_{w2}{::}\kappa_{w2}}\ \overline{y_{w2}{:}\tau_{w2}} \\
\qquad\quad \textbf{provides}\ \overline{s_{p2} = t_{p2}{::}\kappa_{p2}}\ \overline{y_{p2}{:}\tau_{p2}} \\
\longrightarrow\quad \mathcal{T}\cdot \textbf{unit import}\ \overline{s_i = t_i{::}\kappa_i}\ \overline{y_i = x_i{:}\tau_i} \\
\qquad\qquad \textbf{export}\ \overline{s_e = t_e{::}\kappa_e}\ \overline{y_e = x_e{:}\tau_e} \\
\qquad\qquad \triangleright\ \tau_b \\
\qquad\qquad \overline{\textbf{type}\ t_1 = x_{\mathsf{cl1}},x_{\mathsf{dl1}}\ \tau_{\mathsf{l1}} \mid x_{\mathsf{cr1}},x_{\mathsf{dr1}}\ \tau_{\mathsf{r1}} \diamond x_{\mathsf{t1}}} \\
\qquad\qquad \overline{\textbf{type}\ t_2 = x_{\mathsf{cl2}},x_{\mathsf{dl2}}\ \tau_{\mathsf{l2}} \mid x_{\mathsf{cr2}},x_{\mathsf{dr2}}\ \tau_{\mathsf{r2}} \diamond x_{\mathsf{t2}}} \\
\qquad\qquad \overline{\textbf{val}\ x_1{:}\tau_1 = v_1} \\
\qquad\qquad \overline{\textbf{val}\ x_2{:}\tau_2 = v_2} \\
\qquad\qquad \textbf{in}\ e_{b1}\ ;\ e_{b2}
\end{array}
$$

if $\overline{t_1}, \overline{t_2}, \overline{t_i}, \overline{x_1}, \overline{x_{\mathsf{cl1}}}, \overline{x_{\mathsf{dl1}}}, \overline{x_{\mathsf{cr1}}}, \overline{x_{\mathsf{dr1}}}, \overline{x_{\mathsf{t1}}}, \overline{x_2}, \overline{x_{\mathsf{cl2}}}, \overline{x_{\mathsf{dl2}}}, \overline{x_{\mathsf{cr2}}}, \overline{x_{\mathsf{dr2}}}, \overline{x_{\mathsf{t2}}}, \overline{x_i}$ distinct,
$\overline{s_{w1} = t_{w1}} \subseteq \overline{s_i = t_i} \cup \overline{s_{p2} = t_{p2}}$, $\overline{s_{w2} = t_{w2}} \subseteq \overline{s_i = t_i} \cup \overline{s_{p1} = t_{p1}}$, $\overline{s_e = t_e} \subseteq \overline{s_{p1} = t_{p1}} \cup \overline{s_{p2} = t_{p2}}$,
$\overline{y_{w1} = x_{w1}} \subseteq \overline{y_i = x_i} \cup \overline{y_{p2} = x_{p2}}$, $\overline{y_{w2} = x_{w2}} \subseteq \overline{y_i = x_i} \cup \overline{y_{p1} = x_{p1}}$, $\overline{y_e = x_e} \subseteq \overline{y_{p1} = x_{p1}} \cup \overline{y_{p2} = x_{p2}}$,
$\overline{s_{i1} = t_{i1}} \subseteq \overline{s_{w1} = t_{w1}}$, $\overline{s_{p1} = t_{p1}} \subseteq \overline{s_{e1} = t_{e1}}$, $\overline{s_{i2} = t_{i2}} \subseteq \overline{s_{w2} = t_{w2}}$, $\overline{s_{p2} = t_{p2}} \subseteq \overline{s_{e2} = t_{e2}}$,
$\overline{y_{i1} = x_{i1}} \subseteq \overline{y_{w1} = x_{w1}}$, $\overline{y_{p1} = x_{p1}} \subseteq \overline{y_{e1} = x_{e1}}$, $\overline{y_{i2} = x_{i2}} \subseteq \overline{y_{w2} = x_{w2}}$, and $\overline{y_{p2} = x_{p2}} \subseteq \overline{y_{e2} = x_{e2}}$

Figure 3.26 : Reduction rules for $\text{U}{\scriptstyle\text{NIT}}_{\mathsf{c}}$(Part II)

$$|[\overline{t \mapsto \langle \tau_{\mathsf{l}}, \tau_{\mathsf{r}} \rangle}]| = [\overline{\mathsf{injl}\langle t \rangle{:}\tau_{\mathsf{l}} \rightarrow t}, \overline{\mathsf{injr}\langle t \rangle{:}\tau_{\mathsf{r}} \rightarrow t}, \overline{\mathsf{projl}\langle t \rangle{:}t \rightarrow \tau_{\mathsf{l}}}, \overline{\mathsf{projr}\langle t \rangle{:}t \rightarrow \tau_{\mathsf{r}}}, \overline{\mathsf{test}\langle t \rangle{:}t \rightarrow \mathsf{bool}}]$$

Figure 3.27 : Converting a type store to an environment

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 $\mathsf{compound}_{\mathsf{c}}^{\vdash}$ 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.

UNIT$_{\mathsf{c}}$ Evaluation

A reduction semantics for UNIT$_{\mathsf{c}}$ must account for the local type definitions introduced by a **unit** or **letrec** expression. The rules in Figures 3.25 and 3.26 model such types through a type store $\mathcal{T}$, where each reduction maps a store-expression pair $\mathcal{T} \cdot e$ to a new store-expression pair $\mathcal{T}' \cdot e'$. A type store $\mathcal{T}$ maps each type $t \in \mathrm{dom}(\mathcal{T})$ to type expressions τ_{l} and τ_{r} for the type's "left" and "right" variants, respectively. Intermediate expressions in a reduction include pseudo-variables, such as $\mathsf{injl}\langle t \rangle$, which correspond to constructors and selectors for the type t. Type checking treats pseudo-variables as type variables that are bound in the environment $|\mathcal{T}|$, which is the unloading of the type store $\mathcal{T}$ to a type environment (see Figure 3.27).

The $\mathsf{letrec\text{-}types}_{\mathsf{c}}^{\rightarrow}$ 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 x_{cl} with $\mathsf{injl}\langle t \rangle$, etc., within the **letrec** expression. The other reduction rules for UNIT$_{\mathsf{c}}$ closely resemble the rules for UNIT$_{\mathsf{d}}$ in Figure 3.19. Whereas the side conditions for $\mathsf{invoke}_{\mathsf{d}}^{\rightarrow}$ and $\mathsf{compound}_{\mathsf{d}}^{\rightarrow}$ in the untyped semantics enforce safety, the side conditions for $\mathsf{invoke}_{\mathsf{c}}^{\rightarrow}$ and $\mathsf{compound}_{\mathsf{c}}^{\rightarrow}$ in the typed semantics serve merely to require an appropriate α-renaming of the units.

$\textsc{Unit}_{\mathsf{c}}$ Soundness

For a program of type τ, the evaluation rules for $\textsc{Unit}_{\mathsf{c}}$ produce either a value that has a subtype of τ or $\mathsf{variant\ error}$; 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. $[\,] \cdot e \ \longmapsto^* \ \mathcal{T} \cdot \mathsf{variant\ error}$*; or*
3. $[\,] \cdot e \ \longmapsto^* \ \mathcal{T} \cdot v$, $|\mathcal{T}| \vdash v : \tau_0'$*, and* $\tau_0' \ \leq \ \tau_0$.

Proof. Lemma 3.5.3 (Progress) shows that a non-value expression reduces either to $\mathsf{variant\ error}$ or to another expression. Thus, a reduction for e either never ends, ends in $\mathsf{variant\ error}$, 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 $[\,] \cdot e \ \longmapsto^* \ \mathcal{T} \cdot v$ therefore proves the theorem. $\square$

Lemma 3.5.2 (Subject Reduction) *If* $\mathcal{T} \cdot e \ \longmapsto \ \mathcal{T}' \cdot e'$ *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 $\mathcal{T} \cdot e \ \longmapsto \ \mathcal{T}' \cdot e'$. 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. $\mathcal{T} \cdot e \ \longmapsto \ \mathcal{T} \cdot \mathsf{variant\ error}$*; or*
3. $\mathcal{T} \cdot e \ \longmapsto \ \mathcal{T}' \cdot e'$ *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 Λ: **val** *apply* =
$\Lambda\ \alpha{::}\Omega \Rightarrow$
fn $f : (\alpha \rightarrow \alpha) \Rightarrow$ (**fn** $x : \alpha \Rightarrow (f\ x)$)

apply[bool] not true

Polymorphism with **unit**: **val** *apply* =
unit import $\alpha{::}\Omega$
export
$\triangleright\ (\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$
fn $f : (\alpha \rightarrow \alpha) \Rightarrow$ (**fn** $x : \alpha \Rightarrow (f\ x)$)

(**invoke** *apply* **with** $\alpha{::}\Omega \leftarrow$ bool) not true

Figure 3.28 : Polymorphism with Λ versus **unit**

UNIT$_c$ and Polymorphism

Although our UNIT$_c$ model does not support polymorphic functions directly, units can encode polymorphic functions in a straightforward way. Figure 3.28 shows how **unit** with **invoke** supports polymorphic functions in the same manner as Λ with type application.

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

UNIT$_c$ Implementation

Closed units in UNIT$_c$ can be compiled separately in the same way as closed functors 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

$$\begin{array}{rcl}
\textit{definitions} & = & \textit{type-defn}^* \ \textit{datatype-defn}^* \ \textit{value-defn}^* \\
\textit{type-defn} & = & \textbf{type} \ t :: \kappa = \sigma \\
\textbf{signature} & = & \textbf{sig} \ \textbf{import} \ \textit{type-mapping}^* \ \textit{value-var-decl}^* \\
& & \quad \textbf{export} \ \textit{type-mapping}^* \ \textit{value-var-decl}^* \\
& & \quad \textbf{depends} \ \textit{dependency}^* \ \tau \\
\textit{dependency} & = & s \leadsto s
\end{array}$$

Figure 3.29 : Syntax for UNIT_e (type equations)

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

UNIT_c 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 **type** $t = \tau$ defines the type variable t as an abbreviation for the type expression τ.

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.

UNIT_e extends UNIT_c with type dependencies and equations. Figure 3.29 defines syntax extensions for UNIT_e, including a new signature form that contains a **depends** clause. The dependency declaration $t_e \leadsto t_i$ means that an exported type t_e depends on an imported type t_i. When two units are linked with a **compound** expression, tracing the set of dependencies can ensure that linking does not create a cyclic type

$$\frac{\tau_{b1} \leq \tau_{b2} \quad \overline{s_{i1} = t_{i1}::\kappa_{i1}} \subseteq \overline{s_{i2} = t_{i2}::\kappa_{i2}} \quad \overline{s_{e2} = t_{e2}::\kappa_{e2}} \subseteq \overline{s_{e1} = t_{e2}::\kappa_{e1}} \\ \overline{t_{de1} \rightsquigarrow t_{di1}} \subseteq \overline{t_{de2} \rightsquigarrow t_{di2}} \\ \overline{y_{i1}\mathbin{:}\tau_{i1}} \sqsubseteq \overline{y_{i2}\mathbin{:}\tau_{i2}} \quad \overline{y_{e2}\mathbin{:}\tau_{e2}} \sqsubseteq \overline{y_{e1}\mathbin{:}\tau_{e1}}}{\mathsf{sig}[\mathit{i1}, \mathit{e1}, \mathit{di1}, \mathit{de1}, \mathit{b1}] \leq \mathsf{sig}[\mathit{i2}, \mathit{e2}, \mathit{di2}, \mathit{de2}, \mathit{b2}]}$$

Figure 3.30 : Subtyping in UNIT$_\mathsf{e}$ signatures

definition. Also, the signature for a **compound** expression propagates dependency information for types imported into and exported from the compound unit.

UNIT$_\mathsf{e}$ Type Checking

The following abbreviation expresses a UNIT$_\mathsf{e}$ signature:

$$\begin{aligned} \mathsf{sig}[i, e, di, de, b] \equiv \textbf{sig}\ & \textbf{import}\ \overline{s_i = t_i::\kappa_i}\ \overline{y_i\mathbin{:}\tau_i} \\ & \textbf{export}\ \overline{s_e = t_e::\kappa_e}\ \overline{y_e\mathbin{:}\tau_e} \\ & \textbf{depends}\ \overline{s_{de} \rightsquigarrow s_{di}} \\ & \triangleright\ \tau_b \end{aligned}$$

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 UNIT$_\mathsf{e}$ 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:

$$\begin{aligned} \tau \propto_D t \text{ iff } & t \in FTV(\tau) \\ & \text{or } (\exists \langle t' = \tau' \rangle \in D \text{ s.t. } t' \in FTV(\tau) \text{ and } \tau' \propto_D t) \end{aligned}$$

$FTV(\tau)$ denotes the set of type variables in τ 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.

$$
|\tau|_D = \begin{cases}
t & \text{if } \tau{=}t \text{ and } t{\notin}D \\
|\tau'|_D & \text{if } \tau{=}t \text{ and } \langle t = \tau' \rangle \in D \\
|\tau'|_D{\rightarrow}|\tau''|_D & \text{if } \tau{=}\tau'{\rightarrow}\tau'' \\
\begin{array}{l} \textbf{sig} \quad \textbf{import } \overrightarrow{s_i = t_i{::}\kappa_i} \ \overrightarrow{y_i{:}|\tau_i|_{D'}} \ \textbf{export } \overrightarrow{s_e = t_e{::}\kappa_e} \ \overrightarrow{y_e{:}|\tau_e|_{D'}} \\ \qquad\quad \textbf{depends } \overrightarrow{s_{de} \rightsquigarrow s_{di}} \\ \qquad\quad \triangleright\ |\tau_b|_{D'} \end{array} & \begin{array}{l} \text{if } \tau{=}\mathsf{sig}[i, e, di, de, b] \\ \quad \text{where } D' = \{\langle t = \tau \rangle | \langle t = \tau \rangle \in D \\ \qquad\qquad\qquad \text{and } t \notin \overrightarrow{t_i} \cup \overrightarrow{t_e}\} \end{array}
\end{cases}
$$

$$
|e|_D = \begin{cases}
x & \text{if } e{=}x \\
\begin{array}{l} \textbf{unit import } \overrightarrow{s_i = t_i{::}\kappa_i} \ \overrightarrow{y_i = x_i{:}|\tau_i|_{D'}} \\ \qquad\ \ \textbf{export } \overrightarrow{s_e = t_e{::}\kappa_e} \ \overrightarrow{y_e = x_e{:}|\tau_e|_{D'}} \\ \qquad\ \ \triangleright\ |\tau_b|_{D'} \\ \qquad\ \ \overrightarrow{\textbf{type } t_a{::}\kappa_a = |\tau_a|_{D'}} \\ \qquad\ \ \overrightarrow{\textbf{type } t = x_{\mathsf{cl}},x_{\mathsf{dl}}\ |\tau_{\mathsf{l}}|_{D'} \mid x_{\mathsf{cr}},x_{\mathsf{dr}}\ |\tau_{\mathsf{r}}|_{D'} \diamond x_{\mathsf{t}}} \\ \qquad\ \ \overrightarrow{\textbf{val } x{:}|\tau|_{D'} = |v|_{D'}} \ \textbf{in } |e_b|_{D'} \end{array} & \begin{array}{l} \text{if } e{=}\textbf{unit import } \overrightarrow{s_i = t_i{::}\kappa_i} \ \overrightarrow{y_i = x_i{:}\tau_i} \\ \qquad\qquad\quad \textbf{export } \overrightarrow{s_e = t_e{::}\kappa_e} \ \overrightarrow{y_e = x_e{:}\tau_e} \\ \qquad\qquad\quad \triangleright\ \tau_b \\ \qquad\qquad\quad \overrightarrow{\textbf{type } t_a{::}\kappa_a = \tau_a} \\ \qquad\qquad\quad \overrightarrow{\textbf{type } t = x_{\mathsf{cl}},x_{\mathsf{dl}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{cr}},x_{\mathsf{dr}}\ \tau_{\mathsf{r}} \diamond x_{\mathsf{t}}} \\ \qquad\qquad\quad \overrightarrow{\textbf{val } x{:}\tau = v} \ \textbf{in } e_b \\ \quad \text{where } D' = \{\langle t = \tau \rangle | \langle t = \tau \rangle \in D \\ \qquad\qquad\qquad \text{and } t \notin \overrightarrow{t_i} \cup \overrightarrow{t_e} \cup \overrightarrow{t_a} \cup \overrightarrow{t}\} \end{array} \\
\ldots &
\end{cases}
$$

Figure 3.31 : Expanding a set of type abbreviations in a type or expression

Unit$_{\mathsf{e}}$ Evaluation

Given a type equation of the form **type** $t = \tau$, 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 Unit$_{\mathsf{e}}$ are nearly the same as the rules for Unit$_{\mathsf{c}}$ (see Figures 3.25 and 3.26). As in Unit$_{\mathsf{c}}$, Unit$_{\mathsf{e}}$'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 Unit$_{\mathsf{e}}$ would also follow closely the proof for Unit$_{\mathsf{c}}$, based on a new lemma that validates the $\propto_D$ replacements.

$$\mathsf{sig}^{\vdash}_{\mathsf{e}}: \frac{\begin{array}{c}\overrightarrow{s_{de}} \subseteq \overrightarrow{s_e} \quad \overrightarrow{s_{di}} \subseteq \overrightarrow{s_i} \quad \Gamma' = \Gamma, \overrightarrow{t_i{::}\kappa_i}, \overrightarrow{t_e{::}\kappa_e} \quad FTV(\tau_b) \cap \overrightarrow{t_e} = \emptyset \\ \Gamma' \vdash \tau_i :: \kappa_i \quad \Gamma' \vdash \tau_e :: \kappa_e \quad \Gamma' \vdash \tau_b :: \Omega\end{array}}{\Gamma \vdash \mathsf{sig}[i, e, di, de, b] :: \Omega}$$

$$\mathsf{unit}^{\vdash}_{\mathsf{e}}: \frac{\begin{array}{c}\overrightarrow{t_i}, \overrightarrow{t_a}, \overrightarrow{t}, \overrightarrow{x_i}, \overrightarrow{x_{\mathsf{cl}}}, \overrightarrow{x_{\mathsf{dl}}}, \overrightarrow{x_{\mathsf{cr}}}, \overrightarrow{x_{\mathsf{dr}}}, \overrightarrow{x_{\mathsf{t}}}, \overrightarrow{x} \text{ distinct} \quad \overrightarrow{s_i}, \overrightarrow{s_e}, \overrightarrow{y_i}, \overrightarrow{y_e} \text{ distinct} \quad (\overrightarrow{t_i} \cup \overrightarrow{t_a} \cup \overrightarrow{t}) \cap \mathrm{dom}(\Gamma) = \emptyset \\ \overrightarrow{t_e{::}\kappa_e} \subseteq \overrightarrow{t_a{::}\kappa_a} \cup \overrightarrow{t{::}\Omega} \quad \overrightarrow{x_e{:}\tau_e} \sqsubseteq \overrightarrow{x{:}\tau} \cup \overrightarrow{x_{\mathsf{cl}}{:}\tau_{\mathsf{l}} \to t} \cup \overrightarrow{x_{\mathsf{dl}}{:}t \to \tau_{\mathsf{l}}} \cup \overrightarrow{x_{\mathsf{cr}}{:}\tau_{\mathsf{r}} \to t} \cup \overrightarrow{x_{\mathsf{dr}}{:}t \to \tau_{\mathsf{r}}} \cup \overrightarrow{x_{\mathsf{t}}{:}t \to \mathsf{bool}} \\ D = \overrightarrow{\langle t_a = \tau_a \rangle} \quad \tau_a \propto_D t'_a \Rightarrow \tau'_a \not\propto_D t_a \text{ for } \langle t_a = \tau_a \rangle, \langle t'_a = \tau'_a \rangle \in D \\ \overrightarrow{t_{de} \rightsquigarrow t_{di}} = \{ t_a \rightsquigarrow t_i \mid \langle t_a = \tau_a \rangle \in D \text{ and } t_i \in \overrightarrow{t_i} \text{ and } t_a \in \overrightarrow{t_e} \text{ and } \tau_a \propto_D t_i \} \\ \Gamma \vdash \mathsf{sig}[i, e, di, de, b] :: \Omega \quad \Gamma' = \Gamma, \overrightarrow{t_i{::}\kappa_i}, \overrightarrow{t{::}\Omega} \quad \Gamma'_a = \Gamma', \overrightarrow{t_a{::}\kappa_a} \quad \Gamma'_a \vdash \tau_a :: \kappa_a \\ \Gamma' \vdash |\overrightarrow{\tau_{\mathsf{l}}}| :: \Omega \quad \Gamma' \vdash |\overrightarrow{\tau_{\mathsf{r}}}| :: \Omega \quad \Gamma' \vdash |\overrightarrow{\tau}| :: \Omega \\ \Gamma'' = \Gamma', \overrightarrow{x_i{:}|\tau_i|}, \overrightarrow{x{:}|\tau|}, \overrightarrow{x_{\mathsf{cl}}{:}|\tau_{\mathsf{l}}| \to |t|}, \overrightarrow{x_{\mathsf{dl}}{:}|t| \to |\tau_{\mathsf{l}}|}, \overrightarrow{x_{\mathsf{cr}}{:}|\tau_{\mathsf{r}}| \to |t|}, \overrightarrow{x_{\mathsf{dr}}{:}|t| \to |\tau_{\mathsf{r}}|}, \overrightarrow{x_{\mathsf{t}}{:}|t| \to \mathsf{bool}} \\ \Gamma'' \vdash_{\mathsf{s}} |\overrightarrow{v}| : |\overrightarrow{\tau}| \quad \Gamma'' \vdash |e_b| : \tau_b \end{array}}{\begin{array}{l}\Gamma \vdash \textbf{unit import } \overrightarrow{s_i = t_i{::}\kappa_i} \ \overrightarrow{y_i = x_i{:}\tau_i} \textbf{ export } \overrightarrow{s_e = t_e{::}\kappa_e} \ \overrightarrow{y_e = x_e{:}\tau_e} \rhd \tau_b \\ \qquad \overrightarrow{\textbf{type } t_a{::}\kappa_a = \tau_a} \\ \qquad \overrightarrow{\textbf{type } t = x_{\mathsf{cl}}, x_{\mathsf{dl}} \ \tau_{\mathsf{l}} \mid x_{\mathsf{cr}}, x_{\mathsf{dr}} \ \tau_{\mathsf{r}} \diamond x_{\mathsf{t}}} \\ \qquad \overrightarrow{\textbf{val } x{:}\tau = v} \textbf{ in } e_b \\ \ : \mathsf{sig}[i, e, di, de, b]\end{array}}$$

$$\mathsf{compound}^{\vdash}_{\mathsf{e}}: \frac{\begin{array}{c}\overrightarrow{s_i}, \overrightarrow{s_{p1}}, \overrightarrow{s_{p2}}, \overrightarrow{y_i}, \overrightarrow{y_{p1}}, \overrightarrow{y_{p2}} \text{ distinct} \\ \overrightarrow{s_{w1} = t_{w1}{::}\kappa_{w1}} \subseteq \overrightarrow{s_i = t_i{::}\kappa_i} \cup \overrightarrow{s_{p2} = t_{p2}{::}\kappa_{p2}} \quad \overrightarrow{y_{w1}{:}\tau_{w1}} \sqsubseteq \overrightarrow{y_i{:}\tau_i} \cup \overrightarrow{y_{p2}{:}\tau_{p2}} \\ \overrightarrow{s_{w2} = t_{w2}{::}\kappa_{w2}} \subseteq \overrightarrow{s_i = t_i{::}\kappa_i} \cup \overrightarrow{s_{p1} = t_{p1}{::}\kappa_{p1}} \quad \overrightarrow{y_{w2}{:}\tau_{w2}} \sqsubseteq \overrightarrow{y_i{:}\tau_i} \cup \overrightarrow{y_{p1}{:}\tau_{p1}} \\ \overrightarrow{s_e = t_e{::}\kappa_e} \subseteq \overrightarrow{s_{p1} = t_{p1}{::}\kappa_{p1}} \cup \overrightarrow{s_{p2} = t_{p2}{::}\kappa_{p2}} \quad \overrightarrow{y_e{:}\tau_e} \sqsubseteq \overrightarrow{y_{p1}{:}\tau_{p1}} \cup \overrightarrow{y_{p2}{:}\tau_{p2}} \\ \Gamma \vdash e_1 : \mathsf{sig}[i1, e1, di1, de1, b1] \quad \Gamma \vdash e_2 : \mathsf{sig}[i2, e2, di2, de2, b2] \\ \Gamma \vdash \mathsf{sig}[w1, p1, di1, de1, b1] :: \Omega \quad \Gamma \vdash \mathsf{sig}[w2, p2, di2, de2, b] :: \Omega \\ \mathsf{sig}[i1, e1, di1, de1, b1] \leq \mathsf{sig}[w1, p1, di1, de1, b1] \quad \mathsf{sig}[i2, e2, di2, de2, b2] \leq \mathsf{sig}[w2, p2, di2, de2, b] \\ \Gamma \vdash \mathsf{sig}[i, e, di, de, b] :: \Omega \quad \overrightarrow{\langle s_{di1}, s_{de1} \rangle} \cap \overrightarrow{\langle s_{de2}, s_{di2} \rangle} = \emptyset \\ \overrightarrow{s_{de} \rightsquigarrow s_{di}} = \{ s_e \rightsquigarrow s_i \mid s_i \in \overrightarrow{s_i} \text{ and } s_e \in \overrightarrow{s_e} \text{ and } s_e \rightsquigarrow s_i \in \overrightarrow{s_{de1} \rightsquigarrow s_{di1}} \cup \overrightarrow{s_{de2} \rightsquigarrow s_{di2}} \}\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{compound import } \overrightarrow{s_i = t_i{::}\kappa_i} \ \overrightarrow{y_i{:}\tau_i} \textbf{ export } \overrightarrow{s_e = t_e{::}\kappa_e} \ \overrightarrow{y_e{:}\tau_e} \rhd \tau_b \\ \qquad \textbf{link } e_1 \textbf{ with } \overrightarrow{s_{w1} = t_{w1}{::}\kappa_{w1}} \ \overrightarrow{y_{w1}{:}\tau_{w1}} \textbf{ provides } \overrightarrow{s_{p1} = t_{p1}{::}\kappa_{p1}} \ \overrightarrow{y_{p1}{:}\tau_{p1}} \\ \qquad \textbf{and } e_2 \textbf{ with } \overrightarrow{s_{w2} = t_{w2}{::}\kappa_{w2}} \ \overrightarrow{y_{w2}{:}\tau_{w2}} \textbf{ provides } \overrightarrow{s_{p2} = t_{p2}{::}\kappa_{p2}} \ \overrightarrow{y_{p2}{:}\tau_{p2}} \\ \ : \mathsf{sig}[i, e, di, de, b]\end{array}}$$

Figure 3.32 : Type checking for $\textsc{Unit}_{\mathsf{e}}$

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 UNIT_{c}, where modules are implicitly linked by matching names in a global namespace (like conventional `.o` linking). The typing issues in MTAL and 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. Nevertheless, 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.[1] As a result, implementing a mixin protocol with multiple inheritance is error-prone and typically avoided.

---

[1] Dan Friedman determined in an informal poll in 1996 that almost nobody who teaches C++ teaches multiple inheritance [pers. com.].

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

ClassicJava 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^i ...
interface Barrier^i ...
interface Door^i
   extends Place^i, Barrier^i ...
...
class Door^c extends Object
      implements Door^i {
  ...
  Room^c Enter(Person^c p) { ... }
  ...
}
class LockedDoor^c extends Door^c ...
class ShortDoor^c extends Door^c ...
```

$\Rightarrow$

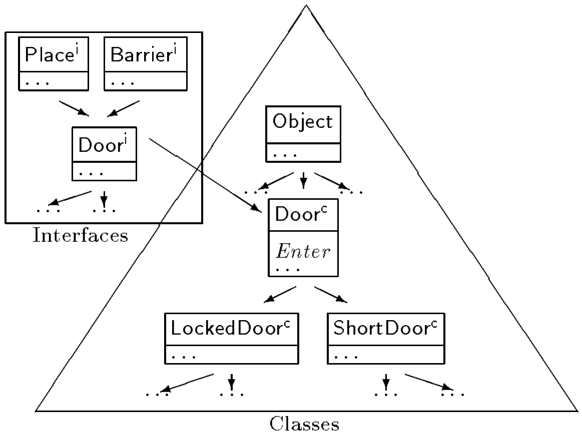


Figure 4.1 : A program determines a static directed acyclic graph of types

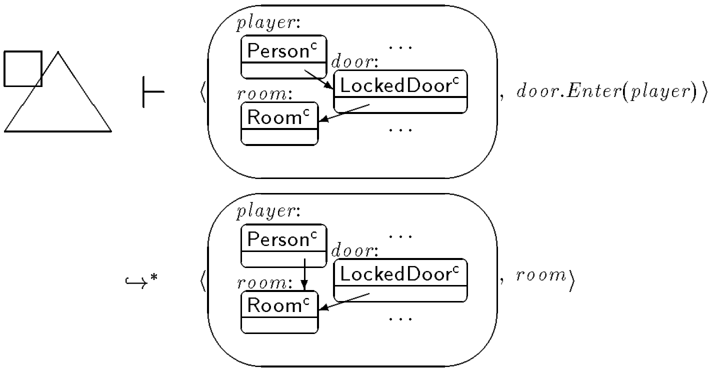


Figure 4.2 : Given a type graph, reductions map a store-expression pair to a new pair

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-

$$
\begin{array}{rcl}
P & = & \mathit{defn}^*\ e \\
\mathit{defn} & = & \textbf{class}\ c\ \textbf{extends}\ c\ \textbf{implements}\ i^*\ \{\ \mathit{field}^*\ \mathit{meth}^*\ \} \\
 & & \mid \textbf{interface}\ i\ \textbf{extends}\ i^*\ \{\ \mathit{meth}^*\ \} \\
\mathit{field} & = & t\ \mathit{fd} \\
\mathit{meth} & = & t\ \mathit{md}\ (\ \mathit{arg}^*\ )\ \{\ \mathit{body}\ \} \\
\mathit{arg} & = & t\ \mathit{var} \\
\mathit{body} & = & e \mid \textbf{abstract} \\
e & = & \textbf{new}\ c \mid \mathit{var} \mid \mathsf{null} \mid e\underline{\ :\ c\ }.\mathit{fd} \mid e\underline{\ :\ c\ }.\mathit{fd} = e \\
 & & \mid e.\mathit{md}\ (e^*\ ) \mid \textbf{super}\underline{\ \equiv \textbf{this}\ :\ c\ }.\mathit{md}\ (e^*\ ) \\
 & & \mid \textbf{view}\ t\ e \mid \textbf{let}\ \mathit{var} = e\ \textbf{in}\ e \\
\mathit{var} & = & \text{a variable name or } \textbf{this} \\
c & = & \text{a class name or } \mathsf{Object} \\
i & = & \text{interface name or } \mathsf{Empty} \\
\mathit{fd} & = & \text{a field name} \\
\mathit{md} & = & \text{a method name} \\
t & = & c \mid i
\end{array}
$$

Figure 4.3 : ClassicJava syntax; underlined phrases are inserted by elaboration

sicJava; instance variables are always initialized to null. Finally, the **view** and **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 **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 ClassesOnce(P) predicate states that each class name is defined at most once in the program P. The relation $\prec^c_P$ associates each class name in P to the class it extends, and the (overloaded) $\Subset^c_P$ 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 $\leq^c_P$, which is a partial order if the CompleteClasses(P) predicate holds and the 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 $(t \ldots \longrightarrow t)$, V for variable lists $(var\ldots)$, and Γ 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).

$\textsc{ClassesOnce}(P)$	Each class name is declared only once $\textbf{class}\ c \cdots \textbf{class}\ c' \cdots$ is in $P \implies c \neq c'$
$\textsc{FieldOncePerClass}(P)$	Field names in each class declaration are unique $\textbf{class} \cdots \{ \cdots fd \cdots fd' \cdots \}$ is in $P \implies fd \neq fd'$
$\textsc{MethodOncePerClass}(P)$	Method names in each class declaration are unique $\textbf{class} \cdots \{ \cdots md\ (\cdots)\ \{ \cdots \} \cdots md'\ (\cdots)\ \{ \cdots \} \cdots \}$ is in $P \implies md \neq md'$
$\textsc{InterfacesOnce}(P)$	Each interface name is declared only once $\textbf{interface}\ i \cdots \textbf{interface}\ i' \cdots$ is in $P \implies i \neq i'$
$\textsc{InterfacesAbstract}(P)$	Method declarations in an interface are **abstract** $\textbf{interface} \cdots \{ \cdots md\ (\cdots)\ \{e\} \cdots \}$ is in $P \implies e$ is **abstract**
$\textsc{MethodArgsDistinct}(P)$	Each method argment name is unique $md\ (t_1\ var_1\ \ldots\ t_n\ var_n)\ \{ \cdots \}$ is in $P \implies var_1, \ldots var_n$, and **this** are distinct
$\prec^c_P$	Class is declared as an immediate subclass $c \prec^c_P c' \Leftrightarrow \textbf{class}\ c\ \textbf{extends}\ c' \cdots \{ \cdots \}$ is in P
$\in\!\!\in^c_P$	Field is declared in a class $\langle c.fd,\ t\rangle \in\!\!\in^c_P c \Leftrightarrow \textbf{class}\ c \cdots \{ \cdots t\ fd \cdots \}$ is in P
$\in\!\!\in^c_P$	Method is declared in class $\langle md,\ (t_1 \ldots t_n \longrightarrow t),\ (var_1\ \ldots\ var_n),\ e\rangle \in\!\!\in^c_P c$ $\Leftrightarrow \textbf{class}\ c \cdots \{ \cdots t\ md\ (t_1\ var_1\ \ldots\ t_n\ var_n)\ \{e\} \cdots \}$ is in P
$\prec^i_P$	Interface is declared as an immediate subinterface $i \prec^i_P i' \Leftrightarrow \textbf{interface}\ i\ \textbf{extends} \cdots i' \cdots \{ \cdots \}$ is in P
$\in\!\!\in^i_P$	Method is declared in an interface $\langle md,\ (t_1 \ldots t_n \longrightarrow t),\ (var_1\ \ldots\ var_n),\ e\rangle \in\!\!\in^i_P i$ $\Leftrightarrow \textbf{interface}\ i \cdots \{ \cdots t\ md\ (t_1\ var_1\ \ldots\ t_n\ var_n)\ \{e\} \cdots \}$ is in P
$\prec\!\!\!\prec^c_P$	Class declares implementation of an interface $c \prec\!\!\!\prec^c_P i \Leftrightarrow \textbf{class}\ c \cdots \textbf{implements} \cdots i \cdots \{ \cdots \}$ is in P
$\leq^c_P$	Class is a subclass $\leq^c_P \equiv$ the transitive, reflexive closure of $\prec^c_P$
$\textsc{CompleteClasses}(P)$	Classes that are extended are defined $\mathrm{rng}(\prec^c_P) \subseteq \mathrm{dom}(\prec^c_P) \cup \{\mathsf{Object}\}$
$\textsc{WellFoundedClasses}(P)$	Class hierarchy is an order $\leq^c_P$ is antisymmetric
$\textsc{ClassMethodsOK}(P)$	Method overriding preserves the type $(\langle md,\ T,\ V,\ e\rangle \in\!\!\in^c_P c$ and $\langle md,\ T',\ V',\ e'\rangle \in\!\!\in^c_P c') \implies (T = T'$ or $c \not\leq^c_P c')$
$\in^c_P$	Field is contained in a class $\langle c'.fd,\ t\rangle \in^c_P c$ $\Leftrightarrow \langle c'.fd,\ t\rangle \in\!\!\in^c_P c'$ and $c' = \min\{c'' \mid c \leq^c_P c''$ and $\exists t'$ s.t. $\langle c''.fd,\ t'\rangle \in\!\!\in^c_P c''\}$
$\in^c_P$	Method is contained in a class $\langle md,\ T,\ V,\ e\rangle \in^c_P c$ $\Leftrightarrow (\langle md,\ T,\ V,\ e\rangle \in\!\!\in^c_P c'$ and $c' = \min\{c'' \mid c \leq^c_P c''$ and $\exists e', V'$ s.t. $\langle md,\ T,\ V',\ e'\rangle \in\!\!\in^c_P c''\})$

Figure 4.4 : Predicates and relations in the model of ClassicJava (Part I)

$\leq^{\mathsf{i}}_P$	Interface is a subinterface $\leq^{\mathsf{i}}_P \equiv$ the transitive, reflexive closure of $\prec^{\mathsf{i}}_P$
$\textsc{CompleteInterfaces}(P)$	Extended/implemented interfaces are defined $\mathrm{rng}(\prec^{\mathsf{i}}_P) \cup \mathrm{rng}(\prec\!\!\!\prec^{\mathsf{c}}_P) \subseteq \mathrm{dom}(\prec^{\mathsf{i}}_P) \cup \{\mathsf{Empty}\}$
$\textsc{WellFoundedInterfaces}(P)$	Interface hierarchy is an order $\leq^{\mathsf{i}}_P$ is antisymmetric
$\ll^{\mathsf{c}}_P$	Class implements an interface $c \ll^{\mathsf{c}}_P i \Leftrightarrow \exists c', i'$ s.t. $c \leq^{\mathsf{c}}_P c'$ and $i' \leq^{\mathsf{i}}_P i$ and $c' \prec\!\!\!\prec^{\mathsf{c}}_P i'$
$\textsc{InterfaceMethodsOK}(P)$	Interface inheritance or redeclaration of methods is consistent $\langle md, T, V, \mathbf{abstract}\rangle \in\!\!\!\in^{\mathsf{i}}_P i$ and $\langle md, T', V', \mathbf{abstract}\rangle \in\!\!\!\in^{\mathsf{i}}_P i'$ $\Longrightarrow (T = T'$ or $\forall i''(i'' \not\leq^{\mathsf{i}}_P i$ or $i'' \not\leq^{\mathsf{i}}_P i'))$
$\in^{\mathsf{i}}_P$	Method is contained in an interface $\langle md, T, V, \mathbf{abstract}\rangle \in^{\mathsf{i}}_P i \Leftrightarrow \exists i'$ s.t. $i \leq^{\mathsf{i}}_P i'$ and $\langle md, T, V, \mathbf{abstract}\rangle \in\!\!\!\in^{\mathsf{i}}_P i'$
$\textsc{ClassesImplementAll}(P)$	Classes supply methods to implement interfaces $c \prec\!\!\!\prec^{\mathsf{c}}_P i \Longrightarrow (\forall md, T, V\ \langle md, T, V, \mathbf{abstract}\rangle \in^{\mathsf{i}}_P i \Longrightarrow \exists e, V'$ s.t. $\langle md, T, V', e\rangle \in^{\mathsf{c}}_P c)$
$\textsc{NoAbstractMethods}(P, c)$	Class has no **abstract** methods (can be instantiated) $\langle md, T, V, e\rangle \in^{\mathsf{c}}_P c \Longrightarrow e \neq \mathbf{abstract}$
$\leq_P$	Type is a subtype $\leq_P \equiv \leq^{\mathsf{c}}_P \cup \leq^{\mathsf{i}}_P \cup \ll^{\mathsf{c}}_P$
$\in_P$	Field or method is in a type $\in_P \equiv \in^{\mathsf{c}}_P \cup \in^{\mathsf{i}}_P$

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 $\in^{\mathsf{c}}_P$ relations. The $\in^{\mathsf{c}}_P$ 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 $\prec^{\mathsf{i}}_P$ induces a subinterface relation $\leq^{\mathsf{i}}_P$. 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 $\in^{\mathsf{i}}_P$, 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 $\prec\!\!\!\prec^{\mathsf{c}}_P$ relation. This relation is a set of edges joining the class tree and the interface graph, completing the sub*type* 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:

$\vdash_{\mathsf{p}} P \Rightarrow P' : t$	P elaborates to P' with type t
$P \vdash_{\mathsf{d}} \mathit{defn} \Rightarrow \mathit{defn}'$	defn elaborates to defn'
$P, t \vdash_{\mathsf{m}} \mathit{meth} \Rightarrow \mathit{meth}'$	meth in t elaborates to meth'
$P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t$	e elaborates to e' with type t in Γ
$P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t$	e has type t using subsumption in Γ
$P \vdash_{\mathsf{t}} t$	t exists

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 $\mathbf{get}^{\mathsf{c}}$ and $\mathbf{set}^{\mathsf{c}}$ rules for fields first determine the type of the instance expression, and then calculate a class-tagged field name using $\in_P$; this yields both the type of the field and the class for the installed annotation. In the $\mathbf{set}^{\mathsf{c}}$ 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 system on pairs of expressions and stores. A store $\mathcal{S}$ is a mapping from *object*s to

$\vdash_{\mathsf{p}}$

$$\frac{\begin{array}{llll} \textsc{ClassesOnce}(P) & \textsc{InterfacesOnce}(P) & \textsc{MethodOncePerClass}(P) & \textsc{FieldOncePerClass}(P) \\ \textsc{CompleteClasses}(P) & \textsc{WellFoundedClasses}(P) & \textsc{CompleteInterfaces}(P) & \textsc{WellFoundedInterfaces}(P) \\ \textsc{InterfaceMethodsOK}(P) & \textsc{InterfacesAbstract}(P) & \textsc{MethodArgsDistinct}(P) & \textsc{ClassesImplementAll}(P) \\ \multicolumn{4}{l}{P \vdash_{\mathsf{d}} \mathit{defn}_j \Rightarrow \mathit{defn}'_j \text{ for } j \in [1,n] \quad P,[\,] \vdash_{\mathsf{e}} e \Rightarrow e' : t \quad \text{where } P = \mathit{defn}_1 \ldots \mathit{defn}_n\ e} \end{array}}{\vdash_{\mathsf{p}} \mathit{defn}_1 \ldots \mathit{defn}_n\ e \Rightarrow \mathit{defn}'_1 \ldots \mathit{defn}'_n\ e' : t}[\mathbf{prog}^{\mathsf{c}}]$$

$\vdash_{\mathsf{d}}$

$$\frac{P \vdash_{\mathsf{t}} t_j \text{ for each } j \in [1,n] \quad P, c \vdash_{\mathsf{m}} \mathit{meth}_k \Rightarrow \mathit{meth}'_k \text{ for each } k \in [1,p]}{\begin{array}{ll} P \vdash_{\mathsf{d}} \mathbf{class}\ c \cdots \{\ t_1\ \mathit{fd}_1 \ldots t_n\ \mathit{fd}_n & \Rightarrow \mathbf{class}\ c \cdots \{\ t_1\ \mathit{fd}_1 \ldots t_n\ \mathit{fd}_n \\ \qquad \mathit{meth}_1 \ldots \mathit{meth}_p\ \} & \qquad \mathit{meth}'_1 \ldots \mathit{meth}'_p\ \} \end{array}}[\mathbf{defn}^{\mathsf{c}}]$$

$$\frac{P, i \vdash_{\mathsf{m}} \mathit{meth}_j \Rightarrow \mathit{meth}_j \text{ for each } j \in [1,p]}{P \vdash_{\mathsf{d}} \mathbf{interface}\ i \cdots \{\ \mathit{meth}_1 \ldots \mathit{meth}_p\ \} \Rightarrow \mathbf{interface}\ i \cdots \{\ \mathit{meth}_1 \ldots \mathit{meth}_p\ \}}[\mathbf{defn}^{\mathsf{i}}]$$

$\vdash_{\mathsf{m}}$

$$\frac{\begin{array}{ll} P \vdash_{\mathsf{t}} t & P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] \\ P,[\mathbf{this} : t_o,\ \mathit{var}_1 : t_1, \ldots \mathit{var}_n : t_n] \vdash_{\mathsf{s}} e \Rightarrow e' : t & \end{array}}{P, t_o \vdash_{\mathsf{m}} t\ \mathit{md}\ (t_1\ \mathit{var}_1 \ldots t_n\ \mathit{var}_n)\ \{\ e\ \} \Rightarrow t\ \mathit{md}\ (t_1\ \mathit{var}_1 \ldots t_n\ \mathit{var}_n)\ \{\ e'\ \}}[\mathbf{meth}]$$

$\vdash_{\mathsf{e}}$

$$\frac{P \vdash_{\mathsf{t}} c \quad \textsc{NoAbstractMethods}(P, c)}{P, \Gamma \vdash_{\mathsf{e}} \mathbf{new}\ c \Rightarrow \mathbf{new}\ c : c}[\mathbf{new}^{\mathsf{c}}] \qquad \frac{\mathit{var} \in \mathrm{dom}(\Gamma)}{P, \Gamma \vdash_{\mathsf{e}} \mathit{var} \Rightarrow \mathit{var} : \Gamma(\mathit{var})}[\mathbf{var}]$$

$$\frac{P \vdash_{\mathsf{t}} t}{P, \Gamma \vdash_{\mathsf{e}} \mathsf{null} \Rightarrow \mathsf{null} : t}[\mathbf{null}] \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \quad \langle c.\mathit{fd},\ t\rangle \in_P t'}{P, \Gamma \vdash_{\mathsf{e}} e.\mathit{fd} \Rightarrow e' \underline{\ : c\ }.\mathit{fd} : t}[\mathbf{get}^{\mathsf{c}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \quad \langle c.\mathit{fd},\ t\rangle \in_P t' \quad P, \Gamma \vdash_{\mathsf{s}} e_v \Rightarrow e'_v : t}{P, \Gamma \vdash_{\mathsf{e}} e.\mathit{fd} = e_v \Rightarrow e' \underline{\ : c\ }.\mathit{fd} = e'_v : t}[\mathbf{set}^{\mathsf{c}}]$$

Figure 4.6 : Context-sensitive checks and type rules for ClassicJava (Part I)

class-tagged field records. A field record $\mathcal{F}$ 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.

$$\frac{\begin{array}{l} P,\Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \ \langle md,\, (t_1 \ldots t_n \longrightarrow t),\, (var_1 \ \ldots \ var_n),\, e_b \rangle \in_P t' \\ P,\Gamma \vdash_{\mathsf{s}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n] \end{array}}{P,\Gamma \vdash_{\mathsf{e}} e.md\ (e_1 \ \ldots \ e_n) \Rightarrow e'.md\ (e'_1 \ \ldots \ e'_n) : t}[\mathbf{call}^{\mathsf{c}}]$$

$$\frac{\begin{array}{l} P,\Gamma \vdash_{\mathsf{e}} \mathbf{this} \Rightarrow \mathbf{this} : c' \quad c' \prec^{\mathsf{c}}_P c \quad \langle md,\, (t_1 \ldots t_n \longrightarrow t),\, (var_1 \ \ldots \ var_n),\, e_b \rangle \in_P c \\ P,\Gamma \vdash_{\mathsf{s}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n] \qquad e_b \neq \mathbf{abstract} \end{array}}{P,\Gamma \vdash_{\mathsf{e}} \mathbf{super}.md(e_1 \ \ldots \ e_n) \Rightarrow \mathbf{super}\underline{\ \equiv \mathbf{this} : c\ }.md(e'_1 \ \ldots \ e'_n) : t}[\mathbf{super}^{\mathsf{c}}]$$

$$\frac{P,\Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t}{P,\Gamma \vdash_{\mathsf{e}} \mathbf{view}\ t\ e \Rightarrow e' : t}[\mathbf{wcast}^{\mathsf{c}}] \qquad \frac{P \vdash_{\mathsf{t}} t}{P,\Gamma \vdash_{\mathsf{e}} \mathbf{abstract} \Rightarrow \mathbf{abstract} : t}[\mathbf{abs}]$$

$$\frac{P,\Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t'\ t \leq_P t' \text{ or } t \in \mathrm{dom}(\prec^{\mathsf{i}}_P) \text{ or } t' \in \mathrm{dom}(\prec^{\mathsf{i}}_P)}{P,\Gamma \vdash_{\mathsf{e}} \mathbf{view}\ t\ e \Rightarrow \mathbf{view}\ t\ e' : t}[\mathbf{ncast}^{\mathsf{c}}]$$

$$\frac{P,\Gamma \vdash_{\mathsf{e}} e_1 \Rightarrow e'_1 : t_1 \qquad P,\Gamma[var : t_1] \vdash_{\mathsf{e}} e_2 \Rightarrow e'_2 : t}{P,\Gamma \vdash_{\mathsf{e}} \mathbf{let}\ var = e_1\ \mathbf{in}\ e_2 \Rightarrow \mathbf{let}\ var = e'_1\ \mathbf{in}\ e'_2 : t}[\mathbf{let}]$$

$\vdash_{\mathsf{s}}, \vdash_{\mathsf{t}}$

$$\frac{P,\Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad t' \leq_P t}{P,\Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t}[\mathbf{sub}^{\mathsf{c}}] \qquad \frac{t \in \mathrm{dom}(\prec^{\mathsf{c}}_P) \cup \mathrm{dom}(\prec^{\mathsf{i}}_P) \cup \{\mathsf{Object},\ \mathsf{Empty}\}}{P \vdash_{\mathsf{t}} t}[\mathbf{type}^{\mathsf{c}}]$$

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 means getting stuck. This property can be formulated as a type soundness theorem.

Theorem 4.1.1 (Type Soundness) *If* $\vdash_{\mathsf{p}} P \Rightarrow P' : t$ *and* $P' = \mathit{defn}_1 \ \ldots \ \mathit{defn}_n \ e$, *then either*

- $P' \vdash \langle e,\, \emptyset \rangle \hookrightarrow^* \langle \mathit{object},\, \mathcal{S} \rangle$ *and* $\mathcal{S}(\mathit{object}) = \langle t',\, \mathcal{F} \rangle$ *and* $t' \leq_P t$; *or*
- $P' \vdash \langle e,\, \emptyset \rangle \hookrightarrow^* \langle \mathsf{null},\, \mathcal{S} \rangle$; *or*
- $P' \vdash \langle e,\, \emptyset \rangle \hookrightarrow^* \langle \mathsf{error\text{: } bad\ cast},\, \mathcal{S} \rangle$; *or*
- $P' \vdash \langle e,\, \emptyset \rangle \hookrightarrow^* \langle \mathsf{error\text{: } dereferenced\ null},\, \mathcal{S} \rangle$.

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

$$
\begin{array}{rcl}
e & = & \ldots \mid \mathit{object} \\
v & = & \mathit{object} \mid \mathsf{null}
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{E} & = & [\,] \mid \mathsf{E}\underline{\ : c\ }.\mathit{fd} \mid \mathsf{E}\underline{\ : c\ }.\mathit{fd} = e \mid v\underline{\ : c\ }.\mathit{fd} = \mathsf{E} \\
 & \mid & \mathsf{E}.\mathit{md}(e \ldots) \mid v.\mathit{md}(v \ldots \mathsf{E}\ e \ldots) \\
 & \mid & \mathbf{super}\underline{\ \equiv v : c\ }.\mathit{md}(v \ldots \mathsf{E}\ e \ldots) \\
 & \mid & \mathbf{view}\ t\ \mathsf{E} \mid \mathbf{let}\ \mathit{var} = \mathsf{E}\ \mathbf{in}\ e
\end{array}
$$

$$
\begin{array}{ll}
P \vdash \langle \mathsf{E}[\mathbf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathit{object}], \mathcal{S}[\mathit{object} \mapsto \langle c, \mathcal{F}\rangle]\rangle & [\mathit{new}] \\
\quad \text{where } \mathit{object} \notin \mathrm{dom}(\mathcal{S}) \text{ and } \mathcal{F} = \{c'.\mathit{fd} \mapsto \mathsf{null} \mid c \leq^{\mathsf{c}}_P c' \text{ and } \exists t \text{ s.t. } \langle c'.\mathit{fd}, t\rangle \in^{\mathsf{c}}_P c'\} & \\
P \vdash \langle \mathsf{E}[\mathit{object}\underline{\ : c'\ }.\mathit{fd}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}\rangle & [\mathit{get}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle \text{ and } \mathcal{F}(c'.\mathit{fd}) = v & \\
P \vdash \langle \mathsf{E}[\mathit{object}\underline{\ : c'\ }.\mathit{fd} = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}[\mathit{object} \mapsto \langle c, \mathcal{F}[c'.\mathit{fd} \mapsto v]\rangle]\rangle & [\mathit{set}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle & \\
P \vdash \langle \mathsf{E}[\mathit{object}.\mathit{md}(v_1, \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[\mathit{object}/\mathbf{this}, v_1/\mathit{var}_1, \ldots\ v_n/\mathit{var}_n]], \mathcal{S}\rangle & [\mathit{call}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle \text{ and } \langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t), (\mathit{var}_1\ \ldots\ \mathit{var}_n), e\rangle \in^{\mathsf{c}}_P c & \\
P \vdash \langle \mathsf{E}[\mathbf{super}\underline{\ \equiv \mathit{object} : c'\ }.\mathit{md}(v_1, \ldots\ v_n)], \mathcal{S}\rangle & [\mathit{super}] \\
\qquad \hookrightarrow \langle \mathsf{E}[e[\overline{\mathit{object}/\mathbf{this},}\ v_1/\mathit{var}_1, \ldots\ v_n/\mathit{var}_n]], \mathcal{S}\rangle & \\
\quad \text{where } \langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t), (\mathit{var}_1\ \ldots\ \mathit{var}_n), e\rangle \in^{\mathsf{c}}_P c' & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\ t'\ \mathit{object}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathit{object}], \mathcal{S}\rangle & [\mathit{cast}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle \text{ and } c \leq_P t' & \\
P \vdash \langle \mathsf{E}[\mathbf{let}\ \mathit{var} = v\ \mathbf{in}\ e], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[v/\mathit{var}]], \mathcal{S}\rangle & [\mathit{let}] \\
 & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\ t'\ \mathit{object}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error\!: bad\ cast}, \mathcal{S}\rangle & [\mathit{xcast}] \\
\quad \text{where } \mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle \text{ and } c \not\leq_P t' & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\ t'\ \mathsf{null}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error\!: bad\ cast}, \mathcal{S}\rangle & [\mathit{ncast}] \\
P \vdash \langle \mathsf{E}[\mathsf{null}\underline{\ : c\ }.\mathit{fd}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error\!: dereferenced\ null}, \mathcal{S}\rangle & [\mathit{nget}] \\
P \vdash \langle \mathsf{E}[\mathsf{null}\underline{\ : c\ }.\mathit{fd} = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error\!: dereferenced\ null}, \mathcal{S}\rangle & [\mathit{nset}] \\
P \vdash \langle \mathsf{E}[\mathsf{null}.\mathit{md}(v_1, \ldots\ v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error\!: dereferenced\ null}, \mathcal{S}\rangle & [\mathit{ncall}]
\end{array}
$$

Figure 4.8 : Operational semantics for ClassicJava

program) [85]. Specifically, for a configuration on the left-hand side of an evaluation step, there exists a type environment that establishes the expression's type as some t. This environment must be consistent with the store.

Definition 4.1.2 (Environment-Store Consistency)

$$
\begin{array}{ll}
 & P,\Gamma \vdash_\sigma \mathcal{S} \\
 & \quad \Leftrightarrow (\mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle \\
\Sigma_1\text{:} & \qquad \Rightarrow \Gamma(\mathit{object}) = c \\
\Sigma_2\text{:} & \qquad\quad \mathit{and}\ \mathrm{dom}(\mathcal{F}) = \{c_1.\mathit{fd} \mid \langle c_1.\mathit{fd}, c_2\rangle \in^{\mathsf{c}}_P c\} \\
\Sigma_3\text{:} & \qquad\quad \mathit{and}\ \mathrm{rng}(\mathcal{F}) \subseteq \mathrm{dom}(\mathcal{S}) \cup \{\mathsf{null}\} \\
\Sigma_4\text{:} & \qquad\quad \mathit{and}\ (\mathcal{F}(c_1.\mathit{fd}) = \mathit{object}'\ \mathit{and}\ \langle c_1.\mathit{fd}, c_2\rangle \in^{\mathsf{c}}_P c) \\
 & \qquad\qquad \Rightarrow ((\mathcal{S}(\mathit{object}') = \langle c', \mathcal{F}'\rangle) \Rightarrow c' \leq_P c_2)) \\
\Sigma_5\text{:} & \qquad \mathit{and}\ \mathit{object} \in \mathrm{dom}(\Gamma) \Rightarrow \mathit{object} \in \mathrm{dom}(\mathcal{S})^2 \\
\Sigma_6\text{:} & \qquad \mathit{and}\ \mathrm{dom}(\mathcal{S}) \subseteq \mathrm{dom}(\Gamma)
\end{array}
$$

Since the rewriting rules reduce *annotated* terms, we derive new type judgements $\vdash_{\underline{\mathsf{s}}}$ and $\vdash_{\underline{\mathsf{e}}}$ 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, $\vdash_{\underline{\mathsf{s}}}$ is derived from $\vdash_{\mathsf{s}}$, and so forth.

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

$$\frac{P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t'}{P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathbf{view}\ t\ e : t}[\mathbf{cast}^{\mathsf{c}}]$$

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

$$\mathbf{let}\ x = o.GetDoor()\ \mathbf{in}\ (\mathbf{view}\ \mathsf{LockedDoor}^{\mathsf{c}}\ x)$$

might reduce to

$$\mathbf{view}\ \mathsf{LockedDoor}^{\mathsf{c}}\ shortDoorObject$$

where *shortDoorObject* is an instance of $\mathsf{ShortDoor}^{\mathsf{c}}$. Unlike the [$\mathbf{wcast}^{\mathsf{c}}$] and [$\mathbf{ncast}^{\mathsf{c}}$] rules in $\vdash_{\mathsf{e}}$, the [$\mathbf{cast}^{\mathsf{c}}$] rule in $\vdash_{\underline{\mathsf{e}}}$ assigns a type to the reduced expression.

The $\vdash_{\underline{\mathsf{e}}}$ relation also generalizes the [$\mathbf{super}^{\mathsf{c}}$] 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 Σ_5, it would be wrong to write $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\mathcal{S})$ because Γ may contain bindings for lexical variables.

Definition 4.1.3 (Well-Formed Super Calls)

$$\textsc{SuperOk}(e) \Leftrightarrow \textit{For all } \mathbf{super}\underline{\ \equiv e_0 : c\ }.md(e_1, \ldots\ e_n) \textit{ in } e,$$
$$\textit{either } e_0 = \mathbf{this} \textit{ or } e_0 = \textit{object for some object}.$$

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

Lemma 4.1.4 (Conserve) *If* $\vdash_{\mathsf{p}} P \Rightarrow P' : t$ *and* $P' = \mathit{defn}_1 \ldots \mathit{defn}_n\ e$*, then* $P',\emptyset \vdash_{\underline{\mathsf{e}}} e' : t$*.*

Proof. The claim follows from a copy-and-patch argument. □

Lemma 4.1.5 (Subject Reduction) *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t$*,* $P,\Gamma \vdash_{\sigma} \mathcal{S}$*,* $\textsc{SuperOk}(e)$*, and* $P \vdash \langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$*, then* e' *is an* error *configuration or there exists a* Γ' *such that*

1. $P,\Gamma' \vdash_{\underline{\mathsf{s}}} e' : t$*,*
2. $P,\Gamma' \vdash_{\sigma} \mathcal{S}'$*, and*
3. $\textsc{SuperOk}(e')$*.*

Proof. The proof examines reduction steps. For each case, if execution has not halted with an error configuration, we construct the new environment Γ' 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. □

Lemma 4.1.6 (Progress) *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t$*,* $P,\Gamma \vdash_{\sigma} \mathcal{S}$*, and* $\textsc{SuperOk}(e)$*, then either* e *is a value or there exists an* $\langle e',\mathcal{S}'\rangle$ *such that* $P \vdash \langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$*.*

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 C.2 for the complete proof. □

By combining the Subject Reduction and Progress lemmas, we can prove that every non-value ClassicJava program reduces while preserving its type, thus establishing the soundness of 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, $\mathsf{LockedDoor}^c$ and $\mathsf{ShortDoor}^c$. An instance of $\mathsf{LockedDoor}^c$ requires a key to open the door, while an instance of $\mathsf{ShortDoor}^c$ 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^c and ShortDoor^c classes cannot be combined to create a new LockedShortDoor^c class for doors that are both locked and short.

```
class LockedDoorc extends Doorc {
  boolean canOpen(Personc 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 ShortDoorc extends Doorc {
  boolean canPass(Personc p) {
    if (p.height() > 1) {
      System.out.println("You are too tall");
      return false;
    }
    System.out.println("Ducking into door...");
    return super.canPass(p);
  }
}

/* Cannot merge for LockedShortDoorc */
```

```
interface Doori {
  boolean canOpen(Personc p);
  boolean canPass(Personc p);
}
mixin Lockedm extends Doori {
  boolean canOpen(Personc 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 Shortm extends Doori {
  boolean canPass(Personc p) {
    if (p.height() > 1) {
      System.out.println("You are too tall");
      return false;
    }
    System.out.println("Ducking into door...");
    return super.canPass(p);
  }
}
class LockedDoorc = Lockedm(Doorc);
class ShortDoorc = Shortm(Doorc);
class LockedShortDoorc = Lockedm(Shortm(Doorc));
```

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^i extends Door^i {
  Object neededItem();
}
mixin Secure^m extends Door^i implements Secure^i {
   Object neededItem() { return null; }
   boolean canOpen(Person^c p) {
      Object item = neededItem();
      if (!p.hasItem(item)) {
        System.out.println("You don't have the " + item);
        return false;
      }
      System.out.println("Using " + item + "...");
      return super.canOpen(p);
   }
}
mixin NeedsKey^m extends Secure^i {
   Object neededItem() {
      return theKey;
   }
}
mixin NeedsSpell^m extends Secure^i {
   Object neededItem() {
      return theSpellBook;
   }
}
mixin Locked^m = NeedsKey^m compose Secure^m;
mixin Magic^m = NeedsSpell^m compose Secure^m;
mixin LockedMagic^m = Locked^m compose Magic^m;
mixin LockedMagicDoor^m = LockedMagic^m compose Door^m;
class LockedDoor^c = Locked^m(Door^c); ...
```

Figure 4.10 : Composing mixins for localized parameterization

The mixin $\mathsf{Locked}^{\mathsf{m}}$ is nearly identical to the original $\mathsf{LockedDoor}^{\mathsf{c}}$ class definition, except that the superclass is specified via the interface $\mathsf{Door}^{\mathsf{i}}$. The new $\mathsf{LockedDoor}^{\mathsf{c}}$ and $\mathsf{ShortDoor}^{\mathsf{c}}$ classes are created by applying $\mathsf{Locked}^{\mathsf{m}}$ and $\mathsf{Short}^{\mathsf{m}}$ to the class $\mathsf{Door}^{\mathsf{c}}$, respectively. Similarly, applying $\mathsf{Locked}^{\mathsf{m}}$ to $\mathsf{ShortDoor}^{\mathsf{c}}$ yields a class for locked, short doors.

Consider another door variation: $\mathsf{MagicDoor}^{\mathsf{c}}$, which is similar to $\mathsf{LockedDoor}^{\mathsf{c}}$ except that the player needs a book of spells instead of a key. We can extract the common parts of the implementation of $\mathsf{MagicDoor}^{\mathsf{c}}$ and $\mathsf{LockedDoor}^{\mathsf{c}}$ into a new mixin, $\mathsf{Secure}^{\mathsf{m}}$. Then, key- or book-specific information is composed with $\mathsf{Secure}^{\mathsf{m}}$ to produce $\mathsf{Locked}^{\mathsf{m}}$ and $\mathsf{Magic}^{\mathsf{m}}$, as shown in Figure 4.10. Each of the new mixins extends $\mathsf{Door}^{\mathsf{i}}$ since the right hand mixin in the composition, $\mathsf{Secure}^{\mathsf{m}}$, extends $\mathsf{Door}^{\mathsf{i}}$.

The $\mathsf{Locked}^{\mathsf{m}}$ and $\mathsf{Magic}^{\mathsf{m}}$ mixins can also be composed to form $\mathsf{LockedMagic}^{\mathsf{m}}$.

This mixin has the expected behavior: to open an instance of $\mathsf{LockedMagic}^{\mathsf{m}}$, 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 $\mathsf{Locked}^{\mathsf{m}}$ and $\mathsf{Magic}^{\mathsf{m}}$ do not interfere with each other because the interface extended by $\mathsf{Locked}^{\mathsf{m}}$ is $\mathsf{Door}^{\mathsf{i}}$, which does not contain *neededItem*. In contrast, $\mathsf{Door}^{\mathsf{i}}$ does contain *canOpen*, so the *canOpen* method in $\mathsf{Locked}^{\mathsf{m}}$ overrides and chains to the *canOpen* in $\mathsf{Magic}^{\mathsf{m}}$.

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.[4] 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 x that is not contained in its inheritance interface, then that declaration never overrides another x.

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

[4] We 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 $\mathsf{LockedMagicDoor}^{\mathsf{m}}$ 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 $\mathsf{Secure}^{\mathsf{m}}$'s *neededItem* to $\mathsf{Magic}^{\mathsf{m}}$'s method because *neededItem* is not included in the $\mathsf{Door}^{\mathsf{i}}$ interface. The *canOpen* method is in both $\mathsf{Door}^{\mathsf{i}}$ and $\mathsf{Secure}^{\mathsf{i}}$, 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

[5]Our 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.

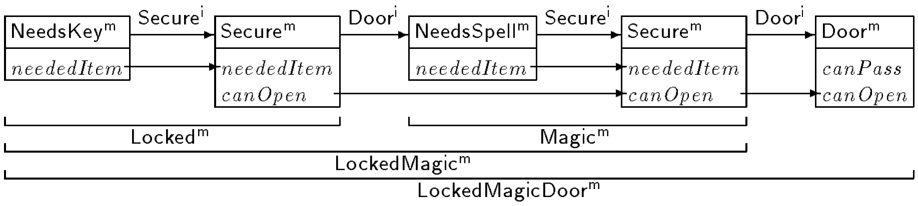


Figure 4.11 : $\mathsf{LockedMagicDoor}^{\mathsf{m}}$ mixin corresponds to a sequence of atomic mixins

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.

$$
\begin{array}{rcl}
\mathit{defn} & = & \textbf{mixin}\ m\ \textbf{extends}\ i\ \textbf{implements}\ i^{*}\ \{\ \mathit{field}^{*}\ \mathit{meth}^{*}\ \} \\
 & & \mid\ \textbf{mixin}\ m = m\ \textbf{compose}\ m \\
 & & \mid\ \textbf{interface}\ i\ \textbf{extends}\ i^{*}\ \{\ \mathit{meth}^{*}\ \} \\
e & = & \textbf{new}\ m \mid \mathit{var} \mid \mathsf{null} \mid e.\mathit{fd} \mid e.\mathit{fd} = e \\
 & & \mid\ e.\mathit{md}\ (e^{*}) \mid \textbf{super}\underline{\ \equiv \textbf{this}\ }.\mathit{md}\ (e^{*}) \\
 & & \mid\ \textbf{view}\underline{\ t\ \textbf{as}\ }t\ e \mid \textbf{let}\ \mathit{var} = e\ \textbf{in}\ e \\
m & = & \text{mixin name} \\
t & = & m \mid i
\end{array}
$$

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

MixinsOnce(P)	Each mixin name is declared only once **mixin** $m \cdots$ **mixin** $m' \cdots$ is in $P \Longrightarrow m \neq m'$
FieldOncePerMixin(P)	Field names in each mixin declaration are unique **mixin** $\cdots \{ \cdots fd \cdots fd' \cdots \}$ is in $P \Longrightarrow fd \neq fd'$
MethodOncePerMixin(P)	Method names in each mixin declaration are unique **mixin** $\cdots \{ \cdots md\ (\cdots)\ \{\cdots\} \cdots md'\ (\cdots)\ \{\cdots\} \cdots \}$ is in $P \Longrightarrow md \neq md'$
NoAbstractMixins(P)	Methods in a mixin are not **abstract** **mixin** $\cdots \{ \cdots md\ (\cdots)\ \{ e \} \cdots \}$ is in $P \Longrightarrow e \neq$ **abstract**
$\prec^{\mathsf{m}}_{P}$	Mixin declares an inheritance interface $m \prec^{\mathsf{m}}_{P} i \Leftrightarrow$ **mixin** m **extends** $i \cdots \{ \cdots \}$ is in P
$\prec\!\!\prec^{\mathsf{m}}_{P}$	Mixin declares implementation of an interface $m \prec\!\!\prec^{\mathsf{m}}_{P} i \Leftrightarrow$ **mixin** $m \cdots$ **implements** $\cdots i \cdots \{ \cdots \}$ is in P
$\bullet \doteq^{\mathsf{m}}_{P} \bullet \circ \bullet$	Mixin is declared as a composition $m \doteq^{\mathsf{m}}_{P} m' \circ m'' \Leftrightarrow$ **mixin** $m =m'$ **compose** m'' is in P
$\Subset^{\mathsf{m}}_{P}$	Method is declared in a mixin $\langle md,\ (t_1 \ldots t_n \longrightarrow t),\ (var_1\ \ldots\ var_n),\ e\rangle \Subset^{\mathsf{m}}_{P} m$ $\Leftrightarrow$ **mixin** $m \cdots \{ \cdots t\ md\ (t_1\ var_1\ \ldots\ t_n\ var_n)\ \{ e \} \cdots \}$ is in P
$\Subset^{\mathsf{m}}_{P}$	Field is declared in a mixin $\langle m.fd,\ t\rangle \Subset^{\mathsf{m}}_{P} m \Leftrightarrow$ **mixin** $m \cdots \{ \cdots t\ fd \cdots \}$ is in P
$\leq^{\mathsf{m}}_{P}$	Mixin is a submixin $m \leq^{\mathsf{m}}_{P} m' \Leftrightarrow m = m'$ or $(\exists m'', m'''$ s.t. $m \doteq^{\mathsf{m}}_{P} m'' \circ m'''$ and $(m'' \leq^{\mathsf{m}}_{P} m'$ or $m''' \leq^{\mathsf{m}}_{P} m'))$
$\trianglelefteq^{\mathsf{m}}_{P}$	Mixin is viewable as a mixin $m \trianglelefteq^{\mathsf{m}}_{P} m' \Leftrightarrow m = m'$ or $(\exists m'', m'''$ s.t. $m \doteq^{\mathsf{m}}_{P} m'' \circ m'''$ and $(m'' \leq^{\mathsf{m}}_{P} m'$ xor $m''' \leq^{\mathsf{m}}_{P} m')$ and $(m'' \trianglelefteq^{\mathsf{m}}_{P} m'$ xor $m''' \trianglelefteq^{\mathsf{m}}_{P} m'))$
CompleteMixins(P)	Mixins that are composed are defined $\mathrm{rng}(\doteq^{\mathsf{m}}_{P}) \subseteq \{m \circ m' \mid m, m' \in \mathrm{dom}(\prec^{\mathsf{m}}_{P}) \cup \mathrm{dom}(\doteq^{\mathsf{m}}_{P})\}$
WellFoundedMixins(P)	Mixin hierarchy is an order $\leq^{\mathsf{m}}_{P}$ is antisymmetric
CompleteInterfaces(P)	Extended/implemented interfaces are defined $\mathrm{rng}(\prec^{\mathsf{i}}_{P}) \cup \mathrm{rng}(\prec^{\mathsf{m}}_{P}) \cup \mathrm{rng}(\prec\!\!\prec^{\mathsf{m}}_{P}) \subseteq \mathrm{dom}(\prec^{\mathsf{i}}_{P}) \cup \{\mathsf{Empty}\}$
$\dashv^{\mathsf{m}}_{P}$	Mixin extends an interface $m \dashv^{\mathsf{m}}_{P} i \Leftrightarrow m \prec^{\mathsf{m}}_{P} i$ or $(\exists m', m''$ s.t. $m \doteq^{\mathsf{m}}_{P} m' \circ m''$ and $m'' \dashv^{\mathsf{m}}_{P} i)$
$\ll^{\mathsf{m}}_{P}$	Mixin implements an interface $m \ll^{\mathsf{m}}_{P} i \Leftrightarrow \exists m', i'$ s.t. $m \leq^{\mathsf{m}}_{P} m'$ and $i' \leq^{\mathsf{i}}_{P} i$ and $(m' \prec^{\mathsf{m}}_{P} i'$ or $m' \prec\!\!\prec^{\mathsf{m}}_{P} i')$
$\lll^{\mathsf{m}}_{P}$	Mixin is viewable as an interface $m \lll^{\mathsf{m}}_{P} i \Leftrightarrow (\exists i'$ s.t. $i' \leq^{\mathsf{i}}_{P} i$ and $(m \prec^{\mathsf{m}}_{P} i'$ or $m \prec\!\!\prec^{\mathsf{m}}_{P} i'))$ or $(\exists m', m''$ s.t. $m \doteq^{\mathsf{m}}_{P} m' \circ m''$ and $(m' \ll^{\mathsf{m}}_{P} i$ xor $m'' \ll^{\mathsf{m}}_{P} i)$ and $(m' \lll^{\mathsf{m}}_{P} i$ xor $m'' \lll^{\mathsf{m}}_{P} i))$
MixinCompositionsOK(P)	Mixins are composed safely $m \doteq^{\mathsf{m}}_{P} m' \circ m'' \Longrightarrow \exists i$ s.t. $m' \dashv^{\mathsf{m}}_{P} i$ and $m'' \lll^{\mathsf{m}}_{P} i$
MixinMethodsOK(P)	Method definitions match inheritance interface $(\langle md,\ T,\ V,\ e\rangle \Subset^{\mathsf{m}}_{P} m$ and $\langle md,\ T',\ V',$ **abstract**$\rangle \Subset^{\mathsf{m}}_{P} i) \Longrightarrow (T = T'$ or $m \not\dashv^{\mathsf{m}}_{P} i)$
$\in^{\mathsf{m}}_{P}$	Field is contained and visible in a mixin $\langle m'.fd,\ t\rangle \in^{\mathsf{m}}_{P} m \Leftrightarrow m \trianglelefteq^{\mathsf{m}}_{P} m'$ and $\{\langle m'.fd,\ t\rangle\} = \{\langle m'.fd,\ t\rangle \mid m \leq^{\mathsf{m}}_{P} m'$ and $\langle m'.fd,\ t\rangle \Subset^{\mathsf{m}}_{P} m'\}$
$\underline{\Subset}^{\mathsf{m}}_{P}$	Method is potentially visible in a mixin (used for $\in^{\mathsf{m}}_{P}$) $\langle md,\ T\rangle \underline{\Subset}^{\mathsf{m}}_{P} m \Leftrightarrow (\exists V, e$ s.t. $\langle md,\ T,\ V,\ e\rangle \Subset^{\mathsf{m}}_{P} m)$ or $(\exists i, V$ s.t. $m \prec^{\mathsf{m}}_{P} i$ and $\langle md,\ T,\ V,$ **abstract**$\rangle \in^{\mathsf{i}}_{P} i)$ or $(\exists m', m''$ s.t. $m \doteq^{\mathsf{m}}_{P} m' \circ m''$ and $(\langle md,\ T\rangle \underline{\Subset}^{\mathsf{m}}_{P} m'$ or $\langle md,\ T\rangle \underline{\Subset}^{\mathsf{m}}_{P} m''))$
$\in^{\mathsf{m}}_{P}$	Method is visible in a mixin $\langle md,\ T\rangle \in^{\mathsf{m}}_{P} m \Leftrightarrow \langle md,\ T\rangle \underline{\Subset}^{\mathsf{m}}_{P} m$ and $((\exists i$ s.t. $m \prec^{\mathsf{m}}_{P} i)$ or $(\exists m', m'', i$ s.t. $m \doteq^{\mathsf{m}}_{P} m' \circ m''$ and $m' \dashv^{\mathsf{m}}_{P} i$ and $(\langle md,\ T'\rangle \underline{\Subset}^{\mathsf{m}}_{P} m' \Rightarrow \langle md,\ T'\rangle \in^{\mathsf{m}}_{P} m')$ and $(\langle md,\ T''\rangle \underline{\Subset}^{\mathsf{m}}_{P} m'' \Rightarrow \langle md,\ T''\rangle \in^{\mathsf{m}}_{P} m'')$ and $(((\langle md,\ T'\rangle \in^{\mathsf{m}}_{P} m'$ and $\langle md,\ T''\rangle \in^{\mathsf{m}}_{P} m'')$ $\Rightarrow (\exists V$ s.t. $\langle md,\ T,\ V,$ **abstract**$\rangle \in^{\mathsf{i}}_{P} i))))$

Figure 4.13 : Predicates and relations in the model of MixedJava (Part I)

$\textsc{MixinsImplementAll}(P)$ — Mixins supply methods to implement interfaces

$$
\begin{aligned}
m \ll^{\mathsf{m}}_P i \Longrightarrow (\forall md, T\ \langle md,\ T,\ V,\ \textbf{abstract}\rangle \in^{\mathsf{i}}_P i \\
\Longrightarrow (\exists\ e\ \text{ s.t. }\ \langle md,\ T,\ V,\ e\rangle \Subset^{\mathsf{m}}_P m \\
\text{or } \exists\ i'\ \ s.t.\ \ (m \lessdot^{\mathsf{m}}_P i' \\
\text{and } \langle md,\ T,\ V,\ \textbf{abstract}\rangle \in^{\mathsf{i}}_P i')))
\end{aligned}
$$

$\in^{\mathsf{i}}_P$ — Method with type in an interface

$$\langle md,\ T\rangle \in^{\mathsf{i}}_P i \Leftrightarrow \exists V\ \text{ s.t. }\ \langle md,\ T,\ V,\ \textbf{abstract}\rangle \in^{\mathsf{i}}_P i$$

$\leq_P$ — Type is a subtype — $\leq_P\ \equiv\ \leq^{\mathsf{m}}_P \cup \leq^{\mathsf{i}}_P \cup \ll^{\mathsf{m}}_P$

$\trianglelefteq_P$ — Type is viewable as another type — $\trianglelefteq_P\ \equiv\ \trianglelefteq^{\mathsf{m}}_P \cup \leq^{\mathsf{i}}_P \cup \lll^{\mathsf{m}}_P$

$\in_P$ — Field or method is in a type — $\in_P\ \equiv\ \in^{\mathsf{m}}_P \cup \in^{\mathsf{i}}_P$

:: and @ — Chain constructors

:: adds an element to the beginning of a chain; @ appends two chains

$\longrightarrow_P$ — Mixin corresponds to a chain of atomic mixins

$$
\begin{aligned}
& m \longrightarrow_P M \\
& \Leftrightarrow (\exists i\ \text{ s.t. }\ m \prec^{\mathsf{m}}_P i \text{ and } M = [m]) \\
& \quad \text{or } (\exists m', m'', M', M''\ \text{ s.t. }\ m \doteq^{\mathsf{m}}_P m' \circ m'' \text{ and } m' \longrightarrow_P M' \\
& \qquad \text{and } m'' \longrightarrow_P M'' \text{ and } M = M'@M'')
\end{aligned}
$$

$\leq^{\mathsf{M}}$ — Chains have an inverted subsequence order

$$M \leq^{\mathsf{M}} M' \Leftrightarrow \exists M''\ \text{ s.t. }\ M = M''@M'$$

$\bullet/\bullet \rhd \bullet/\bullet$ — Mixin view operation selects a new chain

$$
\begin{aligned}
M/m \rhd M'/m' \Leftrightarrow\ & (m = m' \text{ and } M = M') \\
& \text{or } (\exists m'', m'''\ \text{ s.t. }\ m \doteq^{\mathsf{m}}_P m'' \circ m''' \\
& \quad \text{and } ((m'' \leq^{\mathsf{m}}_P m' \text{ and } M/m'' \rhd M'/m') \\
& \qquad \text{or } (m''' \leq^{\mathsf{m}}_P m' \text{ and } \exists M_l, M_r\ \text{ s.t. }\ m'' \longrightarrow_P M_l \text{ and } M = M_l@M_r \\
& \qquad\qquad \text{and } M_r/m''' \rhd M'/m'))
\end{aligned}
$$

$\bullet/\bullet \rhd \bullet/\bullet$ — Interface view operation selects a new chain

$$M/t \rhd M'/i \Leftrightarrow M' = \min\{m{::}M'' \mid m \ll^{\mathsf{m}}_P i \text{ and } M \leq^{\mathsf{M}} m{::}M''\}$$

$\bullet.\bullet \propto \bullet.\bullet$ — Method in a chain is the same as in another chain

$$
\begin{aligned}
m{::}M.md \propto M'.md \Leftrightarrow\ & m{::}M = M' \text{ or } (\exists i, T, V, M''\ \text{ s.t. }\ m \prec^{\mathsf{m}}_P i \text{ and } \langle md,\ T\rangle \in^{\mathsf{i}}_P i \\
& \quad \text{and } M/i \rhd M''/i \text{ and } M''.md \propto M'.md)
\end{aligned}
$$

$\bullet \in^{\mathsf{rt}}_P \bullet \text{ in } \bullet$ — Method selects a view within a chain and subchain

$$
\begin{aligned}
& \langle md,\ T,\ V,\ e,\ m{::}M/m\rangle \in^{\mathsf{rt}}_P M_v \text{ in } M_o \\
& \Leftrightarrow \langle md,\ T,\ V,\ e\rangle \Subset^{\mathsf{m}}_P m \\
& \quad \text{and } m_x{::}M_x = \min\{m_x{::}M_x \mid M_v \leq^{\mathsf{M}} m_x{::}M_x \text{ and } \langle md,\ T\rangle \in^{\mathsf{m}}_P m_x\} \\
& \quad \text{and } M_b = \max\{M' \mid m_x{::}M_x.md \propto M'.md\} \\
& \quad \text{and } m{::}M = \min\{m{::}M \mid M_o \leq^{\mathsf{M}} m{::}M \\
& \qquad \text{and } m{::}M.md \propto M_b.md \\
& \qquad \text{and } \exists V', e'\ \text{ s.t. }\ \langle md,\ T,\ V',\ e'\rangle \Subset^{\mathsf{m}}_P m\}
\end{aligned}
$$

Figure 4.14 : Predicates and relations in the model of MixedJava (Part II)

program. A well-formed program induces a subtype relation $\leq^{\mathsf{m}}_P$ 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, $\mathsf{LockedMagic}^{\mathsf{m}}$ is a subtype of $\mathsf{Secure}^{\mathsf{m}}$, but it contains two copies of $\mathsf{Secure}^{\mathsf{m}}$ (see Figure 4.11). Hence, interpreting an instance of $\mathsf{LockedMagic}^{\mathsf{m}}$ as an instance of $\mathsf{Secure}^{\mathsf{m}}$ is ambiguous. More concretely, the fragment

$$\mathsf{LockedMagicDoor}^{\mathsf{m}}\ door = \textbf{new}\ \mathsf{LockedMagicDoor}^{\mathsf{m}};$$
$$(\textbf{view}\ \mathsf{Secure}^{\mathsf{m}}\ door).neededItem();$$

is ill-formed because $\mathsf{LockedMagic}^{\mathsf{m}}$ is not uniquely viewable as $\mathsf{Secure}^{\mathsf{m}}$. To eliminate such ambiguities, we introduce the "viewable as" relation $\trianglelefteq_P$, which is a restriction on the subtype relation. Subsumption is thus based on $\trianglelefteq_P$ rather than $\leq_P$. The relations $\in^{\mathsf{m}}_P$, 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**$^{\mathsf{i}}$, **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**$^{\mathsf{m}}$ 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**$^{\mathsf{m}}$ rule strips out the **view** part of the expression and delegates all work to the subsumption rules. Third, the **sub**$^{\mathsf{m}}$ 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 **class**es 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

$\vdash_{\mathsf{p}}$

$$\frac{\begin{array}{l}\textsc{MixinsOnce}(P) \quad \textsc{MethodOncePerMixin}(P) \quad \textsc{InterfacesOnce}(P) \quad \textsc{CompleteMixins}(P) \\ \textsc{WellFoundedMixins}(P) \quad \textsc{CompleteInterfaces}(P) \quad \textsc{WellFoundedInterfaces}(P) \\ \textsc{MixinFieldsOK}(P) \quad \textsc{MixinMethodsOK}(P) \quad \textsc{InterfaceMethodsOK}(P) \\ \textsc{InterfacesAbstract}(P) \quad \textsc{NoAbstractMixins}(P) \quad \textsc{MethodArgsDistinct}(P) \\ \textsc{MixinsImplementAll}(P) \quad P \vdash_{\mathsf{d}} \mathit{defn}_j \Rightarrow \mathit{defn}'_j \text{ for } j \in [1,n] \quad P, [\,] \vdash_{\mathsf{e}} e \Rightarrow e' : t \\ \text{where } P = \mathit{defn}_1 \ldots \mathit{defn}_n\ e\end{array}}{\vdash_{\mathsf{p}} \mathit{defn}_1 \ldots \mathit{defn}_n\ e \Rightarrow \mathit{defn}'_1 \ldots \mathit{defn}'_n\ e' : t}[\mathbf{prog}^{\mathsf{m}}]$$

$\vdash_{\mathsf{d}}$

$$\frac{P \vdash_{\mathsf{t}} t_j \text{ for each } j \in [1,n] \quad P, m \vdash_{\mathsf{m}} \mathit{meth}_k \Rightarrow \mathit{meth}'_k \text{ for each } k \in [1,p]}{\begin{array}{ll}P \vdash_{\mathsf{d}} \textbf{mixin}\ m \cdots \{\ t_1\ \mathit{fd}_1 \ldots t_n\ \mathit{fd}_n & \Rightarrow \textbf{mixin}\ m \cdots \{\ t_1\ \mathit{fd}_1 \ldots t_n\ \mathit{fd}_n \\ \qquad \mathit{meth}_1 \ldots \mathit{meth}_p\ \} & \qquad \mathit{meth}'_1 \ldots \mathit{meth}'_p\ \}\end{array}}[\mathbf{defn}^{\mathsf{m}}]$$

$\vdash_{\mathsf{e}}$

$$\frac{\begin{array}{c}P \vdash_{\mathsf{t}} m \\ m \prec\!\!\!\cdot^{\mathsf{m}}_P \mathsf{Empty}\end{array}}{P, \Gamma \vdash_{\mathsf{e}} \textbf{new}\ m \Rightarrow \textbf{new}\ m : m}[\mathbf{new}^{\mathsf{m}}] \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : m \quad \langle m'.\mathit{fd}, t\rangle \in_P m}{P, \Gamma \vdash_{\mathsf{e}} e.\mathit{fd} \Rightarrow e'.\mathit{fd} : t}[\mathbf{get}^{\mathsf{m}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : m \quad \langle m'.\mathit{fd}, t\rangle \in_P m \quad P, \Gamma \vdash_{\mathsf{s}} e_v \Rightarrow e'_v : t}{P, \Gamma \vdash_{\mathsf{e}} e.\mathit{fd} = e_v \Rightarrow e'.\mathit{fd} = e'_v : t}[\mathbf{set}^{\mathsf{m}}]$$

$$\frac{\begin{array}{ll}P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' & \langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t)\rangle \in_P t' \\ P, \Gamma \vdash_{\mathsf{s}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n] & \end{array}}{P, \Gamma \vdash_{\mathsf{e}} e.\mathit{md}\ (e_1 \ldots e_n) \Rightarrow e'.\mathit{md}\ (e'_1 \ldots e'_n) : t}[\mathbf{call}^{\mathsf{m}}]$$

$$\frac{\begin{array}{lll}P, \Gamma \vdash_{\mathsf{e}} \textbf{this} \Rightarrow \textbf{this} : m & m \prec\!\!\!\cdot^{\mathsf{m}}_P i & \langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t)\rangle \in_P i \\ P, \Gamma \vdash_{\mathsf{s}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n] & & \end{array}}{P, \Gamma \vdash_{\mathsf{e}} \textbf{super}.\mathit{md}(e_1 \ldots e_n) \Rightarrow \textbf{super}\underline{\ \equiv \textbf{this}\ }.\mathit{md}(e'_1 \ldots e'_n) : t}[\mathbf{super}^{\mathsf{m}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t}{P, \Gamma \vdash_{\mathsf{e}} \textbf{view}\ t\ e \Rightarrow e' : t}[\mathbf{wcast}^{\mathsf{m}}] \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \quad t' \not\trianglelefteq_P t}{P, \Gamma \vdash_{\mathsf{e}} \textbf{view}\ t\ e \Rightarrow \textbf{view}\underline{\ t'\ \textbf{as}\ }t\ e' : t}[\mathbf{ncast}^{\mathsf{m}}]$$

$\vdash_{\mathsf{s}}$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \quad t' \trianglelefteq_P t}{P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow \textbf{view}\underline{\ t'\ \textbf{as}\ }t\ e' : t}[\mathbf{sub}^{\mathsf{m}}]$$

$\vdash_{\mathsf{t}}$

$$\frac{t \in \mathrm{dom}(\prec^{\mathsf{m}}_P) \cup \mathrm{dom}(\doteq^{\mathsf{m}}_P) \cup \mathrm{dom}(\prec^{\mathsf{i}}_P) \cup \{\mathsf{Empty}\}}{P \vdash_{\mathsf{t}} t}[\mathbf{type}^{\mathsf{m}}]$$

Figure 4.15 : Context-sensitive checks and type elaboration rules for MixedJava

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 run-time 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 $\langle object \| view \rangle$. 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 $\mathsf{LockedMagicDoor}^{\mathsf{m}}$ is used as a $\mathsf{Magic}^{\mathsf{m}}$ instance, the object's view corresponds to the boxed part of the following chain:

$$[\mathsf{NeedsKey}^{\mathsf{m}}\ \mathsf{Secure}^{\mathsf{m}}\ \boxed{\mathsf{NeedsSpell}^{\mathsf{m}}\ \mathsf{Secure}^{\mathsf{m}}}\ \mathsf{Door}^{\mathsf{m}}]$$

The full chain corresponds to $\mathsf{LockedMagicDoor}^{\mathsf{m}}$ and the boxed part corresponds to $\mathsf{Magic}^{\mathsf{m}}$. 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 *neededItem* method of the object begins in the $\mathsf{NeedsSpell}^{\mathsf{m}}$ element of the chain.

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

$$[\boxed{\mathsf{NeedsSpell}^{\mathsf{m}}\ \mathsf{Secure}^{\mathsf{m}}}\ \mathsf{Door}^{\mathsf{m}}]$$

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:[7]

$$[\mathsf{NeedsSpell}^{\mathsf{m}}\ \mathsf{Secure}^{\mathsf{m}}\ \mathsf{Door}^{\mathsf{m}}]/\mathsf{Magic}^{\mathsf{m}}.$$

The view-based dispatching algorithm, described by the $\in^{\mathrm{rt}}_{P}$ relation, proceeds in two phases. The first phase of a search for method x locates the *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 $\propto$ 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 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 $\mathsf{Secure^m}$'s *canOpen* method calls the appropriate *neededItem* method in an instance of $\mathsf{LockedMagicDoor^m}$, sometimes dispatching to the method in $\mathsf{NeedsKey^m}$ and sometimes to the one in $\mathsf{NeedsSpell^m}$. The following example illustrates the essence of dispatching from $\mathsf{Secure^m}$'s *canOpen*:

$$\mathsf{Object}\ \ canOpen(\mathsf{Secure^m}\ \ o)\ \{\ \ldots\ o.neededItem()\ \ldots\ \}$$

$$\begin{array}{l} \textbf{let}\ \ door = \textbf{new}\ \ \mathsf{LockedMagicDoor^m} \\ \ \textbf{in}\ \ canOpen(\underline{\textbf{view}\ \mathsf{Secure^m}}\ \ \textbf{view}\ \ \mathsf{Locked^m}\ \ door)\ \ldots \\ \qquad canOpen(\underline{\textbf{view}\ \mathsf{Secure^m}}\ \ \textbf{view}\ \ \mathsf{Magic^m}\ \ door) \end{array}$$

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

$$[\mathsf{NeedsKey^m}\ \mathsf{Secure^m}\ \mathsf{NeedsSpell^m}\ \mathsf{Secure^m}\ \mathsf{Door^m}]/\mathsf{LockedMagicDoor^m}.$$

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

$$[\mathsf{Secure^m}\ \mathsf{NeedsSpell^m}\ \mathsf{Secure^m}\ \mathsf{Door^m}]/\mathsf{Secure^m}.$$

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

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

$$[\mathsf{Secure^m}\ \mathsf{Door^m}]/\mathsf{Secure^m}.$$

In this view, the base definition of *neededItem* is in the rightmost $\mathsf{Secure^m}$ of the full chain, and it is overridden in $\mathsf{NeedsSpell^m}$. Neither the definition of *neededItem* in $\mathsf{NeedsKey^m}$ nor the one in the leftmost occurrence of $\mathsf{Secure^m}$ is a candidate relative to the given view, because $\mathsf{Secure^m}$ 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 $\in_P^{\mathsf{rt}}$ 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 $\langle object \| view \rangle$ pairs.

$$
\begin{array}{rcl}
e &=& \ldots \mid \langle object \| M/t \rangle \\
v &=& \langle object \| M/t \rangle \mid \mathsf{null}
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{E} &=& [\,] \mid \mathsf{E}.fd \mid \mathsf{E}.fd = e \mid v.fd = \mathsf{E} \\
&\mid& \mathsf{E}.md(e\ \ldots) \mid v.md(v\ \ldots\ \mathsf{E}\ e\ \ldots) \\
&\mid& \mathbf{super}\underline{\ \equiv v\ }.md(v\ \ldots\ \mathsf{E}\ e\ \ldots) \\
&\mid& \mathbf{view}\underline{\ t\ \mathbf{as}\ }t\ \mathsf{E} \mid \mathbf{let}\ var = \mathsf{E}\ \mathbf{in}\ e
\end{array}
$$

$$
\begin{array}{ll}
P \vdash \langle \mathsf{E}[\mathbf{new}\ m], \mathcal{S} \rangle & [new] \\
\quad \hookrightarrow \langle \mathsf{E}[\langle object \| M/m \rangle], \mathcal{S}[object \mapsto \langle m, [M_1.fd_1 \mapsto \mathsf{null}, \ldots M_n.fd_n \mapsto \mathsf{null}] \rangle] \rangle & \\
\quad \text{where } object \notin \mathrm{dom}(\mathcal{S}) \text{ and } m \longrightarrow_P M & \\
\qquad \{M_1.fd_1, \ldots M_n.fd_n\} = \{m'{::}M'.fd \mid M \leq^{\mathsf{M}} m'{::}M' & \\
\qquad\qquad \text{and } \exists t \text{ s.t. } \langle m'.fd, t \rangle \in_P^{\mathsf{m}} m'\} & \\
P \vdash \langle \mathsf{E}[\langle object \| M/m \rangle.fd], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S} \rangle & [get] \\
\quad \text{where } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } \langle m'.fd, t \rangle \in_P m \text{ and } M/m \rhd M'/m' \text{ and } \mathcal{F}(M'.fd) = v & \\
P \vdash \langle \mathsf{E}[\langle object \| M/m \rangle.fd = v], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}[object \mapsto \langle m, \mathcal{F}[M'.fd \mapsto v] \rangle] \rangle & [set] \\
\quad \text{where } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } \langle m'.fd, t \rangle \in_P m \text{ and } M/m \rhd M'/m' & \\
P \vdash \langle \mathsf{E}[\langle object \| M/t \rangle.md(v_1, \ldots v_n)], \mathcal{S} \rangle & [call] \\
\quad \hookrightarrow \langle \mathsf{E}[e[\langle object \| m'{::}M'/m' \rangle/\mathbf{this}, v_1/var_1, \ldots v_n/var_n]], \mathcal{S} \rangle & \\
\quad \text{where } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } m \longrightarrow_P M_o & \\
\qquad \text{and } \langle md, T, (var_1\ \ldots\ var_n), e, m'{::}M'/m' \rangle \in_P^{\mathsf{rt}} M \text{ in } M_o & \\
P \vdash \langle \mathsf{E}[\mathbf{super}\underline{\ \equiv \langle object \| m{::}M/m \rangle\ }.md(v_1, \ldots v_n)], \mathcal{S} \rangle & [super] \\
\quad \hookrightarrow \langle \mathsf{E}[e[\overline{\langle object \| m'{::}M'/m' \rangle}/\mathbf{this}, v_1/var_1, \ldots v_n/var_n]], \mathcal{S} \rangle & \\
\quad \text{where } m \prec_P^{\mathsf{m}} i \text{ and } M/i \rhd M''/i & \\
\qquad \text{and } \langle md, T, (var_1\ \ldots\ var_n), e, m'{::}M'/m' \rangle \in_P^{\mathsf{rt}} M'' \text{ in } M'' & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\underline{\ t'\ \mathbf{as}\ }t\ \langle object \| M/t' \rangle], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[\langle object \| M'/t \rangle], \mathcal{S} \rangle & [view] \\
\quad \text{where } t' \trianglelefteq_P t \text{ and } M/t' \rhd M'/t & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\underline{\ t'\ \mathbf{as}\ }t\ \langle object \| M/t' \rangle], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[\langle object \| M''/t \rangle], \mathcal{S} \rangle & [cast] \\
\quad \text{where } t' \ntrianglelefteq_P t \text{ and } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } m \trianglelefteq_P t \text{ and } m \longrightarrow_P M' \text{ and } M'/m \rhd M''/t & \\
P \vdash \langle \mathsf{E}[\mathbf{let}\ var = v\ \mathbf{in}\ e], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{E}[e[v/var]], \mathcal{S} \rangle & [let] \\
 & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\underline{\ t'\ \mathbf{as}\ }t\ \langle object \| M/t' \rangle], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{error\colon\ bad\ cast}, \mathcal{S} \rangle & [xcast] \\
\quad \text{where } t' \ntrianglelefteq_P t \text{ and } \mathcal{S}(object) = \langle m, \mathcal{F} \rangle \text{ and } m \ntrianglelefteq_P t & \\
P \vdash \langle \mathsf{E}[\mathbf{view}\underline{\ t'\ \mathbf{as}\ }t\ \mathsf{null}], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{error\colon\ bad\ cast}, \mathcal{S} \rangle & [ncast] \\
P \vdash \langle \mathsf{E}[\mathsf{null}.fd], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{error\colon\ dereferenced\ null}, \mathcal{S} \rangle & [nget] \\
P \vdash \langle \mathsf{E}[\mathsf{null}.fd = v], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{error\colon\ dereferenced\ null}, \mathcal{S} \rangle & [nset] \\
P \vdash \langle \mathsf{E}[\mathsf{null}.md(v_1, \ldots v_n)], \mathcal{S} \rangle \hookrightarrow \langle \mathsf{error\colon\ dereferenced\ null}, \mathcal{S} \rangle & [ncall]
\end{array}
$$

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* $\vdash_{\mathsf{p}} P \Rightarrow P' : t$ *and* $P' = defn_1 \ldots defn_n\ e$*, then either*

- $P' \vdash \langle e, \emptyset \rangle \hookrightarrow^* \langle \langle object \| M/t \rangle, \mathcal{S} \rangle$ *and* $\mathcal{S}(object) = \langle t', \mathcal{F} \rangle$ *and* $t' \leq_P t$*; or*
- $P' \vdash \langle e, \emptyset \rangle \hookrightarrow^* \langle \mathsf{null}, \mathcal{S} \rangle$*; or*
- $P' \vdash \langle e, \emptyset \rangle \hookrightarrow^* \langle \mathsf{error\colon bad\ cast}, \mathcal{S} \rangle$*; or*
- $P' \vdash \langle e, \emptyset \rangle \hookrightarrow^* \langle \mathsf{error\colon dereferenced\ null}, \mathcal{S} \rangle$*.*

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 ($\vdash_\sigma$) to reflect the differences between ClassicJava and MixedJava. In MixedJava, the environment Γ maps object-view pairs to the type part of the view, *i.e.*, $\Gamma(\langle object || M/t \rangle) = t$. The updated consistency relation is defined as follows:

Definition 4.3.2 (Environment-Store Consistency for MixedJava)

$$
\begin{array}{ll}
 & P,\Gamma \vdash_\sigma \mathcal{S} \\
 & \quad \Leftrightarrow (\mathcal{S}(object) = \langle m,\mathcal{F} \rangle \\
\Sigma_1\textit{:} & \qquad \Rightarrow (\Gamma(\langle object || M/t \rangle) = t' \\
 & \qquad\quad \Rightarrow (\mathrm{WF}(M/t)\ \textit{and}\ t = t'\ \textit{and}\ m \leq_P t)) \\
\Sigma_2\textit{:} & \qquad\quad \textit{and}\ \mathrm{dom}(\mathcal{F}) = \{m'{::}M'.fd \mid |m| \leq^{\mathsf{M}} m'{::}M'\ \textit{and} \\
 & \qquad\qquad\qquad\qquad \exists\ t\ \text{s.t.}\ \langle m'.fd,t \rangle \in^{\mathsf{m}}_P m'\} \\
\Sigma_3\textit{:} & \qquad\quad \textit{and}\ \{object \mid \langle object || \_\_ \rangle \in \mathrm{rng}(\mathcal{F})\} \subseteq \mathrm{dom}(\mathcal{S}) \cup \{\mathsf{null}\} \\
\Sigma_4\textit{:} & \qquad\quad \textit{and}\ (\mathcal{F}(m'{::}M'.fd) = \langle object' || M''/t' \rangle\ \textit{and}\ \langle m'.fd,t \rangle \in^{\mathsf{m}}_P m') \\
 & \qquad\qquad \Rightarrow (t' = t)) \\
\Sigma_5\textit{:} & \quad \textit{and}\ \langle object || \_\_ \rangle \in \mathrm{dom}(\Gamma) \Rightarrow object \in \mathrm{dom}(\mathcal{S}) \\
\Sigma_6\textit{:} & \quad \textit{and}\ object \in \mathrm{dom}(\mathcal{S}) \Rightarrow \langle object || \_\_ \rangle \in \mathrm{dom}(\Gamma)
\end{array}
$$

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)

$$\begin{aligned} \mathrm{WF}(M/t) \Leftrightarrow\ & \exists\ m_o, M_o \text{ s.t. } m_o \longrightarrow_P M_o \textit{ and } M_o \leq^{\mathsf{M}} M \\ & \textit{and } ((\exists\ M', M'' \text{ s.t. } M = M' \ @\ M'' \textit{ and } t \longrightarrow_P M') \\ & \quad \textit{or } (\exists\ m, M' \text{ s.t. } M = m{::}M' \textit{ and } m \trianglelefteq_P t)) \end{aligned}$$

The lemmata for proving MixedJava soundness are mostly the same as for ClassicJava, based on a revised typing relation $\vdash_{\underline{\mathsf{e}}}$. The annotations in MixedJava programs eliminate implicit subsumption by inserting explicit **view** expressions, so the $\vdash_{\underline{\mathsf{s}}}$ relation for proving MixedJava soundness is the same as $\vdash_{\underline{\mathsf{e}}}$. The $\vdash_{\underline{\mathsf{e}}}$ relation is like $\vdash_{\mathsf{e}}$, except for the handling of **view** expressions:

$$\frac{P, \Gamma \vdash_{\underline{\mathsf{e}}} e : t'}{P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathbf{view}\underline{\ t'\ \mathbf{as}\ }t\ e : t}[\mathbf{cast}^{\mathsf{m}}]$$

Also, as in ClassicJava, we define a SuperOk predicate for validating the shape of **super** calls:

Definition 4.3.4 (Well-Formed Super Calls)

$$\begin{aligned} \textsc{SuperOk}(e) \Leftrightarrow\ & \textit{For all } \mathbf{super}\underline{\ \equiv e_0\ }.md(e_1, \ldots\ e_n) \textit{ in } e, \\ & \textit{either } e_0 = \mathbf{this} \\ & \quad \textit{or } e_0 = \langle object \| m{::}M/m \rangle \textit{ for some object, } m, \textit{ and } M. \end{aligned}$$

Lemma 4.3.5 (Conserve for MixedJava) *If* $\vdash_{\mathsf{p}} P \Rightarrow P' : t$ *and* $P' = defn_1 \ldots defn_n\ e$, *then* $P', \emptyset \vdash_{\underline{\mathsf{e}}} e' : t$.

Proof. The claim follows from a copy-and-patch argument. □

Lemma 4.3.6 (Subject Reduction for MIXEDJAVA**)** *If* $P,\Gamma \vdash_{\underline{e}} e : t$, $P,\Gamma \vdash_{\sigma} \mathcal{S}$, SUPEROK$(e)$, *and* $\langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$, *then* e' *is an* error *configuration or there exists* Γ' *such that*

1. $P,\Gamma' \vdash_{\underline{e}} e' : t$,
2. $P,\Gamma' \vdash_{\sigma} \mathcal{S}'$, *and*
3. 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 Γ' 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. □

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

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. □

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 $\in_P^{\mathsf{rt}}$ 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* (**class**es, 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 MixedJava. 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:

```
(define name-mixin
  (lambda (superclass)
   (class superclass args
     (private [name "no name"])
     (public
       [set-name (lambda (n) (set! name n))]
       [get-name (lambda () name)])
     (sequence (apply super-init args)))))
```

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 run-time 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.

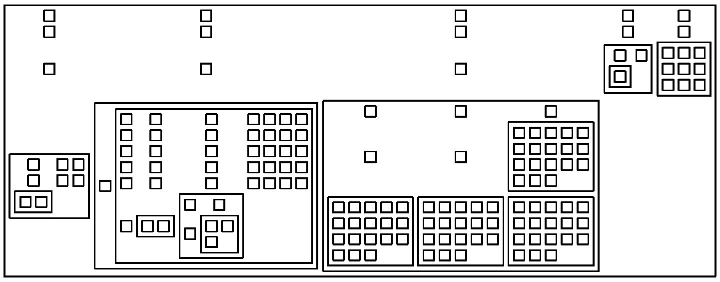
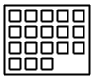


Figure 5.1 : Unit structure of DrScheme version 53

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.

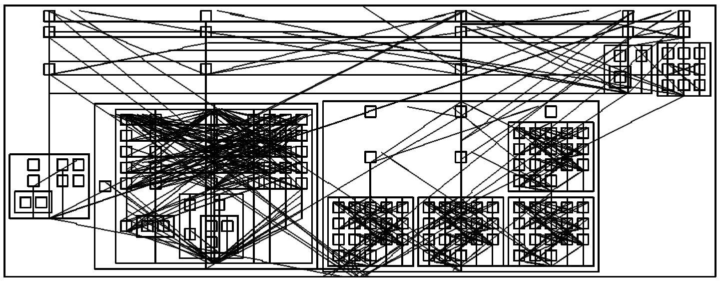


Figure 5.2 : Unit structure of DrScheme with linking

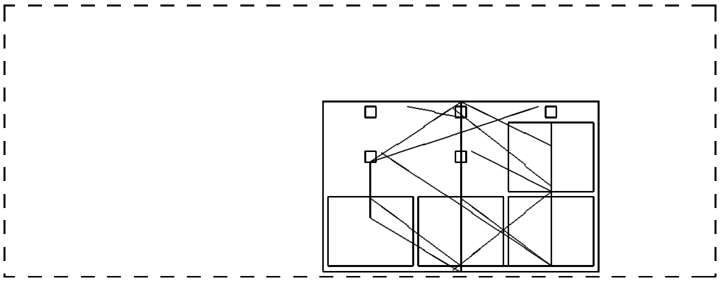


Figure 5.3 : Local unit structure in DrScheme

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 uncompilable 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 pratice, 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:[1]

(**class*** *superclass-expr* (*interface-expr* $\cdots$) (*init-variable* $\cdots$)
instance-variable-clause $\cdots$)

The expression *superclass-expr* determines the superclass for the new class, and the *interface-expr*s specify the interfaces implemented by the class. The *init-variable*s receive instance-specific initialization values when the class is instantiated (like the arguments supplied with **new** in Java). Finally, the *instance-variable-clause*s 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

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

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*

[1] Centered ellipses indicate repeated patterns.

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* $\cdots$)
  variable $\cdots$)

[2]MzScheme's approach to methods avoids duplicating the functionality of procedures with methods. This orthogonal design, however, incurs a substantial cost in practice because each object record must provide a slot for every method in the class, and a closure is created for each method per object. Adding true methods to the object system, like methods in most object-oriented languages, would improve the run-time performance of the object system and would not affect the essence of our presentation.

The *superinterface-expr*s specify all of the superinterfaces for the new interface, and the *variable*s 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* $\cdots$])
    (**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

[3] Since 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_{c} Proofs

B.1 Proof of Subject Reduction

Lemma 3.5.2 (Subject Reduction) *If* $\mathcal{T} \cdot e \longmapsto \mathcal{T}' \cdot e'$ *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 $\mathcal{T} \cdot e \longmapsto \mathcal{T}' \cdot e'$.

Case $e = e_1 \,;\, e_2$.

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

Case $e = v \,;\, e_2$.

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

Case $e = e_1 \,;\, e_2$ where $\mathcal{T} \cdot e_1 \longmapsto \mathcal{T}' \cdot e_1'$, so $e' = e_1' \,;\, e_2$.

By induction, $|\mathcal{T}'| \vdash e_1' : \tau'$ for some τ'. By Lemma B.3.7 (Store Growth) and Lemma B.3.4 (Environment Extension), $|\mathcal{T}'| \vdash e_2 : \tau_0$. Therefore, by $\mathsf{seq}_{\mathsf{c}}^{\vdash}$ again, $|\mathcal{T}'| \vdash e' : \tau_0$.

Case $e = e_1$ **as** τ.

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

Case $e = v$ **as** τ.

By $\mathsf{generalize}_{\mathsf{c}}^{\rightarrow}$, $e' = v$ and $\mathcal{T}' = \mathcal{T}$. Thus, $|\mathcal{T}'| \vdash e' : \tau_0'$.

Case $e = e_1$ **as** τ where $\mathcal{T} \cdot e_1 \longmapsto \mathcal{T}' \cdot e_1'$, so $e' = e_1'$ **as** τ.

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 $\mathsf{generalize}_{\mathsf{c}}^{\vdash}$ again, $|\mathcal{T}'| \vdash e' : \tau_0$.

Case $e = e_1\ e_2$.

By $\mathsf{app}_{\mathsf{c}}^{\vdash}$, $|\mathcal{T}| \vdash e_1 : \tau \to \tau_0$ and $|\mathcal{T}| \vdash e_2 : \tau_2$ for some τ_2 where $\tau_2 \leq \tau$.

There are two possibilities for e_1:

Case $e_1 = v_1$.

We must consider the two cases for e_2:

Case $e_2 = v_2$.

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 $v_1 = \mathsf{projl}\langle t\rangle$ where $\mathcal{T}(t) = \langle \tau_{\mathsf{l}}, \tau_{\mathsf{r}}\rangle$, so $\tau = t$ and $\tau_0 = \tau_{\mathsf{l}}$.

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 $v_2 = \mathsf{injl}\langle t\rangle v_3$ where $|\mathcal{T}| \vdash v_3 : \tau_0'$ and $\tau_0' \leq \tau_0$.

By $\mathsf{projl}_{\mathsf{c}}^{\rightarrow}$, $\mathcal{T}' = \mathcal{T}$ and $e' = v_3$, so $|\mathcal{T}'| \vdash e' : \tau_0'$.

Case $v_2 = \mathsf{injr}\langle t\rangle v_3$.

By $\mathsf{projl\text{-}fail}_{\mathsf{c}}^{\rightarrow}$, $\mathcal{T} \cdot e \longmapsto \mathcal{T} \cdot \mathsf{variant\ error}$; there is no e' such that $\mathcal{T} \cdot e \longmapsto \mathcal{T}' \cdot e'$.

Case $v_1 = \mathsf{projr}\langle t\rangle$ where $\mathcal{T}(t) = \langle \tau_{\mathsf{l}}, \tau_{\mathsf{r}}\rangle$, so $\tau = t$ and $\tau_0 = \tau_{\mathsf{r}}$.

Analogous to previous case.

Case $v_1 = \mathsf{test}\langle t\rangle$ where $\mathcal{T}(t) = \langle \tau_{\mathsf{l}}, \tau_{\mathsf{r}}\rangle$, so $\tau = t$ and $\tau_0 = \mathsf{bool}$.

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 $v_2 = \mathsf{injl}\langle t\rangle v_3$ where $|\mathcal{T}| \vdash v_3 : \tau_{\mathsf{l}}$.

By $\mathsf{test\text{-}true}_{\mathsf{c}}^{\rightarrow}$, $\mathcal{T}' = \mathcal{T}$ and $e' = \mathsf{true}$, which has type bool.

Case $v_2 = \mathsf{injr}\langle t\rangle v_3$ where $|\mathcal{T}| \vdash v_3 : \tau_{\mathsf{l}}$.

By $\mathsf{test\text{-}false}_{\mathsf{c}}^{\rightarrow}$, $\mathcal{T}' = \mathcal{T}$ and $e' = \mathsf{true}$, which has type bool.

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

By $\mathsf{app}_{\mathsf{c}}^{\rightarrow}$, $\mathcal{T}' = \mathcal{T}$ and $e' = [v_2/x]e_0$. By Lemma B.3.3 (Substitution), $|\mathcal{T}| \vdash [v_2/x]e_0 : \tau_0'$ where $\tau_0' \leq \tau_0$.

Case $\mathcal{T} \cdot e_2 \longmapsto \mathcal{T}' \cdot e_2'$ and $e' = v_1\ e_2'$.

By induction, $|\mathcal{T}'| \vdash e_2' : \tau_2'$ where $\tau_2' \leq \tau_2$. By Lemma B.3.7 (Store Growth) and Lemma B.3.4 (Environment Extension), $|\mathcal{T}'| \vdash v_1 : \tau \rightarrow \tau_0$. Since $\tau_2' \leq \tau_2$, $\tau_2' \leq \tau$ by the transitivity of $\leq$. Thus, by $\mathsf{app}_\mathsf{c}^{\vdash}$, $|\mathcal{T}'| \vdash e' : \tau_0$.

Case $\mathcal{T} \cdot e_1 \longmapsto \mathcal{T}' \cdot e_1'$ and $e' = e_1'\ e_2$.

By induction, $|\mathcal{T}'| \vdash e_1' : \tau_1'$ 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 τ_1' is a subtype of $\tau \rightarrow \tau_0$, τ_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 $\mathsf{app}_\mathsf{c}^{\vdash}$, $|\mathcal{T}'| \vdash e' : \tau_0'$ (where $\tau_0' \leq \tau_0$).

Case $\mathrm{e} = \textbf{letrec}\ \overline{\textbf{type}\ t = x_{\mathsf{cl}},\ x_{\mathsf{dl}}\ \tau_{\mathsf{l}} \mid x_{\mathsf{cr}},\ x_{\mathsf{dr}}\ \tau_{\mathsf{r}} \diamond x_{\mathsf{t}}}\ .$
$\overline{\textbf{val}\ x_v : \tau = v}$
$\textbf{in}\ e_b$

The $\mathsf{letrec}_\mathsf{c}^{\vdash}$ rule ensures that $\overline{t} \cap \mathrm{dom}(\mathcal{T}) = \emptyset$. By $\mathsf{letrec\text{-}types}_\mathsf{c}^{\rightarrow}$, $\mathcal{T}' = \mathcal{T}[\overline{t \mapsto \langle \tau_{\mathsf{l}}, \tau_{\mathsf{r}} \rangle}]$ and $e' = S(\textbf{letrec}\ \overline{\textbf{val}\ x_v : \tau = v}\ \textbf{in}\ e_b)$ where S is a replacement on $\overline{x_{\mathsf{cl}}}$, $\overline{x_{\mathsf{dl}}}$, $\overline{x_{\mathsf{cr}}}$, $\overline{x_{\mathsf{dr}}}$, and $\overline{x_{\mathsf{t}}}$.

Let $\Gamma = |\mathcal{T}|$. As in $\mathsf{letrec}_\mathsf{c}^{\vdash}$, let $\Gamma' = \Gamma[\overline{t{::}\Omega}]$ and let Γ'' be the extension of Γ' with types for $\overline{x_{\mathsf{cl}}}, \cdots \overline{x_{\mathsf{t}}}$. Since $\Gamma \vdash e : \tau_0$, by $\mathsf{letrec}_\mathsf{c}^{\vdash}$ we have $\Gamma'' \vdash_\mathsf{s} \overline{v} : \overline{\tau}$ 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 $\overline{t}$, which means that $|\mathcal{T}'|$ extends $|\mathcal{T}|$ with type bindings $\overline{t{::}\Omega}$ and $\overline{\mathsf{injl}\langle t \rangle : \tau_{\mathsf{l}} \rightarrow t}, \cdots \overline{\mathsf{test}\langle t \rangle : t \rightarrow \mathsf{bool}}$.
- By construction, Γ'' adds $\overline{t{::}\Omega}$ and $\overline{x_{\mathsf{cl}} : \tau_{\mathsf{l}} \rightarrow t}, \cdots \overline{x_{\mathsf{t}} : t \rightarrow \mathsf{bool}}$ to Γ.
- S replaces the variables $\overline{x_{\mathsf{cl}}}, \cdots, \overline{x_{\mathsf{t}}}$ with the variables $\mathsf{injl}\langle t \rangle, \cdots \mathsf{test}\langle t \rangle$.

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

Case $\mathrm{e} = \textbf{letrec}\ \overline{\textbf{val}\ x_v : \tau = v}\ \textbf{in}\ e_b$.

By $\mathsf{letrec}_\mathsf{c}^{\rightarrow}$, $\mathcal{T}' = \mathcal{T}$ and $e' = [\overline{\textbf{letrec}\ \overline{\textbf{val}\ x_v : \tau = v}\ \textbf{in}\ v/x_v}]e_b$. The derivation for $|\mathcal{T}| \vdash \textbf{letrec}\ \overline{\textbf{val}\ x_v : \tau = v}\ \textbf{in}\ e_b : \tau_0$ proves in intermediate steps that $|\mathcal{T}|[\overline{x_v{:}\tau}] \vdash_{\mathsf{s}} \overline{v} : \overline{\tau}$. Then, using $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$, we can synthesize the judgement $|\mathcal{T}| \vdash_{\mathsf{s}} \textbf{letrec}\ \overline{\textbf{val}\ x_v : \tau\ =\ v}\ \textbf{in}\ v : \overline{\tau}$. Thus, according to Lemma B.3.3 (Substitution), $|\mathcal{T}'| \vdash e' : \tau_0'$ where $\tau_0' \le \tau_0$.

Case e = $\textbf{invoke}\ e_u\ \textbf{with}\ \overline{s\ =\ t{::}\Omega\ \leftarrow\ \sigma}\ \overline{y{:}\tau\ \leftarrow\ e_i}$.

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

By $\mathsf{invoke}^{\vdash}_{\mathsf{c}}$, $|\mathcal{T}| \vdash e_u : \tau_u$, $|\mathcal{T}| \vdash \overline{e_i} : \overline{\tau}$, and τ_u is a signature such that $\tau_u \le \textbf{sig import}\ \overline{s = t{::}\Omega}\ \overline{y{:}\tau}\ \textbf{export}\ \emptyset\ \triangleright\ \tau_b$ where $[\overline{\sigma/t}]\tau_b = \tau_0$.

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

$$
\begin{array}{l}
\textbf{unit import}\ \overline{s_i\ =\ t_i{::}\Omega}\ \overline{y_i\ =\ x_i{:}\tau_i} \\
\quad\ \textbf{export}\ \overline{s_e\ =\ t_e{::}\Omega}\ \overline{y_e\ =\ x_e{:}\tau_e} \\
\quad\ \triangleright\ \tau_b \\
\quad\ \overline{\textbf{type}\ t_d\ =\ x_{\mathsf{cl}},\ x_{\mathsf{dl}}\ \tau_{\mathsf{l}}\ \mid\ x_{\mathsf{cr}},\ x_{\mathsf{dr}}\ \tau_{\mathsf{r}}\ \diamond\ x_{\mathsf{t}}} \\
\quad\ \overline{\textbf{val}\ x_v : \tau_v\ =\ v_v} \\
\quad\ e_b
\end{array}
$$

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

By $\mathsf{invoke}^{\rightarrow}_{\mathsf{c}}$, $\mathcal{T} \cdot e \longmapsto \mathcal{T} \cdot [\overline{\sigma/t}, \overline{v/x}]e_l$ where

$$
\begin{array}{lll}
e_l & = & \textbf{letrec}\ \overline{\textbf{type}\ t_d\ =\ x_{\mathsf{cl}},\ x_{\mathsf{dl}}\ \tau_{\mathsf{l}}\ \mid\ x_{\mathsf{cr}},\ x_{\mathsf{dr}}\ \tau_{\mathsf{r}}\ \diamond\ x_{\mathsf{t}}} \\
& & \qquad\quad \overline{\textbf{val}\ x_v : \tau_v\ =\ v_v} \\
& & \qquad\quad \textbf{in}\ e_b\ \textbf{as}\ \tau_b.
\end{array}
$$

The expression e_l can be typed in an environment $|\mathcal{T}|[\overline{t_i{:}\Omega}, \overline{x_i{:}\tau_i}]$ because v_u types in $|\mathcal{T}|$. To verify this, we check each antecedent in the $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$ rule applied to e_l with $|\mathcal{T}|[\overline{t_i{:}\Omega}, \overline{x_i{:}\tau_i}]$ and show that it corresponds to an antecedent in the $\mathsf{unit}^{\vdash}_{\mathsf{c}}$ rule applied to v_u with $|\mathcal{T}|$ (which is already proved, by assumption):

- Distinct variable names, and not in dom(Γ): The $\mathsf{unit}^{\vdash}_{\mathsf{c}}$ antecedents include all of the names in the $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$ antecedents.
- $\overline{\tau_{\mathsf{l}}}$, $\overline{\tau_{\mathsf{r}}}$, and $\overline{\tau_v}$ validations: The $\mathsf{unit}^{\vdash}_{\mathsf{c}}$ definition of Γ' adds both $\overline{t_i}$ and $\overline{t_d}$ to Γ, while the $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$ definition of Γ' adds only $\overline{t_d}$. But the $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$ rule is applied with $|\mathcal{T}|[\overline{t_i{:}\Omega}, \overline{x_i{:}\tau_i}]$ instead of $|\mathcal{T}|$, so the final Γ' in $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$ and $\mathsf{unit}^{\vdash}_{\mathsf{c}}$ have the same type variable bindings. The extra value variable bindings

$[\overline{x_i{:}\tau_i}]$ in Γ' for $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$ cannot affect the validation of type expressions by Lemma B.3.4 (Environment Extension).

- $\overline{v_v}$ typings: As above, Γ'' in both $\mathsf{unit}^{\vdash}_{\mathsf{c}}$ and $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$ contain the same bindings, because $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$ is applied with $|\mathcal{T}|[\overline{t_i{:}\Omega}, \overline{x_i{:}\tau_i}]$. Thus, the antecedents in $\mathsf{unit}^{\vdash}_{\mathsf{c}}$ imply the ones in $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$.
- e_b **as** τ_b typing: In $\mathsf{unit}^{\vdash}_{\mathsf{c}}$, $\Gamma'' \vdash e_b : \tau_b'$ for some τ_b' where $\tau_b' \leq \tau_b$. Since Γ'' is the same in $\mathsf{unit}^{\vdash}_{\mathsf{c}}$ and $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$, $\Gamma'' \vdash e_b$ **as** $\tau_b : \tau_b$ in $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$.
- $FTV(\tau_b) \cap \overline{t_d} = \emptyset$ check: The $FTV(\tau_b) \cap \overline{t_d} = \emptyset$ check in $\mathsf{unit}^{\vdash}_{\mathsf{c}}$ implies the equivalent check in $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$.

Thus, $|\mathcal{T}|[\overline{t_i{:}\Omega}, \overline{x_i{:}\tau_i}] \vdash e_l : \tau_b$. By Lemma B.3.5 (Stable Typing), we have $[\overline{\sigma/t_i}](|\mathcal{T}|[\overline{x_i{:}\tau_i}]) \vdash [\overline{\sigma/t_i}]e_l : [\overline{\sigma/t_i}]\tau_b$. Then, using Lemma B.3.3 (Substitution), $[\overline{\sigma/t_i}](|\mathcal{T}|) \vdash [\overline{\sigma/t_i}, \overline{v/x_i}]e_l : [\overline{\sigma/t_i}]\tau_b$, which is equivalent to the judgement $[\overline{\sigma/t_i}](|\mathcal{T}|) \vdash e' : \tau_0'$ where $\tau_0' = [\overline{\sigma/t}]\tau_b$. Since $\mathsf{unit}^{\vdash}_{\mathsf{c}}$ ensures $\overline{t_i} \cap \mathrm{dom}(|\mathcal{T}|) = \emptyset$, $|\mathcal{T}| \vdash e' : \tau_0'$.

Finally, since $\tau_0 = [\overline{\sigma/t}]\tau_b = \tau_0'$, $\mathcal{T} \cdot e \longmapsto \mathcal{T}' \cdot e'$ for $\mathcal{T}' = \mathcal{T}$ and $|\mathcal{T}'| \vdash e' : \tau_0'$ where $\tau_0' \leq \tau_0$.

To complete the **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 is a value for each i.

Covered above.

Case $\mathcal{T} \cdot e_i \longmapsto \mathcal{T}' \cdot e_i'$ 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 $\mathcal{T} \cdot e_u \longmapsto \mathcal{T}' \cdot e_u'$ 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 $e =$ **compound** .

$$
\begin{array}{l}
\textbf{import}\ \overline{s_i\ =\ t_i{::}\Omega}\ \overline{y_i{::}\tau_i} \\
\textbf{export}\ \overline{s_e\ =\ t_e{::}\Omega}\ \overline{y_e{::}\tau_e} \\
\rhd\ \tau_b \\
\textbf{link}\ e_1\ \textbf{with}\ \overline{s_{w1} = t_{w1}{::}\Omega}\ \overline{y_{w1}{::}\tau_{w1}} \\
\qquad\quad \textbf{provides}\ \overline{s_{p1} = t_{p1}{::}\Omega}\ \overline{y_{p1}{::}\tau_{p1}} \\
\textbf{and}\ e_2\ \textbf{with}\ \overline{s_{w2} = t_{w2}{::}\Omega}\ \overline{y_{w2}{::}\tau_{w2}} \\
\qquad\quad \textbf{provides}\ \overline{s_{p2} = t_{p2}{::}\Omega}\ \overline{y_{p2}{::}\tau_{p2}}
\end{array}
$$

The interesting case is where $e_1 = v_1$ and $e_2 = v_2$ and $\mathsf{compound}_{\mathsf{c}}^{\rightarrow}$ applies, so we consider that case first.

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

$$
\begin{array}{l}
\textbf{unit import}\ \overline{s_{i1} = t_{i1}{::}\Omega}\ \overline{y_{i1} = x_{i1}{:}\tau_{i1}} \\
\qquad \textbf{export}\ \overline{s_{e1} = t_{e1}{::}\Omega}\ \overline{y_{e1} = x_{e1}{:}\tau_{e1}} \\
\qquad \rhd\ \tau_{b1} \\
\qquad \overline{\textbf{type}\ t_{d1}\ =\ x_{\mathsf{cl}1},\ x_{\mathsf{dl}1}\ \tau_{\mathsf{l}1}\ \mid\ x_{\mathsf{cr}1},\ x_{\mathsf{dr}1}\ \tau_{\mathsf{r}1}\ \diamond\ x_{\mathsf{t}1}} \\
\qquad \overline{\textbf{val}\ x_{v1}\ :\ \tau_{v1}\ =\ v_{v1}} \\
\qquad e_{b1}
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{unit import}\ \overline{s_i = t_i{::}\Omega}\ \overline{y_i = x_i{:}\tau_i} \\
\qquad \textbf{export}\ \overline{s_e = t_e{::}\Omega}\ \overline{y_e = x_e{:}\tau_e} \\
\qquad \rhd\ \tau_b \\
\qquad \overline{\textbf{type}\ t_d\ =\ x_{\mathsf{cl}},\ x_{\mathsf{dl}}\ \tau_{\mathsf{l}}\ \mid\ x_{\mathsf{cr}},\ x_{\mathsf{dr}}\ \tau_{\mathsf{r}}\ \diamond\ x_{\mathsf{t}}} \\
\qquad \overline{\textbf{val}\ x_v\ :\ \tau_v\ =\ v_v} \\
\qquad e_b
\end{array}
$$

We show that the new unit has the signature type τ_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 $\mathsf{compund}_{\mathsf{c}}^{\rightarrow}$ 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 **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 $\mathsf{compound}_\mathsf{c}^{\vdash}$ (for propagating exports) and $\mathsf{unit}_\mathsf{c}^{\vdash}$ (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 $\overline{\tau_{\mathsf{l}}}, \cdots$ are valid: The $\overline{\tau_{\mathsf{l}}}, \cdots$ expressions in v_3 are the combined type expressions $\overline{\tau_{\mathsf{l}1}}, \cdots$ and $\overline{\tau_{\mathsf{l}2}}, \cdots$. For v_1, the type expressions $\overline{\tau_{\mathsf{l}1}}, \cdots$ are validated in an environment $\Gamma'_1 = \Gamma[\overline{t_{i1}{::}\Omega}, \overline{t_{d1}{::}\Omega}]$. For v_3, the same type expressions must be validated in the environment $\Gamma' = \Gamma[\overline{t_i{::}\Omega}, \overline{t_{d1}{::}\Omega}, \overline{t_{d2}{::}\Omega}]$. But Γ' includes all of the type bindings of Γ'_1, since $\overline{t_{i1}}$ must be a subset of $\overline{t_i} \cup \overline{t_{d2}}]$ by $\mathsf{compound}_\mathsf{c}^{\rightarrow}$. Although Γ' may contain additional type variables, they cannot affect the validation of $\overline{\tau_{\mathsf{l}1}}, \cdots$ by Lemma B.3.4 (Environment Extension). By a similar argument, the type expressions $\overline{\tau_{\mathsf{l}2}}, \cdots$ can be validated in v_3.

- Expression types, $\Gamma'' \vdash_{\mathsf{s}} \overline{v_v} : \overline{\tau_v}$: The $\overline{v_v}$ expressions in v_3 are the combined expressions $\overline{v_{v1}}$ and $\overline{v_{v2}}$. The argument for typing $\overline{v_v}$ in v_3 is similar to the argument above for validating the type expressions, but the environment Γ'' is not merely a superset of Γ''_1 or Γ''_2. Γ'' contains at least as many variable bindings as Γ''_1, but for each variable in Γ''_1, its type binding in Γ'' is a subtype of the one in Γ''_1. The subtyping is possible according to the $\mathsf{compound}_\mathsf{c}^{\vdash}$ rule, which allows a subtype relationship between the types $\overline{\tau_{i1}}$ expected by v_1 and the types $\overline{\tau_{e2}}$ provided by v_2 (or the types $\overline{\tau_i}$ imported by e). As a result, by Lemma B.3.2 (Environment Replacement), if $\Gamma''_1 \vdash \overline{v_{v1}} : \overline{\tau'_{v1}}$ for v_1, $\Gamma'' \vdash \overline{v_{v1}} : \overline{\tau'_v}$ in v_3 where $\tau'_v \leq \tau'_{v1}$. This subtyping relationship is sufficient for validating v_3, which requires the subsumption relation $\Gamma'' \vdash_{\mathsf{s}} \overline{v_v} : \overline{\tau_v}$.

- Initialization expression type, $\Gamma'' \vdash_{\mathsf{s}} e_b : \tau_b$: By $\mathsf{compound}_\mathsf{c}^{\rightarrow}$, $e_b = e_{b1} \,;\, e_{b2}$. The type of e_b is thus the type of e_{b2} in the environment Γ''. By the same argument as for the $\overline{v_v}$ expressions, if e_{b2} has type τ'_{b2} in Γ''_2, it has some type τ'_b in Γ'' such that $\tau'_b \leq \tau'_{b2}$. By assumption, $\tau'_{b2} \leq \tau_{b2}$, and by $\mathsf{compound}_\mathsf{c}^{\vdash}$ $\tau_{b2} \leq \tau_b$, so $\tau'_b \leq \tau_b$ by transitivity.

- $FTV(\tau_b) \cap \overline{t_d} = \emptyset$ check: By $\mathsf{compound}_{\mathsf{c}}^{\vdash}$, τ_b must be well-formed in Γ extended with $\overline{t_i}$, so $FTV(\tau_b) \subseteq (\mathrm{dom}(\Gamma) \cup \overline{t_i})$. Type-checking for v_1 and v_2 ensures that $(\overline{t_{d1}} \cup \overline{t_{d2}}) \cap \mathrm{dom}(\mathcal{T}) = \emptyset$, and $\mathsf{compound}_{\mathsf{c}}^{\rightarrow}$ applies only when $(\overline{t_{d1}} \cup \overline{t_{d2}}) \cap \overline{t_i} = \emptyset$. Since $\overline{t_d} = \overline{t_{d1}} \cup \overline{t_{d2}}$, we have $FTV(\tau_b) \cap \overline{t_d} = \emptyset$.

Thus, all of the antecedents hold; v_3 must have the signature τ_0 because it has the same imports, exports, and declared initialization type as e.

To complete the **compound** case for e, we must consider the remaining possibilities for e_1:

Case $e_1 = v_1$.

Case $e_2 = v_2$.

Covered above.

Case $\mathcal{T} \cdot e_2 \longmapsto \mathcal{T}' \cdot e_2'$.

By induction, $|\mathcal{T}'| \vdash e_2' : \tau_2'$ where $\tau_2' \leq \tau_2$. Substituting e_2' for e_2 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 $\mathcal{T} \cdot e_1 \longmapsto \mathcal{T}' \cdot e_1'$.

By induction, $|\mathcal{T}'| \vdash e_1' : \tau_1'$ where $\tau_1' \leq \tau_1$. Substituting e_1' for e_1 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 $\tau_0 \leq \tau_0$. $\square$

B.2 Proof of Progress

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

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

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

3. $\mathcal{T} \cdot e \longmapsto \mathcal{T}' \cdot e'$ *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.

Case $e = e_1 ; e_2$.

By induction, there are three possibilities for e_1:

Case $e_1 = v$ for some v.

By $\mathsf{seq}_{\mathsf{c}}^{\rightarrow}$, $\mathcal{T}' = \mathcal{T}$ and $e' = e_2$.

Case $\mathcal{T} \cdot e_1 \longmapsto \mathcal{T} \cdot \mathsf{variant\ error}$.

By $\mathsf{variant}_{\mathsf{c}}^{\rightarrow}$, a variant error replaces the entire context of the erroneous subexpression. Thus, $\mathcal{T} \cdot e \longmapsto \mathcal{T} \cdot \mathsf{variant\ error}$.

Case $\mathcal{T} \cdot e_1 \longmapsto \mathcal{T}' \cdot e_1'$.

$e' = e_1' ; e_2$.

Case $e = e_1$ **as** τ.

By induction, there are three possibilities for e_1:

Case $e_1 = v$ for some v.

By $\mathsf{generalize}_{\mathsf{c}}^{\rightarrow}$, $\mathcal{T}' = \mathcal{T}$ and $e' = e_1$.

Case $\mathcal{T} \cdot e_1 \longmapsto \mathcal{T} \cdot \mathsf{variant\ error}$.

By $\mathsf{variant}_{\mathsf{c}}^{\rightarrow}$, $\mathcal{T} \cdot e \longmapsto \mathcal{T} \cdot \mathsf{variant\ error}$.

Case $\mathcal{T} \cdot e_1 \longmapsto \mathcal{T}' \cdot e_1'$.

$e' = e_1'$ **as** τ.

Case $e = e_1\ e_2$.

By $\mathsf{app}_{\mathsf{c}}^{\vdash}$, $|\mathcal{T}| \vdash e_1 : \tau \rightarrow \tau_0$ and $|\mathcal{T}| \vdash e_2 : \tau_2$ for some τ_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 $v_1 = \mathsf{projl}\langle\tau_0, \tau_\mathsf{r}\rangle$ and $v_2 = \mathsf{injr}\langle\tau_0, \tau_\mathsf{r}\rangle v_3$, $\mathcal{T} \cdot e \longmapsto \mathcal{T} \cdot \mathsf{variant\ error}$, and for other cases, $\mathcal{T} \cdot e \longmapsto \mathcal{T}' \cdot e'$ for some $\mathcal{T}'$ and e'.

Case $\mathcal{T} \cdot e_2 \longmapsto \mathcal{T} \cdot \mathsf{variant\ error}$.

By $\mathsf{variant}_\mathsf{c}^{\rightarrow}$, $\mathcal{T} \cdot e \longmapsto \mathcal{T} \cdot \mathsf{variant\ error}$.

Case $\mathcal{T} \cdot e_2 \longmapsto \mathcal{T}' \cdot e_2'$.

$e' = v_1\ e_2'$.

Case $\mathcal{T} \cdot e_1 \longmapsto \mathcal{T} \cdot \mathsf{variant\ error}$.

By $\mathsf{variant}_\mathsf{c}^{\rightarrow}$, $\mathcal{T} \cdot e \longmapsto \mathcal{T} \cdot \mathsf{variant\ error}$.

Case $\mathcal{T} \cdot e_1 \longmapsto \mathcal{T}' \cdot e_1'$.

$e' = e_1'\ e_2$.

Case e = **letrec** $\overline{\textbf{type}\ t = x_\mathsf{cl},\ x_\mathsf{dl}\ \tau_\mathsf{l} \mid x_\mathsf{cr},\ x_\mathsf{dr}\ \tau_\mathsf{r} \diamond x_\mathsf{t}}$.
$\overline{\textbf{val}\ x_v : \tau = v_v}$
in e_b

By $\mathsf{letrec}_\mathsf{c}^{\vdash}$, $\overline{t} \cap \mathrm{dom}(|\mathcal{T}|) = \emptyset$. Since $\mathrm{dom}(|\mathcal{T}|)$ and $\mathrm{dom}(\mathcal{T})$ contain the same types, $\overline{t} \cap \mathrm{dom}(\mathcal{T}) = \emptyset$. Thus, $\mathsf{letrec\text{-}types}_\mathsf{c}^{\rightarrow}$ applies, and $\mathcal{T} \cdot e \longmapsto \mathcal{T}' \cdot S(\textbf{letrec}\ \overline{\textbf{val}\ x_v : \tau = v_v}\ \textbf{in}\ e_b)$.

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

By $\mathsf{letrec}_\mathsf{c}^{\rightarrow}$, $\mathcal{T} \cdot e \longmapsto \mathcal{T} \cdot [\overline{\textbf{letrec}\ \overline{\textbf{val}\ x_v : \tau = v_v}\ \textbf{in}\ v_v / x_v}]e_b$.

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

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

unit import $\overline{s_i = t_i{::}\Omega}\ \overline{y_i = x_i{:}\tau_i}$
export $\overline{s_e = t_e{::}\Omega}\ \overline{y_e = x_e{:}\tau_e}$
$\triangleright\ \tau_b$
$\overline{\textbf{type}\ t_d = x_\mathsf{cl},\ x_\mathsf{dl}\ \tau_\mathsf{l} \mid x_\mathsf{cr},\ x_\mathsf{dr}\ \tau_\mathsf{r} \diamond x_\mathsf{t}}$
$\overline{\textbf{val}\ x_v : \tau_v = v_v}$
e_b

where $\overline{s_i = t_i} \subseteq \overline{s = t}$, and $\overline{y_i} \subseteq \overline{y}$. We can choose $\overline{x}$ so that $\overline{y_i = x_i} \subseteq \overline{y = x}$. Thus, $\mathsf{invoke}_{\mathsf{c}}^{\rightarrow}$ defines $\mathcal{T}'$ and e' when $e_u = v_u$ and $\overline{e_i} = \overline{v_i}$.

To complete the **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 $\mathcal{T} \cdot e_i \longmapsto \mathcal{T} \cdot \mathsf{variant\ error}$ for the first e_i that is not a v.

By $\mathsf{variant}_{\mathsf{c}}^{\rightarrow}$, $\mathcal{T} \cdot e \longmapsto \mathcal{T} \cdot \mathsf{variant\ error}$.

Case $\mathcal{T} \cdot e_i \longmapsto \mathcal{T}' \cdot e'_i$.

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

Case $\mathcal{T} \cdot e_u \longmapsto \mathcal{T} \cdot \mathsf{variant\ error}$.

By $\mathsf{variant}_{\mathsf{c}}^{\rightarrow}$, $\mathcal{T} \cdot e \longmapsto \mathcal{T} \cdot \mathsf{variant\ error}$.

Case $\mathcal{T}' \cdot e_u \longmapsto \mathcal{T}' \cdot e'_u$ 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 $\mathcal{T} \cdot e \longmapsto \mathcal{T}' \cdot e'$.

Case e = **compound**

$$\begin{array}{l}
\textbf{import}\ \overline{s_i = t_i{::}\Omega}\ \overline{y_i = x_i{::}\tau_i} \\
\textbf{export}\ \overline{s_e = t_e{::}\Omega}\ \overline{y_e = x_e{::}\tau_e} \\
\triangleright\ \tau_b \\
\textbf{link}\ e_1\ \textbf{with}\ \overline{t_{w1}{::}\Omega}\ \overline{x_{w1}{::}\tau_{w1}} \\
\qquad\qquad \textbf{provides}\ \overline{t_{p1}{::}\Omega}\ \overline{x_{p1}{::}\tau_{p1}} \\
\textbf{and}\ e_2\ \textbf{with}\ \overline{t_{w2}{::}\Omega}\ \overline{x_{w2}{::}\tau_{w2}} \\
\qquad\qquad \textbf{provides}\ \overline{t_{p2}{::}\Omega}\ \overline{x_{p2}{::}\tau_{p2}}
\end{array}.$$

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

To complete the **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 $\mathcal{T} \cdot e_2 \longmapsto \mathcal{T} \cdot \textsf{variant error}$.

By $\textsf{variant}_{\textsf{c}}^{\rightarrow}$, $\mathcal{T} \cdot e \longmapsto \mathcal{T} \cdot \textsf{variant error}$.

Case $\mathcal{T} \cdot e_2 \longmapsto \mathcal{T}' \cdot e_2'$ 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 $\mathcal{T}, e \longmapsto \mathcal{T}' \cdot e'$.

Case $\mathcal{T} \cdot e_1 \longmapsto \mathcal{T} \cdot \textsf{variant error}$.

By $\textsf{variant}_{\textsf{c}}^{\rightarrow}$, $\mathcal{T} \cdot e \longmapsto \mathcal{T} \cdot \textsf{variant error}$.

Case $\mathcal{T} \cdot e_1 \longmapsto \mathcal{T}' \cdot e_1'$ 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 $\mathcal{T} \cdot e \longmapsto \mathcal{T}' \cdot e'$. □

B.3 Supporting Lemmata

Lemma B.3.1 (Replacement) *If* $\Gamma \vdash C[e] : \tau$ *using* $\Gamma' \vdash e : \tau_e$, *and if* $\Gamma' \vdash e' : \tau_e'$ *where* $\tau_e' \leq \tau_e$, *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 Γ'' and τ_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 $C_2 = \textbf{fn}\ x : \tau_1 \Rightarrow [\,]$.

By $\textsf{lambda}_{\textsf{c}}^{\vdash}$, $\tau_0 = \tau_1 \rightarrow \tau_e$ and $\tau_0' = \tau_1 \rightarrow \tau_e'$. By the definition of $\leq$ on function types, $\tau_0' \leq \tau_0$.

Case $C_2 = [\,]\ e_2$.

By $\textsf{app}_{\textsf{c}}^{\vdash}$, $\Gamma'' \vdash e_2 : \tau_2$ and $\tau_e = \tau_2' \rightarrow \tau_0$ where $\tau_2 \leq \tau_2'$. Similarly, $\tau_e' = \tau_2'' \rightarrow \tau_0'$ for some τ_2'' and τ_0'. By assumption, $\tau_e' \leq \tau_e$, 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 $\textsf{app}_{\textsf{c}}^{\vdash}$. Furthermore, since $\tau_e' \leq \tau_e$, $\tau_0' \leq \tau_0$.

Case $C_2 = e_2\ [\,]$.

The $\mathsf{app}^{\vdash}_{\mathsf{c}}$ rule explicitly allows subsumption for the operand expression, so the type of $C_2[e]$ and $C_2[e']$ is the same in Γ''. In other words, $\tau'_0 = \tau_0$.

Case $C_2 = [\,]$ **as** τ.

The $\mathsf{generalize}^{\vdash}_{\mathsf{c}}$ rule explicitly allows subsumption the expression, so $\tau'_0 = \tau_0$.

Case $C_2 = [\,]\ ;\ e_2$.

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

Case $C_2 = e_1\ ;\ [\,]$.

The type of $C_2[e_0]$ is the type of e_0, so $\tau_0 = \tau_e$ and $\tau'_0 = \tau'_e$. By assumption, $\tau'_e \leq \tau_e$, so $\tau'_0 \leq \tau_0$.

Case $C_2 =$ **letrec** ... **val** $x_v : \tau_v = [\,]$ ... **in** e_b.

The $\mathsf{letrec}^{\vdash}_{\mathsf{c}}$ rule explicitly allows subsumption for **val** expressions, so $\tau'_0 = \tau_0$.

Case $C_2 =$ **letrec** ... **in** $[\,]$.

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

Case $C_2 =$ **invoke** $[\,]$ ....

By $\mathsf{invoke}^{\vdash}_{\mathsf{c}}$, τ_e must be a signature containing an initialization expression type τ_b. Since $\tau'_e \leq \tau_e$, τ'_e must also be a signature with fewer (or the same) imports and an initialization expression type τ'_b where $\tau'_b \leq \tau_b$. By $\mathsf{invoke}^{\vdash}_{\mathsf{c}}$, $\tau_0 = [\overline{\sigma/t}]\tau_b$ and $\tau'_0 = [\overline{\sigma/t}]\tau'_b$, and by Lemma B.3.6 (Type Substitution), $\tau'_0 \leq \tau_0$.

Case $C_2 =$ **invoke** e_u ... $y{:}\tau_i \leftarrow [\,]$ ....

The $\mathsf{invoke}^{\vdash}_{\mathsf{c}}$ rule explicitly allows subsumption for imported expressions, so $\tau'_0 = \tau_0$.

Case unit ... **val** $x_v : \tau_v = [\,]$ ... e_b.

The $\mathsf{unit}^{\vdash}_{\mathsf{c}}$ rule explicitly allows subsumption for **val** expressions, so $\tau'_0 = \tau_0$.

Case unit ... $[\,]$.

By $\mathsf{unit}^{\vdash}_{\mathsf{c}}$, τ_0 is a signature type containing an initialization expression type τ_e. Similarly, τ'_0 is a signature type with the same imports and exports, but the initialization expression type τ'_e. Since $\tau'_e \leq \tau_e$, $\tau'_0 \leq \tau_0$ by $\leq$ for signatures.

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

The $\mathsf{compound}^{\vdash}_{\mathsf{c}}$ 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 $\overline{x_n{:}\tau'}$, then replace $\overline{x}$ with $\overline{x_n}$. By Lemma B.3.1 (Replacement), $\Gamma[\overline{x{:}\tau}, \overline{x_n{:}\tau'}] \vdash [\overline{x_n/x}]e : \tau'_0$ where $\tau'_0 \leq \tau_0$. Because no x is free in $[\overline{x_n/x}]e$, $\Gamma[\overline{x_n{:}\tau'}] \vdash [\overline{x_n/x}]e : \tau'_0$. 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* $\Gamma \vdash \overline{v{:}\tau'}$ *where* $\overline{\tau'} \leq \overline{\tau}$, *then* $\Gamma \vdash [\overline{v/x}]e : \tau'_0$ *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 Γ'. 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* $\Gamma \vdash e : \tau_0$, $\Gamma \vdash \sigma :: \Omega$, *and* $(\overline{t} \cup \overline{x}) \cap \mathrm{dom}(\Gamma) = \emptyset$, *then* $\Gamma[\overline{t{::}\Omega}, \overline{x{::}\tau}] \vdash e : \tau_0$ *and* $\Gamma[\overline{t{::}\Omega}, \overline{x{::}\tau}] \vdash \sigma :: \Omega$.

Proof. Although some type rules depend on $\mathrm{dom}(\Gamma)$, the expression being typed may always be lexically renamed to avoid any conflicts with $\overline{t}$ or $\overline{x}$ without affecting the type of the expression:

- $\mathsf{letrec}_{\mathsf{c}}^{\vdash}$: Types $\overline{tb}$ defined within the **letrec** expression may be renamed without affecting the type, because $\mathsf{letrec}_{\mathsf{c}}^{\vdash}$ requires $FTV(\tau_b) \cap \overline{tb} = \emptyset$.

- $\mathsf{unit}_{\mathsf{c}}^{\vdash}$: Almost like $\mathsf{letrec}_{\mathsf{c}}^{\vdash}$, except that renaming an import variable $\overline{t_i}$ also implies a renaming in the unit's signature $\mathsf{sig}[i,\ e,\ b]$. However, the renaming is lexical within $\mathsf{sig}[i,\ e,\ b]$, making the renamed signature equivalent to the original one. $\square$

Lemma B.3.5 (Stable Typing) *If* $\Gamma[\overline{t{::}\Omega}] \vdash e : \tau_0$, $S = [\overline{\sigma/t}]$, *and* $\Gamma \vdash \overline{\sigma} :: \overline{\Omega}$, *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 σ. Each proof tree's leaf of the form $\Gamma' \vdash t :: \Omega$ is replaced with $\Gamma' \vdash \sigma$, which is provable by assumption (Γ' may have more type bindings than Γ, but the extra bindings cannot affect the validation of σ 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 $S(\Gamma) \vdash S(e) : S(\tau_0)$. $\square$

Lemma B.3.6 (Type Substitution) *If* $\tau \leq \tau'$ *and* $S = [\overline{\sigma/t}]$, *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 σ produces a valid proof of $S(\tau) \leq S(\tau')$, relying only on the equivalence of each σ to itself. $\square$

Lemma B.3.7 (Store Growth) *If* $\mathcal{T} \cdot e \longmapsto \mathcal{T}' \cdot e'$, *then* $\forall\ t \in \mathrm{dom}(\mathcal{T})$, $\mathcal{T}'(t) = \mathcal{T}(t)$ *and* $|\mathcal{T}'|(t) = |\mathcal{T}|(t)$.

Proof. The $\mathsf{letrec\text{-}types}_{\mathsf{c}}^{\rightarrow}$ reduction extends the type store, and no reduction contracts the store, so each reduction step must preserve or extend the type store. The environment $|\mathcal{T}|$ includes type and variable bindings for each type in $\mathcal{T}$, so extending the type store also extends the corresponding environment. $\square$

Lemma B.3.8 (Value Types) *For all* Γ *and* v,

1. *If* $\Gamma \vdash v : t$, *then* v *is either* $\mathsf{injl}\langle t\rangle v'$ *or* $\mathsf{injr}\langle t\rangle v'$ *for some* v'.

2. *If* $\Gamma \vdash v : \tau \rightarrow \tau'$, *then* v *is either* $\mathbf{fn}\ x{:}\tau \Rightarrow e$ *for some* e *or* $\mathsf{injl}\langle t\rangle$, $\mathsf{injr}\langle t\rangle$, $\mathsf{projl}\langle t\rangle$, $\mathsf{projr}\langle t\rangle$, *or* $\mathsf{test}\langle t\rangle$ *for some* t.

3. *If* $\Gamma \vdash v : \tau$ *and* τ *is a signature, then* v *is a* **unit** *expression.*

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

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* $P,\Gamma \vdash_{\underline{e}} e : t$*,* $P,\Gamma \vdash_{\sigma} \mathcal{S}$*,* SuperOk$(e)$*, and* $P \vdash \langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$*, then* e' *is an* error *configuration or there exists a* Γ' *such that*

1. $P,\Gamma' \vdash_{\underline{s}} e' : t$*,*

2. $P,\Gamma' \vdash_{\sigma} \mathcal{S}'$*, and*

3. 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 Γ' and show that the two consequents of the theorem are satisfied relative to the new expression, store, and environment.

Case $[new]$. Set $\Gamma' = \Gamma\ [object : c]$.

1. We have $P,\Gamma \vdash_{\underline{e}} \mathsf{E}[\mathbf{new}\ c] : t$. From $[new]$, $object \notin \mathrm{dom}(\mathcal{S})$. Then, by Σ_5, $object \notin \mathrm{dom}(\Gamma)$. Thus $P,\Gamma' \vdash_{\underline{e}} \mathsf{E}[\mathbf{new}\ c] : t$ by Lemma C.3.1. Since P,Γ' $\vdash_{\underline{e}} \mathbf{new}\ c : c$ and $P,\Gamma' \vdash_{\underline{e}} object : c$, Lemma C.3.2 implies $P,\Gamma' \vdash_{\underline{e}} \mathsf{E}[object]$ $: t$.

2. Let $\mathcal{S}'(object) = \langle c,\mathcal{F}\rangle$. $object$ is the only new element in $\mathrm{dom}(\mathcal{S}')$ and $\mathrm{dom}(\Gamma')$.

   Σ_1: $\Gamma'(object) = c$.

   Σ_2: $\mathrm{dom}(\mathcal{F})$ is correct by construction.

   Σ_3: $\mathrm{rng}(\mathcal{F}) = \{\mathsf{null}\}$.

   Σ_4: Since $\mathrm{rng}(\mathcal{F}) = \{\mathsf{null}\}$, this property is unaffected.

Σ_5 **and** Σ_6: The only change to Γ and $\mathcal{S}$ is *object*.

3. Since $\mathsf{E}[\mathit{object}]$ contains the same **super** expressions as $\mathsf{E}[\mathbf{new}\ c]$, and no instance of **this** or *object* is replaced in the new expression, $\textsc{SuperOk}(e')$ holds.

Case [*get*].

1. Let t' be such that $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathit{object}\underline{\ :\ c'\ }.\mathit{fd} : t'$. If v is null, it can be typed as t', so $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}[v] : t$ by Lemma C.3.2. If v is not null, then by Σ_4, $\mathcal{S}(v) = \langle c'', \_\rangle$ where $c'' \leq_P t'$. By Lemma C.3.4, $P,\Gamma' \vdash_{\underline{\mathsf{s}}} \mathsf{E}[v] : t$.
2. $\mathcal{S}$ and Γ are unchanged.
3. $\textsc{SuperOk}(e')$ holds because no **super** expression is changed.

Case [*set*].

1. The proof is by a straightforward extension of the proof for [*get*].
2. The only change to the store is a field update; thus only Σ_3 and Σ_4 are affected. Let v be the assigned value, and assume that v is not null.

   Σ_3: Since v is typable, it must be in dom(Γ). By Σ_5, it is therefore in dom($\mathcal{S}$).

   Σ_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 Σ_1 indicates that the type tag of v will preserve Σ_4.
3. $\textsc{SuperOk}(e')$ holds because no **super** expression is changed.

Case [*call*].

1. From $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathit{object}.\mathit{md}(v_1, \ldots\ v_n) : t$ we know $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathit{object} : t'$, $P,\Gamma \vdash_{\underline{\mathsf{s}}} v_i : t_i$ for i in $[1, n]$, and $\langle \mathit{md}, (t_1 \ldots\ t_n \rightarrow t), (\mathit{var}_1, ..., \mathit{var}_n), e_m\rangle \in^{\mathsf{c}}_P t'$. The type-checking of P proves that $P,t_0 \vdash_{\underline{\mathsf{m}}} t\ \mathit{md}\ (t_1\ \mathit{var}_1, \ldots\ t_n\ \mathit{var}_n)\ \{e_m\}$, which implies that $P,[\mathbf{this} : t_0,\ \mathit{var}_1 : t_1, \ldots\ \mathit{var}_n : t_n] \vdash_{\underline{\mathsf{s}}} e_m : t$ where t_0 is the defining class of *md*. Further, we know that $t' \leq_P t_0$ from $\in^{\mathsf{c}}_P$ for methods and $\textsc{ClassMethodsOk}(P)$. Thus, Lemma C.3.3 shows that $P,\Gamma \vdash_{\underline{\mathsf{s}}} e_m[\mathit{object}/\mathbf{this},\ v_1/\mathit{var}_1, \ldots\ v_n/\mathit{var}_n] : t$.

2. $\mathcal{S}$ and Γ 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 $[object/\textbf{this}]$ 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, $\textsc{SuperOk}(e')$ holds.

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

Case $[cast]$.

1. By assumption, $\mathcal{S}(object) = \langle c, \_ \rangle$ where $c \leq_P t$. Since $c \leq_P t$, $P,\Gamma \vdash_{\underline{s}} object : t$.

2. $\mathcal{S}$ and Γ are unchanged.

3. $\textsc{SuperOk}(e')$ holds because no **super** expression is changed.

Case $[let]$.

1. $P,\Gamma \vdash_{\underline{e}} \textbf{let}\ var = v\ \textbf{in}\ e : t$ implies $P,\Gamma \vdash_{\underline{e}} v : t'$ for some type t' and $P,\Gamma[var : t'] \vdash_{\underline{e}} e : t$. By Lemma C.3.3, $P,\Gamma' \vdash_{\underline{s}} e\,[v/var] : t$.

2. $\mathcal{S}$ and Γ are unchanged.

3. $\textsc{SuperOk}(e')$ holds because no **super** expression is changed.

Case $[xcast]$, $[ncast]$, $[nget]$, $[nset]$, and $[ncall]$. e' is an error configuration. □

C.2 Proof of Progress

Lemma 4.1.6 (Progress) *If* $P,\Gamma \vdash_{\underline{e}} e : t$, $P,\Gamma \vdash_{\sigma} \mathcal{S}$, *and* $\textsc{SuperOk}(e)$, *then either* e *is a value or there exists an* $\langle e', \mathcal{S}' \rangle$ *such that* $P \vdash \langle e, \mathcal{S} \rangle \hookrightarrow \langle e', \mathcal{S}' \rangle$.

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 c. The $[new]$ reduction rules constructs the appropriate e' and $\mathcal{S}'$.

Case $v\underline{\ :\ c\ }.fd$. If v is null, then the $[nget]$ reduction rule applies. Otherwise, $v = object$, and we show that $[get]$ applies.

Type-checking combined with Σ_5 implies $\mathcal{S}(object) = \langle c, \mathcal{F} \rangle$ for some c and $\mathcal{F}$. Type-checking also implies that $\langle c'.fd, t \rangle \in_P c$ for some c' by $[\mathbf{get}^{\mathsf{c}}]$. By the definition of $\in_P$ ($\in_P^{\mathsf{c}}$ in this case), we have $c \leq_P c'$. Finally, by Σ_2, $c'.fd \in \mathrm{dom}(\mathcal{F})$.

Case $v\underline{\ :\ c\ }.fd = v'$. Similar to $v\underline{\ :\ c\ }.fd$, 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 Σ_5 implies $\mathcal{S}(object) = \langle c, \mathcal{F} \rangle$ for some c and $\mathcal{F}$, and type-checking also implies $\langle md, T, V, e_m \rangle \in_P^{\mathsf{c}} c$.

Case $\mathbf{super}\underline{\equiv v : c\ }(v_1, \ldots\ v_n)$. By SuperOk($e$), v must be of the form $object$. Type-checking ensures $\langle md, T, V, e_m \rangle \in_P^{\mathsf{c}} c$.

Case $\mathbf{view}\ t\ v$. If v is null, then $[ncast]$ applies. Otherwise, $v = object$, and by Σ_5, $\mathcal{S}(object) = \langle c, \mathcal{F} \rangle$ for some c and $\mathcal{F}$. Either $[cast]$ or $[xcast]$ applies, depending on whether $c \leq_P t$.

Case $\mathbf{let}\ var = v \in e$. The $[let]$ reduction always applies, constructing an e' and $\mathcal{S}'$ $(= \mathcal{S})$. □

C.3 Supporting Lemmata

The supporting lemmata are due to Shriram Krishnamurthi.

Lemma C.3.1 (Free) *If* $P, \Gamma \vdash_{\underline{\mathsf{e}}} e : t$ *and* $a \notin \mathrm{dom}(\Gamma)$, *then* $P, \Gamma\ [a : t'] \vdash_{\underline{\mathsf{e}}} e : t$.

Proof. The claim follows by reasoning about the shape of the derivation. □

Lemma C.3.2 (Replacement) *If* $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e] : t$, $P, \Gamma \vdash_{\underline{\mathsf{e}}} e : t'$, $P, \Gamma \vdash_{\underline{\mathsf{e}}} e' : t'$, *then* $P, \Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e'] : t$.

Proof. The proof is a replacement argument in the derivation tree. □

Lemma C.3.3 (Substitution) *If* $P, \Gamma\ [var_1 : t_1, \ldots\ var_n : t_n] \vdash_{\underline{\mathsf{e}}} e : t$ *and* $\{var_1, \ldots\ var_n\} \cap \mathrm{dom}(\Gamma) = \emptyset$ *and* $P, \Gamma \vdash_{\underline{\mathsf{s}}} v_i : t_i$ *for* $i \in [1,n]$, *then* $P, \Gamma \vdash_{\underline{\mathsf{s}}} e\ [v_1/var_1, \ldots\ v_n/var_n] : t$.

Proof. Let σ denote the substitution $[v_1/var_1, \ldots\ v_n/var_n]$, let γ 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 $P,\Gamma\gamma \vdash_{\underline{e}} e : t$. We perform a case analysis on the last step in the derivation.

Case $e = \mathbf{new}\ c$. Since $e' = \mathbf{new}\ c$ and its type does not depend on Γ, then $P,\Gamma \vdash_{\underline{e}} e' : c$.

Case $e = var$. If $var \notin \mathrm{dom}(\sigma)$, then $e' = var$ and $\Gamma(var) = t$, so $P,\Gamma \vdash_{\underline{e}} e' : t$. Otherwise, $var = var_i$ for some $i \in [1,n]$, and $e' = \sigma(var_i) = v_i$. By assumption, $P,\Gamma \vdash_{\underline{s}} v_i : t_i$, so $P,\Gamma \vdash_{\underline{s}} e' : t_i$.

Case $e = \mathsf{null}$. By [**null**], any type is derivable for null.

Case $e = e_1\underline{\ :\ c\ }.fd$. By [$\mathbf{get}^{\mathsf{c}}$], $P,\Gamma\gamma \vdash_{\underline{e}} e_1 : t'$ and $\langle c.fd,t\rangle \in_P t'$ from some t'. By induction, $P,\Gamma \vdash_{\underline{e}} \sigma(e_1) : t''$ where t'' is a sub-type of t'. Since $\in_P$ for fields is closed over subtypes on the right-hand side, $\langle c.fd,t\rangle \in_P t''$. Thus, by [$\mathbf{get}^{\mathsf{c}}$] on $e' = \sigma(e_1)\underline{\ :\ c\ }.fd$, $P,\Gamma \vdash_{\underline{e}} e' : t$.

Case $e = e_1\underline{\ :\ c\ }.fd = e_2$. This case is similar to the previous case, relying on subsumption for the right-hand side of an assignment as allowed by [$\mathbf{set}^{\mathsf{c}}$].

Case $e = \mathbf{view}\ t\ e_1$. By [$\mathbf{cast}^{\mathsf{c}}$], $P,\Gamma\gamma \vdash_{\underline{e}} e_1 : t'$ for some t'. By induction, $P,\Gamma \vdash_{\underline{s}} \sigma(e_1) : t'$. Since $e' = \sigma(\mathbf{view}\ t\ e_1) = \mathbf{view}\ t\ \sigma(e_1)$, [$\mathbf{cast}^{\mathsf{c}}$] gives $P,\ \Gamma \vdash_{\underline{e}} e' : t$.

Case $e = \mathbf{let}\ var = e_1\ \mathbf{in}\ e_2$. Let $\sigma_1 = \sigma$ and $\gamma_1 = \gamma$, and lexically rename var in e so that $var \notin \mathrm{dom}(\gamma_1)$. By [**let**], $P,\Gamma\gamma_1 \vdash_{\underline{e}} e_1 : t_1$. By induction, $P,\Gamma \vdash_{\underline{s}} \sigma_1(e_1) : t_1$. Let $\gamma_2 = [var : t_1]$, so that $P,\Gamma\gamma_1\gamma_2 \vdash_{\underline{e}} e_2 : t$. Since $var \notin \mathrm{dom}(\gamma_1)$, we can reverse the order of the γ_1 and γ_2 extensions to Γ, so $P,\Gamma\gamma_2\gamma_1 \vdash_{\underline{e}} e_2 : t$. By induction, $P,\Gamma\gamma_2 \vdash_{\underline{s}} \sigma_1(e_2) : t$. Finally, by [**let**] on $e' = \sigma_1(\mathbf{let}\ var = e_1\ \mathbf{in}\ e)$, $P,\Gamma \vdash_{\underline{s}} e' : t$.

Case $e = e_0.md\ (e_1, \ldots\ e_n)$. By [$\mathbf{call}^{\mathsf{c}}$], $P,\Gamma\gamma \vdash_{\underline{s}} e_i : t_i$ for $i \in [1,n]$ and $P,\Gamma\gamma \vdash_{\underline{e}} e_0 : t_0$ such that $\langle md, (t_1 \ldots\ t_n \rightarrow t), (var_1, ..., var_n), e_m\rangle \in_P t_0$. By induction, $P,\Gamma \vdash_{\underline{s}} \sigma(e_i) : t_i$ for each e_i, and $P,\Gamma \vdash_{\underline{e}} \sigma(e_0) : t_0'$ where $t_0' \leq_P t_0$. Since $\in_P$ must preserve the type of methods over subtypes on the right-hand side (by ClassMethodsOk), $\langle md,(t_1 \ldots\ t_n \rightarrow t),(var_1', \ldots var_n'),e_m'\rangle \in_P t_0'$. Thus, by [$\mathbf{call}^{\mathsf{c}}$] on $e' = \sigma(e_0.md\ (e_1, \ldots\ e_n))$, $P,\Gamma \vdash_{\underline{e}} e' : t$.

Case $e =$ **super** $\underline{\equiv \textbf{this} : c}\,.md\ (e_1, \ldots\ e_n)$. This case is similar to the previous case. □

Lemma C.3.4 (Replacement with Subtyping) *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e] : t$*,* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t'$*, and* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e' : t''$ *where* $t'' \leq_P t'$*, then* $P,\Gamma \vdash_{\underline{\mathsf{s}}} \mathsf{E}[e'] : t$*.*

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 $\mathsf{E}[e] = \mathsf{E}_1[\mathsf{E}_2[e]]$ where E_2 is a singular evaluation context, *i.e.*, a context whose depth is one. Consider the shape of $\mathsf{E}_2[\bullet]$, which must be one of:

Case $\bullet\underline{\ :\ c\ }.fd$. Since c is fixed, $\bullet$'s type does not matter; the expression's type is the type of the field.

Case $\bullet\underline{\ :\ c\ }.fd = e$. Same as the previous case.

Case $v\underline{\ :\ c\ }.fd = \bullet$. Since $[\textbf{set}^{\mathsf{c}}]$ allows subsumption on the right-hand side of the assignment, the type of the expression is the same replacing $\bullet$ with e or e'.

Case $\bullet.md(e\ \ldots)$. 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 e or e'.

Case $v.md(v\ \ldots\ \bullet\ e\ \ldots)$. Since $[\textbf{call}^{\mathsf{c}}]$ allows subsumption on method arguments, the type of the expression is the same replacing $\bullet$ with e or e'.

Case **super** $\underline{\equiv v : c\ }.md\ (v\ \ldots\ \bullet\ e\ \ldots)$. Same as the previous case.

Case **view** $t\ \bullet$. Since t is fixed, $\bullet$'s type does not matter in $[\textbf{cast}^{\mathsf{c}}]$ (our less restrictive typing rule); the expression's type is t.

Case **let** $var = \bullet$ **in** e_2. By $[\textbf{let}]$ with $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t'$, $P,\Gamma\gamma_1 \vdash_{\underline{\mathsf{e}}} e_2 : t_1$ for some type t_1 where γ_1 is $[var : t']$. We must show that $P,\Gamma\gamma_2 \vdash_{\underline{\mathsf{s}}} e_2 : t_1$ where $\gamma_2 = [var : t'']$, which follows from Lemma C.3.6. □

Definition C.3.5 $\Gamma \leq_\Gamma \Gamma'$ *if* $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$ *and* $\forall\ v \in \mathrm{dom}(\Gamma)$*,* $\Gamma'(v) \leq_P \Gamma(v)$*.*

Lemma C.3.6 *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t$ *and* $\Gamma \leq_\Gamma \Gamma'$*, then* $P,\Gamma' \vdash_{\underline{\mathsf{s}}} e : t$*.*

Proof. The proof is a straightforward adaptation of the proof for Lemma C.3.3. □

Appendix D

MixedJava Proofs

D.1 Proof of Subject Reduction

Lemma 4.3.6 (Subject Reduction for MixedJava**)** *If* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e : t$, $P,\Gamma \vdash_{\sigma} \mathcal{S}$, $\textsc{SuperOk}(e)$, *and* $\langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$, *then* e' *is an* error *configuration or there exists* Γ' *such that*

1. $P,\Gamma' \vdash_{\underline{\mathsf{e}}} e' : t$,
2. $P,\Gamma' \vdash_{\sigma} \mathcal{S}'$, *and*
3. $\textsc{SuperOk}(e')$.

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 Γ' and show that the two consequents of the theorem are satisfied relative to the new expression, store, and environment.

Case [*new*]. Set $\Gamma' = \Gamma\ [\langle object||M/m\rangle : m]$.

1. We have $P,\Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[\mathbf{new}\ m] : t$. From Σ_5, $object \notin \mathrm{dom}(\mathcal{S}) \Rightarrow \langle object||\_\_\rangle \notin \mathrm{dom}(\Gamma)$. Thus $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}[\mathbf{new}\ m] : t$ by Lemma D.3.1. Since $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathbf{new}\ m : m$ and $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \langle object||M/m\rangle : m$, Lemma D.3.2 implies $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}[\langle object||M/m\rangle] : t$.
2. Let $\mathcal{S}'(object) = \langle m,\mathcal{F}\rangle$, so *object* is the only new element in $\mathrm{dom}(\mathcal{S}')$ and $\langle object||M/m\rangle$ is the only new element in $\mathrm{dom}(\Gamma')$.

   Σ_1: $\Gamma'(\langle object||M/m\rangle) = m$ and $m \leq_P m$. Since $m \longrightarrow_P M$, $\mathrm{WF}(M/m)$.

   Σ_2: $\mathrm{dom}(\mathcal{F})$ is correct by construction.

   Σ_3: $\mathrm{rng}(\mathcal{F}) = \{\mathsf{null}\}$.

   Σ_4: Since $\mathrm{rng}(\mathcal{F}) = \{\mathsf{null}\}$, this property is unaffected.

Σ_5 **and** Σ_6: The only addition to the domains of Γ and $\mathcal{S}$ is *object*.

3. Since $\mathsf{E}[\langle object||M/m\rangle]$ contains the same **super** expressions as $\mathsf{E}[\mathbf{new}\ m]$, and no instance of **this** or *object* is replaced in the new expression, $\textsc{SuperOk}(e')$ holds.

Case [*get*]. Set $\Gamma' = \Gamma$.

1. If v is null, it can be typed as t, so $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}[v] : t$ by Lemma D.3.2. If v is not null, then by Σ_4, $v = \langle object||M/t\rangle$ for some *object* and M. By Σ_1, $\Gamma(v) = t$, so by Lemma D.3.2, $P,\Gamma' \vdash_{\underline{\mathsf{e}}} \mathsf{E}[v] : t$.
2. $\mathcal{S}$ and Γ are unchanged.
3. $\textsc{SuperOk}(e')$ holds because no **super** expression is changed.

Case [*set*]. Set $\Gamma' = \Gamma$.

1. The proof is by a straightforward extension of the proof for [*get*].
2. The only change to the store is a field update; thus only Σ_3 and Σ_4 are affected. Let v be the assigned value, and assume that v is not null.

   Σ_3: Since v is typable, it must be in $\mathrm{dom}(\Gamma)$. By Σ_5, its object part is therefore in $\mathrm{dom}(\mathcal{S})$.

   Σ_4: The typing of the assignment expression indicates that the type of v is t, so v must be of the form $\langle object'||M''/t\rangle$.
3. $\textsc{SuperOk}(e')$ holds because no **super** expression is changed.

Case [*call*]. Set $\Gamma' = \Gamma\ [\langle object||m'{::}M'/m'\rangle : m']$.

1. We are given $\langle md, (t_1 \ldots\ t_n \to t), (var_1, ..., var_n), e_m, m' :: M'/m'\rangle \in^{\mathsf{rt}}_P M_v$ in M_o, which implies $\langle md, T, V, e_m\rangle \in^{\mathsf{m}}_P m'$ by the definition of $\in^{\mathsf{rt}}_P$, where $T = (t_1\ \ldots\ t_n \to t)$ and $V = (var_1\ \ldots\ var_n)$.

   We are also given $P,\Gamma \vdash_{\underline{\mathsf{e}}} \langle object||M_v/t'\rangle.md(v_1, \ldots\ v_n) : t$, which implies $\langle md, T'\rangle \in_P t'$. By Lemma D.3.6, $T' = T$. Since the method call type-checks and $T' = T$, $P,\Gamma \vdash_{\underline{\mathsf{e}}} v_i : t_i$ for i in $[1, n]$.

   Type-checking for the program P ensures that $P,m' \vdash_{\underline{\mathsf{m}}} t\ md\ (t_1\ var_1, \ldots\ t_n\ var_n)\ \{e_m\}$, and thus $P,[\mathbf{this} : m',\ var_1 : t_1, \ldots\ var_n : t_n] \vdash_{\underline{\mathsf{e}}} e : t$.

Hence, by Lemma D.3.3, P,$\Gamma' \vdash_{\mathsf{e}} e_m[\langle object||m'::M'/m'\rangle/\mathbf{this}, v_1/var_1, \ldots v_n/var_n] : t$.

2. $\mathcal{S}' = \mathcal{S}$. If Γ' contained a mapping for $\langle object||m'::M'/m'\rangle$, it was m' by Σ_1, so $\Gamma' = \Gamma$. Otherwise, $\langle object||m'::M'/m'\rangle$ is new in Γ', which might affect Σ_1, Σ_5, and Σ_6:

   Σ_1: $\Gamma'(\langle object||m'::M'/m'\rangle) = m'$. The $\in_P^{\mathsf{rt}}$ relation ensures that $m \leq_P m'$ because $M_o \leq^{\mathsf{M}} m'::M'$. WF$(m'::M'/m')$ is immediate.

   Σ_5 and Σ_6: Since $\Gamma(\langle object||M_v/t'\rangle) = t'$, $object \in \mathrm{dom}(\mathcal{S})$ by Σ_5 on $\mathcal{S}$ and Γ. Thus, adding $\langle object||m'::M'/m'\rangle$ to Γ' does not require any new elements in $\mathcal{S}'$.

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 $[\langle object||m'::M'/m'\rangle/\mathbf{this}]$ part of the substitution may replace **this** in a **super** annotation with $\langle object||m'::M'/m'\rangle$, but no other part of the substitution can affect **super** annotations. Thus, SUPEROK(e') holds.

Case [*super*]. Set $\Gamma' = \Gamma\ [\langle object||m'::M'/m'\rangle : m']$.

1. Similar to [*call*]. The object for dispatching is $\langle object||m::M/m\rangle$, and we are given $m \prec_P^{\mathsf{m}} i$ and $\langle md,T\rangle \in_P i$. (The i in [**super**$^{\mathsf{m}}$] 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 $\langle object||m'::M'/m'\rangle$ is new:

   Σ_1: $\Gamma'(\langle object||m'::M'/m'\rangle) = m'$. If $\mathcal{S}(object) = \langle m_o,\_\rangle$, $m_o \leq_P m'$ because Σ_1 on Γ ensures that the original view $m::M$ is part of m_o, $\triangleright$ selects a sub-view of M as M'', and $\in_P^{\mathsf{rt}}$ selects $m'::M'$ within M''.

   Σ_5 and Σ_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, $P,\Gamma' \vdash_{\underline{e}} \mathsf{E}[\langle object||M'/t\rangle] : t$.
2. Similar to [*call*]. If $\langle object||M'/t\rangle \notin \Gamma$:

   Σ_1: $\Gamma'(\langle object||M'/t\rangle) = t$. The side condition for [*view*] requires $t' \trianglelefteq_P t$, which implies $t' \leq_P t$. Σ_1 on Γ ensures $m \leq_P t'$ when $\mathcal{S}(object) = \langle m, \_\rangle$, so $m \leq_P t$ by transitivity.

   Σ_5 and Σ_6: 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:

   Σ_1: $\Gamma'(\langle object||M''/t\rangle) = t$. The side condition for [*cast*] requires $m \trianglelefteq_P t$, which implies $m \leq_P t$.

   Σ_5 and Σ_6: Same as [*call*].
3. SuperOk(e') holds because no **super** expression is changed.

Case [*let*]. $P,\Gamma \vdash_{\underline{e}}$ **let** $var = v$ **in** $e : t$ implies $P,\Gamma \vdash_{\underline{e}} v : t'$ for some type t' and $P,\Gamma[var : t'] \vdash_{\underline{e}} e : t$. Set $\Gamma' = \Gamma$.

1. By Lemma D.3.3, $P,\Gamma' \vdash_{\underline{e}} e\,[v/var] : t$.
2. $\mathcal{S}$ and Γ are unchanged.
3. SuperOk(e') holds because no **super** expression is changed.

Case [*xcast*], [*ncast*], [*nget*], [*nset*] and [*ncall*]. e' is an error configuration. $\square$

D.2 Proof of Progress

Lemma 4.3.7 (Progress for MixedJava) *If $P,\Gamma \vdash_{\underline{e}} e : t$, $P,\Gamma \vdash_{\sigma} \mathcal{S}$, and* SuperOk($e$)*, then either e is a value or there exists an $\langle e',\mathcal{S}'\rangle$ such that $\langle e,\mathcal{S}\rangle \hookrightarrow \langle e',\mathcal{S}'\rangle$.*

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 $\mathcal{S}'$.

Case $v.fd$. If v is null, then the $[nget]$ reduction rule applies. Otherwise, $v = \langle object||M/t\rangle$, and we show that $[get]$ applies.

Type-checking combined with Σ_5 implies $\mathcal{S}(object) = \langle m_o,\mathcal{F}\rangle$ for some m_o and $\mathcal{F}$. Type-checking also implies that $t = m$ for some mixin m, and $\langle m'.fd,t\rangle \in_P m$ for some m' by $[\mathbf{get}^{\mathsf{m}}]$. By the definition of $\in_P$ ($\in_P^{\mathsf{m}}$ in this case), we have $m \trianglelefteq_P m'$, so there is a unique m' such that $M/m \triangleright M'/m'$, and $M \leq^{\mathsf{M}} M'$.

Environment-store consistency implies $M_o \leq^{\mathsf{M}} M$, where $m_o \longrightarrow_P M_o$. By transitivity, $M_o \leq^{\mathsf{M}} M'$. Finally, by Σ_2, $M'.fd \in \mathrm{dom}(\mathcal{F})$.

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, $v = \langle object||M/t'\rangle$, and we show that $[call]$ applies.

Type-checking combined with Σ_5 implies $\mathcal{S}(object) = \langle m_o,\mathcal{F}\rangle$ for some m_o and $\mathcal{F}$. Define M_o as $m_o \longrightarrow_P M_o$.

By $[\mathbf{call}^{\mathsf{m}}]$, $\langle md,\ T\rangle \in_P t'$. The $\in_P^{\mathsf{rt}}$ 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 $\propto$ 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 $\langle md,T,V,e\rangle \in_P^{\mathsf{m}} m_b$:
   - If there is no i such that $m_x \prec_P^{\mathsf{m}} i$ and $\langle md,T\rangle \in_P i$, then $\langle md,T,V,e\rangle \in_P^{\mathsf{m}} m_x$ by the definition of $\in_P^{\mathsf{m}}$.
   - 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::$M_x \leq^{\mathsf{M}} m_y$::$M_y$. This argument on m_x::M_x applies inductively to m_y::M_y, showing that $\langle md,T,V,e\rangle \in_P^{\mathsf{m}} m_b$.

3. By WF(M/t') and $M \leq^{\mathsf{M}} M_b$, then $M_o \leq^{\mathsf{M}} M_b$, so the final phase of the $\in_P^{\mathsf{rt}}$ calculation must succeed (although the selected view is not necessarily M_b).

Case super $\underline{\equiv v}$ $(v_1, \ldots\ v_n)$. By SuperOk(e), v must be a reference of the form $\langle object \| m{::}M/m \rangle$. Thus, we show that the [*super*] reduction applies.

Since $m \prec_P^{\mathsf{m}} i$, 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 $\in_P^{\mathsf{rt}}$ relation necessarily selects some $m'{::}M'/m'$ and e.

Case view $\underline{t'\ \mathsf{as}}$ $t\ v$. If v is null, then [*ncast*] applies. Otherwise, $v = \langle object \| M/t' \rangle$ for some M, and by Σ_5 $\mathcal{S}(object) = \langle m_o{,}\mathcal{F} \rangle$ for some m_o and $\mathcal{F}$.

If $t' \trianglelefteq_P t$, then [*view*] applies since M'/t clearly exists for $M/t \triangleright M'/t$. Assume that $t' \ntrianglelefteq_P t$. If $m_o \ntrianglelefteq_P t'$, then [*xcast*] applies. Otherwise, [*cast*] applies since $m_o \trianglelefteq_P t$ means that M''/t clearly exists for $M'/m \triangleright M''/t$, where $m \longrightarrow_P M'$.

Case let $var = v \in e$. The [*let*] reduction always applies, constructing an e' and $\mathcal{S}'$ $(= \mathcal{S})$. □

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* $P{,}\Gamma \vdash_{\underline{\mathsf{e}}} e : t$ *and* $a \notin \mathrm{dom}(\Gamma)$*, then* $P{,}\Gamma[a : t'] \vdash_{\underline{\mathsf{e}}} e : t$.

Proof. This follows by reasoning about the shape of the derivation. □

Lemma D.3.2 (Replacement for MixedJava) *If* $P{,}\Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e] : t$, $P{,}\Gamma \vdash_{\underline{\mathsf{e}}} e : t'$, *and* $P{,}\Gamma \vdash_{\underline{\mathsf{e}}} e' : t'$, *then* $P{,}\Gamma \vdash_{\underline{\mathsf{e}}} \mathsf{E}[e'] : t$.

Proof. The proof is a replacement argument in the derivation tree. □

Lemma D.3.3 (Substitution for MIXEDJAVA) *If* $P,\Gamma\ [var_1 : t_1, \ldots\ var_n : t_n] \vdash_{\underline{\mathsf{e}}} e : t$ *and* $\{var_1, \ldots\ var_n\} \cap \mathrm{dom}(\Gamma) = \emptyset$ *and* $P,\Gamma \vdash_{\underline{\mathsf{s}}} v_i : t_i$ *for* $i \in [1,n]$, *then* $P,\Gamma \vdash_{\underline{\mathsf{e}}} e\ [v_1/var_1, \ldots\ v_n/var_n] : t$.

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 (•/• ▹ •/• Preserves Well-Formedness) *If* WF(M/t) *and* $M/t \triangleright M'/t'$, *then* 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 is $[m]$ (the base case), then $t' = m$, $M' = M$, and WF($[m']/m'$). Also, if $m = m'$ and $M = M'$, then WF(M/m) = WF(M'/m').

Otherwise, $m' = m'' \circ m'''$, and there are two sub-cases:

- If $m'' \leq_P^{\mathsf{M}} m'$ and $M/m'' \triangleright M'/m'$, then WF($M'/m'$) by induction: m'' is smaller than m, and WF(M/m'') since m'' is a prefix of m.
- If $m''' \leq_P^{\mathsf{M}} m'$ and $M_r/m''' \triangleright M'/m'$, then WF($M'/m'$) by induction: M_r is smaller than M, and 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 $m \prec\!\!\!\prec_P^{\mathsf{m}} i$. Thus, WF($M'/i$) because $M' = m$::M'' and $m \trianglelefteq_P i$. □

Lemma D.3.5 (Consistency of •.• ∝ •.•) *If* m::$M.md \propto m'$::$M'.md$ *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 ▹ (used in the definition of ∝) cannot select any chain other than $[m]$.

Otherwise, $m \prec_P^{\mathsf{m}} i$ for some i where $\langle md,T\rangle \in_P^{\mathsf{i}} i$. Since $M/i \triangleright M''/i$ and ▹ preserves well-formedness of views, M'' is of the form m''::M''' for some m'' and M'''

where $m'' \prec^{\mathsf{m}}_P i$. By MixinsImplementAll, MixinMethodsOk, $m'' \prec^{\mathsf{m}}_P i$, and $\langle md,T\rangle \in^{\mathsf{i}}_P i$, 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. $\square$

Lemma D.3.6 (Soundness of $\bullet \in^{\mathsf{rt}}_P \bullet$ in $\bullet$) *If* $\langle md,T,\_\_,\_\_,m{::}M/m\rangle \in^{\mathsf{rt}}_P M_v$ *in* M_o*,* $\mathrm{WF}(M_v/t_v)$*, and* $\langle md,T'\rangle \in_P t_v$*, then* $T = T'$*.*

Proof. To get $\langle md,T,\_\_,\_\_,m{::}M/m\rangle$, the $\in^{\mathsf{rt}}_P$ relation first finds $m_x{::}M_x$ such that $\langle md,T''\rangle \in_P m_x$. Since $\mathrm{WF}(M_v/t_v)$ 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, $\mathrm{WF}(M_v/t_v)$ implies $m_x{::}M_x = M_v$ and $m_x \trianglelefteq_P t$. Since $m_x \trianglelefteq_P t$, $T'' = T'$ by MixinMethodsOk and MixinsImplementAll.

Case t_v is an atomic mixin m'. Then, $\mathrm{WF}(M_v/t_v)$ implies $m_x{::}M_x = M_v$ and $m_x = t$, so $T'' = T'$ by MethodOncePerMixin.

Case t_v is a composite mixin m'. Then, $\langle md,T'\rangle \in_P t_v$ implies $\langle md,T'\rangle \in^{\mathsf{m}}_P m''$ for some m'' where $m' \leq_P m''$, which means that some candidate for m_x exists within m'. Furthermore, $\langle md,T'''\rangle \in^{\mathsf{m}}_P m''$ where $m' \leq_P m''$ implies $T''' = T''$ by the definition of $\in^{\mathsf{m}}_P$ (because $\in^{\mathsf{m}}_P$ eliminates ambiguities), so every candidate for m_x with m' gives the same type to method md. Since M_v starts with the chain of atomic mixins of m', then m_x must be part of m'. Finally, $T'' = T'$ because $m' \leq_P m_x$.

Next, $\in^{\mathsf{rt}}_P$ finds an M_b such that $m_x{::}M_x.md \propto M_b.md$. From $\mathrm{WF}(m_x{::}M_x/m_x)$, $\langle md,T'\rangle \in_P m_x$, 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 $\in^{\mathsf{rt}}_P$ relation selects a $m{::}M$ such that $\langle md,T,\_\_,\_\_\rangle \in^{\mathsf{m}}_P m$ 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, $\in^{\mathsf{m}}_P$ implies $\Subset^{\mathsf{m}}_P$, so we have both $\langle md,T,\_\_,\_\_\rangle \Subset^{\mathsf{m}}_P m_b$ and $\langle md,T',\_\_,\_\_\rangle \Subset^{\mathsf{m}}_P m_b$. By MethodOncePerMixin, those must be the same method in m_b, so $T = T'$. $\square$

Bibliography

[1] Abadi, M. and L. Cardelli. A theory of primitive objects — untyped and first-order systems. In Hagiya, M. and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320. Springer-Verlag, April 1994.

[2] Abadi, M. and L. Cardelli. A theory of primitive objects: second-order systems. In Sannella, D., editor, *Proc. European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, 1994.

[3] Abadi, M. and L. Cardelli. An imperative object calculus. In Mosses, P. D., M. Nielsen and M. I. Schwartzbach, editors, *Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 471–485. Springer-Verlag, May 1995.

[4] Agesen, O., S. Freund and J. C. Mitchell. Adding type parameterization to Java. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 49–65, October 1997.

[5] Ancona, D. and E. Zucca. An algebraic approach to mixins and modularity. In Hanus, M. and M. Rodríguez-Artalejo, editors, *Proc. Conference on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, pages 179–193. Springer-Verlag, 1996.

[6] Biswas, S. K. Higher-order functors with transparent signatures. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 154–163, Janurary 1995.

[7] Bracha, G. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.

[8] Bracha, G. and W. Cook. Mixin-based inheritance. In *Proc. Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming*, October 1990.

[9] Bracha, G. and G. Lindstrom. Modularity meets inheritance. In *Proc. IEEE Computer Society International Conference on Computer Languages*, pages 282–290, April 1992.

[10] Cardelli, L. Program fragments, linking, and modularization. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 266–277, Janurary 1997.

[11] Clinger, W. and Rees, J. (Eds.). The revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

[12] Cook, W. R. *A Denotational Semantics of Inheritance.* Ph.D. thesis, Department of Computer Science, Brown University, Providence, RI, May 1989.

[13] Cook, W. R. Object-oriented programming versus abstract data types. In *Proc. ACM International Workshop on Foundations of Object-Oriented Languages*, pages 151–178, June 1990.

[14] Crary, K., R. Harper and S. Puri. What is a recursive module? In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 50–63, May 1999.

[15] Curtis, P. and J. Rauen. A module system for Scheme. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 13–28, 1990.

[16] Drossopolou, S. and S. Eisenbach. Java is typesafe – probably. In *Proc. European Conference on Object Oriented Programming*, June 1997.

[17] Ducournau, R., M. Habib, M. Huchard and M. L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 16–24, October 1992.

[18] Duggan, D. and C. Sourelis. Mixin modules. In *Proc. ACM International Conference on Functional Programming*, pages 262–273, May 1996.

[19] Eifrig, J., S. Smith, V. Trifonov and A. Zwarico. Application of OOP type theory: State, decidability, integration. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 16–30, October 1994.

[20] Felleisen, M. Programming languages and lambda calculi. `www.cs.rice.edu/~matthias/411web/mono.ps`.

[21] Felleisen, M. and D. P. Friedman. *A Little Java, A Few Patterns*. The MIT Press, 1998.

[22] Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report 100, Rice University, June 1989. *Theoretical Computer Science*, volume 102, 1992, pp. 235–271.

[23] Findler, R. B., C. Flanagan, M. Flatt, S. Krishnamurthi and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 369–388, September 1997.

[24] Findler, R. B. and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. ACM International Conference on Functional Programming*, September 1998.

[25] Flanagan, C., M. Flatt, S. Krishnamurthi, S. Weirich and M. Felleisen. Finding bugs in the web of program invariants. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[26] Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[27] Flatt, M. and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 236–248, June 1998.

[28] Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 171–183, Janurary 1998.

[29] Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Massachusetts, 1994.

[30] Glew, N. and G. Morrisett. Type-safe linking and modular assembly language. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 250–261, Janurary 1999.

[31] Gosling, J., B. Joy and G. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[32] Harbison, S. P. *Modula-3.* Prentice Hall, 1991.

[33] Harper, R. and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 123–137, Janurary 1994.

[34] Harper, R., J. Mitchell and E. Moggi. Higher-order modules and the phase distinction. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 341–354, Janurary 1990.

[35] Harper, R. and C. Stone. A type-theoretic interpretation of Standard ML. In Plotkin, G., C. Stirling and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner.* MIT Press, 1998.

[36] Hollingsworth, J. *Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada.* PhD thesis, The Ohio State University, 1992.

[37] Hudak, P. and Wadler, P. (Eds.). Report on the programming language Haskell. Technical Report YALE/DCS/RR777, Yale University, Department of Computer Science, August 1991.

[38] International Organization for Standardization. *Ada 95 Reference Manual. The Language. The Standard Libraries*, Janurary 1995.

[39] Jagannathan, S. Metalevel building blocks for modular systems. *ACM Transactions on Programming Languages and Systems*, 16(3):456–492, May 1994.

[40] JavaSoft. *JavaBeans*, 1.0 edition, October 1996. `http://java.sun.com/beans`.

[41] Kamin, S. Inheritance in SMALLTALK-80: a denotational definition. In *Proc. ACM Symposium on Principles of Programming Languages*, Janurary 1988.

[42] Kelsey, R. A. Fully-parameterized modules or the missing link. Technical Report 97-3, NEC Research Institute, 1997.

[43] Kessler, R. R. *LISP, Objects, and Symbolic Programming.* Scott, Foresman and Company, Glenview, IL, USA, 1988.

[44] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier and J. Irwin. Aspect-oriented programming. In *Proc. European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1997.

[45] Koschmann, T. *The Common LISP Companion.* John Wiley and Sons, New York, N.Y., 1990.

[46] Krishnamurthi, S., Y.-D. Erlich and M. Felleisen. Expressing structural properties as language constructs. In *Proc. European Symposium on Programming*, 1999.

[47] Krishnamurthi, S. and M. Felleisen. Toward a formal theory of extensible software. In *Proc. ACM Conference on Foundations of Software Engineering*, 1998.

[48] Krishnamurthi, S., M. Felleisen and D. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *Proc. European Conference on Object-Oriented Programming*, 1998.

[49] Kühne, T. The translator pattern—external functionality with homomorphic mappings. In *Proceedings of TOOLS 23, USA*, pages 48–62, July 1997.

[50] Lee, S.-D. and D. P. Friedman. Quasi-static scoping: Sharing variable bindings across multiple lexical scopes. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 479–492, Janurary 1993.

[51] Lee, S.-D. and D. P. Friedman. Enriching the lambda calculus with context toward a theory of incremental program construction. In *Proc. ACM International Conference on Functional Programming*, pages 239–250, 1996.

[52] Leroy, X. Unboxed objects and polymorphic typing. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 177–188, Janurary 1992.

[53] Leroy, X. Manifest types, modules, and separate compilation. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 109–122, Janurary 1994.

[54] Leroy, X. Applicative functions and fully transparent higher-order modules. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 142–153, Janurary 1995.

[55] Leroy, X. *The Objective Caml system*, 1996. `http://pauillac.inria.fr/ocaml/`.

[56] MacQueen, D. Modules for Standard ML. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 198–207, 1984.

[57] MacQueen, D. B. and M. Tofte. A semantics for higher-order functors. In *Proc. European Symposium on Programming*, Lecture Notes in Computer Science, pages 409–423. Springer-Verlag, April 1994.

[58] Mason, I. A. and C. L. Talcott. Reasoning about object systems in VTLoE. *International Journal of Foundations of Computer Science*, 6(3):265–298, September 1995.

[59] McIlroy, M. D. Mass produced software components. In Naur, P. and B. Randell, editors, *Report on a Conference of the NATO Science Committee*, pages 138–150, 1968.

[60] Mezini, M. and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 97–116, 1998.

[61] Miller, J. and G. Rozas. Free variables and first-class environments. *Lisp and Symbolic Computation: An International Journal*, 3(4):107–141, 1991.

[62] Milner, R., M. Tofte and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts and London, England, 1990.

[63] Mitchell, J. G., W. Mayberry and R. Sweet. *Mesa Language Manual*, 1979.

[64] OMG. *The Common Object Request Broker: Architecture and Specification*, 2.0 edition, July 1995. formal document 97-02-25.

[65] Palsberg, J. and C. B. Jay. The essence of the Visitor pattern. Technical Report 05, University of Technology, Sydney, 1997.

[66] Reddy, U. S. Objects as closures: Abstract semantics of object oriented languages. In *Proc. Conference on Lisp and Functional Programming*, pages 289–297, July 1988.

[67] Rémy, D. Programming objects with ML-ART: An extension to ML with abstract and record types. In Hagiya, M. and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, pages 321–346, New York, N.Y., April 1994. Springer-Verlag.

[68] Rémy, D. Introduction aux objets. Unpublished manuscript, lecture notes for *course de magistère*, Ecole Normale Supérieure, 1996.

[69] Rémy, D. and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 40–53, Paris, France, 15–17 Janurary 1997.

[70] Reppy, J. and J. Riecke. Simple objects for Standard ML. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 171–180, 1996.

[71] Reynolds, J. C. User-defined types and procedural data structures as complementary approaches to data abstraction. In Schuman, S. A., editor, *New Directions in Algorithmic Languages*, pages 157–168. IFIP Working Group 2.1 on Algol, 1975.

[72] Rogerson, D. *Inside COM: Microsoft's Component Object Model*. Microsoft Press, 1997.

[73] Rossie, J. G., D. P. Friedman and M. Wand. Modeling subobject-based inheritance. In Cointe, P., editor, *Proc. European Conference on Object-Oriented Pro-*

gramming, volume 1098 of *Lecture Notes in Computer Science*, pages 248–274. Springer-Verlag, July 1996.

[74] Saraswat, V. Java is not type-safe, August 1997. `www.research.att.com/~vj/bug.html`.

[75] Smaragdakis, Y. and D. Batory. Implementing layered designs with mixin layers. In *Proc. European Conference on Object-Oriented Programming*, pages 550–570, 1998.

[76] Snyder, A. Inheritance and the development of encapsulated software components. In *Research Directions in Object-Oriented Programming*, pages 165–188. MIT Press, 1987.

[77] SunSoft. *SunOS 5.5 Linker and Libraries Manual*, 1996.

[78] Syme, D. Proving Java type soundness. Technical Report 427, University of Cambridge, July 1997.

[79] Szyperski, C. Independently extensible systems: Software engineering potential and challenges. In *Proc. Australian Computer Science Conference*, 1996.

[80] Szyperski, C. *Component Software*. Addison-Wesley, 1998.

[81] Tarditi, D., G. Morrisett, P. Cheng, C. Stone, R. Harper and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[82] Tofte, M. Principal signatures for higher-order program modules. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 189–199, Janurary 1992.

[83] Weide, B. W., W. F. Ogden and S. H. Zweben. Reusable software components. In Yovits, M. C., editor, *Advances in Computers*, volume 33. Academic Press, 1991.

[84] Wirth, N. *Programming in Modula-2*. Springer-Verlag, 1983.

[85] Wright, A. and M. Felleisen. A syntactic approach to type soundness. Technical Report 160, Rice University, 1991. *Information and Computation*, volume 115(1), 1994, pp. 38–94.