Much Ado About Nothing:
Putting Java's Null in its Place

A dissertation presented by

Richard Cobbe

to the College of Computer and Information Science

in partial fulfillment of the requirements for
the degree of Doctor of Philosophy

in the field of

Computer Science

Northeastern University
Boston, Massachusetts

December 1, 2008

Abstract

Mainstream object-oriented languages include a *null* value that inhabits every object type and that pervades programs. It exists both because the language semantics requires it in certain places and because it is the most convenient representation for common patterns, such as for sentinel values indicating failure. Safety requires implementations of these languages to insert run-time checks throughout programs to determine whether object references are null at each field lookup and method call.

The ubiquity of *null* in object-oriented programs leads to severe engineering problems for programmers. First, the error messages issued by the run-time checks are typically not sufficiently informative to help the programmer find the source of the error. Second, the type systems in OO languages generally do not distinguish *null* from other values of (object) type, preventing the programmer from stating important invariants about the flow of *null* in the type system. Third, programmers' standard use of *null* as a sentinel does not unambiguously represent failures. To resolve or avoid these ambiguities, component authors must incorporate additional complexity into their interfaces, and this complexity can lead to subtle bugs.

In this dissertation, we propose two changes to Java that allow us to completely remove the *null* value. Doing so addresses the problems above and provides significant engineering benefits to the programmer. Further, we demonstrate the practical feasibility of our proposal with a migration path that allows programmers to shift large codebases from Java to our new language, one class at a time.

Acknowledgments

I am deeply grateful to many people for the support and assistance they have provided during while I have worked on my dissertation. Chief among them are the members of my committee, Professors Karl Lieberherr, Olin Shivers, both of Northeastern University, and Professor Robert Muller of Boston College. Without their support, suggestions, and insight I could never have completed this. My advisor, Matthias Felleisen, has repeatedly demonstrated his devotion to his students, spending long hours reading drafts, discussing the technical points in detail, always spurring us on to better and better performance, and supporting us in other ways too many to mention here.

Just as important to this dissertation and to my development as a graduate student are the members of the Programming Research Laboratory at Northeastern University. This large, vibrant group has been an ideal environment for the study of programming languages, and I cannot begin to list all of the things I have learned from my classmates and professors. I especially want to thank Professor Mitchell Wand, Ryan Culpepper, David Herman, and Theo Skotiniotis; our many discussions have helped me to refine and polish this work in ways I could not have done otherwise. Jesse Tov provided invaluable help in evaluating OOHaskell. Thanks also to Ryan Culpepper and Cormac Flanagan for suggesting my title and subtitle, respectively.

My employers at The MathWorks have very graciously allowed me the time and flexibility that I needed to finish this dissertation. I am also grateful that they have provided me with an opportunity to work on projects that involve ideas that, while perhaps not directly connected to this dissertation, are at least closely related.

Finally, I must also acknowledge my debt to many friends who provided personal support during my time in graduate school. They have spent far more time than I had a right to expect listening to me as I adapted to the academic research community and struggled to find my place in it. In addition to the classmates and professors above, I also owe a debt to many others: in no particular order, Robert Mack, Dean Lampros, Joe Teja, Terry Lewis, and of course the many members of the Back Bay Chorale and its Parish Committee. These people have provided support, empathy, fun, friendship, and most importantly an opportunity to reconnect with a part of myself I had long neglected.

Contents

List of Figures

Chapter 1

Introduction

Mainstream object-oriented languages like Java [20] and C# [32] contain a special *null* value, distinct from all other values in the language, that inhabits all (object) types. Linguistically, this *null* value serves two major purposes. First, it serves as a value for uninitialized fields, guaranteeing that references to those fields are type-safe. Second, programmers use it to encode a "missing" value or result, much as programmers use `NONE` in SML [33] and `Nothing` in Haskell [40]. Following Haskell, we refer to this role as *Maybe*.

Pragmatically, *null*-as-*Maybe* serves several functions:

- A field that represents an optional property is *null* when the property is not present.
- Programmers frequently use *null* as the default value of optional method parameters or optional method results. For example, a program may pass *null* as the first argument to

  ```
  java.util.Formatter.format(Locale, String, Object...)
  ```

  to indicate that the formatter should not perform localization. Similarly, implementations of the `java.util.Map` interface return *null* from their `get` methods to indicate that the desired key does not exist in the mapping.
- In a particularly significant instance of the previous case, programmers use fields that denote optional properties in the representation of cyclical or mutually referential data structures. Constructing cyclical data structures in mainstream OO languages requires programmers to initialize some fields in the cycle to temporary values until all of the objects in the cycle exist. In Java, *null* is the most convenient and thus the most frequent such temporary value.

All of these uses depend on *null* inhabiting every type.

1.1 The Problems with *null*

Because uses of *null* pervade Java and C# programs, and because *null* serves multiple purposes, programmers suffer from significant engineering problems. First, the error messages produced by the run-time system upon misuses of *null* do not generally contain enough information to help programmers find and fix the underlying bugs. Second, although there may be many invariants about how *null* flows through a program, programmers cannot state these invariants in the language itself, so they must assume all the effort of checking and enforcing them. Third, representing missing values and results with *null* is sometimes ambiguous, and this ambiguity leads to unnecessarily complicated interfaces and the potential for subtle bugs.

Uninformative Error Messages

When a Java program attempts to dereference *null*, the run-time system throws a null-pointer exception, which contains only the stack trace current at the time of the dereference, and no additional information. This is not, in general, sufficient for the programmer to find and fix the underlying cause of the problem, the injection of *null* into the flow of data.

Consider the program in figure 1.1. The `map` field associates strings with strings, with the additional invariant that the range of the table does not include *null*. The `main` method invokes the `initTable` to initialize this table, then calls `readTable` to perform computation on the table's contents. However, `initTable` violates the table's invariant in the second call to `put`, and as a result, the program stops with the indicated null-pointer exception during the evaluation of the highlighted expression.

This exception is not sufficient to help the programmer find and fix the error, the insertion of the invalid mapping into the table. The method which actually violates `map`'s invariant, `initTable`, does not appear in the exception's stack trace, because Java does not (indeed, cannot) detect the error within that method's dynamic extent.[1]

Java cannot detect the error at its actual location because it does not know the invariant governing *null*s and the table `map`. More generally, Java's run-time system can only detect the symptom of *null* bugs, i.e., that the program has attempted to dereference *null*, not the underlying cause, i.e., that *null* has flowed into a context that does not expect it. As the example above demonstrates, this flow can occur at any point in the execution of the program before the run-time system detects the failure. The exception's stack trace, therefore, is often irrelevant and misleading.

Finally, Java's error messages do not indicate whether the dereferenced *null* came from an uninitialized field or a missing value. This information could help program-

[1] The error message from Sun's Java compiler and JVM implementation on Mac OS X even provides the wrong line number, reporting the location of the error at the beginning of the statement, not the specific subexpression that causes the error.

```
import java.util.HashMap;

class C {
    private HashMap<String, String> map =
        new HashMap<String, String>();
    // INVARIANT: null is not in map's range

    private void initTable() {
        map.put("x", "valid mapping");
        map.put("y", null);          // invalid mapping!
    }

    private void readTable() {
        System.out.println(map.get("x").length() +
                           map.get("y").length());
    }

    public static void main(String[] args) {
        C c = new C();
        c.initTable();
        c.readTable();
    }
}
```

```
Exception in thread "main" java.lang.NullPointerException
        at C.readTable(C.java:14)
        at C.main(C.java:20)
```

Figure 1.1: An uninformative null-pointer exception

mers find and fix the underlying error, but because Java uses the same value in both cases, Java's built-in error detection and reporting mechanism cannot distinguish between them without additional information.

Inexpressible Invariants

Java does not provide a convenient way for programmers to state invariants about the flow of *null* within the programming language. In addition to making it impossible for the language to detect and report violations of these invariants, this also deprives programmers of an important form of documentation.

Consider again figure 1.1 from the previous section. The class definition contains a comment stating the invariant that *null* is not in `map`'s range. If Java allowed the pro-

grammer to specify this invariant in the program itself, rather than in documentation, then the run-time system could watch for violations of the invariant and signal them to the programmer. In this specific case, Java would be able to detect that the second call to `map.put` in `initTable` violates the invariant and signal an error accordingly.

Java programmers typically specify program invariants in the source code in one of two ways. Either they encode the invariant in the type system, or they use Java's `assert` statements. Programmers cannot, however, use the type system to state invariants concerning the flow of *null*, as Java's type system does not distinguish *null* from the other values of any given object type.

While programmers can use Java's assertions to state *null* invariants, this is not practical in many situations. First, the programmer must write the assertion statements manually, and this can result in a significant amount of code that clutters up method definitions and distracts the reader from the method's main functionality. Second, programmers can add assertions to pre-existing classes (for which they do not have the source code) only with difficulty. As an example, consider `HashMap` in our running example. If this code's author does not want to annotate *every* call to `map.put` in the program with an assertion (impractical in all but the smallest programs), then he must instead construct a wrapper class that implements `HashMap`'s interface and functionality but also checks the desired assertions. Such wrapper classes, however, do not generally integrate well into existing programs that already use the wrapped class.

Imprecise Indications of Missing Values

The second major use of *null* in Java programs is to indicate that an optional value or method result is not present. However, because *null* cannot unambiguously represent such a missing value, this encoding leads to severe drawbacks. As a concrete example, consider the class `java.util.HashMap`, in the current Java API [53]. The `get` method, according to its contract, returns either the value for the specified key or *null* when the specified key is not present, which is consistent with the conventional Java encoding of this condition. However, a result of *null* is ambiguous: it could mean either that the key is not in the table or that the key is present in the table and maps to *null*. In order to distinguish these two cases, the programmer must call the `containsKey` method to determine if the specified key exists in the table.[2] Figure 1.2 demonstrates this protocol.

The interface for `HashMap` must contain the `containsKey` method and is therefore larger and more complicated that it otherwise needs to be. Additionally, clients must follow a more complex protocol than those required by a more precise encoding of this "optional" result. Finally, this protocol requires two observations on the table,

[2]The `keySet` and `entrySet` methods also allow the programmer to determine if a given key is present in the table, but these methods suffer from the same problems that affect `containsKey`.

```
HashMap<String, Integer> map = ...;

if (map.containsKey("x")) {
    System.out.println("Key not found");
} else {
    Integer val = map.get("x");
    ...
}
```

Figure 1.2: The `java.util.HashMap` lookup protocol

highlighted in the figure above. Generally, the client expects these observations to be consistent, even in a concurrent program in which `map` is a shared data structure. It is the programmer's responsibility in such circumstances to wrap these observations in the necessary synchronization mechanism. Omitting the synchronization can lead to hard-to-find race conditions, with no warning from the language.

This ambiguity arises because programmers are attempting to use *null* to represent a disjoint union—either the value is present or it is not—but the *null*-based encoding of this union can never actually be disjoint. Java's type system requires that if *null* is a value of the type used to represent the union, then it must also be a value of both sides of the union, because only object types, and *all* object types, contain *null*. While Java provides the necessary facilities to define a true disjoint union, the fact that most programmers simply rely on *null* suggests that direct linguistic support for this particular union can help.

1.2 Thesis and Contributions

We claim that, with two relatively small changes to the Java programming language, we can remove *null* from the language, thus yielding significant engineering benefits to programmers. We also provide a migration path for existing programs. In support of this thesis, we propose and describe MIJAVA, a language based on Java that contains two primary contributions: a new object construction mechanism that removes the need for a default value for uninitialized fields, and an explicit disjoint union, analogous to Haskell's `Maybe`, that integrates well with other features of the language. We also describe a compiler from MIJAVA to Java that allows programmers to migrate Java classes to MIJAVA and thus take advantage of MIJAVA's benefits in existing programs.

1.3 Roadmap

Chapters 2 and 3 explore the problems with Java's current approach and describe MIJAVA's new features in detail. Chapter 4 discusses object creation with delayed initialization, including the important special case of constructing cyclical data structures, and how MIJAVA addresses this problem. Chapter 5 defines MIJMODEL, a formal model of the core of MIJAVA that includes our two proposed features and omits *null*, and proves a type soundness theorem. Chapter 6 defines the translation from MIJAVA to Java, and Chapter 7 describes our experience working with MIJAVA in the context of existing Java codebases. Finally, Chapter 8 discusses related works, and Chapter 9 concludes.

Chapter 2

Object Initialization

A Java program can easily refer to an uninitialized field with, in general, no indication that the results of the computation are likely to be incorrect. For safety, Java specifies that all uninitialized fields have the value *null* (or `0`, `0.0`, or `false`, depending on the field's type), but as we have argued, any run-time errors that result from the use of this "uninitialized" value are likely to be so far from the actual source of the error that the programmer cannot easily diagnose the problem.

Established Java coding practices call for classes to provide constructors that completely initialize every field in the class. However, the language does not enforce this requirement. In particular, a class may choose to define no constructors, leaving only the automatically-generated default constructor, which initializes no fields. If a class does provide a custom constructor, Java compilers generally provide no warning if the constructor fails to initialize one or more fields. Finally, even if every constructor in a class initializes all of the class's fields, it is still possible for the program to refer to one of the class's fields before it has been initialized, as we demonstrate in the next section.

Elements of arrays created without explicit initializers suffer from the same problem. Method parameters are immune, however, as the language both allocates and initializes these as part of the method-call mechanism, and the program cannot refer to parameters in a partly-initialized state. Local variables are also immune, as Java performs a static definite-assignment analysis [19, Ch. 16] to ensure that the program assigns to each local variable before referring to its value.[1]

In order to remove *null* from Java, we must modify object and array initialization to ensure that programs initialize all fields and array elements before referring to them. After a more detailed investigation of how programs can observe uninitialized fields,

[1] Various components of the run-time system, such as the garbage collector, may impose certain requirements on the values of otherwise uninitialized local variables. Because the definite-assignment analysis guarantees that the program cannot observe the values of these fields, we can ignore the implementation's invariants and focus instead on those of the program.

```
class Base {
    public Base() { System.out.println(this.m()); }

    int m() { return 0; }
}

class Derived extends Base {
    private String s;

    public Derived() { s = "Hello, world!"; }

    int m() { return s.length(); }
}
```

Figure 2.1: Interleaving initialization and computation in Java

we propose an object construction mechanism for MIJAVA, and we discuss possible array instantiation forms.

2.1 Observing Uninitialized Fields

Java programs can easily dereference uninitialized instance fields in a newly-created object. Even if all of the class's constructors initialize all of the object's fields, a program can still perform such a field reference by interleaving code that performs computation on the new object with the code that initializes its fields.

As an example of this interleaving, consider the class definitions in figure 2.1. Here, the `Base` class contains no fields, and its constructor merely invokes the method `m` and prints its result. Instantiating `Base` directly causes no problems. Its subclass `Derived`, however, is a different story. This class defines a single field `s` and a constructor that initializes `s`. Further, `Derived` overrides `m`, and the overriding definition invokes a method on the value of the field `s`.

As written, instantiating `Derived` causes a null-pointer exception at the highlighted expression, because of the order of operations specified for object initialization in Java. For this example, the key point is that `Derived`'s constructor immediately invokes its superclass's constructor, which in turn calls `this.m`. Because `this` refers to an instance of `Derived` (albeit an incompletely initialized one), Java's dynamic dispatch rules specify that this method call executes the body of `Derived`'s implementation of method `m`.

Since the `Derived` constructor does not initialize `s` until after the superclass constructor returns, `s` is still *null* during the execution of `Derived.m`, so the highlighted expression throws a null-pointer exception. This inversion of initialization and use occurs because the program invokes a method on `this` before the relevant object's constructors have completed.

Java allows class authors to specify initialization expressions in field declarations, making constructor assignments for those fields unnecessary. These are not, however, a practical solution to this problem, for two reasons. First, unlike statements in constructors, field initializers cannot be parameterized over arguments.[2] Second, Java's specification states that programs evaluate these field initializers only after the superclass's constructor returns. Therefore, supplying the field `s` with a field initializer in the example above does not change the program's behavior at all.[3]

Programs may also refer to uninitialized variables without invoking a method on `this` and without exploiting the complexities of subclass initialization. The code in figure 2.2 demonstrates that invoking a method on another class, passing `this` as an argument, can also cause undesired behavior, if the called method refers to its argument in a way that refers to an uninitialized field. In this specific example, `C`'s constructor invokes `Utils.m` on the newly-created object before initializing the field `s`. This method in turn calls `m` on the new, incompletely initialized instance of `C`, which causes a null-pointer exception at the highlighted expression in the body of `C.m`.[4]

Each of these examples demonstrates the dangers of interleaving initialization code with computation that operates on the object under construction. To prevent access to uninitialized fields, therefore, MIJAVA must enforce a separation between these two activities, allowing computation on an object only *after* the program has completely initialized it.

2.2 A New Object Construction Mechanism

The examples of the previous section demonstrate that interleaving object initialization with computation on the object under construction can easily lead to accesses to uninitialized fields and null-pointer exceptions. To prevent these exceptions, MIJAVA enforces a separation between these two activities, allowing computation on an ob-

[2] Because of the order of execution during object creation, these field initializers can in fact see the superclass's fields, the values of which can depend on arguments to the superclass's constructor. This strategy by itself is not a practical object initialization mechanism because it requires a protocol between superclass and subclass, and this protocol must extend all the way up the class hierarchy.

[3] Changing Java's semantics to evaluate field initializer expressions before invoking the superclass's constructor is not by itself sufficient. This merely inverts the problem, as a field initializer could refer to a field inherited from the superclass, which would not yet be initialized.

[4] This example demonstrates the problem by calling a static method, but calling an instance method on another object can also produce this behavior.

```
class C {
    String s;

    public C() {
        Utils.m(this);
        s = "Hello, world!";
    }

    public int m() { return s.length(); }
}

class Utils {
    public static int m(C c) {
        return c.m();
    }
}
```

Figure 2.2: Uninitialized fields through foreign method calls

ject only after the program has completely initialized it. The primary mechanism that MIJAVA uses to enforce this separation is a restriction on uses of the keyword `this`: MIJAVA constructors cannot refer to `this` during object initialization.

Because constructors cannot refer to `this` during field initialization, even implicitly, they cannot initialize fields with assignment statements. Instead, each constructor of a class *C* creates and returns a *C* "pre-object," which is a record of the information necessary to construct a fully-initialized instance of *C*—that is, it contains values for each of *C*'s fields. Pre-objects are not instances, however, and programs may not invoke methods on them or otherwise use them in a context that expects full instances. For example, given the following definition of the `Point` class,

```
class Point {
    public double x;
    public double y;

    public Point(double x, double y) { ... }
    public double distanceToOrigin() { ... }
}
```

a `Point` pre-object has exactly two fields, `x` and `y`, both of type `double`, but it does *not* support `Point`'s `distanceToOrigin` method, and an attempt to invoke this method on

a pre-object would cause the compiler to signal an error. When the program finishes building a complete pre-object, the run-time system reifies this pre-object into a full instance of the corresponding class, after which computation proceeds as in Java.

Figure 2.3 contains a feasible concrete syntax for MIJAVA's constructor mechanism; the highlighted expressions (explained below) demonstrate pre-object creation. In this figure, we define a library of classes to represent shapes on a plane. The `Shape` class contains a single `anchorPoint` field that specifies the shape's location, and a constructor that takes an initial value for this field. (We assume a `Point` class as defined above.) `Shape` has a single constructor that initializes the shape's anchor point with the supplied location.

The `Rect` class interprets the inherited `locn` field as the lower-left corner and adds a new field to represent the rectangle's upper-right corner. The programmer has equipped `Rect` with two initialization paths: the first merely initializes the rectangle's location, and the second also draws the new rectangle on a supplied `Canvas`.

The `ColorRect` class extends `Rect` to include a color attribute. The two constructors accordingly take the new rectangle's color as an additional parameter, and `ColorRect` also overrides `draw` to take the color into account.

`Shape`'s constructor demonstrates how to build a pre-object. The `preobj` form creates a pre-object corresponding to the containing class. That is, inside `Shape`'s constructor, the `preobj` form creates a `Shape` pre-object, and inside one of `Rect`'s constructors, it creates a `Rect` pre-object.

The `preobj` expression has two component parts, here separated by a semicolon. The first part, `super()`, invokes a constructor in the superclass, which in this case is `Object`'s only constructor and takes no arguments. This constructor invocation returns a pre-object for the superclass. The second part is a series of field initialization clauses for all of the fields defined directly within `Shape`.

MIJAVA imposes the following restrictions on constructors in order to enforce its static guarantees about uninitialized fields:

1. Each class must have at least one constructor. (Section 9.1 discusses the possibility of automatically synthesized constructors, similar to those currently provided by Java.)

2. Each constructor for a class *C* must return a *C* pre-object. (Other methods may also create and return pre-objects, but this is less common.)

3. Every `preobj` expression must provide values for every field declared directly in the containing class and no other fields.

4. No constructor may refer to `this`, even implicitly. This restriction prevents the interleaving of initialization and computation described above. (The occurrence

```
class Shape {
  Point anchorPoint;

  public Shape(Point locn) {
    return preobj { super(); anchorPoint <- locn; };
  }
}

class Rect extends Shape {
  // anchor point is lower-left corner
  Point upperRight;

  public Rect(Point lowLft, Point upRt) {
    return preobj { super(lowLft); upperRight <- upRt; };
  }

  public Rect(Point lowLft, Point upRt, Canvas c) {
    return Rect(lowLft, upRt);
  } postinit {
    this.draw(c);
  }

  protected void draw(Canvas c) { ... }
}

class ColorRect extends Rect {
  Color color;

  public ColorRect(Point lowLft, Point upRt, Color c) {
    return preobj { super(lowLft, upRt); color <- c; };
  }

  public ColorRect(Point lowLft, Point upRt, Canvas cvs, Color clr) {
    return preobj { super(lowLft, upRt, cvs); color <- clr; };
  }

  protected void draw(Canvas c) { ... color ... }
}
```

Figure 2.3: Leak-proof constructors in MIJAVA

of anchorPoint in the preobj expression in Shape's constructor is not a reference to this.anchorPoint. Instead, it is a label used to create the pre-object record, similar to the use of field labels in ML's record creation syntax.)

Rect's first constructor is similar to Shape's. The call to Shape's constructor produces a Shape pre-object, and the initializer for the upperRight field completes the Rect pre-object. This preobj expression does *not* contain an initializer for the inherited anchorPoint field; the Shape pre-object contains values for all such inherited fields.

As an example of the run-time behavior of the pre-object mechanism, consider the expression new Rect(p1, p2), where p1 and p2 are variables of type Point. Evaluation begins by invoking Rect's first constructor to compute a Rect pre-object. Rect passes p1 to Shape's constructor, which creates and returns a Shape pre-object in which the anchorPoint field has the value p1. When Shape's constructor returns, the preobj form in Rect extends the Shape pre-object with the field upperRight, which has the value p2. Rect returns this new, larger pre-object to the new expression, which converts the pre-object into a Rect instance. With this, instantiation is complete, and computation proceeds with the rest of the program.

Rect's second constructor demonstrates two additional features of the system, using pre-objects as first-class values, and writing code that requires this but must run at object creation time. Instead of creating a pre-object directly, this constructor instead calls the class's other constructor, in the highlighted expression, and returns its pre-object as the final result.

Rect's second constructor also has a "postinit block," which looks like a second body, separated from the first by the postinit keyword. This optional block may refer to this and to the constructor's formal parameters, and it may perform arbitrary computation on the object being constructed. MIJAVA executes postinit blocks only for effect; they do not return a value.

These postinit blocks allow classes to perform actions that require access to this during construction. This occurs frequently in Java programs; for example, in Sun's implementation of java.lang.Thread [52], each constructor registers the newly-created thread with its thread group, and this would be impossible if constructors could not refer to this.

Because postinit blocks may refer to this, MIJAVA delays the execution of these blocks until it has converted the pre-object into a full instance. This delay ensures that the program has provided initial values for *all* of the object's fields, including those defined in subclasses, before executing code that may refer to those fields. For instance, although ColorRect's second constructor begins by calling Rect's second constructor, MIJAVA delays the execution of Rect's postinit block until after the full ColorRect pre-object has been created and reified into a ColorRect instance. This

delay is necessary here in order to prevent a premature reference to `ColorRect`'s `color` field. The postinit block in `Rect`'s second constructor invokes `draw`, which in this case executes `ColorRect`'s implementation of the method, and this implementation refers to the value of the `color` field.

MIJAVA invokes the postinit blocks in the reverse order in which the program called the constructors. So, for instance, if `ColorRect`'s second constructor also had a postinit block, MIJAVA would execute `Rect`'s postinit before `ColorRect`'s. When a constructor invokes another constructor in the same class, the invoked constructor's postinit block is executed before that of the invoking constructor. MIJAVA's static guarantees do not require a particular evaluation order for postinit blocks, so long as the object is completely initialized first. The order described here, however, essentially executes postinit blocks starting with the base class and moving to the derived class, which is the order that Java programmers currently expect from their constructors. This order allows subclass postinit blocks to assume that their superclasses' representation invariants hold.

MIJAVA's constructors are much more flexible than Java's constructors. Java's syntax *requires* that the first statement in the body of a constructor must be an explicit constructor invocation. If the programmer does not provide one, Java inserts a synthesized call to the superclass's default (nullary) constructor. In contrast, MIJAVA allows constructors to perform (almost) arbitrary computation before the `preobj` expression that calls the next constructor in the sequence. MIJAVA's mechanism is even more flexible than the above examples demonstrate; a programmer may, for instance, write the following class:

```
class ColorRect extends Rect {
    java.awt.Color color;

    public ColorRect(Point lowLft, Point upRt,
                     java.awt.Color color, Canvas c) {
        if (lowLft.x >= 0 && lowLft.y >= 0) {
            return preobj { super(lowLft, upRt, c); color <- color; };
        } else {
            return preobj { super(lowLft, upRt); color <- color; };
        }
    }
}
```

In this class, the `preobj` expressions appear in the branches of an `if` statement.[5] As a result, this constructor invokes `Rect`'s third constructor when all the rectangle's co-

[5] MIJAVA's constructor mechanism is much more general than this example demonstrates. The branches of the `if` may also contain additional computation, or the constructor may use other control constructs, up to and including delegating the construction of the pre-object to another (static) method of the same class.

ordinates are positive and `Rect`'s first constructor otherwise. This kind of conditional constructor dispatch is simply not possible in Java.

The Java Language Specification is silent on why it requires constructors to begin with a constructor call, but we conjecture that it is an attempt to ensure that programs initialize superclass fields before referring to them. The first section of this chapter demonstrates conclusively that this restriction is not sufficient to prevent such uninitialized references, and the formal model of Chapter 5 demonstrates that it is not necessary, either. We believe strongly that MIJAVA—and languages in general—should not impose unnecessary restrictions such as this one, especially when the restrictions present significant barriers. Indeed, much of the complexity of the translation defined in Chapter 6 arises because of the need to accommodate this apparently needless requirement.

Delayed Representation Invariants

The construction mechanism that we describe here does represent a change in the establishment of object invariants for classes in inheritance hierarchies, and specifically the invariants on which subclass constructors may rely. Java programmers generally believe that a subclass constructor may assume that the superclass's invariants hold when the superclass's constructor returns, although as we have seen this is not in fact the case in many situations. In MIJAVA, on the other hand, because object construction proceeds in two phases, each class (or, rather, each constructor) has two sets of invariants: the first set is established during the creation of the pre-object, and the second set is not established until the postinit block executes. As a consequence, a subclass constructor may not assume that all of the superclass invariants hold for the superclass's pre-object.

While this difference certainly requires investigation, we do not expect it to be a significant issue in practice. For this change in initialization order to affect program behavior, a subclass constructor would have to observe that the fields in a pre-object fail to satisfy certain invariants. MIJAVA, however, provides no way for a program to inspect the properties of a pre-object before reifying it into a complete instance, at which point the postinit blocks begin to execute. By the time the program executes the postinit block for a particular constructor, all of the postinit blocks for later constructors in the invocation chain—including all those corresponding to constructors in superclasses—have completed, so the superclass representation invariants should in principle hold. Therefore, we do not believe that a program can observe this difference in initialization order.

2.3 Array Elements

In Java, array initialization proceeds much as object initialization does, except that arrays do not have explicit constructors. As with object fields, Java initializes all elements of a newly-allocated array to *null*, and it is the program's responsibility to initialize the array elements to the desired values through assignment statements. Java does allow the programmer to specify array initializers, but such initializers effectively hard-code the size of the array in the program and cannot be used with arrays whose size is not determined until run-time.

To ensure that all array elements are initialized before the program refers to them, MIJAVA must provide some array initialization mechanism beyond Java's array initializers. One possibility is a `buildArray` library method that takes two arguments: the size of the array and (the Java encoding of) a closure that maps the new array's indices to the corresponding initial values. Because the syntax for closures is so cumbersome in Java, however, this is hardly a practical solution. In the absence of a light-weight syntax for closures, an array comprehension form is the realistic alternative.

Constructing arrays from values read from an input source (keyboard, disk, network) is more difficult, and additional work is necessary to find the right abstractions. Array comprehensions and `buildArray` could work, as long as MIJAVA specifies the order in which they compute the element values. Alternatively, programmers may elect to assemble the values into a `List` or `Vector`, and then use the existing `Component.toArray` library method to produce the array.

2.4 Non-Object Variables

The discussion above concentrates solely on variables of object type, but Java also specified default initial values for variables of primitive type (`boolean`, `long`, `double`, and their subtypes). Unlike *null*, however, these default values are behaviorally members of their respective types. While invoking a method on an uninitialized field of object type raises a null-pointer exception, applying a primitive operation to an uninitialized primitive variable simply produces an answer, with no indication that the answer is most likely incorrect. As a result, such bugs can go unnoticed for long periods of time and can be difficult to diagnose and fix when found.

Because MIJAVA's constructor and array initialization mechanisms apply to primitive types as well as object types, its guarantees extend even to fields of non-object type. That is, each field is initialized intentionally, not by accidental default. Therefore, we can ensure that no program references an uninitialized variable of primitive type and thus eliminate the errors described above.

2.5 Design Alternatives

MIJAVA's constructors are very general and therefore somewhat complicated. Other languages, like Spec# [4, 13], offer alternatives that are simpler but more restrictive. In Spec#, constructors look much more like Java's, in that they are a normal sequence of assignment statements, with a required call to one of the superclass's constructors, written `base(...)`. Unlike Java, however, this `base` call may appear anywhere within the body of the constructor, subject to the following restrictions:

- Each control path through the constructor must initialize all (non-inherited) fields at least once before calling `base`;[6] and
- No control path may refer to `this` before calling `base`. (The implicit occurrences of `this` in the field assignments are allowed, but the implicit `this` in references to inherited fields are not.)

A straightforward static analysis checks that constructors satisfy the above requirements, which appear to be sufficient to guarantee, statically, that no program accesses an uninitialized field.[7] As we discuss in Chapter 8, though, MIJAVA's constructors are more general. And while MIJAVA and (this aspect of) Spec# were developed in parallel, we provide three important features that they do not: we demonstrate how to add uninitialized-field guarantees to an existing language, rather than developing an entire new language from scratch; we provide a migration path from Java to MIJAVA; and we have succeeded in completely banishing *null* from the language.

2.6 Scaling to the Full Language: Exception Handling

Thus far, we have considered MIJAVA's constructor mechanism in isolation, and the model of Chapter 5 includes only core language features. To complete the discussion of this proposed change, we must investigate how we would scale MIJAVA's constructors and static guarantees to the full language and how they interact with other features of Java and with the idioms that Java programmers use in large projects. In particular, we must discuss whether exceptions could violate MIJAVA's guarantees.

MIJAVA constructors may throw exceptions rather than returning normally, in both the pre-object creation and the postinit sections. In both cases, using Java's existing exception semantics gives reasonable behavior and preserves MIJAVA's static guarantees about uninitialized field access.

If the pre-object creation section throws an exception, the run-time system transfers control to the nearest dynamically-enclosing handler. Because the program raised the

[6] Spec# also supports "delayed" constructors, which relax this restriction; see section 8.1 for details.
[7] We have not seen a formal proof of this statement in the Spec# literature.

exception before converting the pre-object into a full instance, no references to the partially-constructed object exist. Therefore, the program cannot refer to uninitialized fields in this object, and MIJAVA's guarantees still hold. Because pre-objects are first-class values, however, the constructor *could* allow a reference to the pre-object (or the superclass's pre-object) to "leak" out to the handler, either by placing a reference to the pre-object into the exception object, or by storing it in some mutable field whose scope includes the handler. However, MIJAVA's type system ensures that any such pre-object contains values for all of the necessary fields, so the program may safely reify it and be assured that the resulting object is fully initialized.

If a postinit block throws an exception, then all of the relevant field initialization sections have provided values for the object's fields, and the run-time system has reified the pre-object into a full instance. However, the program may not have established all of the object's representation invariants before throwing the exception. Unlike the previous case, however, the incomplete object may be visible to the rest of the program. A postinit block may have stored `this` in a variable visible to the handler's context, or it may have put `this` into one of the exception object's fields. In such cases, it is up to the programmer to recognize that the object may not behave as expected and to take appropriate action. This does not invalid MIJAVA's guarantees, because we claim only that MIJAVA programs must initialize all fields before referring to them.

Even with this potential problem, MIJAVA still represents an improvement over existing Java programs. Java constructors may throw exceptions before establishing all of the object's representation invariants, and if `this` leaks out to the rest of the program as above, Java programs can manipulate the incompletely-initialized object just as easily as in MIJAVA. Further, a Java constructor may throw an exception and leak `this` before initializing all of the object's fields, which is not possible in MIJAVA.

Chapter 3

A New *Maybe*

To allow programmers to express *Maybe*—that is, the possibility for "missing" values or results—without *null*, we add a parameterized `Maybe` type to MIJAVA, similar to the `Maybe` type in Haskell and the `Option` type in SML. The introduction of this type, which we write in Java-like syntax as `Maybe<T>`, changes the interpretation of types in MIJAVA: the type `String` now contains *only* instances of the class `String`, and not *null* or any other distinguished value. Programmers must write `Maybe<String>` to indicate that a particular context allows *null*.

Introducing a `Maybe` type inspired by functional programming poses the challenge of integrating this type with the rest of MIJAVA. In particular, using Haskell-like pattern matching to decompose `Maybe` values would be a particularly bad fit, as it would require making significant syntactic and semantic changes to the language that would be of limited utility.[1] Instead, we look for a more Java-like interface to this type.

3.1 *Maybe* in Java

The `Maybe` type implements the class interface specified in figure 3.1, although MIJAVA implementations may choose not to represent `Maybe` as a class. The first constructor injects a value into the `Maybe` type, and the second constructor creates a `Maybe` value that indicates that the optional property is not present. The `isSome` method indicates whether a value is present, and the `valOf` method returns the underlying value, raising a `MaybeException` when invoked on a `Maybe` object that represents a missing value. For convenience, we define two additional keywords, `some` and `none`, which correspond to `Maybe`'s two constructors, respectively.[2]

[1] Without additional effort along the lines of Scala [35], the pattern matching mechanism would apply only to `Maybe` types.

[2] In an actual implementation, we could even spell `none` as `null`, as the two values are used in many of the same contexts and generally behave the same, although this new `null` would in fact respond to certain method calls. For clarity, however, we leave `null` at its current meaning in Java and use `none`

```
class Maybe<T> {
    public Maybe(T val) { ... }
    public Maybe() { ... }

    public boolean isSome() { ... }
    public T valOf() { ... }
}

class MaybeException extends RuntimeException { }
```

Figure 3.1: The `Maybe` interface

Because the pattern matching forms of functional languages do not lend themselves well to Java or other object-oriented languages, we instead provide implicit coercions between the types `T` and `Maybe<T>` for any `T`. These coercions are similar to those between, for instance, `int` and `Integer` in Java 5. While the coercion from `T` to `Maybe<T>` always succeeds,[3] the reverse may fail with a `MaybeException`. This coercion is essential for clarity, brevity, and programmer convenience. Without it, the programmer would have to pollute the code base with many calls to `isSome` and `valOf`. (These calls to `isSome` correspond directly to the `(obj != null)` checks that Java programmers are *supposed* to make, but in reality most programmers prefer not to go to the effort.)

Without additional constraints, MIJAVA's insertion of coercions is underspecified, and this ambiguity can produce programs that behave in unexpected ways. As an example, consider the following partial program:

```
class C {
    void m(Maybe<String> s) { ... }
    // assume no overloading on method m()

    void m2() {
        Maybe<String> x = ...;

        m(x);
    }
}
```

for the `Maybe` constructor in this dissertation.

[3] This coercion may in fact fail with an `OutOfMemoryError`, just like coercions from `int` to `Integer`, but we ignore this possibility here.

In the highlighted method call expression, the program uses `x`, a variable of type `Maybe<String>`, in a context that expects an expression of the same type. Here, the programmer presumably expects the language to leave the expression unchanged and not insert any coercions. However, MIJAVA could insert other coercions for `x`, such as `some(x)`, `valOf(some(x))`, or `some(valOf(x))`.[4] Some possible coercions, like the first above, we reject as producing ill-typed programs. In this particular example, the resulting expression has type `Maybe<Maybe<String>>`, which does not match the context. The second possibility yields an expression of the correct type, but at the cost of the unnecessary dynamic check in `valOf`. The third possibility demonstrates the danger of these "padding" coercions: while the resulting expression has the correct type, the inserted runtime check throws a `MaybeException` whenever `x` is `none`, even though the original program (with no coercions) completes successfully in this case.

To avoid this problem and to make coercion introduction deterministic, MIJAVA always avoids padding when inserting coercions. For the program above, then, the language inserts no coercions. To convert an expression of type `Maybe<String>` to type `Maybe<Maybe<String>>`, MIJAVA always inserts a single `some` coercion, and so on. With this restriction, and as long as the compiler can unambiguously determine what type a context expects, coercion introduction is deterministic. Section 3.2 addresses those cases where ambiguity arises from the context.

While it may seem at first glance that these coercions give MIJAVA's `none` value all of the drawbacks of Java's *null*, including delayed detection of null-pointer references, there is a key difference, best illustrated with an example. Consider a `Map` that associates integers with strings, with the additional invariant that every integer in the domain of the structure maps to an actual string and not to *null*. Figure 3.2 demonstrates how one would write this in Java 5 and in MIJAVA:

- In the Java code on the left, the programmer cannot state the invariant within the language. Therefore, the second call to `put` completes normally, even though it violates this invariant. The program cannot detect this error until it attempts to *use* (not just retrieve) the incorrect value. Further, although this example places the error and its detection close together textually, they may be arbitrarily far apart in actual code.

- On the right, in MIJAVA, the use of `String` instead of `Maybe<String>` in the declaration of `m` specifies to the compiler that `m`'s range may contain only `String`s. Because the arguments in both calls to `m.put` are of type `Maybe<String>`, the compiler inserts coercions, and the second coercion fails with a `MaybeException`. In MIJAVA, then, the run-time system throws an exception as soon as the program violates this invariant, instead of much later, when the program executes code that depends on an invariant previously violated.

[4] For clarity, we assume here a `valOf` keyword, such that `valOf(x)` is equivalent to `x.valOf()`.

```
java.util.Map<Integer, String> m = ...;

m.put(13, "Baker's Dozen");
m.put(42, null);  // oops!

System.out.println(m.get(13).length());
System.out.println(m.get(42).length());
    // NullPointerException!
```

```
java.util.Map<Integer, String> m = ...;

m.put(13, some("Baker's Dozen"));
m.put(42, none);   // MaybeException

System.out.println(m.get(13).length());
System.out.println(m.get(42).length());
```

Figure 3.2: The *Maybe* coercion at an abstraction boundary

MIJAVA's use of *Maybe* also allows programmers to design simpler data structure interfaces than in Java without loss of expressiveness. Consider the `Map` interface again, which would have the following (partial) signature in MIJAVA:

```
public interface Map<K, V> {
    Maybe<V> get(K key);
    ...
}
```

The `get` method would return `none` to indicate that the specified key is not in the map. In a map of type `Map<K, Maybe<V>>`, `get` could return one of three kinds of values:

1. `none`, to indicate that the key is not present;
2. `some(none)`, to indicate that the key is present and maps to `none`; and
3. `some(some(`v`))`, to indicate that the key maps to the value v.

In Java, `Map.get` returns *null* in both the first and second cases, so `Map`'s client must call the map's `containsKey` method to distinguish between them.[5] Because the Java map requires two observations to distinguish these cases, subtle concurrency bugs can arise if the programmer neglects to perform the necessary synchronization.

Additionally, this example demonstrates the importance of nested *Maybe* types, like `Maybe<Maybe<String>>`. In a language that cannot express such types (such as languages that approximate *Maybe* with non-null reference types) the result of the `get` method in the `Map` interface defined above would discard information whenever `Map` is used at a *Maybe* type. For instance, the `get` method in `Map<Object, Maybe<String>>`

[5]The `MaybeException` raised by a failed coercion from type `Maybe<Maybe<T>>` to `T` is not enough to distinguish between the two cases, so the programmer must instead fall back on the `isSome` method. Unlike Java, however, the return value itself suffices to distinguish these two cases, leading to a cleaner `Map` interface.

could return only `Maybe<String>`, and programs could not rely on its result to distinguish between cases 1 and 2 above.

The behavior of `Map.get` also provides a new answer to the question posed by other statically-typed languages about how functions like `get` should signal failure. In Haskell, operations that may not succeed typically return `Maybe t`, and the programmer uses the `Maybe` monad and the `do` syntax to evaluate several such expressions in sequence and abort the sequence if any of the intermediate operations fail. OCaml, on the other hand, does not have specialized monad syntax, so in the standard idiom such functions throw an exception. This spares programmers the syntactic overhead of checking each `option` result in turn. Neither language's standard practice has a clear advantage over the other: Haskell's solution requires additional syntax, and OCaml's makes the case of a single `option` operation more expensive, syntactically. With MIJAVA's behavior, we get the best of both worlds, and the `get` method allows both lightweight syntax for a single `Maybe` operation and sequences of `Maybe` operations without checks after each one.

3.2 Scaling *Maybe* to the Full Language

The model of Chapter 5 specifies the behavior of *Maybe* types in the simplified core of Java. To complete the proposal, we must discuss how it interacts with the features of the full language, including `Object` as unique supertype, Java's system of generics, contexts that lead to ambiguous coercions, and `Maybe`'s effect on existing libraries.

Maybe and `Object`

Interpreting the type `Object` in MIJAVA in the manner described at the beginning of this chapter is not consistent with Java's use of `Object` as the unique maximum of the subtyping order. (For this section, we ignore non-object types like `int` and `double`.) If, as we said earlier, the type `Number` contains only instances of the class `Number` and its subclasses, but not `none`, then the type `Object` should include only instances of the class `Object` and its subclasses, but not `none`. Indeed, this interpretation is necessary in order to achieve the run-time enforcement of the *Maybe* invariants described in the previous section. As a consequence of this interpretation, `Maybe<T>` *cannot* be a subtype of `Object`, even though it may be implemented as an object, i.e., as an instance of some class. The same logic also implies that `Maybe` cannot implement any interfaces.

Additionally, it is impossible to designate a single type as the unique supertype and retain the desired expressiveness of the MIJAVA type hierarchy. In particular, we cannot designate `Maybe<Object>` as the maximal type for two reasons. First, `Maybe<String>` is *not* a subtype of `Maybe<Object>`. Second, it leaves no room for `Maybe<Maybe<Object>>`, and so on. While MIJAVA's coercions can convert values of

any type to (a subtype of) `Object`, these coercions are not the same as Java's upcasts, as they can fail at run-time.

Adding *Maybe* to MIJAVA thus means that there is no single type at the root of the subtyping hierarchy. Further evaluation work is required to determine if this has significant practical drawbacks, but we do not expect serious problems due to the limited ways in which programmers use `Object`.

While other object-oriented languages allow a multi-rooted class hierarchy, we must briefly check that introducing this complication into Java does not interfere with the uses of `Object`. The class `Object` serves the following functions in Java programs:

1. It defines operations that are available on all (non-primitive) values in the language: `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`.

2. It allows for heterogeneous data structures, such as `List<Object>`.

3. It serves as the default bound on generic type arguments.

All the methods currently defined on `Object` have reasonable implementations for `Maybe` values, including `none`, so MIJAVA could easily provide those operations.[6] For the second point, we conjecture that when programmers want heterogeneity, they generally want *bounded* heterogeneity: rather than absolutely any value, they want any value whose type is a subtype of some specific type `T`. Finally, instead of `Object` as the default bound on unqualified type variables, MIJAVA could instead assume *nothing* about those types. Alternatively, because `Maybe` values satisfy `Object`'s (implicit) interface, MIJAVA could assume this interface for unqualified type variables. This is, however, a potentially awkward exception to Java's type rules, as it essentially assumes the existence of a maximal type but does not allow the programmer to name it.

Even with the issues raised above, the lack of a unique supertype does have certain benefits. Because programmers cannot simply insert values of `Maybe` type into a data structure that expects `Object`, they must think consciously about where `none` is allowed in the program and where it is not, and they must express that decision in the type system.

Maybe and Java's Generics

The model of Chapter 5 does not specify whether the `Maybe<T>` type constructor behaves like other parameterized types in Java with respect to type erasure and so forth, although the concrete syntax that we use in this chapter strongly suggests that it does.

[6] This remains true even if `Maybe` is not implemented as a class. Because Java already has `Class` instances for primitive types like `int`, we could easily create a new `Class` instance for `Maybe`.

If MIJAVA treats `Maybe` as a normal generic type, then Java's restrictions on generic types and overloaded methods would also apply to `Maybe`. In particular, the following interface definition would not compile, as the three methods would have exactly the same type signature for the purposes of overloading:

```
interface I {
    void m(Maybe<String> s);
    void m(Maybe<Record> r);
    void m(Maybe<Maybe<Record>> r);
}
```

Java programmers have developed standard techniques to address this problem. The easiest solution, of course, is to give the methods different names. If that is undesirable or impossible (as with constructors), then the programmer can add additional "dummy" parameters that serve only to distinguish otherwise equivalent methods.

In addition, treating `Maybe` as other Java generic types would allow programmers to refer to the raw type `Maybe` with no type arguments. As a result, `Maybe` would be susceptible to heap pollution [20, §4.12.2.1], like other generic types, and a variable of type `Maybe<Number>` could refer to a value that was instantiated as a `Maybe<String>`, as in the following example:

```
Maybe<Number> mn = some(new Integer(42));
Maybe bare_m = n;
Maybe<String> ms = bare_m;  // generates "unchecked" warning

System.out.println(ms.valOf());
    // generates ClassCastException at run-time
```

Fortunately, because `Maybe` instances are immutable, we avoid the additional problems of mutable generic types, in which it is possible to insert an instance of `String` into an object that was originally created as an instance of `ArrayList<Number>`.

Heap pollution would unfortunately allow programmers to bypass MIJAVA's type system and its enforcement of the invariants about the flow of *null*, as the following example demonstrates:

```
Map<String, String> map = ...;
Map bareMap = map;
Map<String, Maybe<String>> nullableMap = bareMap;
    // generates "unchecked" warning, which programmer may ignore

nullableMap.put("foo", none);   // also "unchecked" warning

System.out.println(map.get("foo"));  // ClassCastException
```

Because this code inserts the record for "foo" into nullableMap, which is identical to map but has a type that explicitly allows none in the range of the table, MIJAVA does not insert a coercion around put's second argument. The program thus violates the invariant on map, but the runtime system cannot detect this until the program attempts to read the invalid entry, in the call to get. This behavior is not limited to Maybe but rather follows from the specified semantics for generics in Java, and we cannot fix this problem without dramatically altering those semantics.

If, on the other hand, we treat Maybe<T> as a special case rather than a typical Java generic type, we avoid the problems discussed above, but at the cost of complicating the language definition. Additionally, this exception would likely have high engineering costs, as it would require changes to both the compiler and the Java Virtual Machine. As we discuss below, however, *Maybe* precludes backward compatibility, regardless of whether it is a standard Java generic type, so changes to the JVM may be acceptable under the circumstances.

Maybe and Ambiguous Coercions

As discussed above, MIJAVA never "pads" inserted *Maybe* coercions. Treating coercions in this fashion reduces, but does not eliminate, situations that lead to ambiguous coercions. Coercions can still be ambiguous in certain contexts: arguments to overloaded methods, the branches of conditional expressions, and the first argument to instanceof expressions. In each case, the larger expression does not uniquely specify the type of its subexpressions.

For example, an invocation of an overloaded method may lead to an ambiguous coercion, as in the following example:

```
class C {
    public static void m(D x) { ... }
    public static void m(Maybe<Maybe<D>> x) { ... }
}

Maybe<D> d = none;
C.m(d);
```

(Note that C's two methods *are* distinct for the purposes of overloading, since the argument types erase to D and Maybe, respectively.) The application on the final line is ambiguous: should the compiler coerce d up to Maybe<Maybe<D>> or down to D?[7]

[7]This example also demonstrates that MIJAVA must insert coercions *before* erasing types. If a program calls a method that expects a single argument of type Maybe<String> and the actual parameter has type Maybe<Maybe<String>>, then MIJAVA must insert the coercion in order to protect the program's invariants. After erasure, though, both formal and actual arguments have type Maybe, so no coercion would appear necessary.

One possible solution is to declare that MIJAVA always resolves these ambiguities in the same direction: always adding `Maybe`s, or always removing them. Indeed, adding `Maybe`s is presumably preferable, since this operation cannot fail. Due to our experience with the consequences of similar explicit coercions in other languages, however, we suspect that a better solution is to have MIJAVA detect such ambiguities and signal a compile-time error to the programmer, who would then resolve the ambiguity by inserting casts or otherwise making the desired type of the expression explicit.

A similar case arises with Java's conditional operator. In the excerpt

```
class C {
    public static D m1() { ... }
    public static Maybe<D> m2() { ... }
}

... (e ? C.m1() : C.m2()) ...
```

MIJAVA must clearly insert a coercion on (at least) one branch of the conditional expression. It is not obvious, however, whether MIJAVA should coerce the first branch to `Maybe<D>` or the second branch to `D`.

In addition to the strategies discussed above for resolving overloaded methods, MIJAVA could attempt to disambiguate based on the type expected by the conditional expression's context, in cases where the context's expected type is itself clear. This does, however, differ from Java's type system, in which neither the type of a conditional expression nor the `int`/`Integer` coercions it introduces currently depend on the expression's context. Exploring the consequences of this strategy requires further work.

Finally, the `instanceof` operator defines a context that could lead to ambiguous coercions. If `d` has type `Maybe<Number>`, should `(d instanceof Integer)` perform the coercion, possibly with a `MaybeException`, or should it unconditionally evaluate to `false`? Because `Maybe` does not admit subtypes, and because MIJAVA does not include the value *null*, there is no need to check at run time whether a given object is an instance of `Maybe`, as the type system can prove this statically.[8]

One solution, then, is to simply prohibit the use of the `instanceof` operator with `Maybe` types. Programmers can still determine if a given `Maybe<Number>` contains an `Integer` object by inserting an explicit cast to `Number`, forcing the coercion. As a consequence, given the method definition

```
boolean m(Maybe<Number> n) {
    return ((Number) n) instanceof Integer;
}
```

[8] Java 5 permits only raw types in `instanceof` expressions, so it would be impossible to check, for example, whether an object is an instance of `Maybe<String>`.

the expression `m(none)` terminates with an exception, because the highlighted cast causes the compiler to insert a coercion, which fails.

Another alternative would modify the `instanceof` operator to insert necessary coercions but still prohibit the appearance of `Maybe` types on the right-hand side of the operator. (As a consequence of this restriction, `instanceof` would only coerce from `Maybe<T>` to `T` and not in the opposite order.) In keeping with Java's current behavior, where `null instanceof T` is false for any `T`, any failed coercion would cause the entire `instanceof` operation to return `false` rather than throwing an exception.

Further work is required to determine which of these two behaviors is more practical. The first alternative, however, looks more promising, as the second represents a significant exception to the semantics of MIJAVA's coercions. Additionally, experience has shown that masking errors, as in the second alternative, usually makes debugging and maintenance significantly harder.

Effects on Existing Java Libraries

Because this chapter's proposal for *Maybe* requires re-interpreting existing Java types (such that `none`, or `null`, is no longer a value of the `String` type), the types on the classes and interfaces of the Java standard library are no longer correct for MIJAVA. Adapting the Java libraries to MIJAVA would thus require pervasive changes. While an analysis similar to that proposed by Ekman and Hedin [11] could help to automate part of this transition, such changes would unavoidably break backward compatibility with previous versions of the Java language and libraries.

3.3 Design Issues

We considered and rejected two alternative designs for MIJAVA's representation of *Maybe*. The first is a slight variation on the proposal described above, in which the `Maybe` type is covariant. As we explain below, however, doing so could introduce additional problems with ambiguous coercions, primarily in the context of method overloading.

The second alternative is a completely different way of representing *null*-as-*Maybe*. In this scheme, rather than adding a parameterized type constructor to add *null* to any type (and simultaneously restrict the type's interface), we instead give each class in the program a dedicated *null* representation that implements the same interface as the original class. While this strategy is closer to the core spirit of object-oriented programming, it interacts poorly with the rest of the language, including such core features as subclassing.

Maybe and Covariance

Because `Maybe` objects are immutable, we believe that MIJAVA could safely make the `Maybe` type covariant in its type argument. This would represent a significant departure from Java's semantics, in which generic type constructors are invariant, but the combination of added programmer convenience and a simpler model suggested to us that this idea was worth pursuing. As we demonstrate below, however, covariant `Maybe` interacts poorly with method overload resolution, so the `Maybe` type is invariant in MIJAVA, like other generic types.

Consider the following class definition:

```
class C {
    public static void m(Maybe<Number> x) { ... }
    public static void m(Number x) { ... }
    public static void m(Integer x) { ... }

    public static void main(String[] args) {
        Maybe<Integer> i = some(new Integer(3));
        C.m(i);
    }
}
```

In the shaded method call expression, the argument's static type is `Maybe<Integer>`. If `Maybe` were covariant, then MIJAVA could resolve the overloading in either of two ways:

1. MIJAVA could treat the `Maybe<Integer>` as a `Maybe<Number>` by subsumption and call the first definition of `m`.

2. MIJAVA could insert a coercion from `Maybe<Integer>` to `Integer` and call the third definition of `m`.

(The third possibility, coercing `Maybe<Integer>` to `Integer` and then treating the result as a `Number`, is inconsistent with Java's existing overload resolution rules, which specify that the language always chooses a method with a more specific type over a method with a less specific type.)

This ambiguity arises because of the combination of implicit coercions and subtyping. We could introduce disambiguation rules to resolve this, such as specifying that subsumption always takes precedence over coercion. The example of C++ [51], however, is instructive in this case. C++ has both (programmer-defined) implicit coercions and subsumption, and it does use complex disambiguation rules to address this problem. As a result, its method overload resolution frequently causes behavior that is confusing even to experienced C++ programmers.

To avoid this problem, then, MIJAVA treats `Maybe` as invariant, like other generic types. As a result, the example above is clear: the actual parameter in the highlighted expression has type `Maybe<Integer>`, which is not a subtype of `Maybe<Number>`, so the only applicable method is the third definition, with the addition of a coercion.

It is still possible to add explicit casts to the example's highlighted expression to cause it to call one of the other two methods:

- Writing `C.m((Number) i)` coerces `i` to `Integer` and casts the result to `Number`, so the expression calls the second definition of `m`.
- Writing `C.m(some((Number) i))` calls the first definition of `m`, by coercing `i` to `Integer`, casting that to `Number`, and then constructing a `Maybe<Number>` out of the result.

Keeping `Maybe` invariant also simplifies type-checking for conditional expressions, like `e1 ? e2 : e3`. The type of this expression is, in many cases, the least-upper bound of the types of `e2` and `e3`, and allowing covariance in `Maybe` would complicate this semantics in much the same way as with overload resolution. For instance, consider the code excerpt below:

```
Integer i = ...;
Number n = ...;
Boolean b = ...;

... (b ? some(i) : n) ...
```

If `Maybe` were covariant, the type of the expression would be ambiguous: MIJAVA could coerce `n` to `Maybe<Number>`, for an overall type of `Maybe<Number>`; or it could coerce `some(i)` to `Integer`, for an overall type of `Number`. With `Maybe` invariant, only the second strategy leads to a well-typed expression.

If one branch has type `Maybe<`C`>` and the other type `Maybe<`D`>`, where $C \neq D$, then MIJAVA must coerce the `Maybe`s away for both branches, regardless of the relationship (if any) between C and D, because `Maybe` is invariant. This applies even if the context expects a `Maybe` type, in which case MIJAVA must coerce the result of the entire conditional expression back to the `Maybe` type, leaving open the possibility that the first coercion can fail at runtime. Programmers can avoid this by casting one or both expressions to the desired type.

Making `Maybe` invariant does require programmers to insert explicit casts in some contexts in which casts are unnecessary in Java without `Maybe`. Consider the example method, in both Java and MIJAVA, in figure 3.3. In the Java variant, on top, the programmer can return an `Integer` from a method whose declared return type is `Number`, through subsumption due to subtyping. The equivalent method in MIJAVA, however, requires an explicit cast, as shown on the bottom. Without the cast, `return`'s argument

```
Number stringToNum(String s) {
    try {
        return new Integer(Integer.parseInt(s));
    } catch (NumberFormatException e) {
        return null;
    }
}
```

```
Maybe<Number> stringToNum(String s) {
    try {
        return some((Number) (new Integer(Integer.parseInt(s))));
    } catch (NumberFormatException e) {
        return none;
    }
}
```

Figure 3.3: Returning a subtype in Java and MIJAVA

would have type `Maybe<Integer>`, which is not a subtype of `Maybe<Number>`. Casting the `some` expression to `Maybe<Number>` instead of the cast above does not compile, as Java does not allow casts between unrelated types.

Alternatives to *Maybe*

In addition to the design described in the first part of this chapter, we considered another representation of *null* that, at least initially, appears to fit better into a class-based object-oriented language like Java. As we demonstrate here, however, the idea has several drawbacks that make it unsuitable.

Rather than a single *null* value that inhabits every type, as in Java, or a separate type intended solely to represent *null*, we instead create separate *null*-like values for each type. These specialized values inhabit their respective types and thus implement all of the same methods. These methods, however, perform actions appropriate to *null* in a manner specified by the class author.

As an example, consider the `List` implementation in figure 3.4. The highlighted clauses depart from existing Java syntax, and we explain these below. With this definition, the programmer defines not one but four separate (though related) classes in a single definition: an abstract base class `List` with the specified interface; two implementing subclasses, `RealList` and `NullList`; and a separate `NullListException`

```
class List implementations RealList, NullList {
    private Object head;
    private List tail;

    public List(Object hd, List tl) {
        head = hd;
        tail = tl;
    }

    public Object first() { return head; }
    public List rest() { return tail; }
    public int length() { return 1 + tail.length(); }
        when null { return 0; }
}
```

Figure 3.4: A `List` implementation

class. The `implementations` clause merely specifies the names of the two subclasses.[9] (For clarity, we refer to the programmer's definition of `List` as $\texttt{List}_p$ and the generated abstract class as $\texttt{List}_a$.) With this strategy, programmers would not use the literal `null`; instead, they would use instances of `NullList`.

The generated classes appear in figure 3.5. The abstract class $\texttt{List}_a$ defines the public interface of $\texttt{List}_p$, excluding all fields. The class `RealList` contains all of the fields and method implementations from $\texttt{List}_p$. The `NullList` class also implements the methods of $\texttt{List}_a$ to throw a specialized `NullListException` that provides information beyond what a null-pointer exception would contain in a similar context. For the `length()` method, though, throwing an exception is not desirable, as there is a reasonable and indeed necessary implementation of this method on *null* lists. The `when null` clause from figure 3.4 allows the programmer to specify the behavior of the method for `NullList`.

This proposal is, in some sense, closest to the ideal of object-oriented programming, which specifies that dynamic dispatch should be the primary (and in extreme forms, the only) mechanism for computation, including conditionals. In this case, the check to see whether a given $\texttt{List}_a$ is *null* is subsumed by the normal dynamic method dispatch mechanism. This encoding of *null*, however, suffers from serious incompatibilities with other features of Java:

[9] In a real implementation, the programmer would be able to specify the name of the exception class, just as with the two subclasses.

```
abstract class List_a {
    public abstract Object first();
    public abstract List_a rest();
    public abstract int length();
}

class RealList extends List_a {
    private Object head;
    private List_a tail;

    public RealList(Object hd, List_a tl) {
        head = hd;
        tail = tl;
    }

    public Object first() { return head; }
    public List_a rest() { return tail; }
    public int length() { return 1 + tail.length; }
}

class NullList extends List_a {
    public Object first() { throw new NullListException("first"); }
    public List_a rest() { throw new NullListException("rest"); }
    public int length() { return 0; }
}

class NullListException extends RuntimeException {
    public NullListException(String msg) { super(msg); }
}
```

Figure 3.5: Classes generated from figure 3.4

1. This implementation strategy complicates access to private members in other objects of the same type. If $\texttt{List}_p$ contains a private method `m`, then code in $\texttt{List}_p$ may refer to, e.g., `tail.m`. However, because private methods cannot be abstract, `m` cannot be declared in $\texttt{List}_a$ but may appear only in `RealList` and `NullList`. Therefore, to access such fields and methods, programmers must insert an explicit check and downcast.

2. This implementation of $\texttt{List}_a$ and its subclasses is incompatible with subtyping. That is, if the programmer writes a class $\texttt{L2}_p$ that extends `List`, the compiler must create classes $\texttt{L2}_a$, `RealL2`, and `NullL2` such that the following subclassing relationships hold:

   - $\texttt{L2}_a$ extends $\texttt{List}_a$, and
   - `RealL2` extends both $\texttt{L2}_a$ and `RealList`, and
   - `NullL2` extends both $\texttt{L2}_a$ and `NullList`.

   This, however, is impossible, as Java allows only single inheritance.

3. This implementation does not support final classes. If the programmer defines $\texttt{List}_p$ to be final, the compiler cannot make $\texttt{List}_a$ final, because it has two subclasses. While the compiler could make `RealList` and `NullList` final, programmers would still be able to define another class `ExtraList` that extends $\texttt{List}_a$, violating the spirit of the `final` declarations.

Additionally, we hoped that in raising the `NullListException`, the compiler could provide additional information about the cause of the error beyond that currently available in Java's null-pointer exception. In practice, however, it is hard to see how the compiler could synthesize and include any information beyond that included in a stack trace.

We did consider alternatives to the implementation of figure 3.5, but neither solved all three issues. One possibility uses an interface instead of the abstract class to allow subclassing but not `final` classes, and the other uses nested classes to allow `final` classes but not subclassing. Neither strategy provides access to private members.

In the first alternative, in figure 3.6, $\texttt{List}_a$ is an interface, and the classes `RealList` and `NullList` implement this interface. The exception `NullListException` is as before. With this strategy, it is possible to define $\texttt{L2}_p$ as a subclass of $\texttt{List}_p$. Now, $\texttt{L2}_a$ and $\texttt{List}_a$ are interfaces, so the class `RealL2` extends only `RealList` and implements $\texttt{List}_a$ and $\texttt{L2}_a$, and likewise for `NullL2`. However, the problems with private methods and final classes remain.

The second alternative, in figure 3.7, is a possible translation for classes declared `final` by the programmer. Here, $\texttt{List}_a$ is again an abstract class, but now the concrete

```
interface List_a {
    public Object first();
    public List_a rest();
    public int length();
}

class RealList implements List_a {
    private Object head;
    private List_a tail;
    public RealList(Object hd, List_a tl) {
        head = hd; tail = tl;
    }
    public Object first() { return head; }
    public List_a rest() { return tail; }
    public int length() { return 1 + tail.length(); }
}

class NullList implements List_a {
    public Object first() { throw new NullListException("first"); }
    public List_a rest() { throw new NullListException("rest"); }
    public int length() { return 0; }
}
```

Figure 3.6: The interface implementation of $\texttt{List}_p$

```
public abstract class List_a {
    private List_a() { }

    public abstract Object first();
    public abstract List_a rest();
    public abstract int length();

    public static final class RealList extends List_a {
        private Object head;
        private List_a tail;
        // ... methods as above
    }

    public static final class NullList extends List_a {
        // methods as above
    }
}
```

Figure 3.7: The inner class implementation of $\texttt{List}_p$

classes `RealList` and `NullList` are defined as final inner classes of `List`$_a$. While we cannot make `List`$_a$ final, we can prevent programmers from defining additional subclasses by giving `List`$_a$ an empty private constructor. This prevents Java from synthesizing the standard (public) default constructor and ensures that `List`$_a$'s constructors are visible only within the class. Because `RealList` and `NullList` are inner classes, they can see `List`$_a$'s constructor and are thus instantiable.

Unfortunately, while this implementation solves the problem with `final`, it fails to address the other two problems. Private methods and fields remain inaccessible without a downcast, and the use of an abstract base class instead of an interface reintroduces the difficulties with subclassing. Specifically, one would not be able to use this strategy if `List`$_p$ extended another class. By using the interface technique for some classes and the inner class technique for others, the programmer can define subclasses and classes that are final, but not final subclasses.

Also, this encoding of *Maybe* has no support for `Maybe<Maybe<T>>`, dramatically limiting its usefulness. For instance, consider a class that implements a map from `String` to `DatabaseRecord`$_a$. Since the domain of the table already includes (the encoding of) *null*, then the map class would suffer from the same problems as the current Java implementation: a result of *null* (i.e., an instance of `NullDatabaseRecord`) from the `get` method would be ambiguous, and all of the consequences discussed earlier would apply.

Finally, several additional questions about this solution remain unanswered:

- What variants, if any, should the compiler generate for interfaces? Generating these specialized types seems necessary to allow programmers to specify invariants about the flow of *null* within the type system, but it is not at all clear whether the `when null` mechanism discussed above is appropriate for interfaces. We would also need to investigate how these generated classes interact with subtyping, and whether programmer-specified classes that implement such interfaces could override the `when null` blocks in the interface definition.

- How can we integrate these parallel classes with Java's generics? Consider a data structure that maps strings to instances of some `DatabaseRecord`$_p$ class, with the invariant that *null* is not in the table's range. In this system, one would specify its (partial) interface as follows:

```
interface DBRecordMap {
    void put(String key, RealDatabaseRecord value);
    DatabaseRecord_a get(String key);
}
```

The return type of `get` is $\texttt{DatabaseRecord}_a$, not `RealDatabaseRecord`, to indicate that the method returns *null* to indicate that `key` is not in the map's domain. But a programmer cannot easily parameterize this interface over the types involved. An attempt follows:

```
interface DBRecordMap<K, V> {
    void put(K key, V value);
    V get(K key);
}
```

Now, if the programmer uses this interface at the type `DBRecordMap<String, RealDatabaseRecord>`, the `get` method's return type is `RealDatabaseRecord`, not $\texttt{DatabaseRecord}_a$ as intended. To allow the desired return type, MIJAVA would have to support a new type operator that maps `RealDatabaseRecord` to $\texttt{DatabaseRecord}_a$, which would introduce significant additional complexity to the language (especially considering the probable difficulties in the interaction between this type operator and subtyping).

Because of all of these problems, we abandoned this strategy of encoding *null* in favor of the *Maybe* approach discussed in this chapter.

Chapter 4

MIJava and Delayed Initialization

MIJava's constructors as discussed in Chapter 2 guarantee that programs cannot refer to fields before initializing them, but at the cost of requiring constructors to provide values for every field during object creation. There are three scenarios in which a program cannot satisfy this requirement:

- Unavailable values: a program may need to delay the initialization of a field if its value is not available during object creation time, especially if there is no placeholder value of the correct type available.
- Cycle creation: the program must delay the initialization of at least one field until all of the objects in the cycle exist. For the present discussion, cycles come in two variants: "static" cycles, in which the structure of the object graph is known at compile time, and "dynamic" cycles, in which the structure is known only at runtime (such as when reading the graph structure from a file, from user input, or from a network connection).
- Laziness: a program may delay the computation of a field value until it is required, even though the value may be accessible during object construction.

MIJava must support some kind of delayed initialization to be useful to programmers in these situations.

4.1 Unavailable Values

In Java, constructors are free to leave fields uninitialized until after object construction, as in situations where the field value is not available to the constructor. The standard technique in these cases is to initialize the field to `null`, either explicitly with an assignment in the constructor or implicitly by relying on the behavior of Java's object construction mechanism.

In MIJAVA, on the other hand, the programmer must give these delayed fields `Maybe` type. This does impose the additional overhead of the `Maybe` coercions on all subsequent references to this field, but this is still an improvement over the corresponding situation in Java. First, as we discuss in Chapter 2, the `Maybe` coercions allow the runtime system to detect accesses to uninitialized fields earlier (or at least no later) than Java can, and thus MIJAVA provides more precise diagnostics. Second, in the case of fields that represent optional properties of the object, the field would have `Maybe<Maybe<T>>` type, which allows the program to distinguish between a field that has not yet been initialized and a field that has been initialized but denotes the lack of the optional property. Third, the explicit `Maybe` type draws the programmer's attention to the fact that the constructor does not necessarily initialize the field in question and serves as a warning that subsequent methods may not rely on this field having a defined value.

Graph structures, especially cyclical graphs, represent an important case of object creation where not all of the field values are available at object construction time. These graphs include the object-oriented design notions of 'roles' and 'mutual connections' and therefore arise frequently in real-world programs. As an example, consider an object representation of a bibliography or book listing. The obvious implementation includes `Book` and `Author` objects, and there is a mutual relationship between the two: an `Author` may write more than one `Book`, and a `Book` may have more than one `Author`.

In general, when the graph structure is not known at compile time (as when, for instance, the graph is specified by a file on disk), MIJAVA offers no better support than the explicit use of `Maybe` types above. However, as the next section discusses, we conjecture that we can extend MIJAVA to provide better support for the creation of graphs and other cyclical structures whose shape is known at compile time.

4.2 Static Cycle Creation

In Java, *null* is essential to the construction of statically cyclical data structures. As an example, consider the construction of a circular list containing instances of the class defined in figure 4.1. To construct a circular list containing the numbers 1, 2, and 3, a Java programmer must create the three nodes, storing a placeholder value in at least one of the `next` fields, and then use field assignment to complete the cycle once all three objects exist. The following code demonstrates a technique with the minimum possible number of assignments:

```
CircularList node3 = new CircularList(3, tempList);
CircularList node2 = new CircularList(2, node3);
CircularList node1 = new CircularList(1, node2);
node3.next = node1;
```

```
class CircularList {
    int datum;
    CircularList next;

    CircularList(int d, CircularList n) {
        datum = d;
        next = n;
    }
}
```

Figure 4.1: Defining a circular list in Java

where *tempList* is some temporary value used to indicate that the field hasn't been completely initialized; `null` is the most convenient and thus the most common choice for this value.

Because MIJAVA does not include *null*, programmers must find another value to use in these situations, especially since MIJAVA's constructors do not allow the possibility of merely leaving the fields uninitialized. An MIJAVA program could use a distinguished instance of `CircularList` as the temporary value, but this is not always practical. First, using such a sentinel value requires the programmer to explicitly check whether the `next` field contains this sentinel value in all of the places where the Java runtime system currently checks for *null*. Second, constructing a sentinel value may not be feasible for all types. Indeed, it is hardly feasible even for `CircularList`: what value should the `next` field have in this sentinel?

As an alternative in MIJAVA, the `next` field could have type `Maybe<CircularList>`, which would allow the program to initialize `node3.next` to the temporary value `none` and mutate it to the final value later. However, this field type does not correctly express the class's invariants. (We assume for the purposes of this discussion that all instances of `CircularList` do in fact appear within circular lists, so the `next` field can be `none` only during initialization.) As a consequence, any reference to the `next` field, even after initialization is complete, would have to check for `none` at runtime.

Extending MIJAVA to support Cycles

Here, we set forth an informal proposal of an extension to MIJAVA's constructor mechanism to support circular data structures. Our ideas here are inspired by similar features in PLT Scheme [15, §2.11], OCaml [27, §7.3], and Spec# [13]. We propose a new `cyclical` language construct to support the "simultaneous" creation of multiple ob-

```
class CircularList {
    int datum;
    CircularList next;

    CircularList(int d, interim CircularList n) {
        return preobj { super(); datum <- d; next <- n; };
    }
}

cyclical {
    CircularList x = new CircularList(1, y);
    CircularList y = new CircularList(2, z);
    CircularList z = new CircularList(3, x);
} in {
    ...
}
```

Figure 4.2: Proposal for creating cyclical data structures

jects. An example of a possible concrete syntax appears in figure 4.2.

The `cyclical` form consists of two parts. The first part consists entirely of variable declarations. These declarations differ from normal Java local variable declarations in several respects:

- Each declaration must provide an initializer expression.
- The initializer expressions must be instance creation (i.e., `new`) expressions.
- Each identifier bound in this section of a `cyclical` form is visible in *all* of the form's variable initializers.
- Identifiers bound by the `cyclical` form may appear only as arguments to constructors, as in the figure.
- The corresponding constructor argument must have an "interim" type, as in the figure's class definition. This means that in the field initialization section, the argument may appear only as the (entire) right-hand side of a field initialization clause inside a `preobj` expression. Interim arguments are unconstrained in the constructor's postinit block.
- Only constructor arguments may be declared as interim types. Fields, method results, method arguments (including `this`), and local variables may not have

interim type. Further, constructor arguments may not have interim type in subclasses of `java.lang.Exception`, for reasons discussed below.

The identifiers bound in the first part of the `cyclical` form are visible in the second part, which contains unconstrained code that may treat the identifiers as full (i.e., not interim) values of the indicated types.

When a `cyclical` form is executed, the runtime system starts by allocating but not initializing space for all of the variables declared in the first part of the form.[1] Then, it executes the field initialization sections of the constructors for the various objects, in some order, using the references to the allocated but not-yet-initialized space as the values of the declared variables. When all of the field initialization sections have completed, MIJAVA reifies all of the objects "simultaneously," at which point the temporary references are now valid. After executing the postinit blocks for the new objects, MIJAVA continues with the second part of the `cyclical` form.

We conjecture that the `cyclical` form and MIJAVA's constructors ensure that all references in a statically cyclical data structure are fully initialized before the program can perform any operations on that structure. However, the model of Chapter 5 does not include this form, and thus we have not proved this conjecture. Indeed, as the next section demonstrates, there is one case where this guarantee does not hold, although it is beyond the scope of our model.

Exception Handling

We must consider the effect of exception handling on this language feature. In particular, if we are to maintain MIJAVA's guarantees, we need to be sure that throwing an exception cannot "leak" an interim object out to the rest of the program.

As in the previous chapter, the key issue is whether an exception thrown before the program reifies the pre-objects can leak an incompletely-initialized object. Because only variable declarations may appear in the declaration section of a `cyclical` form, this section cannot throw an exception directly, and thus any exception that escapes this section must arise from one of the two following sources:

1. the dynamic extent of the field initialization section of one of the constructors in the declaration section—i.e., in our example, the field initialization section of `CircularList`'s constructor.

2. the dynamic extent of one of the expressions that appears as an actual parameter to one of these constructors.

[1] Language implementation invariants, especially those pertaining to garbage collection, may require initializing the newly allocated regions to all-bits-zero, but this is not important to a discussion of the semantics of the feature.

In the first case, when the field initialization section of a constructor such as that in `CircularList` throws an exception, then the exception may not contain one of the `cyclical`-bound objects. The throwing constructor may use these interim objects only in field initialization clauses, and the type system therefore prevents the construction of an exception object that directly contains such an interim object. However, because MIJAVA supports first-class pre-objects, it is possible for a constructor to throw an exception that contains a pre-object that in turn contains an interim object. Reifying this pre-object could lead to a full instance with uninitialized fields, in violation of MIJAVA's static guarantees, and the type system does not currently detect such violations.

Exceptions thrown by expressions that appear as actual parameters in the `cyclical` form's declaration section do not have access to the partially-initialized objects, due to the restrictions on the use of identifiers bound here. They cannot, therefore, raise an exception object containing an incompletely cyclical object, and they cannot store such an object in an externally-visible location before throwing an exception.

The postinit section of one of the constructors may throw an exception that contains a reference to an object created by a `cyclical` form, and it may store a reference to such an object in an externally-visible location before throwing an exception. Neither situation represents a leak, though, because the postinit section does not execute until after all of the objects in the cycle have been fully constructed and initialized and are thus no longer interim.[2]

OCaml avoids leaking exceptions by restricting the expressions on the right-hand side of the bindings and thus preventing them from throwing an exception. PLT Scheme allows incompletely initialized cycles to escape in several ways, including through exceptions. Spec#'s mechanism, similar to the one we propose here, also allows programmers to exploit exception handling to instantiate incompletely-initialized objects, as we discuss further in Chapter 8. In response, Fähndrich and Xia suggest that constructors should be prohibited from handling exceptions (but not from throwing them), but they neither provide a formal argument that this restriction is sufficient, nor do they discuss the practical consequences of this prohibition. More work remains to investigate potential solutions to this apparent leak.

4.3 Lazy Initialization

For many objects, the value for a specific field may be available (or computable) at object construction time, but the programmer may wish to delay initializing the field until the program requires its value, especially when computing the value would consume excessive time or memory, such as when the program is loading objects corresponding

[2] As with non-cyclical object creation, other representation invariants may not yet hold, but as before, this problem also exists in Java.

to entries in a database. Since each field initialization requires a database lookup and the creation of potentially many objects, Java programmers have developed the "lazy load" pattern [16, Ch. 11] to address this situation. As an example, consider a program that processes `Student` records from a database; each student has a list of associated `Course` records. Because this list is potentially quite long and loading it from the database may take quite some time, the `Student` class author delays its initialization using the following idiom:

```
class Student {
    private List<Course> courses = null;

    public List<Course> getCourses() {
        if (courses == null)
            courses = loadCourses();
        return courses;
    }

    private List<Course> loadCourses() {
        // load the list of courses from the database
    }
    ...
}
```

In this code, the `courses` field is `null` until the `loadCourses` method initializes it from the database. The straightforward MIJAVA translation of this class would have to give `courses` the type `Maybe<List<Course>>`. While it may appear that this places an undue burden on references to this field, the pattern limits these references to the corresponding getter method (`getCourses` above), and this method must already explicitly check whether the field is `null`. The explicit `Maybe` therefore adds no overhead beyond that present in Java and can serve as a useful signal to the programmer that the field's value may not be present, much as in section 4.1 above.

Because this pattern comes up so frequently (and because it requires programmers to exercise great care when dealing with exceptions and concurrency), Warth has proposed LazyJ [55], a variant of Java with explicit support for laziness. We conjecture that incorporating LazyJ's `lazy` types into MIJAVA would be feasible, but we have not yet investigated this in detail.

Chapter 5

Formal Model and Soundness

To ensure that our type system provides the guarantees described in the previous chapters, we have constructed MIJMODEL, a formal model of MIJAVA based on CLASSICJAVA [14]. In place of *null*, MIJMODEL includes *Maybe* types and the constructor model of Chapter 2. For simplicity, the model omits several features of Java and CLASSICJAVA, such as interfaces, static methods, access controls, method overloading, and field shadowing.

As with CLASSICJAVA, MIJMODEL has two separate syntaxes: a raw syntax and an elaborated syntax. The programmer writes the original program in the raw syntax, and an elaboration phase simultaneously type-checks the program and converts it to the elaborated syntax. The elaborated syntax makes all *Maybe* coercions explicit and inserts tags that indicate various other static properties that are not otherwise visible to the dynamic semantics.

MIJMODEL's dynamic semantics is defined on programs in the elaborated semantics. A second type system applies to the elaborated syntax, for use in the soundness and subject reduction proofs, and we prove that the elaboration system and type system are consistent.

5.1 Syntax

We assume the following disjoint, countably infinite sets, and the metavariables that range over them:

$$\begin{array}{rcl} C, D, E & \in & \mathit{ClassName} \\ md & \in & \mathit{MethodName} \\ fd & \in & \mathit{FieldName} \\ x, y & \in & \mathit{VarName} \\ addr & \in & \mathit{Address} \end{array}$$

We also assume two distinguished names, $\mathsf{Object} \in \mathit{ClassName}$ and $\mathsf{self} \in \mathit{VarName}$.

$t, s, u \in \mathit{Type}$	$::=$	C	$\mid$	$\mathbf{Preobj}(C)$	$\mid$ $\mathbf{Maybe}(t)$	types
gt, gs, gu	$::=$	C	$\mid$	$\mathbf{Preobj}(C)$		ground types
ot, os	$::=$	C	$\mid$	$\mathbf{Maybe}(ot)$		object types

Figure 5.1: MIJMODEL's type language

Figure 5.1 defines the syntax for MIJMODEL's type language, which is the same in raw and elaborated programs. A type is either a class name C, a *Maybe* type $\mathbf{Maybe}(t)$, or a pre-object type $\mathbf{Preobj}(C)$. We also define two restricted sets of types, ground types (gt) and object types (ot), for use in specific contexts in the model. The elaboration system uses ground types to determine what coercions to introduce; to simplify these rules, we disallow *Maybe* types in this context. Similarly, to reduce the number of rules in the operational semantics, typecasts are defined only in terms of object types, which do not allow pre-object types or any types constructed from them, such as $\mathbf{Maybe}(\mathbf{Preobj}(C))$. As a notational convenience, we use exponentiation to indicate repeated application, so $\mathbf{Maybe}^2(t) = \mathbf{Maybe}(\mathbf{Maybe}(t))$, and similarly for any non-negative exponent.

Raw Abstract Syntax

Figure 5.2 defines the abstract syntax for unelaborated programs. Following Barendregt [3], we use vector notation to denote repeated elements: $\vec{re}$ is short for the sequence $re_1, \ldots, re_n$, where $n \geq 0$. We generalize this notation to more complicated terms under the arrow; $\overrightarrow{(fd, t)}$ is short for $(fd_1, t_1), \ldots, (fd_n, t_n)$, $n \geq 0$. We represent the empty sequence as ε and use the infix operator $+\!\!+$ to indicate sequence concatenation. We overload the element-of operator $\in$ to indicate membership in sequences as well as in sets. Finally, multiple occurrences of the same vector sequence within the same context (inference rule, proof clause, etc.) preserve the order of the items under the arrow.

A program Φ consists of a sequence of definitions and an expression. Each definition is a single class, which contains constructor definitions, field definitions, and method definitions. MIJMODEL constructors have explicit names, unlike those in Java, to allow us to model multiple constructors per class without the significant overhead of Java-style overload resolution for methods and constructors.[1]

[1]While this is not related to our thesis, we believe that named constructors would be a valuable addition to Java-like languages. Programmers could use them to define two constructors with the same signature. For instance, the `Point` class of the previous chapter could define two constructors, `fromCartesian` and `fromPolar`, each of which takes two `double` arguments; currently, Java programmers have to add a dummy parameter to one of these methods.

Φ	::=	$\Theta\ re$	raw program
Θ	::=	$\overrightarrow{rdefn}$	raw definition sequence
$rdefn$	::=	**class** C **ext** D { $\overrightarrow{rcdefn}\ \overrightarrow{fdefn}\ \overrightarrow{rmdefn}$ }	raw class definition
$rcdefn$	::=	$md(\overrightarrow{t\ x})$ { re } **postinit** { re }	raw constr definition
$fdefn$	::=	$t\ fd$	field definition
$rmdefn$	::=	$t\ md(\overrightarrow{t\ x})$ { re }	raw method definition
re, rf, rg			raw expression:
	::=	x	lexical variable
	|	**new** re	object creation
	|	$re.fd$	field reference
	|	$re.fd = re$	field assignment
	|	$re.md(\overrightarrow{re})$	method call
	|	**super**$.md(\overrightarrow{re})$	super call
	|	$C.md(\overrightarrow{re})$	constr call
	|	**cast**$(ot,\ re)$	typecast
	|	**some** re	*Maybe* injection
	|	$\mathbf{none}_t$	*Maybe* injection
	|	**valOf** re	*Maybe* projection
	|	**preobj**$(C,\ re,\ \overrightarrow{fd \leftarrow re})$	pre-object creation
	|	**seq** { $\overrightarrow{re}$; }	sequential evaluation

Figure 5.2: MIJMODEL's raw abstract syntax

Constructors have two bodies, as in Chapter 2. The first body is concerned with field initialization, and the second is the postinit block. For simplicity, the model requires every constructor to contain a postinit block; to model constructors without this feature, we simply supply a postinit block of self or some other effect-free expression.

Expressions are, for the most part, patterned after the corresponding syntax in Java or CLASSICJAVA. Field references, field assignments, method calls, and super calls behave exactly as in Java. The **new** expression takes any pre-object and reifies it into a full instance of the corresponding class. The (dynamic) type of the pre-object value, rather than an explicit name in the **new** expression, specifies the class to instantiate. In most cases, programmers write object creation expressions in the form **new** $C.md(\overrightarrow{re})$, a concrete syntax essentially identical to Java's, but with an explicit constructor name.

MIJMODEL's casts, which have the form **cast**$(ot,\ re)$, extend Java's casts to support *Maybe* types. Casts to and from pre-object types are forbidden by the type system.[2]

[2]These casts do not appear useful enough to justify the added complexity in the dynamic semantics.

The type in a cast expression thus always has the form **Maybe**$^n(C)$ for some $n \geq 0$ and some class C. The elaboration phase inserts any necessary coercions around the argument *re* to convert it to **Maybe**$^n(D)$, for some class D that may or may not be related to C. At runtime, casting a value of (dynamic) type D to C succeeds when D is a subtype of C, just as in Java. Casting $\mathbf{none}_t$ to **Maybe**$^n(C)$ always succeeds, and casting (**some** v) to **Maybe**$^n(C)$ succeeds exactly when casting the value v to **Maybe**$^{(m-1)}(C)$ succeeds.

As a concrete example, consider the following method definitions:

```
Maybe(Number) m1(Maybe^2(Integer) mmint) { cast(Maybe(Number), mmint) }
Maybe^2(Integer) m2(Maybe(Number) mnum) { cast(Maybe^2(Integer), mnum) }
```

In m1, **cast**'s argument elaborates to (**valOf** mmint), which has type **Maybe**(Integer). If this coercion succeeds at runtime, the entire cast succeeds. In m2, the elaboration phase converts mnum to (**some** mnum) at type **Maybe**^{2}(Integer). At runtime, the cast succeeds if mnum = **none** or if mnum = **some** v, where v is an instance of Integer.[3]

The **some** and **none** keywords construct values of *Maybe* type, and the **valOf** keyword projects *Maybe* values to the contained value. The type annotation on **none** specifies the type of the expression; for any type t, the expression $\mathbf{none}_t$ has exactly (and only) type **Maybe**(t). Without this annotation, the elaboration system would not be deterministic, as we explain below.

Pre-object creation expressions have the form **preobj**$(C,\ re_0,\ \overrightarrow{fd \leftarrow re})$. The first sub-form, C, identifies the class of pre-object to be constructed.[4] The second, re_0, is an expression that evaluates to a D pre-object, where D is C's immediate superclass. In the most common case, this expression is a call to one of D's constructors. The remaining sub-forms specify the initial values for all fields defined directly within C.

The **seq** expression denotes sequential evaluation. MIJMODEL evaluates **seq**'s subexpressions for their effects in left-to-right order, and the value of the overall **seq** expression is the value of its last subexpression.

[3] MIJMODEL allows more casts than MIJAVA does, assuming that MIJAVA follows Java's general rules for cast expressions. In particular, MIJAVA issues compile-time errors for both of the casts in the above example. In method m1, the cast fails because **Maybe**(Number) is not a subtype of **Maybe**(Integer), and vice versa, and MIJAVA, like Java, rejects all casts in which the two types are unrelated. The MIJAVA programmer can work around this by rewriting the body of m1, for instance, as **cast**(Number, (**valOf** mmint)), but at the cost of introducing a coercion that may fail at runtime.

[4] MIJMODEL's operational semantics needs the explicit type tag, but it is feasible for a full implementation to infer this information from context, using only a straightforward but intricate addition to the elaboration system of section 5.2. We assume here that the programmer includes this information directly, or that an earlier unspecified transformation of the source code inserts the annotation.

Π	::=	$\Delta\ e$	program
Δ	::=	$\overrightarrow{defn}$	definition sequence
defn	::=	**class** C **ext** D { $\overrightarrow{cdefn}$ $\overrightarrow{fdefn}$ $\overrightarrow{mdefn}$ }	class definition
cdefn	::=	$md(\overrightarrow{t\ x})$ { e } **postinit** { e }	constr definition
fdefn	::=	$t\ fd$	field definition
mdefn	::=	$t\ md(\overrightarrow{t\ x})$ { e }	method definition
v, w			value:
	::=	$\mathbf{none}_t$	value *Maybe* injection
	|	$\mathbf{some}_t\ v$	value *Maybe* injection
	|	*addr*	object address
	|	$\mathbf{preobj\text{-}val}(C,\ \overrightarrow{fd \leftarrow v},\ e)$	pre-object value
e, f, g			expression:
	::=	v	value
	|	x	lexical variable
	|	$\mathbf{new}\ e$	object creation
	|	$e.fd$	field reference
	|	$e.fd = e$	field assignment
	|	$e.md(\overrightarrow{e})$	method call
	|	$\mathbf{super} \equiv e : C\ .md(\overrightarrow{e})$	super call
	|	$C.md(\overrightarrow{e})$	constr call
	|	$\mathbf{cast}(ot,\ e)$	typecast
	|	$\mathbf{some}_t\ e$	non-value *Maybe* injection
	|	$\mathbf{valOf}\ e$	*Maybe* projection
	|	$\mathbf{preobj}(C,\ e,\ \overrightarrow{fd \leftarrow e})$	pre-object creation
	|	$\mathbf{seq}$ { $\overrightarrow{e}$; }	sequential evaluation
	|	$\mathbf{init}(e,\ e)$	object initialization

Figure 5.3: MIJMODEL's elaborated abstract syntax

Elaborated Abstract Syntax

Figure 5.3 defines MIJMODEL's elaborated syntax. Highlighted productions and nonterminals indicate the differences between the elaborated and raw syntaxes. We add a syntactic class of values, consisting of *Maybe* values, references into the store, and pre-object values. Like **preobj** expressions, **preobj-val** forms have three sub-parts. The first is the pre-object's class, and the second specifies all of the pre-object's fields. The third part is an unevaluated expression, a sequence of all the postinit blocks that MIJMODEL must execute immediately upon reifying this pre-object into a full instance.

The elaboration syntax also adds a new expression, **init**(e_1, e_2), which the operational semantics uses to track the delayed postinit blocks during object construction. The first subexpression e_1 is the pre-object under construction, and the second, e_2, is a delayed postinit block. Also, elaborated **super** expressions include explicit references to self (or its corresponding address) and its static type, as required by the dynamic semantics. Finally, elaborated **some** expressions contain a tag denoting the type of the argument; the type soundness proof requires this but the runtime system does not.

Explicitly-typed none

Both the raw and elaborated syntaxes require the programmer to annotate each occurrence of **none** with its type. This is necessary to keep the elaboration system deterministic.

As an example, consider the following class declaration:

```
class Example ext Object {
    Example m1(Maybe(Example) obj) { self }
    Example m2() { self.m1(none) }
```

In the highlighted method call, we have omitted the type annotation on **none**. As a result, the type of the method's actual parameter is ambiguous. While the intended type reconstruction is **Maybe**(Example), the elaboration system could reconstruct type **Maybe**^{n}(Example) for any $n > 1$ and insert the corresponding **valOf** coercion(s), the innermost of which would always fail at runtime. The explicit type tag prevents this problem.

In a real implementation of the full Java language plus *Maybe*, the context of the expression would in most cases provide enough information to guide the reconstruction (in an application of bidirectional type checking [41]) and thus uniquely determine the coercions to be inserted. In situations where this is not the case, the compiler could signal the ambiguity, and the programmer could insert an explicit cast to resolve the problem, much as is necessary in certain circumstances with the literal `null` in Java today.

Method arguments are the primary context in which `null`'s ambiguous type causes compile-time errors. If a programmer writes a method call with a literal `null` as an argument upon whose precise type method overload resolution depends, the compiler signals an error, and the programmer must add a cast or otherwise indicate the type of `null` more precisely. Such annotations would not, however, be required for **none**. With type erasure, Java does not distinguish between `Maybe<Integer>` and `Maybe<String>` for the purpose of method overloading. The types `Integer` and `Maybe<Integer>` are clearly distinct in this context, but the argument **none** admits only the second type.

$\vdash_{\mathsf{ep}} \Phi \Rightarrow \Pi : t$	Φ elaborates to Π with type t
$\Theta \vdash_{\mathsf{ed}} \mathit{rdefn} \Rightarrow \mathit{defn}$	*rdefn* elaborates to *defn* under Θ
$\Theta, C \vdash_{\mathsf{ec}} \mathit{rcdefn} \Rightarrow \mathit{cdefn}$	*rcdefn* elaborates to *cdefn* in class C under Θ
$\Theta, C \vdash_{\mathsf{ef}} \mathit{fdefn}$	*fdefn* is well-formed in class C under Θ
$\Theta, C \vdash_{\mathsf{em}} \mathit{rmdefn} \Rightarrow \mathit{mdefn}$	*rmdefn* elaborates to *mdefn* in class C under Θ
$\Theta, \Gamma \vdash_{\mathsf{ee}} \mathit{re} \Rightarrow e : t$	*re* elaborates to e with type t in Γ under Θ
$\Theta \vdash_{\mathsf{et}} t$	t exists under Θ

Figure 5.4: Summary of elaboration system judgments

5.2 Elaboration Semantics

The elaboration system checks types of raw terms and produces the corresponding elaborated terms, making the implicit coercions implicit and adding type annotations to **some** and **super** terms. As in ClassicJava, we define the elaboration semantics as a set of judgments on terms, listed in figure 5.4. Those judgments that apply to expressions (possibly as sub-terms of their direct arguments) produce elaborated copies of the expression, class member, definition, or program, as appropriate. Expression elaboration and type-checking requires a type environment Γ, which is a finite map from *VarName* to *Type*.

The elaboration semantics also uses several helper functions and relations, listed in figure 5.5 and described below. The subclassing relation $\prec_\Theta$ is not transitive as it includes only immediate parent/child relationships. The subtype relation $<:_\Theta$ is (in well-formed programs) a partial order. The **Maybe** and **Preobj** type constructors are invariant in their arguments, in keeping with previous chapters.

The subsumption relation $\lesssim:_\Theta$ is an extension of subtyping: $s \lesssim:_\Theta t$ only if an expression of type s can appear in a context that expects a value of type t, either through subtyping or through the addition of *Maybe* coercions, but not both.

The coercion function $\mathit{coerce}(e, s, t)$ takes an expression e of type s and adds the necessary coercions (without padding) to produce an expression of type t.[5] The casting-coercion function *castCoerce* has the same essential purpose as *coerce*, but it is used for the subexpression in a **cast** expression. We state without proof that *castCoerce* and *coerce* always produce identical results; we treat these two cases separately for the sake of the soundness proof.

The next relations, *constr*, *immFd*, *field*, and *meth*, look up the types of various class members in the definition table Θ. Neither *constr* nor *immFd* considers members

[5] As lemma 3 in section 5.5 demonstrates, the result is in fact of type t only when a certain relationship holds between s and t.

$C \prec_\Theta D$	C is an immediate subclass of D
$s <:_\Theta t$	s is a subtype of t
$s \lesssim_\Theta t$	s is subsumed by t
$coerce(e, s, t) = e'$	e' is the coercion of e from s to t
$castCoerce(e, os, ot) = e'$	e' is the coercion of e from os to ot in a cast
$constr_{\Theta C}(md) = \vec{t}$	C defines constructor md with argument types $\vec{t}$
$immFd_{\Theta C}(fd) = t$	fd defined directly in C with type t
$field_{\Theta C}(fd) = t$	fd in C has type t, possibly inherited
$meth_{\Theta C}(md) = \vec{s} \to t$	md has type $\vec{s} \to t$ in class C, possibly inherited
$validOverride_\Theta(C, md, t_0, \vec{t})$	C's override of md with type $\vec{t} \to t_0$ preserves type
$coreType(t) = gt$	t without **Maybe**s is gt

Figure 5.5: Summary of elaboration system auxiliary relations

defined in supertypes; both *field* and *meth* search up the subclassing tree as necessary.

The *validOverride* relation ensures that overriding methods have the same signature as the overridden method in the superclass. Specifically, $validOverride_\Theta(C, md, t_0, \vec{t})$ is true if C's superclass contains method md with the type $\vec{t} \to t_0$, or if the superclass does not contain the method at all.

Finally, for contexts that require a class type, like field reference and method calls, the *coreType* relation "strips off" all of the **Maybe** wrappers on a type and produces the underlying ground type.

Definition Elaboration

Figure 5.6 defines the elaboration rules for programs, classes, constructors, methods, and fields. We extend the arrow notation to cover judgments in the straightforward way. For instance, when we apply the ELAB-PROG rule to the program $rdefn_1\ \ rdefn_2\ \ re$, then $\Theta = rdefn_1\ \ rdefn_2$, and the penultimate antecedent of the rule expands into *two* antecedents, $\Theta \vdash_{\text{ed}} rdefn_1 \Rightarrow defn_1$ and $\Theta \vdash_{\text{ed}} rdefn_2 \Rightarrow defn_2$, both of which must be proved in order to prove the rule's conclusion.

The ELAB-PROG rule ensures that the class definition sequence contains no duplicates or subclassing cycles. It elaborates the individual raw definitions $\overrightarrow{rdefn}$ to create elaborated definitions $\overrightarrow{defn}$, elaborates the top-level expression, and constructs the elaborated program.

The ELAB-DEFN rule ensures that the superclass exists and that the class does not contain duplicate constructor, field, or method names. If the definition satisfies these properties, the rule elaborates and type-checks each of the class's members.

Program Elaboration $\boxed{\vdash_{\mathsf{ep}} \Phi \Rightarrow \Pi : t}$

[ELAB-PROG]
$$\frac{\Theta = \overrightarrow{rdefn} \quad \text{class names in } \Theta \text{ are distinct} \quad (\textbf{class Object } \ldots) \notin \Theta \quad <:_\Theta \text{ is anti-symmetric} \quad \overrightarrow{\Theta \vdash_{\mathsf{ed}} rdefn \Rightarrow defn} \quad \Theta, \varnothing \vdash_{\mathsf{ee}} re \Rightarrow e : t}{\vdash_{\mathsf{ep}} \overrightarrow{rdefn}\ re \Rightarrow \overrightarrow{defn}\ e : t}$$

Definition Elaboration $\boxed{\Theta \vdash_{\mathsf{ed}} rdefn \Rightarrow defn}$

[ELAB-DEFN]
$$\frac{\text{names in } \overrightarrow{rcdefn}, \overrightarrow{fdefn}, \overrightarrow{rmdefn} \text{ distinct} \quad \Theta \vdash_{\mathsf{et}} D \quad \overrightarrow{\Theta, C \vdash_{\mathsf{ec}} rcdefn \Rightarrow cdefn} \quad \overrightarrow{\Theta, C \vdash_{\mathsf{ef}} fdefn} \quad \overrightarrow{\Theta, C \vdash_{\mathsf{em}} rmdefn \Rightarrow mdefn}}{\begin{array}{c}\Theta \vdash_{\mathsf{ed}} \textbf{class } C \textbf{ ext } D \ \{\ \overrightarrow{rcdefn}\ \overrightarrow{fdefn}\ \overrightarrow{rmdefn}\ \} \\ \Rightarrow \textbf{class } C \textbf{ ext } D \ \{\ \overrightarrow{cdefn}\ \overrightarrow{fdefn}\ \overrightarrow{mdefn}\ \}\end{array}}$$

Constructor Elaboration $\boxed{\Theta, C \vdash_{\mathsf{ec}} rcdefn \Rightarrow cdefn}$

[ELAB-CDEFN]
$$\frac{\vec{x} \text{ distinct} \quad \mathsf{self} \notin \vec{x} \quad \Theta, \{\overrightarrow{(x : t)}\} \vdash_{\mathsf{ee}} re_b \Rightarrow e'_b : t_b \quad t'_b \lesssim_\Theta \textbf{Preobj}(C) \quad e_b = coerce(e'_b, t'_b, \textbf{Preobj}(C)) \quad \Theta, \{\overrightarrow{(x : t)}, (\mathsf{self} : C)\} \vdash_{\mathsf{ee}} re_p \Rightarrow e_p : t_p}{\Theta, C \vdash_{\mathsf{ec}} md(\overrightarrow{t\ x})\ \{\ re_b\ \}\ \textbf{postinit}\ \{\ re_p\ \} \Rightarrow md(\overrightarrow{t\ x})\ \{\ e_b\ \}\ \textbf{postinit}\ \{\ e_p\ \}}$$

Field Declaration $\boxed{\Theta, C \vdash_{\mathsf{ef}} fdefn}$

[ELAB-FDEFN]
$$\frac{\Theta \vdash_{\mathsf{et}} t \quad C \prec_\Theta D \quad fd \notin \mathrm{dom}(field_{\Theta D})}{\Theta, C \vdash_{\mathsf{ef}} t\ fd}$$

Method Elaboration $\boxed{\Theta, C \vdash_{\mathsf{em}} rmdefn \Rightarrow mdefn}$

[ELAB-MDEFN]
$$\frac{\Theta \vdash_{\mathsf{et}} t_0 \quad \overrightarrow{\Theta \vdash_{\mathsf{et}} t} \quad \vec{x} \text{ distinct} \quad \mathsf{self} \notin \vec{x} \quad validOverride_\Theta(C, md, t_0, \vec{t}) \quad \Theta, \{\overrightarrow{(x : t)}, (\mathsf{self} : C)\} \vdash_{\mathsf{ee}} re \Rightarrow e' : t' \quad t' \lesssim_\Theta t_0 \quad e = coerce(e', t, \textbf{Preobj}(C))}{\Theta, C \vdash_{\mathsf{em}} t_0\ md(\overrightarrow{t\ x})\ \{\ re\ \} \Rightarrow t_0\ md(\overrightarrow{t\ x})\ \{\ e\ \}}$$

Figure 5.6: MIJMODEL elaboration rules, part I

Constructor elaboration, in the ELAB-CDEFN rule, is more complex. The antecedents perform the following checks and operations:

1. Check that the formal arguments have distinct names and do not include self.
2. Elaborate the main body re_b to e'_b with type t_b in a type environment containing the constructor's formal parameters but *not* self. The main body's type must be coercible to **Preobj**(C), where C is the the containing class.
3. Construct the elaborated body e_b by adding the necessary coercions to e'_b to transform it from type t'_b to type **Preobj**(C).
4. Elaborate the postinit body re_p to e_p with type t_p. Here, the type environment includes both the formal parameters and self. No coercions are necessary because MIJMODEL executes the postinit block purely for its effects.

Field elaboration, ELAB-FDEFN, simply ensures that the field's type exists and that the field does not shadow a field from the superclass. No elaboration is necessary here.

Method elaboration in ELAB-MDEFN is similar to constructor elaboration, except that self is available to the body, and that the body's type must be coercible to the stated return type. Additionally, the *validOverride* antecedent ensures that C's implementation of the method *md* has the same type as the implementation, if any, in C's superclasses.

Expression Elaboration

Figures 5.7 and 5.8 specify the elaboration rules for MIJMODEL expressions. These rules explicitly check for subsumption (using the $\lesssim:_\Theta$ relation) and insert coercions for all subexpressions that appear in strictness positions.

The ELAB-NEW rule expects its argument to have type **Preobj**(C) for some class C and inserts coercions accordingly. (While we do not expect many programs to involve values of type **Maybe**^{n}(**Preobj**(C)) where $n > 0$, MIJMODEL allows this with the obvious semantics.) Because **new** reifies its argument into a full instance, the result type is always C.

ELAB-FIELD-REF and ELAB-FIELD-SET coerce the expression in front of the dot to its underlying class type (as computed by the *coreType* auxiliary) and then look up the type of the field in the resulting class. In both cases, the type of the entire expression is the field's type, even if the type of the expression on the right-hand side of a field assignment is a subtype of the field's type.

Similarly, ELAB-CALL coerces the expression to the left of the dot to its core type and uses this type to look up the signature of the called method. The method's actual parameters must have types subsumed by the argument types in the method's signature.

Expression Elaboration $\boxed{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} re \Rightarrow e : t}$

[ELAB-VAR]
$$\frac{\Gamma(x) = t}{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} x \Rightarrow x : t}$$

[ELAB-NEW]
$$\frac{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} re \Rightarrow e' : t \qquad t \lesssim\!\!:_\Theta \textbf{Preobj}(C) \qquad e = coerce(e', t, \textbf{Preobj}(C))}{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} \textbf{new}\ re \Rightarrow \textbf{new}\ e : C}$$

[ELAB-FIELD-REF]
$$\frac{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} re \Rightarrow e' : t' \qquad C = coreType(t') \qquad e = coerce(e', t', C) \qquad field_{\Theta C}(fd) = t}{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} re.fd \Rightarrow e.fd : t}$$

[ELAB-FIELD-SET]
$$\frac{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} re_1 \Rightarrow e_1' : t_1 \qquad \Theta, \Gamma \mid\!\!\frac{}{\text{ee}} re_2 \Rightarrow e_2' : t_2 \qquad C = coreType(t_1) \qquad e_1 = coerce(e_1', t_1, C) \qquad field_{\Theta C}(fd) = t \qquad t_2 \lesssim\!\!:_\Theta t \qquad e_2 = coerce(e_2', t_2, t)}{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} re_1.fd = re_2 \Rightarrow e_1.fd = re_2 : t}$$

[ELAB-CALL]
$$\frac{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} re_0 \Rightarrow e_0' : t_0 \qquad \overrightarrow{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} re \Rightarrow e' : t'} \qquad C = coreType(t_0) \qquad e_0 = coerce(e_0', t_0, C) \qquad meth_{\Theta C}(md) = \vec{s} \to t \qquad \overrightarrow{t' \lesssim\!\!:_\Theta s} \qquad \overrightarrow{e = coerce(e', t', s)}}{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} re_0.md(\vec{re}) \Rightarrow e_0.md(\vec{e}) : t}$$

[ELAB-SUPER]
$$\frac{\Gamma(\mathsf{self}) = C \qquad C \prec_\Theta D \qquad meth_{\Theta D}(md) = \vec{s} \to t \qquad \overrightarrow{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} re \Rightarrow e' : t'} \qquad \overrightarrow{t' \lesssim\!\!:_\Theta s} \qquad \overrightarrow{e = coerce(e', t', s)}}{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} \textbf{super}.md(\vec{re}) \Rightarrow \textbf{super} \equiv \mathsf{self}{:}C.md(\vec{e}) : t}$$

[ELAB-CONSTR]
$$\frac{\Theta \mid\!\!\frac{}{\text{et}} C \qquad \overrightarrow{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} re \Rightarrow e' : t'} \qquad constr_{\Theta C}(md) = \vec{t} \qquad \overrightarrow{t' \lesssim\!\!:_\Theta t} \qquad \overrightarrow{e = coerce(e', t', t)}}{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} C.md(\vec{re}) \Rightarrow C.md(\vec{e}) : \textbf{Preobj}(C)}$$

[ELAB-CAST]
$$\frac{\Theta \mid\!\!\frac{}{\text{et}} ot \qquad \Theta, \Gamma \mid\!\!\frac{}{\text{ee}} re \Rightarrow e' : ot' \qquad e = castCoerce(e', ot', ot)}{\Theta, \Gamma \mid\!\!\frac{}{\text{ee}} \textbf{cast}(ot,\ re) \Rightarrow \textbf{cast}(ot,\ e) : ot}$$

Figure 5.7: MIJMODEL elaboration rules, part II

[ELAB-SOME]
$$\frac{\Theta \vdash_{\mathsf{et}} t \qquad \Theta,\Gamma \vdash_{\mathsf{ee}} re \Rightarrow e : t}{\Theta,\Gamma \vdash_{\mathsf{ee}} \mathbf{some}\ \ re \Rightarrow \mathbf{some}_t\ \ e : \mathbf{Maybe}(t)}$$

[ELAB-NONE]
$$\frac{\Theta \vdash_{\mathsf{et}} t}{\Theta,\Gamma \vdash_{\mathsf{ee}} \mathbf{none}_t \Rightarrow \mathbf{none}_t : \mathbf{Maybe}(t)}$$

[ELAB-VALOF]
$$\frac{\Theta,\Gamma \vdash_{\mathsf{ee}} re \Rightarrow e : \mathbf{Maybe}(t)}{\Theta,\Gamma \vdash_{\mathsf{ee}} \mathbf{valOf}\ \ re \Rightarrow \mathbf{valOf}\ \ e : t}$$

[ELAB-SEQ]
$$\frac{n > 0 \qquad \forall i \in [1,n]\ .\ \Theta,\Gamma \vdash_{\mathsf{ee}} re_i \Rightarrow e_i : t_i}{\Theta,\Gamma \vdash_{\mathsf{ee}} \mathbf{seq}\ \{\ re_1 \ldots re_n\ ;\ \} \Rightarrow \mathbf{seq}\ \{\ e_1 \ldots e_n\ ;\ \} : t_n}$$

[ELAB-PREOBJ]
$$\frac{\begin{array}{c}\mathsf{self} \notin \mathrm{dom}(\Gamma) \qquad \Theta \vdash_{\mathsf{et}} C \qquad \Theta,\Gamma \vdash_{\mathsf{ee}} re_0 \Rightarrow e_0' : t_0 \\ C \prec_\Theta D \qquad t_0 \lesssim_\Theta \mathbf{Preobj}(D) \qquad e_0 = coerce(e_0', t_0, \mathbf{Preobj}(D)) \\ \Theta,\Gamma \vdash_{\mathsf{ee}} re \Rightarrow e' : t' \qquad immFd_{\Theta C} = \{(fd, t)\} \qquad t' \lesssim_\Theta t \qquad e = coerce(e', t', t)\end{array}}{\Theta,\Gamma \vdash_{\mathsf{ee}} \mathbf{preobj}(C,\ re_0,\ fd \leftarrow re) \Rightarrow \mathbf{preobj}(C,\ e_0,\ fd \leftarrow e) : \mathbf{Preobj}(C)}$$

Type Existence $\boxed{\Theta \vdash_{\mathsf{et}} t}$

[ELAB-EXIST-OBJECT]
$$\frac{}{\Theta \vdash_{\mathsf{et}} \mathsf{Object}}$$

[ELAB-EXIST-CLASS]
$$\frac{\mathbf{class}\ C\ \ldots \in \Theta}{\Theta \vdash_{\mathsf{et}} C}$$

[ELAB-EXIST-PREOBJ]
$$\frac{\Theta \vdash_{\mathsf{et}} C}{\Theta \vdash_{\mathsf{et}} \mathbf{Preobj}(C)}$$

[ELAB-EXIST-MAYBE]
$$\frac{\Theta \vdash_{\mathsf{et}} t}{\Theta \vdash_{\mathsf{et}} \mathbf{Maybe}(t)}$$

Figure 5.8: MIJMODEL elaboration rules, part III

The ELAB-SUPER rule performs the same checks and coercions as the ELAB-CALL rule, except that there is no expression to the left of the dot. To allow the dynamic semantics to resolve the called method correctly, the rule inserts a reference to self, tagged with its type, as in CLASSICJAVA. During evaluation, the reference to the calling object (here, self) provides the value that the reduction substitutes for self in the body of the called method. The class tag C guides method lookup, which always starts in C's immediate superclass, regardless of the dynamic type of self.

To elaborate constructor calls, the ELAB-CONSTR rule performs the same essential checks and coercions as ELAB-SUPER, although without the annotations. Constructors are required to return pre-object types.

To simplify the operational semantics, the ELAB-CAST rule restricts typecasts to class and *Maybe* types. Unlike the other elaboration rules, however, it does not enforce any relationship between the actual type ot' of the argument and its expected type ot, instead relying on the the operational semantics to enforce coerceability and safety.

As defined here, MIJMODEL does not reject provably impossible casts at compile time, as both Java and CLASSICJAVA do. The following alternative to ELAB-CAST would

implement this behavior:

[ELAB-CAST-ALT]

$$\frac{\Theta \vdash_{\text{et}} C \qquad \Theta \vdash_{\text{et}} D \qquad C <:_\Theta D \text{ or } D <:_\Theta C \qquad \begin{array}{c} ot = \mathbf{Maybe}^n(C) \qquad \Theta, \Gamma \vdash_{\text{ee}} re \Rightarrow e' : \mathbf{Maybe}^m(D) \\ e = \mathit{castCoerce}(e', \mathbf{Maybe}^m(D), ot) \end{array}}{\Theta, \Gamma \vdash_{\text{ee}} \mathbf{cast}(ot,\ re) \Rightarrow \mathbf{cast}(ot,\ e) : ot}$$

However, this rule in conjunction with the explicit type tags on **none** would forbid casts like $\mathbf{cast}(\mathsf{Integer},\ \mathbf{none}_{\mathsf{String}})$, even though the equivalent Java cast, `(Integer) null`, succeeds.[6]

The ELAB-SOME rule annotates the **some** keyword with the type of its argument for use in the soundness proof. Otherwise, it and the ELAB-NONE and ELAB-VALOF rules are straightforward.

The ELAB-SEQ rule simply type-checks all subexpressions and adds no coercions beyond those introduced during the elaboration of the subexpressions.

The ELAB-PREOBJ rule must ensure that the second subexpression is a pre-object of the correct type, that the field initialization clauses provide initial values for all (and only) the necessary fields. The auxiliary function $\mathit{immFd}_{\Theta C}$, treated as a set, lists all of the necessary fields and their types. Additionally, the rule ensures that the new object is not accessible during field initialization by guaranteeing that `self` is not bound in the type environment Γ.

Figure 5.8 also defines the type-existence judgment. The types that exist in a program are the classes defined in the definition sequence Θ, the class `Object`, all **Preobj** types based on existing classes, and **Maybe** types based on existing types.

Auxiliary elaboration functions

Figures 5.9, 5.10, and 5.11 define the auxiliary functions and relations for the elaboration semantics.

The rule for immediate subclassing defines the obvious relation: $C \prec_\Theta D$ if and only if class C exists and directly extends D. The subtype relation is the reflexive-transitive closure of the subclassing relation, across all types. We specifically do not include any rules for **Preobj** or **Maybe** types, as they are invariant by design and are not related to other types in the model.

The subsumption relation, $s \lesssim_\Theta t$, holds whenever a value of type s may appear in a context that expects type t, either through coercion or subtyping, but not both (because **Maybe** types are invariant). As such, we do not include a separate transitivity rule, because this would allow $C \lesssim_\Theta \mathbf{Maybe}(D)$ and $\mathbf{Maybe}(C) \lesssim_\Theta \mathbf{Maybe}(D)$ if $C <:_\Theta D$.[7]

[6] Of course, the Java expression `(Integer) s` does fail to compile when `s` has type `String`.

[7] The programmer can add explicit casts to work around some of these restrictions; see section 3.3 for a discussion of the issues surrounding *Maybe* invariance.

Immediate Subclassing $\boxed{C \prec_\Theta D}$

$$\frac{\textbf{class } C \textbf{ ext } D \{ \ldots \} \in \Theta}{C \prec_\Theta D} \text{ [ELAB-IMM-SUBCLASS]}$$

Subtyping $\boxed{s <:_\Theta t}$

$$\frac{}{t <:_\Theta t} \text{ [ELAB-SUBTYPE-REFL]} \qquad \frac{s <:_\Theta u \qquad u <:_\Theta t}{s <:_\Theta t} \text{ [ELAB-SUBTYPE-TRANS]} \qquad \frac{C \prec_\Theta D}{C <:_\Theta D} \text{ [ELAB-SUBTYPE-CLASS]}$$

Subsumption $\boxed{s \lesssim:_\Theta t}$

$$\frac{s <:_\Theta t}{s \lesssim:_\Theta t} \text{ [E-SUBSUMPTION-SUBTYPE]} \qquad \frac{n, m \geq 0}{\textbf{Maybe}^m(gt) \lesssim:_\Theta \textbf{Maybe}^n(gt)} \text{ [E-SUBSUMPTION-MAYBE]}$$

Coercion Introduction (general contexts) $\boxed{coerce(e, s, t) = e'}$

$$\frac{}{coerce(e, \textbf{Maybe}^n(gt_1), \textbf{Maybe}^n(gt_2)) = e} \text{ [E-COERCE-SAME]}$$

$$\frac{n > m}{coerce(e, \textbf{Maybe}^n(gt_1), \textbf{Maybe}^m(gt_2)) = \textbf{valOf}^{(n-m)}\ e} \text{ [E-COERCE-VALOF]}$$

$$\frac{m > n \qquad e' = coerce(e, \textbf{Maybe}^n(gt_1), \textbf{Maybe}^{(m-1)}(gt_2))}{coerce(e, \textbf{Maybe}^n(gt_1), \textbf{Maybe}^m(gt_2)) = \textbf{some}_{\textbf{Maybe}^{(m-1)}(gt_2)}\ e'} \text{ [E-COERCE-SOME]}$$

Figure 5.9: MIJMODEL elaboration rules: auxiliary relations

Casting Coercion $\boxed{castCoerce(e, os, ot) = e'}$

[ELAB-CASTCOERCE-SIMPLE]

$$\frac{}{castCoerce(e, \mathbf{Maybe}^n(C), \mathbf{Maybe}^n(D)) = e}$$

[ELAB-CASTCOERCE-VALOF]

$$\frac{n > m}{castCoerce(e, \mathbf{Maybe}^n(C), \mathbf{Maybe}^m(D)) = \mathbf{valOf}^{(n-m)}\ e}$$

[ELAB-CASTCOERCE-SOME]

$$\frac{n < m \qquad e' = castCoerce(\mathbf{some}_{\mathbf{Maybe}^n(C)}\ e, \mathbf{Maybe}^{(n+1)}(C), \mathbf{Maybe}^m(D))}{castCoerce(e, \mathbf{Maybe}^n(C), \mathbf{Maybe}^m(D)) = e'}$$

Immediate Field Lookup $\boxed{immFd_{\Theta C}(fd) = t}$

[ELAB-IMMFD]

$$\frac{\mathbf{class}\ C\ \mathbf{ext}\ D\ \{\ \overrightarrow{rcdefn}\ \overrightarrow{fdefn}\ \overrightarrow{rmdefn}\ \} \in \Theta \qquad t\ fd \in \overrightarrow{fdefn}}{immFd_{\Theta C}(fd) = t}$$

Field Lookup $\boxed{field_{\Theta C}(fd) = t}$

[ELAB-FIELD-IMM]

$$\frac{immFd_{\Theta C}(fd) = t}{field_{\Theta C}(fd) = t}$$

[ELAB-FIELD-INHERIT]

$$\frac{fd \notin \mathrm{dom}(immFd_{\Theta C}) \qquad C \prec_\Theta D \qquad field_{\Theta D}(fd) = t}{field_{\Theta C}(fd) = t}$$

Figure 5.10: MIJMODEL elaboration auxiliary relations, part II

The coercion-introduction rules ensure that MIJMODEL inserts only those coercions required, with no additional padding, in a purely deterministic fashion, as discussed in section 3.1.

Field, method, and constructor lookup are all straightforward inspections of the class definitions in the program. Because Θ may not contain a definition for Object, these rules imply that $immFd_{\Theta,\mathsf{Object}} = field_{\Theta,\mathsf{Object}} = meth_{\Theta,\mathsf{Object}} = \emptyset$. For constructors, however, the ELAB-CONSTR-OBJECT rule specifies that Object has one constructor, mkObj, that takes no arguments.

Method Lookup $\boxed{meth_{\Theta C}(md) = \overrightarrow{s} \to t}$

[ELAB-METH-IMM]
$$\frac{\textbf{class } C \textbf{ ext } D \ \{ \ \overrightarrow{rcdefn} \ \overrightarrow{fdefn} \ \overrightarrow{rmdefn} \ \} \in \Theta \qquad t \ md(\overrightarrow{s \ x}) \ \{ \ re \ \} \in \overrightarrow{rmdefn}}{meth_{\Theta C}(md) = \overrightarrow{s} \to t}$$

[ELAB-METH-INHERIT]
$$\frac{\textbf{class } C \textbf{ ext } D \ \{ \ \overrightarrow{rcdefn} \ \overrightarrow{fdefn} \ \overrightarrow{rmdefn} \ \} \in \Theta \quad md \notin \overrightarrow{rmdefn} \qquad meth_{\Theta D}(md) = \overrightarrow{s} \to t}{meth_{\Theta C}(md) = \overrightarrow{s} \to t}$$

Constructor Lookup $\boxed{constr_{\Theta C}(md) = \overrightarrow{t}}$

[ELAB-CONSTR-OBJECT]
$$\frac{}{constr_{\Theta,\mathsf{Object}}(\mathsf{mkObj}) = \varepsilon}$$

[ELAB-CONSTR-SUBCLASS]
$$\frac{\textbf{class } C \textbf{ ext } D \ \{ \ \overrightarrow{rcdefn} \ \overrightarrow{fdefn} \ \overrightarrow{rmdefn} \ \} \in \Theta \quad (md(\overrightarrow{t \ x}) \ \{ \ re_b \ \} \ \textbf{postinit} \ \{ \ re_p \ \}) \in \overrightarrow{rcdefn}}{constr_{\Theta C}(md) = \overrightarrow{t}}$$

Valid Method Override $\boxed{validOverride_{\Theta}(C, md, t_0, \overrightarrow{t})}$

[ELAB-VALID-OVERRIDE]
$$\frac{C \prec_{\Theta} D \qquad \text{If } md \in \mathrm{dom}(meth_{\Theta D}) \text{ then } meth_{\Theta D}(md) = \overrightarrow{t} \to t_0}{validOverride_{\Theta}(C, md, t_0, \overrightarrow{t})}$$

Underlying Ground Type $\boxed{coreType(t) = gt}$

[CORETYPE]
$$\frac{n \geq 0}{coreType(\textbf{Maybe}^n(gt)) = gt}$$

Figure 5.11: MIJMODEL elaboration auxiliary relations, part III

$\vdash_{\mathsf{tp}} \Pi : t$	Π has type t
$\Delta \vdash_{\mathsf{td}} defn$	$defn$ is well-formed under Δ
$\Delta, C \vdash_{\mathsf{tc}} cdefn$	$cdefn$ in class C is well-formed under Δ
$\Delta, C \vdash_{\mathsf{tf}} fdefn$	$fdefn$ in class C is well-formed under Δ
$\Delta, C \vdash_{\mathsf{tm}} mdefn$	$mdefn$ in class C is well-formed under Δ
$\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e : t$	e has type t under Δ, Γ, Σ
$\Delta \vdash_{\mathsf{tt}} t$	t exists under Δ

$C \prec_\Delta D$
: C is an immediate subclass of D

$s <:_\Delta t$
: s is a subtype of t

$immFd_{\Delta C}(fd) = t$
: C defines field fd with type t

$field_{\Delta C}(fd) = t$
: C contains field fd, possibly inherited, with type t

$immMeth_{\Delta C}(md) = (\vec{s} \to t, \vec{x}, e)$
: method md defined in C with signature $\vec{s} \to t$, arguments $\vec{x}$, and body e

$meth_{\Delta C}(md) = (\vec{s} \to t, \vec{x}, e)$
: method md present in C with signature $\vec{s} \to t$, arguments $\vec{x}$, and body e

$constr_{\Delta C}(md) = (\vec{t}, \vec{x}, e_b, e_p)$
: constructor md defined in C with argument types $\vec{t}$, argument names $\vec{x}$, main body e_b, and postinit body e_p.

$validOverride_\Delta(C, md, t_0, \vec{t})$
: C's override of md with type $\vec{t} \to t_0$ preserves md's type in superclass

$os \approx ot$
: os and ot have the same number of **Maybe** constructors

Figure 5.12: Type system judgments and auxiliary relations

5.3 Typing Rules

The typing rules apply to elaborated terms, for use in the subject reduction proof. Figure 5.12 defines the judgments and auxiliary relations, which parallel those of the elaboration system. In addition to the type environment Γ of the elaboration semantics, the post-elaboration semantics also uses a store typing Σ, a finite map from addresses to types.

The subsumption relation $\lesssim:_\Theta$, however, does not have an equivalent here, as the

post-elaboration system assumes that all necessary coercions are stated explicitly, so subtyping suffices. The $meth_{\Delta C}$ and $constr_{\Delta C}$ functions expand on the results of their equivalents in the elaboration system to return the formal argument names and body expression(s), for use in the dynamic semantics. The type system also adds the $immMeth_{\Delta C}$ relation, which bears the same relation to $meth_{\Delta C}$ as $immFd_{\Delta C}$ does to $field_{\Delta C}$. Finally, we introduce the relation $os \approx ot$, which holds when types os and ot have the same number of **Maybe** constructors, even if their core types are different.

Figure 5.13 specifies the type rules for programs, class definitions, and class members. These directly parallel the corresponding rules from the elaboration system, except that they do not construct new expressions, and they assume that all *Maybe*-related coercions are in place. Figures 5.14 and 5.15 show the type rules for expressions. Most are copies of the corresponding elaboration rules, with the subsumption and coercions removed, but some require further explanation.

The TYPE-PREOBJVAL rule checks pre-object values. Since a pre-object value is ready for reification, it must contain values for all of the corresponding class's fields, including those inherited from superclasses. So, this rule checks the field initialization clauses not against $immFd_{\Delta C}$ like ELAB-PREOBJ, but against $field_{\Delta C}$. Additionally, the rule ensures that all well-typed pre-object values are closed by type-checking all field values in the empty type environment and checking the delayed postinit expression in a type environment containing only a binding for self.[8]

The rule for **super** expressions, TYPE-SUPER, is almost identical to the corresponding elaboration rule. However, TYPE-SUPER adds two restrictions on e_0, the expression added by the elaboration phase. First, e_0 must be either self or an address; and second, it must have the type C added by the annotation. Like CLASSICJAVA, MIJMODEL requires this restriction on the form of the expression e_0 because the dynamic semantics does not evaluate this position but rather expects it to be a value.[9]

For cast expressions, the TYPE-CAST rule requires only that the type of the expression and the type of the cast operation have the same number of **Maybe**s. Further restrictions on the relationship between the two types are not compatible with the soundness lemma, as in earlier models of Java [14, 24].

The TYPE-INIT rule, for **init** expressions, also ensures that self is not in scope, as **init** expressions arise during object initialization. As the final antecedent indicates, **init** binds self in the second subexpression, whose type is unconstrained as it is executed purely for effects.

Figures 5.16 and 5.17 specify the auxiliary relations and functions for the type system. Most rules are identical to those in the elaboration system, *mutatis mutandis.* The rules for $meth_{\Delta}$, $immMeth_{\Delta}$, and $constr_{\Delta}$, in comparison to the elaboration system,

[8] A trivial induction on the syntax for values shows that, with this definition for TYPE-PREOBJVAL, which effectively binds self in the postinit expression, all well-typed values are closed terms.

[9] The antecedent restricting the form of e_0 corresponds to CLASSICJAVA's SUPEROK() predicate.

Program Typing $\boxed{\vdash_{\mathsf{tp}} \Pi : t}$

[TYPE-PROG]

$$\frac{\Delta = \overrightarrow{defn} \qquad \text{class names in } \Delta \text{ are distinct} \qquad (\textbf{class Object } \ldots) \notin \Delta \qquad <:_\Delta \text{ is anti-symmetric} \qquad \overrightarrow{\Delta \vdash_{\mathsf{td}} defn} \qquad \Delta, \emptyset, \emptyset \vdash_{\mathsf{te}} e : t}{\vdash_{\mathsf{tp}} \overrightarrow{defn}\ e : t}$$

Definition Typing $\boxed{\Delta \vdash_{\mathsf{td}} defn}$

[TYPE-DEFN]

$$\frac{\text{names in } \overrightarrow{cdefn}, \overrightarrow{fdefn}, \overrightarrow{mdefn} \text{ distinct} \qquad \Delta \vdash_{\mathsf{tt}} D \qquad \overrightarrow{\Delta, C \vdash_{\mathsf{tc}} cdefn} \qquad \overrightarrow{\Delta, C \vdash_{\mathsf{tf}} fdefn} \qquad \overrightarrow{\Delta, C \vdash_{\mathsf{tm}} mdefn}}{\Delta \vdash_{\mathsf{td}} \textbf{class } C \textbf{ ext } D \ \{\ \overrightarrow{cdefn}\ \overrightarrow{fdefn}\ \overrightarrow{mdefn}\ \}}$$

Constructor Typing $\boxed{\Delta, C \vdash_{\mathsf{tc}} cdefn}$

[TYPE-CDEFN]

$$\frac{\vec{x} \text{ distinct} \qquad \mathsf{self} \notin \vec{x} \qquad \overrightarrow{\Delta \vdash_{\mathsf{tt}} t} \qquad \Delta, \{\overrightarrow{(x : t)}\}, \emptyset \vdash_{\mathsf{te}} e_b : \textbf{Preobj}(C) \qquad \Delta, \{(\mathsf{self} : C), \overrightarrow{(x : t)}\}, \emptyset \vdash_{\mathsf{te}} e_p : s}{\Delta, C \vdash_{\mathsf{tc}} md(\overrightarrow{t\ x})\ \{\ e_b\ \}\ \textbf{postinit}\ \{\ e_p\ \}}$$

Field Typing $\boxed{\Delta, C \vdash_{\mathsf{tf}} fdefn}$

[TYPE-FDEFN]

$$\frac{\Delta \vdash_{\mathsf{tt}} t \qquad C \prec_\Delta D \qquad fd \notin \mathrm{dom}(field_{\Delta D})}{\Delta, C \vdash_{\mathsf{tf}} t\ fd}$$

Method Typing $\boxed{\Delta, C \vdash_{\mathsf{tm}} mdefn}$

[TYPE-MDEFN]

$$\frac{\vec{x} \text{ distinct} \qquad \mathsf{self} \notin \vec{x} \qquad \Delta \vdash_{\mathsf{tt}} t_0 \qquad \overrightarrow{\Delta \vdash_{\mathsf{tt}} t} \qquad \Delta, \{(\mathsf{self} : C), \overrightarrow{(x : t)}\}, \emptyset \vdash_{\mathsf{te}} e : t_0' \qquad t_0' <:_\Delta t_0 \qquad validOverride_\Delta(C, md, t_0, \vec{t})}{\Delta, C \vdash_{\mathsf{tm}} t_0\ md(\overrightarrow{t\ x})\ \{\ e\ \}}$$

Figure 5.13: MIJMODEL type rules for definitions

Expression Typing $\boxed{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e : t}$

[TYPE-PREOBJVAL]
$$\frac{\Delta \vdash_{\mathsf{tt}} C \qquad \{\overrightarrow{fd}\} = \mathrm{dom}(\mathit{field}_{\Delta C}) \qquad \forall (fd' \leftarrow v') \in \overrightarrow{fd \leftarrow v} \,.\, \Delta, \emptyset, \Sigma \vdash_{\mathsf{te}} v' : t' \text{ and } t' <:_\Delta \mathit{field}_{\Delta C}(fd') \qquad \Delta, \{(\mathsf{self} : C)\}, \Sigma \vdash_{\mathsf{te}} e : t_p}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} \textbf{preobj-val}(C,\ \overrightarrow{fd \leftarrow v},\ e) : \textbf{Preobj}(C)}$$

[TYPE-ADDR]
$$\frac{\Sigma(\mathit{addr}) = t}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} \mathit{addr} : t}$$

[TYPE-VAR]
$$\frac{\Gamma(x) = t}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} x : t}$$

[TYPE-NEW]
$$\frac{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e : \textbf{Preobj}(C)}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} \textbf{new}\ e : C}$$

[TYPE-FIELD-REF]
$$\frac{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e : C \qquad \mathit{field}_{\Delta C}(fd) = t}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e.fd : t}$$

[TYPE-FIELD-SET]
$$\frac{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e_0 : C \qquad \mathit{field}_{\Delta C}(fd) = t \qquad \Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e_1 : t_1 \qquad t_1 <:_\Delta t}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e_0.fd = e_1 : t}$$

[TYPE-CALL]
$$\frac{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e_0 : C \qquad \mathit{meth}_{\Delta C}(md) = (\overrightarrow{s} \to t, \_, \_) \qquad \overrightarrow{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e : s'} \qquad \overrightarrow{s' <:_\Delta s}}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e_0.md(\overrightarrow{e}) : t}$$

[TYPE-SUPER]
$$\frac{e_0 \in \{\mathsf{self}\} \cup \mathit{Address} \qquad \Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e_0 : C_0 \qquad C_0 <:_\Delta C \qquad C \prec_\Delta D \qquad \mathit{meth}_{\Delta D}(md) = (\overrightarrow{s} \to t, \_, \_) \qquad \overrightarrow{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e : s'} \qquad \overrightarrow{s' <:_\Delta s}}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} \textbf{super} \equiv e_0 : C.md(\overrightarrow{e}) : t}$$

[TYPE-CONSTR]
$$\frac{\Delta \vdash_{\mathsf{tt}} C \qquad \mathit{constr}_{\Delta C}(md) = (\overrightarrow{t}, \_, \_, \_) \qquad \overrightarrow{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e : t'} \qquad \overrightarrow{t' <:_\Delta t}}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} C.md(\overrightarrow{e}) : \textbf{Preobj}(C)}$$

[TYPE-CAST]
$$\frac{\Delta \vdash_{\mathsf{tt}} ot \qquad \Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e : ot' \qquad ot \approx ot'}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} \textbf{cast}(ot,\ e) : ot}$$

[TYPE-SOME]
$$\frac{\Delta \vdash_{\mathsf{tt}} t \qquad \Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e : s \qquad s <:_\Delta t}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} \textbf{some}_t\ e : \textbf{Maybe}(t)}$$

Figure 5.14: MIJMODEL type rules for expressions, part I

[TYPE-NONE]
$$\frac{\Delta \vdash_{\mathsf{tt}} t}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} \mathbf{none}_t : \mathbf{Maybe}(t)}$$

[TYPE-VALOF]
$$\frac{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e : \mathbf{Maybe}(t)}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} \mathbf{valOf}\ e : t}$$

[TYPE-PREOBJ]
$$\frac{\begin{array}{c}\mathsf{self} \notin \mathrm{dom}(\Gamma) \qquad \Delta \vdash_{\mathsf{tt}} C \\ \Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e_0 : \mathbf{Preobj}(D) \qquad C \prec_\Delta D \qquad \{\overrightarrow{fd}\} = \mathrm{dom}(immFd_{\Delta C}) \\ \forall (fd' \leftarrow e') \in \overrightarrow{fd \leftarrow e}\ .\ \Theta, \Gamma \vdash_{\mathsf{ee}} \Sigma \Rightarrow e' : t' \text{ and } t' <:_\Delta immFd_{\Delta C}(fd')\end{array}}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} \mathbf{preobj}(C,\ e_0,\ \overrightarrow{fd \leftarrow e}) : \mathbf{Preobj}(C)}$$

[TYPE-SEQ]
$$\frac{n > 0 \qquad \forall i \in [1, n]\ .\ \Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e_i : t_i}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} \mathbf{seq}\ \{\ e_1\ ;\ \ldots\ ;\ e_n\ ;\ \} : t_n}$$

[TYPE-INIT]
$$\frac{\mathsf{self} \notin \mathrm{dom}(\Gamma) \qquad \Delta \vdash_{\mathsf{tt}} C \qquad \Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e_1 : \mathbf{Preobj}(C) \qquad \Delta, \Gamma[\mathsf{self} : C], \Sigma \vdash_{\mathsf{te}} e_2 : t}{\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} \mathbf{init}(e_1,\ e_2) : \mathbf{Preobj}(C)}$$

Type Existence $\boxed{\Delta \vdash_{\mathsf{tt}} t}$

[TEXIST-OBJECT]
$$\frac{}{\Delta \vdash_{\mathsf{tt}} \mathsf{Object}}$$

[TEXIST-CLASS]
$$\frac{\mathbf{class}\ C\ \ldots \in \Delta}{\Delta \vdash_{\mathsf{tt}} C}$$

[TEXIST-PREOBJ]
$$\frac{\Delta \vdash_{\mathsf{tt}} C}{\Delta \vdash_{\mathsf{tt}} \mathbf{Preobj}(C)}$$

[TEXIST-MAYBE]
$$\frac{\Delta \vdash_{\mathsf{tt}} t}{\Delta \vdash_{\mathsf{tt}} \mathbf{Maybe}(t)}$$

Figure 5.15: MIJMODEL type rules for expressions, part II

add information about formal parameter names and method and constructor bodies, for the dynamic semantics. The T-CONSTR-OBJECT rule also specifies the behavior of Object's sole constructor, which returns a pre-object value that contains no fields and whose postinit block has no effects. Finally, we also add the $\approx$ relation, which defines an equivalence relation on object types for use in the TYPE-CAST rule.

In order to prove MIJMODEL's soundness, we must establish a connection between the elaboration semantics of section 5.2 and the type rules in this section, which apply to elaborated terms. The soundness theorem, stated in section 5.5, requires that the program be well-typed under the elaboration semantics, thus guaranteeing that the necessary coercions and annotations are present. But the subject reduction lemma describes expressions that have already been elaborated, so it uses the type semantics to prove its invariants. Theorem 1 establishes the necessary relationship between the two systems.

Subtyping $\boxed{s <:_\Delta t}$

[T-SUBTYPE-REFL]
$$\frac{}{t <:_\Delta t}$$

[T-SUBTYPE-TRANS]
$$\frac{s <:_\Delta t \qquad t <:_\Delta u}{s <:_\Delta u}$$

[T-SUBTYPE-CLASS]
$$\frac{C \prec_\Delta D}{C <:_\Delta D}$$

Immediate Subclassing $\boxed{C \prec_\Delta D}$

[T-IMM-SUBCLASS]
$$\frac{\textbf{class } C \textbf{ ext } D \ \{ \ \ldots \ \} \in \Delta}{C \prec_\Delta D}$$

Field Lookup $\boxed{field_{\Delta C}(fd) = t}$

[T-FIELD-IMM]
$$\frac{immFd_{\Delta C}(fd) = t}{field_{\Delta C}(fd) = t}$$

[T-FIELD-INHERIT]
$$\frac{fd \notin \text{dom}(immFd_{\Delta C}) \qquad C \prec_\Delta D \qquad field_{\Delta D}(fd) = t}{field_{\Delta C}(fd) = t}$$

Immediate Field Lookup $\boxed{immFd_{\Delta C}(fd) = t}$

[T-IMMFD]
$$\frac{\textbf{class } C \textbf{ ext } D \ \{ \ \overrightarrow{cdefn} \ \overrightarrow{fdefn} \ \overrightarrow{mdefn} \ \} \in \Delta \qquad (t \ fd) \in \overrightarrow{fdefn}}{immFd_{\Delta C}(fd) = t}$$

Valid Method Override $\boxed{validOverride_\Delta(C, md, t_0, \overrightarrow{t})}$

[T-VALIDOVERRIDE]
$$\frac{C \prec_\Delta D \qquad md \in \text{dom}(meth_{\Delta D}) \text{ implies } meth_{\Delta D}(md) = (\overrightarrow{t} \to t_0, \_, \_)}{validOverride_\Delta(C, md, t_0, \overrightarrow{t})}$$

Figure 5.16: MIJMODEL type auxiliaries, part I

Method Lookup $\boxed{meth_{\Delta C}(md) = (\vec{s} \to t, \vec{x}, e)}$

$$\text{[T-Meth-Imm]}\quad \frac{immMeth_{\Delta C}(md) = (\vec{s} \to t, \vec{x}, e)}{meth_{\Delta C}(md) = (\vec{s} \to t, \vec{x}, e)}$$

$$\text{[T-Meth-Inherit]}\quad \frac{md \notin \mathrm{dom}(immMeth_{\Delta C}) \qquad C \prec_\Delta D \qquad meth_{\Delta D}(md) = (\vec{s} \to t, \vec{x}, e)}{meth_{\Delta C}(md) = (\vec{s} \to t, \vec{x}, e)}$$

Immediate Method Lookup $\boxed{immMeth_{\Delta C}(md) = (\vec{s} \to t, \vec{x}, e)}$

$$\text{[T-ImmMeth]}\quad \frac{\textbf{class } C \textbf{ ext } D \ \{ \ \overrightarrow{cdefn} \ \overrightarrow{fdefn} \ \overrightarrow{mdefn} \ \} \qquad (t \ md(\overrightarrow{s \ x}) \ \{ \ e \ \}) \in \overrightarrow{mdefn}}{immMeth_{\Delta C}(md) = (\vec{s} \to t, \vec{x}, e)}$$

Constructor Lookup $\boxed{constr_{\Delta C}(md) = (\vec{t}, \vec{x}, e, e)}$

$$\text{[T-Constr-Object]}\quad \frac{}{constr_{\Delta,\mathsf{Object}}(\mathsf{mkObj}) = (\varepsilon, \varepsilon, \textbf{preobj-val}(\mathsf{Object}, \ \varepsilon, \ \mathsf{self}), \mathsf{self})}$$

$$\text{[T-Constr]}\quad \frac{\begin{array}{c}\textbf{class } C \textbf{ ext } D \ \{ \ \overrightarrow{cdefn} \ \overrightarrow{fdefn} \ \overrightarrow{mdefn} \ \} \in \Delta \\ (md(\overrightarrow{t \ x}) \ \{ \ e_b \ \} \ \textbf{postinit} \ \{ \ e_p \ \}) \in \overrightarrow{cdefn}\end{array}}{constr_{\Delta C}(md) = (\vec{t}, \vec{x}, e_b, e_p)}$$

Equivalent Type Structure $\boxed{ot \approx ot}$

$$\text{[T-Equiv-Type]}\quad \frac{n \geq 0}{\textbf{Maybe}^n(C) \approx \textbf{Maybe}^n(D)}$$

Figure 5.17: MIJModel type relations, part II

Theorem 1 (Program Elaboration Preserves Type)
If $\vdash_{\text{ep}} \Theta\ re \Rightarrow \Delta\ e : t$, *then* $\vdash_{\text{tp}} \Delta\ e : t$.

Proof sketch

Because the elaboration rules for class members, class definitions, and programs modify only the expressions and not, for example, the method signatures, elaboration does not change the type structure of the program. Therefore, we have the following equivalences:

- $\Theta \vdash_{\text{et}} t \Leftrightarrow \Delta \vdash_{\text{tt}} t$
- $immFd_{\Theta C} = t \Leftrightarrow immFd_{\Delta C}(fd) = t$
- $field_{\Theta C}(fd) = t \Leftrightarrow field_{\Delta C}(fd) = t$
- $meth_{\Theta C}(md) = (\vec{s} \to t) \Leftrightarrow \exists \vec{x}, e\,.\, meth_{\Delta C}(md) = (\vec{s} \to t, \vec{x}, e)$
- $constr_{\Theta C}(md) = \vec{t} \Leftrightarrow \exists \vec{x}, e_b, e_p\,.\, constr_{\Delta C}(md) = (\vec{t}, \vec{x}, e_b, e_p)$
- $\prec_\Theta = \prec_\Delta$
- $<:_\Theta = <:_\Delta$

Because $\vdash_{\text{ep}} \Theta\ re \Rightarrow \Delta\ e : t$, ELAB-PROG implies that $\Theta \vdash_{\text{ed}} rdefn \Rightarrow defn$ for each $rdefn \in \Theta$, and that $\Theta, \varnothing \vdash_{\text{ee}} re \Rightarrow e : t$.

By lemma 2, $\Delta, \varnothing, \varnothing \vdash_{\text{te}} e : t$. Similar lemmas for class definitions, constructor definitions, field definitions, and method definitions imply that $\Delta \vdash_{\text{td}} defn$ for each $defn \in \Delta$ and that type is preserved for expressions in constructor and method bodies. Therefore, $\vdash_{\text{tp}} \Delta\ e : t$, as required. ■

The key lemma establishes that the two type systems reconstruct the same type for related expressions.

Lemma 2 (Expression Elaboration Preserves Type)
If $\vdash_{\text{ep}} \Theta\ re_0 \Rightarrow \Delta\ e_0 : t_0$, *and if* $\Theta, \Gamma \vdash_{\text{ee}} re \Rightarrow e : t$, *then* $\Delta, \Gamma, \varnothing \vdash_{\text{te}} e : t$.

Proof

Induction on the structure of the derivation of $\Theta, \Gamma \vdash_{\text{ee}} re \Rightarrow e : t$. We show the cases for ELAB-FIELD-REF, ELAB-CAST, and ELAB-PREOBJ here; the others are similar to ELAB-FIELD-REF.

- Case ELAB-FIELD-REF. We have $re = (re'\,.\,fd)$. From the rule, we know that
  - $\Theta, \Gamma \vdash_{\text{ee}} re' \Rightarrow e'' : t'$; and
  - $C = coreType(t')$; and

- $e' = \mathit{coerce}(e'', t', C)$; and
- $\mathit{field}_{\Delta C}(fd) = t$; and
- $e = (e' \,.\, fd)$.

By induction, $\Delta, \Gamma, \varnothing \mid\!\!\!\frac{}{\mathsf{te}} e'' : t'$. Because $C = \mathit{coreType}(t')$, we must have $t' = \mathbf{Maybe}^n(C)$ for some $n \geq 0$. By E-SUBSUMPTION-MAYBE, then $t \lesssim\!\!:_\Theta C$. By lemma 3, $\Delta, \Gamma, \varnothing \mid\!\!\!\frac{}{\mathsf{te}} e' : s'$, where $s' <\!\!:_\Delta C$. The definition of subtyping implies that $s' = D$ for some class D.

An induction on the derivation of $D <\!\!:_\Delta C$ shows that $\mathit{field}_{\Delta D}(fd) = t$. Therefore, by T-FIELD-REF, $\Delta, \Gamma, \varnothing \mid\!\!\!\frac{}{\mathsf{te}} e : t$.

· Case ELAB-CAST. We have $re = \mathbf{cast}(ot,\ re')$. From the rule, we know that

- $\Theta \mid\!\!\!\frac{}{\mathsf{et}} ot$; and
- $\Theta, \Gamma \mid\!\!\!\frac{}{\mathsf{ee}} re' \Rightarrow e'' : ot'$; and
- $e' = \mathit{castCoerce}(e'', ot', ot)$; and
- $e = \mathbf{cast}(ot,\ e')$; and
- $t = ot$.

Because elaboration preserves the type structure of the program, $\Delta \mid\!\!\!\frac{}{\mathsf{tt}} ot$. By induction, $\Delta, \Gamma, \varnothing \mid\!\!\!\frac{}{\mathsf{te}} e'' : ot'$. By lemma 4, there exists a type os such that $\Delta, \Gamma, \varnothing \mid\!\!\!\frac{}{\mathsf{te}} e' : os$ and $os \approx ot$. Therefore, we have $\Delta, \Gamma, \varnothing \mid\!\!\!\frac{}{\mathsf{te}} e : ot$, by TYPE-CAST.

· Case ELAB-PREOBJ. We have $re = \mathbf{preobj}(C,\ re_s,\ \overrightarrow{fd \leftarrow re_f})$. From the rule, we know that

- $\mathsf{self} \notin \mathrm{dom}(\Gamma)$; and
- $\Theta \mid\!\!\!\frac{}{\mathsf{et}} C$; and
- $\Theta, \Gamma \mid\!\!\!\frac{}{\mathsf{ee}} re_s \Rightarrow e'_s : t_s$; and
- $C \prec_\Theta D$; and
- $t_s \lesssim\!\!:_\Theta \mathbf{Preobj}(D)$; and
- $e_s = \mathit{coerce}(e'_s, t_s, \mathbf{Preobj}(D))$; and
- $\overrightarrow{\Theta, \Gamma \mid\!\!\!\frac{}{\mathsf{ee}} re_f \Rightarrow e'_f : t'_f}$; and
- $\mathit{immFd}_{\Theta C} = \{\overrightarrow{(fd, t_f)}\}$; and
- $\overrightarrow{t'_f \lesssim\!\!:_\Theta t_f}$; and
- $\overrightarrow{e_f = \mathit{coerce}(e'_f, t'_f, t_f)}$; and
- $e = \mathbf{preobj}(C,\ e_s,\ \overrightarrow{fd \leftarrow e_f})$; and

– $t = \mathbf{Preobj}(C)$.

Lemma 3 and induction imply that $\Delta, \Gamma, \varnothing \vdash_{\mathsf{te}} e_s : \mathbf{Preobj}(D)$ and

$$\overrightarrow{\Delta, \Gamma, \varnothing \vdash_{\mathsf{te}} e_f : t_f}.$$

Because $immFd_{\Theta C} = immFd_{\Delta C}$ and $\prec_\Delta = \prec_\Theta$ as argued above, we have satisfied TYPE-PREOBJ's antecedents, and thus $\Delta, \Gamma, \varnothing \vdash_{\mathsf{te}} e : t$ as required. ■

While proving that elaboration preserves expression type is primarily a matter of ensuring that the elaboration rules' antecedents imply the antecedents of the corresponding typing rules, we do need two additional lemmas that guarantee that a coerced expression has the expected type, for each kind of coercion.

Lemma 3 (Coercing between related types produces the expected type)
If

- $\vdash_{\mathsf{ep}} \Theta\ re_0 \Rightarrow \Delta\ e_0 : t_0$*; and*
- $\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e' : s$*; and*
- $s \lesssim_\Theta t$*; and*
- $e = coerce(e', s, t)$,

then $\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} e : t'$*, where* $t' <:_\Delta t$.

Proof

Induction on the derivation of $e = coerce(e', s, t)$:

- Case E-COERCE-SAME, in which $s = \mathbf{Maybe}^n(gt_1)$ and $t = \mathbf{Maybe}^n(gt_2)$ and $e = e'$.

  If $n = 0$, then $s = gt_1$ and $t = gt_2$. Then E-SUBSUMPTION-SUBTYPE implies that $s <:_\Theta t$. Because elaboration does not change the subtyping relation, $s <:_\Delta t$ and the desired result follows.

  If $n > 0$, then E-SUBSUMPTION-MAYBE implies $s = t$, and the conclusion follows trivially.

- Case E-COERCE-VALOF, in which $s = \mathbf{Maybe}^n(gt_1)$, $t = \mathbf{Maybe}^m(gt_2)$, $n > m$, and $e = \mathbf{valOf}^{(n-m)}\ e'$.

  By induction on the exponent of **valOf**, we have

  $$\Delta, \Gamma, \Sigma \vdash_{\mathsf{te}} \mathbf{valOf}^{(n-m)}\ e' : \mathbf{Maybe}^m(gt_1).$$

By E-SUBSUMPTION-MAYBE, $gt_1 = gt_2$. Therefore,

$$\Theta, \Gamma \mid\!\!\frac{}{\mathsf{ee}} \Sigma \Rightarrow (\mathbf{valOf}^{(n-m)}\ e') : t.$$

· Case E-COERCE-SOME, in which

- $s = \mathbf{Maybe}^n(gt_1)$; and
- $t = \mathbf{Maybe}^m(gt_2)$, $m > n$; and
- $e = \mathbf{some}_{\mathbf{Maybe}^{(m-1)}(gt_2)}\ e''$; and
- $e'' = coerce(e', \mathbf{Maybe}^n(gt_1), \mathbf{Maybe}^{(m-1)}(gt_2))$.

By E-SUBSUMPTION-MAYBE, $m, n \geq 0$ and $gt_1 = gt_2$; call both types gt.

Case analysis on m:

- Case $m = 1$. Because $m > n$, $n = 0$, so $s = gt$ and $t = \mathbf{Maybe}(gt)$.
  Then $e'' = e'$ by E-COERCE-SAME, and $e = \mathbf{some}_{gt}\ e'$. By TYPE-SOME, $\Delta, \Gamma, \Sigma \mid\!\!\frac{}{\mathsf{te}} \mathbf{some}_{gt}\ e' : t$, as required.
- Case $m > 1$.
  E-SUBSUMPTION-MAYBE implies that $\mathbf{Maybe}^n(gt) \lesssim_\Theta \mathbf{Maybe}^{(m-1)}(gt)$. By induction, $\Delta, \Gamma, \Sigma \mid\!\!\frac{}{\mathsf{te}} e'' : t''$ where $t'' <:_\Delta \mathbf{Maybe}^{(m-1)}(gt)$. From the definition of subtyping, we know that $t'' = \mathbf{Maybe}^{(m-1)}(gt)$. TYPE-SOME and the definition of e above imply that $\Delta, \Gamma, \Sigma \mid\!\!\frac{}{\mathsf{te}} e : \mathbf{Maybe}^m(gt)$, as required.
- Case $m < 1$. Since $0 \leq n < m$, this case cannot happen. ■

Lemma 4 (Cast-coercing between related types produces the expected type)
If $\Delta, \Gamma, \Sigma \mid\!\!\frac{}{\mathsf{te}} e' : ot'$ *and* $e = castCoerce(e', ot', ot)$*, then there exists a type os such that* $\Delta, \Gamma, \Sigma \mid\!\!\frac{}{\mathsf{te}} e : os$ *and* $os \approx ot$.

Proof

Induction on the derivation of $e = castCoerce(e', ot', ot)$.[10]

· Case E-CASTCOERCE-SIMPLE. For this case:

- $ot' = \mathbf{Maybe}^n(C)$; and
- $ot = \mathbf{Maybe}^n(D)$; and
- $e = e'$.

[10] This one lemma is the reason for the separate *castCoerce* auxiliary. Because we cannot assume a relationship between *ot* and *ot'*, unlike in lemma 3, we have to structure the definition of *castCoerce* differently for the proof to go through, even though it always produces the same answer as *coerce*.

Let $os = ot'$. Then $os \approx ot$ by T-EQUIVTYPE.

· Case E-CASTCOERCE-REMOVE. For this case:

- $ot' = \mathbf{Maybe}^n(C)$; and
- $ot = \mathbf{Maybe}^m(D)$; and
- $n > m$; and
- $e = \mathbf{valOf}^{(n-m)}\ e'$.

Let $os = \mathbf{Maybe}^m(C)$. By T-EQUIVTYPE, $os \approx ot$. Induction on $n - m$ proves that $\Delta, \Gamma, \varnothing \vdash_{\mathsf{te}} e : os$.

· Case E-CASTCOERCE-ADD. For this case:

- $ot' = \mathbf{Maybe}^n(C)$; and
- $ot = \mathbf{Maybe}^m(D)$; and
- $n < m$; and
- $e = castCoerce(\mathbf{some}_{\mathbf{Maybe}^n(C)}\ e', \mathbf{Maybe}^{(n+1)}(C), \mathbf{Maybe}^m(D))$.

By TYPE-SOME, $\Delta, \Gamma, \varnothing \vdash_{\mathsf{te}} \mathbf{some}_{\mathbf{Maybe}^n(C)}\ e' : \mathbf{Maybe}^{(n+1)}(C)$. By induction, there exists a type os such that $\Delta, \Gamma, \varnothing \vdash_{\mathsf{te}} e : os$ and $os \approx \mathbf{Maybe}^m(D)$. ■

5.4 Dynamic Semantics

We define MIJMODEL's dynamic semantics as a reduction semantics with explicit evaluation contexts, after CLASSICJAVA. Figure 5.18 defines the evaluation contexts. For the dynamic semantics, we add a store σ, which is a finite map from addresses to object instances. An object instance is a pair $(C, \mathcal{F})$ containing the object's dynamic type C and a field map $\mathcal{F}$. This field map is a finite map from field names to values. Only object instances appear in the store; the operational semantics does not perform additional allocation for pre-object and **Maybe** values.

Figure 5.19 specifies the reduction rules that participate in object creation. As this is a complex process, these rules are best explained with a detailed example. Given the class definition in figure 5.20, object creation proceeds according to the reduction sequence below. In each step, the redex is underlined. (For clarity, we omit the store and flatten nested **seq** expressions.)

1. **new** Symbol.interned("foo")

   The first step is the object creation expression, as originally written by the programmer. By rule R-CONSTR, the constructor call becomes the following **init** expression, containing the two bodies of the constructor.

$$
\begin{array}{rcl}
\mathcal{E} & ::= & \Box \mid \textbf{new}\ \mathcal{E} \mid \mathcal{E}.fd \mid \mathcal{E}.fd = e \mid v.fd = \mathcal{E} \mid \mathcal{E}.md(\vec{e}) \\
& \mid & v.md(\vec{v},\ \mathcal{E},\ \vec{e}) \mid \textbf{super} \equiv v\!:\!C.md(\vec{v},\ \mathcal{E},\ \vec{e}) \mid C.md(\vec{v},\ \mathcal{E},\ \vec{e}) \\
& \mid & \textbf{cast}(ot,\ \mathcal{E}) \mid \textbf{some}_t\ \mathcal{E} \mid \textbf{valOf}\ \mathcal{E} \mid \textbf{preobj}(C,\ \mathcal{E},\ \overrightarrow{fd \leftarrow e}) \\
& \mid & \textbf{preobj}(C,\ v,\ \overrightarrow{fd \leftarrow v},\ fd \leftarrow \mathcal{E},\ \overrightarrow{fd \leftarrow v}) \mid \textbf{seq}\ \{\ \mathcal{E};\ \vec{e};\ \} \mid \textbf{init}(\mathcal{E},\ e)
\end{array}
$$

Figure 5.18: MIJMODEL evaluation contexts

$$
\Delta \vdash_{\Gamma} \langle \mathcal{E}[\textbf{new preobj-val}(C,\ \overrightarrow{fd \leftarrow v},\ e)], \sigma\rangle \rightarrow \quad \text{[R-NEW]}
$$
$$
\langle \mathcal{E}[[addr/\mathsf{self}](\textbf{seq}\ \{\ e;\ addr;\ \})], \sigma'\rangle
$$
$$
\text{where } addr \notin \mathrm{dom}(\sigma) \text{ and } \sigma' = \sigma[addr : (C, \mathcal{F})]
$$
$$
\text{and } \mathcal{F} = \{\overrightarrow{(fd, v)}\}
$$

$$
\Delta \vdash_{\Gamma} \langle \mathcal{E}[C.md(\vec{v})], \sigma\rangle \rightarrow \langle \mathcal{E}[[\overrightarrow{v/x}](\textbf{init}(e_b,\ e_p))], \sigma\rangle \quad \text{[R-CONSTR]}
$$
$$
\text{where } constr_{\Delta C}(md) = (\_, \vec{x}, e_b, e_p)
$$

$$
\Delta \vdash_{\Gamma} \langle \mathcal{E}[\textbf{preobj}(C,\ \textbf{preobj-val}(D,\ \overrightarrow{fd' \leftarrow v'},\ e'),\ \overrightarrow{fd \leftarrow v})], \sigma\rangle \rightarrow \quad \text{[R-PREOBJ]}
$$
$$
\langle \mathcal{E}[\textbf{preobj-val}(C,\ \overrightarrow{fd' \leftarrow v'} +\!\!+ \overrightarrow{fd \leftarrow v},\ e')], \sigma\rangle
$$

$$
\Delta \vdash_{\Gamma} \langle \mathcal{E}[\textbf{init}(\textbf{preobj-val}(C,\ \overrightarrow{fd \leftarrow v},\ e),\ e')], \sigma\rangle \rightarrow \quad \text{[R-INIT]}
$$
$$
\langle \mathcal{E}[\textbf{preobj-val}(C,\ \overrightarrow{fd \leftarrow v},\ \textbf{seq}\ \{\ e;\ e';\ \})], \sigma\rangle
$$

Figure 5.19: MIJMODEL reductions for object construction

2. **new init**(Symbol.uninterned("foo"), Symbol.register(self))

   Because the original constructor Symbol.interned dispatches to another constructor in the same class, R-CONSTR applies again. Here, though, the main body of the called constructor is a **preobj** expression.

3. **new init**(**init**(**preobj**(Symbol, Object.mkObj(), name ← "foo"),
   System.out.println("uninterned")),
   SymTable.register(self))

   Once again, MIJMODEL uses the R-CONSTR rule to evaluate the constructor application Object.mkObj. The main body of the called constructor is hardwired by T-CONSTR-OBJECT to produce a pre-object value directly.

```
class Symbol extends Object {
    String name;

    // constructor
    uninterned(String s) {
        preobj(Symbol, Object.mkObj(); name <- s);
    } postinit {
        System.out.println("uninterned");
    }

    // constructor
    interned(String s) {
        Symbol.uninterned(s);
    } postinit {
        SymTable.register(self);
    }
}
```

Figure 5.20: A Symbol class

4. **new init**(**init**(**preobj**(Symbol,
   init(**preobj-val**(Object, ε, self), self),
   name ← "foo"),
   System.out.println("uninterned")),
   SymTable.register(self))

   Now that the program has a full pre-object for the superclass, R-INIT moves the first postinit block, still unevaluated, into this pre-object.

5. **new init**(**init**(**preobj**(Symbol,
   preobj-val(Object, ε, **seq** { self; self; }),
   name ← "foo"),
   System.out.println("uninterned")),
   SymTable.register(self))

   The redex of this step triggers R-PREOBJ, meaning the Object pre-object is extended with the initialization clause for name to create a Symbol pre-object.

6. **new init**(**init**(**preobj-val**(Symbol, name ← "foo", **seq** { self; self; }),
   System.out.println("uninterned")),
   SymTable.register(self))

R-INIT fires again to move another postinit block into the pre-object. This postinit block originated with Symbol.uninterned.

7. **new init**(**preobj-val**(Symbol,
   name ← "foo",
   seq { self; self; System.out.println("uninterned"); }),
   SymTable.register(self))

   R-INIT fires one last time to move the postinit block from Symbol.interned into the pre-object.

8. **new preobj-val**(Symbol,
   name ← "foo",
   seq { self; self;
   System.out.println("uninterned");
   SymTable.register(self); })

   Finally, with a completed pre-object, MIJMODEL uses rule R-NEW to reify the pre-object into a full instance. In the result below, the **seq**'s subexpressions are the delayed postinit expressions of this step, with all occurrences of self replaced by *obj*, a fresh reference to the newly created instance. Also, R-NEW inserts one last reference to self/*obj* at the end of the **seq**, so that the entire expression evaluates to a reference to the new object.

9. **seq** { *obj*; *obj*;
   System.out.println("uninterned");
   SymTable.register(*obj*);
   obj; }

At this point, the reductions for **seq** expressions take effect, evaluating the subexpressions from left to right. This executes the delayed postinit blocks, in the correct order, and finally reduces to a reference to the newly-created instance, as desired.

Figure 5.21 contains the remaining reduction rules, most of which are straightforward adaptations of CLASSICJAVA's reduction rules. The R-FIELD-REF rule looks the object up in the store and then looks up the specified field in the instance's field table. The R-FIELD-SET rule is similar, but it updates the instance's field table instead and evaluates to the value of the right-hand side of the expression.

The R-CALL rule obtains the instance from the store purely for its class tag, which it uses to find the correct implementation of the indicated method. R-SUPER, in contrast, uses the type annotation to guide the method lookup. Both rules replace self with the value to the left of the dot during application.

Casting is implemented with four reduction rules. The first two, R-CLASS-CAST and R-BAD-CAST, address casts to a class type. Both rules look the object up in the store to

$\Delta \models_r \langle \mathcal{E}[addr.fd], \sigma \rangle \to \langle \mathcal{E}[\mathcal{F}(fd)], \sigma \rangle$ [R-FIELD-REF]
where $\sigma(addr) = (C, \mathcal{F})$

$\Delta \models_r \langle \mathcal{E}[addr.fd = v], \sigma \rangle \to \langle \mathcal{E}[v], \sigma' \rangle$ [R-FIELD-SET]
where $\sigma(addr) = (C, \mathcal{F})$ and $\sigma' = \sigma[addr : (C, \mathcal{F}[fd : v])]$

$\Delta \models_r \langle \mathcal{E}[addr.md(\vec{v})], \sigma \rangle \to \langle \mathcal{E}[[addr/\mathsf{self}, \overrightarrow{v/x}]e], \sigma \rangle$ [R-CALL]
where $\sigma(addr) = (C, \mathcal{F})$ and $meth_{\Delta C}(md) = (\_, \vec{x}, e)$

$\Delta \models_r \langle \mathcal{E}[\mathbf{super} \equiv addr{:}C.md(\vec{v})], \sigma \rangle \to \langle \mathcal{E}[[addr/\mathsf{self}, \overrightarrow{v/x}]e], \sigma \rangle$ [R-SUPER]
where $C \prec_\Delta D$ and $meth_{\Delta D}(md) = (\_, \vec{x}, e)$

$\Delta \models_r \langle \mathcal{E}[\mathbf{cast}(C,\ addr)], \sigma \rangle \to \langle \mathcal{E}[addr], \sigma \rangle$ [R-CLASS-CAST]
where $\sigma(addr) = (D, \mathcal{F})$ and $D <:_\Delta C$

$\Delta \models_r \langle \mathcal{E}[\mathbf{cast}(C,\ addr)], \sigma \rangle \to \langle \mathsf{error{:}\ bad\ cast}, \sigma \rangle$ [R-BAD-CAST]
where $\sigma(addr) = (D, \mathcal{F})$ and $D \not<:_\Delta C$

$\Delta \models_r \langle \mathcal{E}[\mathbf{cast}(\mathbf{Maybe}(t),\ \mathbf{some}_s\ v)], \sigma \rangle \to$ [R-SOME-CAST]
$\langle \mathcal{E}[\mathbf{some}_t\ \mathbf{cast}(t,\ v)], \sigma \rangle$

$\Delta \models_r \langle \mathcal{E}[\mathbf{cast}(\mathbf{Maybe}(t),\ \mathbf{none}_s)], \sigma \rangle \to \langle \mathcal{E}[\mathbf{none}_t], \sigma \rangle$ [R-NONE-CAST]

$\Delta \models_r \langle \mathcal{E}[\mathbf{valOf}\ \mathbf{some}_t\ v], \sigma \rangle \to \langle \mathcal{E}[v], \sigma \rangle$ [R-VALOF-SOME]

$\Delta \models_r \langle \mathcal{E}[\mathbf{valOf}\ \mathbf{none}_t], \sigma \rangle \to \langle \mathsf{error{:}\ valOf\ none}, \sigma \rangle$ [R-VALOF-NONE]

$\Delta \models_r \langle \mathcal{E}[\mathbf{seq}\ \{\ e;\ \}], \sigma \rangle \to \langle \mathcal{E}[e], \sigma \rangle$ [R-SEQ-DONE]

$\Delta \models_r \langle \mathcal{E}[\mathbf{seq}\ \{\ v;\ e_0;\ \vec{e};\ \}], \sigma \rangle \to \langle \mathcal{E}[\mathbf{seq}\ \{\ e_0;\ \vec{e};\ \}], \sigma \rangle$ [R-SEQ]

Figure 5.21: Other MIJMODEL reductions

obtain its dynamic type and perform a subtyping check. The other two reductions, R-SOME-CAST and R-NONE-CAST, handle casts to **Maybe** type. Both of these casts always succeed, although R-SOME-CAST just pushes the cast inside one layer of **Maybe**, and this new cast can of course fail.

The next two reduction rules, R-VALOF-SOME and R-VALOF-NONE, handle **valOf** expressions, either by stripping off the outermost **some** or by signaling an error if the argument is **none**. The final two rules address **seq** expressions. If a **seq** form has a single subexpression e, then R-SEQ-DONE applies, and e is the contractum. If, on the other hand, the **seq** has multiple subexpressions, and the first is a value, then R-SEQ specifies that evaluation simply drops the first subexpression.

5.5 Soundness

MIJMODEL satisfies a type soundness theorem. To state this result formally, we need the notion of store-typing consistency.

Definition 5 (Store-typing consistency)
A store typing Σ is consistent with a store σ under the definitions Δ, written $\Delta, \Sigma \vdash_{\mathsf{st}} \sigma$, if the following conditions hold:

1. $\mathrm{dom}(\Sigma) = \mathrm{dom}(\sigma)$; and
2. For all $addr$, C, $\mathcal{F}$ such that $\sigma(addr) = (C, \mathcal{F})$:
   a) $\Sigma(addr) = C$; and
   b) $\mathrm{dom}(\mathcal{F}) = \mathrm{dom}(field_{\Delta C})$; and
   c) For each $fd \in \mathrm{dom}(\mathcal{F})$, $\Delta, \emptyset, \Sigma \vdash_{\mathsf{te}} \mathcal{F}(fd) : t$ where $t <:_\Delta field_{\Delta C}(fd)$.

We can now state and prove a type soundness theorem.

Theorem 6 (Type Soundness for MIJMODEL)
If $\vdash_{\mathsf{ep}} \Theta\ re \Rightarrow \Delta\ e : t$, *then one of the following conditions must be true:*

1. $\Delta \vdash_{\mathsf{r}} \langle e, \emptyset \rangle \rightarrow^* \langle v, \sigma \rangle$, *where there exists a store typing* Σ *such that* $\Delta, \Sigma \vdash_{\mathsf{st}} \sigma$; *and* $\Delta, \emptyset, \Sigma \vdash_{\mathsf{te}} v : t'$; *and* $t' <:_\Delta t$; *or*
2. $\Delta \vdash_{\mathsf{r}} \langle e, \emptyset \rangle \rightarrow^* \langle \mathsf{error{:}\ bad\ cast}, \sigma \rangle$; *or*
3. $\Delta \vdash_{\mathsf{r}} \langle e, \emptyset \rangle \rightarrow^* \langle \mathsf{error{:}\ valOf\ none}, \sigma \rangle$; *or*
4. $\Delta \vdash_{\mathsf{r}} \langle e, \emptyset \rangle \Uparrow$.

Proof sketch

Standard strategy due to Wright and Felleisen [56]: induction on the length of the reduction sequence, using lemmas 7 and 10. Lemma 1 and the first antecedent prove that the elaborated program and the expression e are well-typed, allowing us to apply the subject reduction lemma. ■

As the soundness theorem says, a MIJMODEL program can terminate with an error in two cases: a bad cast or an attempt to evaluate **valOf none**$_t$. The bad cast exception is present in CLASSICJAVA, and although the valof-none error is new to MIJMODEL, we have also removed CLASSICJAVA's null-pointer exception. Because MIJMODEL does not include a default value (*null* or otherwise) for uninitialized fields, the soundness theorem implies that a program cannot read a field before initializing it, as such a reference could not produce a value of the correct type. Further, MIJMODEL does not add any new run-time errors for references to uninitialized fields, so the soundness theorem proves that MIJMODEL's type system prevents all such accesses statically.

Lemma 7 (Subject Reduction)
If

- $\vdash_{\mathsf{tp}} \Delta\ e_0 : t_0$*; and*
- $\Delta, \varnothing, \Sigma \vdash_{\mathsf{te}} e : t$*; and*
- $\Delta, \Sigma \vdash_{\mathsf{st}} \sigma$*; and*
- $\Delta \vdash_{\mathsf{r}} \langle e, \sigma\rangle \to \langle e', \sigma'\rangle$*;*

then there exists a store typing Σ' *such that*

- $\Delta, \Gamma, \Sigma' \vdash_{\mathsf{te}} e' : t'$*; and*
- $t' <:_\Delta t$*; and*
- $\Delta, \Sigma' \vdash_{\mathsf{st}} \sigma'$.

Proof: Case analysis on the reduction, with the standard substitution and context-replacement lemmas. In order to prove the R-PREOBJ case, we also need lemma 8, which allows us to replace a bound variable's type with a subtype.

Lemma 8 (Bound-Variable Subtyping)
Let Δ *and* e_0 *be a definition sequence and expression, respectively, such that* $\vdash_{\mathsf{tp}} \Delta\ e_0 : t_0$. *If* $\Delta, \Gamma[x : s], \Sigma \vdash_{\mathsf{te}} e : t$ *and* $s' <:_\Delta s$*, then* $\Delta, \Gamma[x : s'], \Sigma \vdash_{\mathsf{te}} e : t'$*, where* $t' <:_\Delta t$.

Proof sketch

Induction on the derivation of $\Delta, \Gamma[x : s], \Sigma \vdash_{\mathsf{te}} e : t$. Key to this proof are two properties of the type system: first, a class may not contain a field definition that

shadows one of its superclass's fields;[11] and second, if a class overrides one of its superclass's methods, the overriding method must have the same signature as the original. (These allow us to specialize the type of the expression before the dot in a field reference, field assignment, or method call.) In all other contexts, the type rules explicitly allow expressions to have a subtype of the expected type. ■

To state the progress lemma, we need the notion of a "hole term."

Definition 9 (Hole Term)
The metavariable *ht* ranges over those expressions that can appear within the hole of an evaluation context on the left-hand side of a reduction rule. (This definition deliberately does not consider any side conditions on reduction rules.)

$$
\begin{array}{rcl}
ht & ::= & \textbf{new preobj-val}(C,\ \overrightarrow{fd \leftarrow v},\ e) \mid addr.fd \mid addr.fd = v \mid addr.md(\vec{v}) \\
& \mid & \textbf{super} \equiv addr : C.md(\vec{v}) \mid C.md(\vec{v}) \mid \textbf{cast}(C,\ addr) \\
& \mid & \textbf{cast}(\textbf{Maybe}(ot),\ \textbf{some}_t\ v) \mid \textbf{cast}(\textbf{Maybe}(ot),\ \textbf{none}_t) \mid \textbf{valOf some}_t\ v \\
& \mid & \textbf{valOf none}_t \mid \textbf{preobj}(C,\ \textbf{preobj-val}(C,\ \overrightarrow{fd \leftarrow v},\ e),\ \overrightarrow{fd \leftarrow v}) \\
& \mid & \textbf{seq}\ \{\ v;\ \vec{e};\ e;\ \} \mid \textbf{seq}\ \{\ e;\ \} \mid \textbf{init}(\textbf{preobj-val}(C,\ \overrightarrow{fd \leftarrow v},\ e),\ e)
\end{array}
$$

Lemma 10 (Progress)
If $\vdash_{\mathsf{tp}} \Delta\ e_0 : t_0$ *and* $\Delta, \varnothing, \Sigma \vdash_{\mathsf{te}} e : t$ *and* $\Delta, \Sigma \vdash_{\mathsf{st}} \sigma$*, then either*

1. *e is a value; or*
2. *there exist* e', σ' *such that* $\Delta \vdash_{\mathsf{r}} \langle e, \sigma\rangle \rightarrow \langle e', \sigma'\rangle$*; or*
3. *there exists* σ' *such that* $\Delta \vdash_{\mathsf{r}} \langle e, \sigma\rangle \rightarrow \langle \mathsf{error}\text{: bad cast}, \sigma'\rangle$*; or*
4. *there exists* σ' *such that* $\Delta \vdash_{\mathsf{r}} \langle e, \sigma\rangle \rightarrow \langle \mathsf{error}\text{: valOf none}, \sigma'\rangle$.

Proof sketch
If *e* is a value, then the lemma holds trivially. Otherwise, a standard decomposition lemma applies, and there exist context $\mathcal{E}$ and hole term *ht* such that $e = \mathcal{E}[ht]$. The proof then proceeds by case analysis on *ht* in which we demonstrate that the side conditions for the matching reductions are either exhaustive or that they follow from the assumption that *e* is well-typed. ■

Soundness's Effects on the Semantics

Like many other models, MIJMODEL's soundness proof requires annotations on the abstract syntax that neither the static nor dynamic semantics uses. Specifically, subject reduction requires type annotations on **some** expressions.

[11] Implementations of MIJAVA can circumvent this restriction by alpha-renaming all field definitions. For example, the CLASSICJAVA model supports field shadowing by (effectively) tagging each field name with the name of the defining class.

The type tag on **some** expressions allows subject reduction to succeed even with the invariance of **Maybe** types. Consider the following MIJMODEL program excerpt:

```
class C ext D {
    ...
    Maybe(D) m() { some (cast(D, self)) }
}
```

The highlighted expression has type **Maybe**(D), as required by the method's signature, and elaborates to $\textbf{some}_{\mathsf{D}}$ (**cast**(D, self)). (The cast is necessary here because **Maybe** types are invariant.) This elaborated expression reduces to $\textbf{some}_{\mathsf{D}}$ (self). According to the TYPE-SOME rule, this expression has type **Maybe**(D), which satisfies subject reduction. Without the type tag, the post-elaboration type system could only reconstruct the type of the expression as **Maybe**(C), which is *not* a subtype of **Maybe**(D). Reducing this term would therefore violate subject reduction.

Chapter 6

Translating MIJava to Java

In order to support the claim that we provide a migration path from existing Java programs to MIJava, and in order to allow exploration of the practical consequences of MIJava's features in production code, we have defined an automatic translation from MIJava to Java. This translation takes a single MIJava class definition and produces an equivalent Java class definition that can be compiled by a stock Java compiler as part of a larger program.

Our translation must meet the following requirements:

1. It must allow each constructor to contain arbitrarily many `preobj` expressions.
2. It must allow different `preobj` expressions within the same constructor to invoke different constructors in the superclass.
3. It must preserve object creation interfaces for interoperability with Java classes. That is, Java code must be able to instantiate a translated MIJava class with an expression of the form `new C(...)`.

Also, the translation would ideally support first-class pre-objects, and allow `preobj` expressions to appear in any context.[1] Since migration from Java to MIJava does not generally require first-class pre-objects, we focus on the remaining requirements.

Java places two restrictions on constructors that prevent us from implementing MIJava constructors purely as Java constructors. First, a Java constructor must begin with a (possibly implicit) invocation of a constructor in the superclass or another constructor in the same class. Second, Java prohibits access to `this` in the argument list to this constructor invocation. Taken together, these constraints imply that the superclass constructor executes completely before the subclass constructor can begin to initialize the subclass's fields. This is inconsistent with MIJava's order of evaluation, so our translation must use a different strategy.

[1] Providing these two additional capabilities would appear to require the use of a heap-allocated representation of pre-objects, which we avoid in our translation strategy.

```
class Shape {
    Point anchorPoint;

    public Shape(Point locn) {
        return preobj { super(); anchorPoint <- locn; };
    }
}

class Rect extends Shape {
    // anchor point is lower-left corner
    Point upperRight;

    public Rect(Point lowLft, Point upRt) {
        return preobj { super(lowLft); upperRight <- upRt; };
    }

    public Rect(double x, double y, double w, double h, Canvas c) {
        return Rect(new Point(x, y), new Point(x + w, y + h));
    } postinit {
        this.draw(c);
    }

    protected void draw(Canvas c) { ... }
}
```

Figure 6.1: Translation Example

6.1 Overview of Object Creation

The translated constructors, in addition to initializing the new object's fields, must also maintain a history of the constructors invoked on the object in order to execute the postinit blocks in the correct order. Managing this history requires a complex protocol between superclasses and subclasses, so the translation is best described with an extended example. For our example, we use a slightly modified version of the classes from Chapter 2, reproduced here in figure 6.1. In order to demonstrate certain features of the translation, we have modified `Rect`'s second constructor to take four `double` arguments, rather than two `Points`, and to construct the superclass constructor argument and field initializer from them.

We begin with an overview of the object creation protocol, focusing on control flow and the accumulation and use of the constructor history record. In this overview, we refer to the original MIJAVA classes as $\texttt{Shape}_M$ and $\texttt{Rect}_M$ and to their Java translations as $\texttt{Shape}_J$ and $\texttt{Rect}_J$, respectively; we omit the subscript when the meaning is clear from context or when the distinction is immaterial.

In our example, $\texttt{Shape}_M$ has one constructor, and $\texttt{Rect}_M$ has two. As a shorthand, we introduce names for the three constructors: $\texttt{Shape}_M\texttt{.Shape}$ for Shape's only constructor, $\texttt{Rect}_M\texttt{.Rect}_2$ for $\texttt{Rect}_M$'s two-argument constructor, and $\texttt{Rect}_M\texttt{.Rect}_5$ for its five-argument constructor.

As described above, each MIJAVA constructor becomes three members in Java: a constructor, a field initialization method, and a postinit method. We continue the naming convention above, yielding the following members in the Java classes:

$\texttt{Shape}_J\texttt{.Shape}$	$\texttt{Rect}_J\texttt{.Rect}_2$	$\texttt{Rect}_J\texttt{.Rect}_5$
$\texttt{Shape}_J\texttt{.fieldinit}$	$\texttt{Rect}_J\texttt{.fieldinit}_2$	$\texttt{Rect}_J\texttt{.fieldinit}_5$
$\texttt{Shape}_J\texttt{.postinit}$	$\texttt{Rect}_J\texttt{.postinit}_2$	$\texttt{Rect}_J\texttt{.postinit}_5$

In the actual results of the translation, all `fieldinit` methods have the same name, as do all `postinit` methods, and we rely on Java method overloading to distinguish between multiple methods in the same class.

For the overview, we include only the arguments and return values necessary to understand the initialization protocol, principally those involved in the creation, modification, and use of the construction history. Each `fieldinit` method builds and returns the construction history, and each `postinit` method takes the construction history as an argument. We delay discussion of the other arguments until the detailed discussion of $\texttt{Shape}_J$ and $\texttt{Rect}_J$.

To explain the protocol, we step through the evaluation of the expression

```
new Rect(1.0, 2.0, 3.0, 4.0, new Canvas(...)).
```

The table in figure 6.2 shows the call stack and the constructor history at every step during execution. At each step, the currently executing method is listed at the top of the call stack. Steps 1–7, above the horizontal line, represent the part of the process concerned with pre-object creation and field initialization. In steps 5–7, we show the accumulated constructor history in the right column. In each step, the constructor history is that returned by the currently executing method in the same step. This returned constructor history represents the constructors invoked during the current method's dynamic extent, in order by invocation, with the first constructor invoked at the top of the list.[2] Equivalently, the history indicates the list of postinit blocks that must run before the currently executing method's postinit block, in reverse order—that is, the run-time system must execute the postinit block on the *bottom* of the history list first. (We explain this apparent inconsistency in the order of the history list below.)

[2] This list contains only constructors invoked on the object whose initialization protocol is executing. If a constructor evaluates a `new` expression, then the program starts a new initialization protocol for the second object. This second "copy" of the protocol has its own constructor history, which is separate from the first.

	call stack	constructor history
1.	$\texttt{Rect}_J.\texttt{Rect}_5$	—
2.	$\texttt{Rect}_J.\texttt{fieldinit}_5$ $\texttt{Rect}_J.\texttt{Rect}_5$	—
3.	$\texttt{Rect}_J.\texttt{fieldinit}_2$ $\texttt{Rect}_J.\texttt{fieldinit}_5$ $\texttt{Rect}_J.\texttt{Rect}_5$	—
4.	$\texttt{Shape}_J.\texttt{fieldinit}$ $\texttt{Rect}_J.\texttt{fieldinit}_2$ $\texttt{Rect}_J.\texttt{fieldinit}_5$ $\texttt{Rect}_J.\texttt{Rect}_5$	—
5.	$\texttt{Rect}_J.\texttt{fieldinit}_2$ $\texttt{Rect}_J.\texttt{fieldinit}_5$ $\texttt{Rect}_J.\texttt{Rect}_5$	$\texttt{Shape}_M.\texttt{Shape}$
6.	$\texttt{Rect}_J.\texttt{fieldinit}_5$ $\texttt{Rect}_J.\texttt{Rect}_5$	$\texttt{Rect}_M.\texttt{Rect}_2$ $\texttt{Shape}_M.\texttt{Shape}$
7.	$\texttt{Rect}_J.\texttt{Rect}_5$	$\texttt{Rect}_M.\texttt{Rect}_2$ $\texttt{Shape}_M.\texttt{Shape}$
8.	$\texttt{Rect}_J.\texttt{postinit}_5$ $\texttt{Rect}_J.\texttt{Rect}_5$	$\texttt{Rect}_M.\texttt{Rect}_2$ $\texttt{Shape}_M.\texttt{Shape}$
9.	$\texttt{Rect}_J.\texttt{postinit}_2$ $\texttt{Rect}_J.\texttt{postinit}_5$ $\texttt{Rect}_J.\texttt{Rect}_5$	$\texttt{Shape}_M.\texttt{Shape}$
10.	$\texttt{Shape}_J.\texttt{postinit}$ $\texttt{Rect}_J.\texttt{postinit}_2$ $\texttt{Rect}_J.\texttt{postinit}_5$ $\texttt{Rect}_J.\texttt{Rect}_5$	—
11.	$\texttt{Rect}_J.\texttt{postinit}_2$ $\texttt{Rect}_J.\texttt{postinit}_5$ $\texttt{Rect}_J.\texttt{Rect}_5$	—
12.	$\texttt{Rect}_J.\texttt{postinit}_5$ $\texttt{Rect}_J.\texttt{Rect}_5$	—
13.	$\texttt{Rect}_J.\texttt{Rect}_5$	—

Figure 6.2: Stack trace and constructor history during `Rect` construction

The remainder of the process, steps 8–13, executes the constructors' postinit actions. These `postinit` methods take the constructor history as a parameter, and the history listing on each line of the table shows the history as received by the method at the top of the stack during the same execution step. Steps 8 and 9 move entries from the constructor history onto the call stack by invoking the necessary `postinit` methods. Once control has reached the base class's `postinit` method, in step 10, the program begins executing the actual postinit actions, with those at the top of the inheritance hierarchy running first, as required.

In more detail, evaluation of the expression

```
new Rect(1.0, 2.0, 3.0, 4.0, new Canvas(...))
```

proceeds as follows:

1. Execution begins by calling $\texttt{Rect}_J\texttt{.Rect}_5$, which initiates the object creation protocol. Its first action is to call $\texttt{Rect}_J\texttt{.fieldinit}_5$, the fieldinit method corresponding to $\texttt{Rect}_M\texttt{.Rect}_5$.

2. $\texttt{Rect}_J\texttt{.fieldinit}_5$ performs the actions from the pre-object-construction section of $\texttt{Rect}_M\texttt{.Rect}_5$. Since this section begins by invoking $\texttt{Rect}_M\texttt{.Rect}_2$, the $\texttt{fieldinit}_5$ method calls $\texttt{Rect}_J\texttt{.fieldinit}_2$, the corresponding field initialization method.

3. $\texttt{Rect}_J\texttt{.fieldinit}_2$ performs the actions corresponding to the `preobj` expression in $\texttt{Rect}_M\texttt{.Rect}_2$. This expression begins by calling $\texttt{Shape}_M\texttt{.Shape}$, so the translated method calls `Shape.fieldinit`.

4. `Shape.fieldinit` performs the computation equivalent to the `preobj` form in $\texttt{Shape}_M\texttt{.Shape}$. Here, however, the superclass is `Object`, which has an empty constructor, so no call to a `fieldinit` method is necessary, and the program simply initializes `Shape`'s field. (By initializing the fields "in place," we avoid the need for an explicit representation of pre-objects.)

   Like every `fieldinit` method, $\texttt{Shape}_J\texttt{.fieldinit}$ must return a record of the constructors that it called, so the program can later invoke the correct postinit methods. `Shape` is a special case because it is an immediate subclass of `Object`, so $\texttt{Shape}_J\texttt{.fieldinit}$ returns an empty constructor history, indicating that it did not call (the equivalent of) a MIJAVA constructor.

5. Control returns to $\texttt{Rect}_J\texttt{.fieldinit}_2$, which continues with the field initialization clauses by performing the corresponding field assignment to `upperRight`. Before returning, it adds `Shape.Shape` to the head of the constructor history list and returns the augmented list.

6. Control returns to $\texttt{Rect}_J\texttt{.fieldinit}_5$. Since this method corresponds to a constructor without an explicit `preobj` expression, it does not need to perform any additional field initialization here. Therefore, it merely adds $\texttt{Rect}_J\texttt{.Rect}_2$ to the head of the constructor history list and returns.

7. In this step, control returns to $\texttt{Rect}_J\texttt{.Rect}_5$. By this point, the program has initialized all of the fields in the newly-created `Rect`; all that remains is to execute the postinit blocks. To that end, the constructor starts the postinit part of the protocol by invoking $\texttt{Rect}_J\texttt{.postinit}_5$. (Unlike the calls to `postinit` methods below, this particular call is hard-wired, because the call in step 2 above to $\texttt{Rect}_J\texttt{.fieldinit}_5$ is also hard-wired. Therefore, $\texttt{Rect}_J\texttt{.Rect}_5$ does not need to inspect the constructor history to determine which `postinit` method to call.)

8. Because superclass postinit blocks must execute before subclass postinit blocks, $\texttt{Rect}_J\texttt{.postinit}_5$ begins by calling the postinit method corresponding to the constructor called by $\texttt{Rect}_J\texttt{.fieldinit}_5$. Specifically, it removes a record from the head of the constructor history list and invokes the corresponding `postinit` method, $\texttt{Rect}_J\texttt{.postinit}_2$.

9. $\texttt{Rect}_J\texttt{.postinit}_2$ begins the same way: it removes the first item from the constructor history list and invokes the corresponding method, $\texttt{Shape}_J\texttt{.postinit}$. Notice that the constructor history returned from $\texttt{Rect}_J\texttt{.fieldinit}_2$ is the same as that passed to $\texttt{Rect}_J\texttt{.postinit}_2$; this correspondence holds throughout the protocol.

10. $\texttt{Shape}_J\texttt{.postinit}$ does not need to process any further entries in the constructor history list, because $\texttt{Shape}_M\texttt{.Shape}$ invoked `Object`'s constructor in step 4, and because `Object`'s constructor does not have a postinit block. (Indeed, if this list is not empty by the time execution reaches this point, then at least one of the classes has not followed the protocol correctly.) Therefore, `Shape.postinit` executes the statements from $\texttt{Shape}_M\texttt{.Shape}$'s postinit block (none, in this case) and returns.

11. $\texttt{Rect}_J\texttt{.postinit}_2$ regains control. Because `Shape`'s postinit actions have completed, it is free to execute the postinit block for `Rect`'s first constructor. Since $\texttt{Rect}_M\texttt{.Rect}_2$'s postinit block is also empty, the method returns immediately.

12. Control now returns to $\texttt{Rect}_J\texttt{.postinit}_5$. Here, the corresponding constructor $\texttt{Rect}_M\texttt{.Rect}_5$ has a non-empty postinit block, so the method executes all the statements in this block and returns.

13. With the last of the postinit actions complete, control returns to the constructor $\texttt{Rect}_J\texttt{.Rect}_5$, which has no further actions to perform. It therefore returns control to the context of the original `new` expression.

In addition to the above calls, Java requires $\texttt{Rect}_J\texttt{.Rect}_5$ to begin with a call to a constructor, either in `Rect` or in `Shape`. Given the class interfaces as we have presented them, it cannot invoke any of the available constructors, because all constructors in both classes trigger the execution of the complete protocol, including the execution of postinit blocks. If, for instance, $\texttt{Shape}_J\texttt{.Shape}_1$ had a postinit block, then $\texttt{Rect}_J\texttt{.Rect}_5$ could not invoke $\texttt{Shape}_J\texttt{.Shape}_1$, even indirectly, because its postinit block would execute before `Rect` had a chance to initialize its fields, which could lead to an uninitialized field reference as in Chapter 2.

To satisfy this requirement, the translation gives each class a "stub" constructor that takes a single argument of type `StubConstr`. This class, defined in a support library and described below, has no fields or methods and serves only to distinguish this stub constructor from the other constructors in the class. The stub constructor's body contains only a call to the superclass's stub constructor, except in classes that, like `Shape`, directly extend `Object`. Since `Object`'s constructor performs no actions, `Shape`'s stub constructor invokes `Object`'s default constructor. The constructors that, like $\texttt{Rect}_J\texttt{.Rect}_5$, correspond to constructors in the original MIJAVA definition of the class simply invoke this stub constructor in the superclass, satisfying Java's requirements while retaining the correct order of evaluation.

6.2 Support Classes

In this overview, we have represented the constructor history as a stack or sequence of constructor labels. In the actual implementation, we represent the history as an instance of `ClosureList`, a class defined in a support library that contains a sequence of `Closure` objects. Each `Closure` implements a `call` method that, when invoked, calls the correct `postinit` method.

Figure 6.3 defines `Closure`, `ClosureList`, and other support classes used by the output of the translation.[3] The `Closure` interface serves as a tag type, similar to Java's `Serializable` interface; we discuss its implementing classes (and why the interface is empty) below. `ClosureList` is an applicative list of `Closure`s with the obvious implementation. The translation uses the exception `ProtocolErrorException` to signal protocol failures such as a `ClosureList` that is unexpectedly empty or that begins with a `Closure` of unexpected type, situations that should never arise during the execution

[3] In the actual implementation, these support classes are in a separate package to avoid name collisions. We omit the package declarations here for brevity.

```
public interface Closure { }

public abstract class ClosureList {
    private ClosureList () { }

    public static ClosureList empty()   { /* elided */ }
    public ClosureList cons(Closure hd) { /* elided */ }

    public abstract Closure head();
    public abstract ClosureList tail();
    public abstract boolean isEmpty();

    private static class Empty extends ClosureList { /* elided */ }
    private static class Cons extends ClosureList  { /* elided */ }
}

public class ProtocolErrorException extends RuntimeException {
    public ProtocolErrorException() {
        super("Initialization protocol error");
    }
}

public final class StubConstr { }
```

Figure 6.3: Support classes for MIJAVA-to-Java translation

of translated code. The final class, `StubConstr`, ensures that the stub constructors described in the previous section have unique signatures for method overloading.

6.3 Example Translation: Shape

To see some of the details of the object initialization process, consider the result of translating the `Shape` class, in figure 6.4. The `anchorPoint` field declaration is the same as in the original; the translation never modifies existing field or method declarations. The translation adds the constructor `Shape(StubConstr)` for the benefit of subclasses, as discussed above. Since `Shape` extends `Object`, there is no need to perform any action here, as Java automatically synthesizes a call to `Object`'s default constructor, which is the desired behavior.

The `fieldinit` method is the translation of $\texttt{Shape}_M\texttt{.Shape}$'s field initialization section. It expects the same arguments as the original constructor, plus an additional argument of type `Shape`, representing the object under construction. This argument gives `fieldinit`, a static method, access to the object's fields to initialize them.

```
class Shape {
    Point anchorPoint;

    protected Shape(StubConstr dummy) { }

    protected static ClosureList fieldinit(Shape self, Point locn) {
        // return preobj { super(); anchorPoint <- locn; };
        self.anchorPoint = locn;
        return ClosureList.empty();
    }

    protected static void postinit(Shape self,
                                   ClosureList closures,
                                   Point locn) {
        Closure first = (closures.isEmpty()) ? null : closures.head();
        if (first == null) {
            // fieldinit() called Object's constr, so NOP.
        } else {
            throw new ProtocolErrorException());
        }
        // original postinit block, if any, goes here.
    }

    public Shape(Point locn) {
        Shape.postinit(this, Shape.fieldinit(this, locn), locn);
    }
}
```

Figure 6.4: Translation of Shape to Java

This method performs all of the actions from $\texttt{Shape}_M\texttt{.Shape}$'s field initialization section, with the `return preobj` statement replaced by code that initializes the local fields and computes the constructor history. Since the only action is the `return preobj` statement, the translated method begins by initializing the `self.anchorPoint` field. (Had the `preobj` expression invoked a constructor other than `Object`'s, that call would have preceded the field initialization, as we demonstrate below with `Rect`'s translation.) Finally, the `fieldinit` method returns an empty list of closures, indicating that the only invoked constructor is in `Object`, so the `postinit` method does not have to call a superclass method before executing the postinit block.

The `postinit` method is the translation of the original constructor's postinit block, which is empty. Like the `fieldinit` method, it expects the same arguments as the original constructor plus a `self` reference, but it also expects the list of remaining `Closures` as its second argument. Because `Shape.fieldinit` always returns an empty

list, `postinit` expects the list to be empty and throws an exception if it is not. Since $\texttt{Shape}_M\texttt{.Shape}$'s postinit block is empty, `postinit` returns without performing any other actions.

Finally, the `Shape(Point)` constructor allows clients to instantiate `Shape` directly. It begins by calling the `fieldinit` method above, passing a reference to the current object and the original constructor argument and obtaining a list of closures that invoke the necessary `postinit` methods. The constructor finishes by passing a reference to the object being initialized, the closure list, and the original constructor arguments to the `postinit` method.

6.4 Example Translation: `Rect`

The `Rect` class is more complex, so its translation is correspondingly longer and more intricate than `Shape`'s. Figures 6.5 and 6.6 show the resulting class definition. As before, the translation preserves the field declaration for `upperRight` and the `draw` method definition, in the second figure. Also, the translation synthesizes the constructor that takes a single `StubConstr` argument; in this case, the generated constructor invokes `Shape`'s stub constructor, as discussed in the overview above.

The next two members are implementations of the `Closure` interface, corresponding to the constructors that `Rect` invokes: $\texttt{Shape}_M\texttt{.Shape}$ and $\texttt{Rect}_M\texttt{.Rect}_2$. Each closure expects a reference to the object under construction and the remaining closures and invokes the appropriate `postinit` method. The closure's creator is responsible for providing values for the rest of the `postinit` method's arguments, as arguments to the closure's constructor.

The next three members correspond to the first constructor in $\texttt{Rect}_M$. As before, the `fieldinit` method is the translation of field initialization section, which consists entirely of a `return preobj { ... }` statement. Here, the `preobj` expression invokes `Shape`'s constructor, so `Rect.fieldinit` invokes `Shape.fieldinit` and saves the resulting `ClosureList`. Now that the superclass constructor has returned, `Rect.fieldinit` initializes `self`'s `upperRight` field, conses a closure that invokes `Shape.postinit` onto the front of the list, and returns the augmented list.

The `postinit` method is similar to `Shape`'s. Because, however, $\texttt{Rect}_M\texttt{.Rect}_2$'s field initialization section invokes only $\texttt{Shape}_M\texttt{.Shape}$ and no other constructors, the translation's invariants specify that the `closures` list must not be empty, and its first element must be an instance of `ShapeConstr`. If either of these two conditions does not hold, $\texttt{Rect}_J\texttt{.postinit}_2$ throws an exception. Otherwise, it downcasts the `Closure` object to `ShapeConstr` and invokes it, then returns. (The dynamic type check and downcast are necessary because `Closure` cannot declare the `call` method, due to Java's exception-handling rules. We discuss this further after completing the `Rect` example.)

```
class Rect extends Shape {
    Point upperRight;

    protected Rect(StubConstr dummy) { super(dummy); }

    private static final class ShapeClosure implements Closure {
        Point lowLft;
        public ShapeClosure(Point p) { lowLft = p; }
        public void call(Rect self, ClosureList closures) {
            Shape.postinit(self, closures, lowLft);
        }
    }

    private static final class RectClosure implements Closure {
        Point lowLft;
        Point upRt;

        public RectClosure(Point p1, Point p2) {
            lowLft = p1;
            upRt = p2;
        }

        public void call(Rect self, ClosureList closures) {
            Rect.postinit(self, closures, lowLft, upRt);
        }
    }

    protected ClosureList fieldinit(Rect self, Point lowLft, Point upRt) {
        ClosureList closures = Shape.fieldinit(self, lowLft);
        self.upperRight = upRt;
        return closures.cons(new ShapeClosure(lowLft));
    }

    protected void postinit(Rect self, ClosureList closures,
                            Point lowLft, Point upRt) {
        Closure c = (closures.isEmpty()) ? null : closures.head();
        if (c == null)
            throw new ProtocolErrorException();
        else if (c instanceof ShapeClosure)
            ((ShapeClosure) c).call(self, closures.tail());
        else
            throw new ProtocolErrorException();
    }

    public Rect(Point lowLft, Point upRt) {
        super((StubConstr) null);
        Rect.postinit(this,
                      Rect.fieldinit(this, lowLft, upRt);
                      lowLft, upRt);
    }
    // ... continued
```

Figure 6.5: Translation of `Rect` to Java

```
// class Rect extends Shape, continued:
    protected ClosureList fieldinit(Rect self, double x, double y,
                                    double w, double h, Canvas c) {
        Point tmp1 = new Point(x, y);
        Point tmp2 = new Point(x + w, y + h);
        ClosureList closures = Rect.fieldinit(self, tmp1, tmp2);
        return closures.cons(new RectClosure(tmp1, tmp2));
    }

    protected void postinit(Rect self, ClosureList closures,
                            double x, double y, double w, double h,
                            Canvas c) {
        Closure first = (closures.isEmpty()) ? null : closures.head();
        if (first == null)
            throw new EmptyListException();
        else if (first instanceof RectClosure)
            ((RectClosure) first).call(self, closures.tail());

        self.draw(c);
    }

    public Rect(double x, double y, double w, double h, Canvas c) {
        super((StubConstr) null);
        Rect.postinit(this,
                      Rect.fieldinit(this, x, y, w, h, c),
                      x, y, w, h, c);
    }

    protected void draw(Canvas c) { ... }
}
```

Figure 6.6: Translation of `Rect` to Java, continued

Finally, the `Rect(Point, Point)` constructor provides the client instantiation interface. As discussed in the overview, the constructor begins by invoking the superclass's stub constructor, then calls $\texttt{Rect}_J\texttt{.fieldinit}_2$ on `this` and the original arguments to initialize the fields and produce a `ClosureList`. Finally, this constructor executes the pending postinit blocks by passing `this`, the computed `ClosureList`, and the original arguments to $\texttt{Rect}_J\texttt{.postinit}_2$.

Figure 6.6 contains the translation of $\texttt{Rect}_M\texttt{.Rect}_5$, which differs from the first constructor $\texttt{Rect}_M\texttt{.Rect}_2$ in two significant respects. First, instead of calling a superclass constructor, it delegates to another constructor in `Rect`. Second, the arguments to the called constructor are expressions that require evaluation with side effects, instead of being simple identifier references. Both require slight adjustments to the translation.

Because $\texttt{Rect}_J\texttt{.fieldinit}_5$ uses $\texttt{Shape}_M\texttt{.Shape}$'s arguments twice, once in the arguments to $\texttt{Shape}_J\texttt{.fieldinit}$ and once to `RectClosure`'s constructor, the translator produces local variable declarations to hold their results, to avoid executing the expressions multiple times. Additionally, because this constructor delegates to another rather

than creating a pre-object directly, the `fieldinit` method simply conses an entry onto the closure list and does not assign to the fields.

$\texttt{Rect}_J\texttt{.postinit}_5$ is essentially the same as $\texttt{Rect}_J\texttt{.postinit}_2$, although the expected type of the first closure is different. Also, $\texttt{Rect}_M\texttt{.Rect}_5$ has a non-empty postinit block, so the translation must copy its statements into the `postinit` method, after replacing all instances of `this` with `self`. The client constructor $\texttt{Rect}_J\texttt{.Rect}_5$ works exactly like $\texttt{Rect}_J\texttt{.Rect}_2$ does.

6.5 Closure Implementation and Invocation

In the definition above, the `Closure` interface defines no members, not even the `call` method. As a result, the `postinit` methods created by the translation must perform a type-check and downcast on `Closure` instances before using them. This is widely regarded as bad object-oriented style, but it is unavoidable in this situation due to Java's rules regarding declared exceptions.

Consider an alternative definition of the `Closure` interface:

```
public interface Closure {
    public call(ClosureList closures) throws E1, E2, E3;
}
```

Here, we have specified the `call` method with a specific set of exceptions that it may throw, in the highlighted annotation.[4] With this declaration, no class that implements `Closure` may throw anything other than `E1`, `E2`, or `E3` from its `call` method. Similarly, any code that invokes the `call` method through the `Closure` interface must be prepared to handle exceptions of type `E1`, `E2`, and `E3`. Because of `Closure`'s role in the constructor protocol, this `throws` declaration simultaneously prevents every constructor in the program from throwing an exception not listed *and* requires the context of every `new` expression in the program to handle all of the listed exception types. This is clearly impractical. Omitting the `throws` clause entirely does not improve the situation, because this is equivalent to specifying an empty set of exceptions, which would prevent any constructor from ever throwing an exception.

Only by not declaring the `call` method in the `Closure` interface can the translation allow constructors to throw arbitrary exceptions without placing an undue burden on the rest of the program. Since the program cannot, as a result, assume the existence of the `call` method in the closure, it must perform the dynamic type-check and downcast in the `postinit` methods, as in the examples above. As long as the translation copies any `throws` clause present on the original MIJAVA constructor to the `fieldinit`

[4] We are concerned solely with this `throws` declaration and thus ignore unchecked exceptions in this section.

method, `postinit` method, and synthesized constructor, the resulting program compiles, and constructor authors have the necessary flexibility with respect to uncaught exceptions.

Note

In addition to the `throws` clause, the `call` method's first argument, the object being initialized, also prevents us from declaring `call` in the `Closure` interface. Because the type of this argument must correspond to the class in which the closure implementation appears, the only type that works for all possible closures is `Object`, which would require every implementation of the `call` method to downcast the argument.

However, this is not in fact a serious problem. We could refactor the `Closure` protocol to expect the object under construction as an argument to the `Closure`'s implementation's constructor rather than as an argument to the `call` method. In other words, we could simply close the function over `this` as well as over the arguments to the called constructor.

The branches in the type dispatch correspond to those constructors invoked by the original MIJAVA constructor (and thus the corresponding `fieldinit` method); the compiler can easily compute this list statically. We also include a catch-all case to deal with unexpected closure types, which throws a `ProtocolErrorException`, but this situation should never arise in programs created by this translation.

6.6 Complete Specification of the Translation

From detailed examples we proceed to a complete specification of the translation of a MIJAVA class C, whose superclass is D. For simplicity, we assume for the moment that D is either `Object` or that it is also defined in MIJAVA and thus subject to the same translation. As above, the translation of class C has several components: the classes that represent closures, the stub constructor, the three members corresponding to each MIJAVA constructor, and finally the remaining members of the original class, copied without modification.

Generating the Closure Definitions

For each constructor *invoked* by a class, except for `Object`'s constructor, the translation synthesizes a closure, represented as a nested class, that invokes the appropriate `postinit` method. For the invoked constructor

$$C'(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ \texttt{throws}\ E_1, \ldots, E_m,$$

Constructor access	`fieldinit` access
`public`	`protected`
`protected`	`protected`
`private`	`private`

Table 6.1: Access control for generated methods

with formal arguments $x_1,\ldots,x_n$ of types $\tau_1,\ldots,\tau_n$ and declared exception types $E_1,\ldots,E_m$, the translation synthesizes the following closure definition:

```
private static final class CClosure extends Closure {
  τ1 x1;
  ...
  τn xn;
  public CClosure(τ1 x1', ..., τn xn') {
      x1 = x1'; ...; xn = xn';
  }

  public void call(C self, ClosureList closures) throws E1, ..., Em {
    C'.postinit(self, closures, x1, ..., xn);
  }
}
```

The names *CClosure* and $x'_1,\ldots,x'_n$ are fresh. The fields $x_1,\ldots,x_n$ store the values of the arguments supplied to the original constructor for use with the postinit block, and the closure's constructor merely initializes those fields. The `call` method invokes the appropriate `postinit` method, passing the object under construction, the remaining closures, and the saved parameter values. The translation propagates the original constructor's exception clause to the `call` method, in case any of those exceptions originate in the postinit block.

Generating the `fieldinit` Methods

The `fieldinit` method has the same signature, including exception declaration, as the original MIJAVA constructor, except that it takes an additional parameter of type C at the beginning of the argument list. We assume here that this parameter has the name `self`, but it actually has a fresh name to avoid conflict with any static fields in the class and any local variables declared within the fieldinit section. The method's access control is based on that of the original constructor, according to table 6.1. Java and MIJAVA do not allow other modifiers like `synchronized` on constructors, so the generated methods cannot have additional modifiers.

To generate the `fieldinit` method's body, the translation copies the contents of the original field initialization section unchanged, except for statements that involve

constructor invocations and pre-object creations. These statements can take one of two forms:

- `return preobj` { `super`($arg_1, \ldots, arg_n$); $fd_1 \leftarrow e_1$; ...; $fd_m \leftarrow e_m$; }

  where arg_i and e_j are expressions, for $i \in [1,n]$ and $j \in [1,m]$, and $n, m \geq 0$. Let D be the superclass of C, the class being translated.

  For such a statement, where $D \neq$ `Object`, the compiler emits a block of the following form:

  ```
  {
      τ1 x1 = arg1;
      ...
      τn xn = argn;
      ClosureList closures = D.fieldinit(self, x1, ..., xn);
      self.fd1 = e1;
      ...
      self.fdm = em;
      return closures.cons(new CClosure(x1, ..., xn));
  }
  ```

  where $x_1, \ldots, x_n$ and *closures* are fresh identifiers, and *CClosure* is the name of the closure class generated above for the invoked superclass constructor. The types τ_i are the types of the formal parameters of the invoked constructor, obtained from its declaration. This block saves the values of the arguments to the superclass constructor to avoid repeated execution of those expressions, invokes the superclass constructor, initializes the local fields, and returns an updated construction history list.

  If, however, $D =$ `Object`, then $n = 0$, and there is no `fieldinit` method to call. In this case, the block has the following form:

  ```
  {
      self.fd1 = e1;
      ...
      self.fdm = em;
      return ClosureList.empty();
  }
  ```

- `return this(`arg_1`, ..., `arg_n`);` where arg_i is an expression.

  The generated block is the same as the previous case, except that the invoked `fieldinit` method is in the same class, and the translation omits the field assignment statements. The output is as follows:

```
{
    τ1 x1 = arg1;
    ...
    τn xn = argn;
    return C.fieldinit(self, x1, ..., xn)
             .cons(new CClosure(x1, ..., xn));
}
```

where the variables x_i are fresh and the types τ_i are the types of the formal parameters of the invoked constructor, as before, and *CClosure* is the closure synthesized for the invoked constructor.

The generated `fieldinit` method is static and therefore cannot contain any references to `this`. However, such references do not appear in the result of the translation, as long as the original program is well-typed, because MIJAVA's type system forbids occurrences of `this` within the body of a field initialization section.

Generating the `postinit` Method

For each MIJAVA constructor with the signature

$$C(\tau_1\ x_1, \ldots, \tau_n\ x_n)\ \texttt{throws}\ E_1, \ldots, E_m$$

the translation generates a `postinit` method with the signature

$$\texttt{void postinit}(C\ \texttt{self, ClosureList cls,}\ \tau_1\ x_1, \ldots, \tau_n\ x_n)\ \texttt{throws}\ E_1, \ldots, E_m.$$

The arguments `self` and `cls` have freshly generated names. The `postinit` method has the same access control as the `fieldinit` method, as specified in table 6.1.

The body of the `postinit` method consists of two sections: the first section uses the list of closures to invoke the next `postinit` section in the chain, and the second executes the statements from the original constructor's postinit blocks. To generate the first section, the translation first computes the set of all of the constructors, either in this class C or its superclass D, that the original constructor may invoke directly on any control path through its field initialization section. This set cannot be empty in a well-typed MIJAVA program.

If `Object`'s constructor is not in this set, then the translator maps each invoked constructor to the name of the inner class representing the associated closure. For a closure set $F_1, \ldots, F_p$, the translation generates the code in the top half of figure 6.7, where *cl* is a fresh identifier. This code throws an exception if the closure list is empty, or if the first closure is not of one of the expected types.

If the set of invoked constructors includes `Object`'s as well as the constructors corresponding to the closures $F_1, \ldots, F_p$, the translation generates the code in the bottom

```
Closure cl = (cls.isEmpty()) ? null : cls.head();
if (cl == null) {
    throw new ProtocolErrorException();
} else if (cl instanceof F1) {
    ((F1) cl).call(self, cls.tail());
}
...
else if (cl instanceof Fp) {
    ((Fp) cl).call(self, cls.tail());
} else {
    throw new ProtocolErrorException();
}
```

```
Closure cl = (cls.isEmpty()) ? null : cls.head();
if (cl == null) {
    // NOP
} else if (cl instanceof F1) {
    ((F1) cl).call(self, cls.tail());
}
...
else if (cl instanceof Fp) {
    ((Fp) cl).call(self, cls.tail());
} else {
    throw new ProtocolErrorException();
}
```

Figure 6.7: Generated closure dispatch code without and with `Object`

half of figure 6.7, which differs only in the highlighted line. An empty list of closures indicates that the `fieldinit` method invoked `Object`'s constructor. Since there is no corresponding `postinit` method, no action is necessary.

For the second part of the `postinit` method, the translator simply emits the code from the original postinit block, replacing all references to `this` (even implicit references) with `self`.

6.7 Limitations

While our experience suggests that the translation described here suffices to migrate most existing Java programs to MIJAVA, it does have two restrictions. First, our translation does not support first-class pre-objects. Therefore, `preobj` expressions may appear only as the direct argument of `return` statements inside constructors. This is a consequence of the lack of an explicit representation of pre-objects in the output of the translation. It should be possible to devise a translation strategy to support this language feature, involving an explicit representation of pre-objects, but we leave this for future work.

Second, Java's reflection system allows clients to observe the implementation details of our translation and to invoke the generated methods in a way that violate the object-creation protocol's invariants. This problem is shared by most compilers that generate Java source code, like AspectJ and related technologies.

6.8 Translating Mixed Programs

The translation described in the previous section assumes that the program contains only MIJAVA classes (except for `Object`). As part of providing a smooth migration path from Java to MIJAVA, our implementation must support programs containing a mixture of Java and MIJAVA classes, which requires some modifications to the implementation and restrictions on the MIJAVA class definitions.

Java Extending MIJAVA

Our translation, as described above, works for programs in which Java classes extend MIJAVA classes in the typical way, but MIJAVA's guarantees about uninitialized field access apply only to those fields in MIJAVA classes. It is possible to extend the translation slightly to allow the authors of the Java classes to preserve these invariants, but only at the cost of a more complicated initialization protocol for Java subclasses.

In the common case (especially prevalent when programmers are migrating existing programs from Java to MIJAVA), a Java subclass follows the standard Java initialization protocol, in which subclass constructors invoke a constructor in their superclass. As an

```
class ColorRect extends Rect {
    Color color;

    public ColorRect(Point lowLft, Point upRt, Color c) {
        super(lowLft, upRt);
        color = c;
    }

    public ColorRect(Point lowLft, Point upRt, Canvas cnvs, Color clr) {
        super(lowLft.x, lowLft.y, upRt.x, upRt.y, cnvs);
        color = clr;
    }

    protected void draw(Canvas c) { ... color ... }
}
```

Figure 6.8: A Java implementation of `ColorRect`

example, consider the `ColorRect` class definition in figure 6.8, which is a Java class that extends $\texttt{Rect}_M$ from figure 6.1. Because the translation preserves $\texttt{Rect}_M$'s constructor interface, this class compiles successfully against $\texttt{Rect}_J$. However, because $\texttt{Rect}_J$'s constructors execute $\texttt{Rect}_M$'s postinit blocks, creating an instance of `ColorRect` executes these postinit blocks before `ColorRect`'s constructors have the chance to initialize the `color` field. If the client invokes `ColorRect`'s second constructor, the program executes `ColorRect.draw` early and gets a null-pointer exception upon reference to the `color` field. As Java does not allow `ColorRect`'s constructors to initialize fields before invoking $\texttt{Rect}_J$'s constructors, there is no way to initialize `color` before the `draw` method executes.

It is possible to preserve MIJAVA's guarantees about uninitialized fields, but only by requiring additional effort on the part of the Java subclass author. Since the various methods used by the translation are protected and thus visible to subclasses, we could extend the protocol described above to include `ColorRect` as well. Most of `ColorRect` remains the same as in figure 6.8; only its second constructor changes:

```
public ColorRect(Point lowLft, Point upRt, Canvas c, Color clr) {
    super((StubConstr) null);
    ClosureList closures = Rect.fieldinit(this, lowLft, upRt, c);
    color = c;
    Rect.postinit(this, closures, lowLft, upRt, c);
}
```

In this version, the constructor directly invokes the static methods produced by the translation and initializes `color` directly at the appropriate time.

Requiring subclasses to follow this protocol manually, however, is not realistic. First, it exposes far too many details of the translation's implementation. Second, it requires modifying Java class definitions, which defeats the purpose of supporting class-by-class migration of large programs. Finally, the programmer would have to propagate this protocol all the way down the class hierarchy, covering (in the worst case) the entire program. Accepting the weakened form of MIJAVA's guarantees, especially at an interim point in a migration, is more likely to be practical—especially because, even with the guarantees weakened, we can still ensure that fields in MIJAVA classes are initialized before their use, so the partially-migrated program is *still* more robust than the Java original.

MIJAVA Extending Java

Our translation also supports MIJAVA classes that extend Java classes, but with severe limitations. We cannot in general guarantee that the MIJAVA class will initialize all its fields before the rest of the program references them. Additionally, there are significant restrictions on how the MIJAVA class may invoke its superclass's constructor, as described below.

As an example of the problems the translation faces in this situation, consider the code in figure 6.9. Here, the `Base` class is written in Java, and its constructor invokes the method `m`. Our translation cannot produce correct code for `Derived`. Not only can we not preserve `Derived`'s constructor semantics, but the result also cannot provide MIJAVA's guarantees with respect to uninitialized field access. Again, we use $\texttt{Derived}_M$ to refer to the original form of the class and $\texttt{Derived}_J$ to refer to its translation.

First, $\texttt{Derived}_M$'s constructor presents severe problems to the translation. Because `Base` defines only a two-argument constructor, Java requires all of $\texttt{Derived}_J$'s constructors to begin with an invocation of `Base(int, int)`. However, $\texttt{Derived}_M$'s constructor calls the `computeArgs` method to compute the arguments to `Base`'s constructor, which is impossible in Java. Additionally, $\texttt{Derived}_M$'s constructor chooses between two different constructor invocations based on `computeArgs`'s result, and Java simply does not allow programs to choose between constructor invocations at run-time.

In some situations, it might be possible to factor the computation of the superclass constructor arguments out into newly-synthesized static methods, which $\texttt{Derived}_J$'s constructor could invoke directly as arguments to `Base`'s constructor. This is not, however, possible in general. To do so in this example (ignoring the multiple constructor invocations), the translation would have to split the body of the `computeArgs` method into two separate methods, one to compute the first argument and one the second. If these computations are at all interrelated, or if they have interleaving effects, this separation is not possible without changing the behavior of the program.

```
class Base {
    public Base(int i, int j) {
        System.out.println(i + " " + j + " " + this.m());
    }

    public void m() { return 0; }
}

class Derived extends Base {
    private String str;

    public Derived(String s) {
        Pair<Integer, Integer> args = computeArgs();
        if (args.fst < args.snd)
            return preobj { Base(args.fst, args.snd); str <- s; };
        else
            return preobj { Base(args.snd, args.fst); str <- s; };
    }

    private static Pair<Integer, Integer> computeArgs() {
        ... body elided ...
    }

    public void m() {
        return str.length();
    }
}
```

Figure 6.9: Translating when MIJAVA extends Java

Second, because `Base`'s constructor has no restrictions on computation with `this`, it is free to invoke a method on itself that is overridden in the base class, just as in the example in Chapter 2. Therefore, even if we could fix the problem in the previous paragraphs, instantiating `Derived` would cause an null-pointer exception in the body of `Derived.m`, because `str` is still `null` at this point during program execution.

As an important special case, however, MIJAVA classes that directly extend `Object` avoid both of these problems. `Object` defines only a nullary constructor, which has an empty (effect-free) body, so the translation can easily move the superclass constructor invocation to the front of the translated constructor without changing the program's behavior. Additionally, since `Object`'s constructor's body invokes no methods in the subclass, MIJAVA can still guarantee that fields in the subclass are initialized before the program references them.

6.9 Separate Translation

The translation supports separate compilation in much the same way that Java's compiler currently does. Beyond the information about other classes already required to compile a Java class, our translation needs only the following additional pieces of information in order to translate and compile a class *C*:

- the constructor signatures of *C*'s superclass, including the `throws` clause;
- whether *C*'s superclass was originally defined in MIJAVA or in Java; and
- if *C*'s superclass was originally written in MIJAVA, the names of the static methods corresponding to the field initialization and postinit sections of the superclass's constructors.[5]

The first piece of information is already present in both `.java` and `.class` files. The latter two are currently present only in the source code or implicit in the translation, but we could add them to the `.class` file in a backward-compatible way, either by adding new items to the end of the `class` structure, or by defining new attributes for inclusion in the `class` structure's `attributes` array [28, §4.7].

6.10 Implementation Shortcuts

For the purposes of the evaluation described in Chapter 7, we have implemented a prototype of the implementation discussed here. The prototype compiles MIJAVA programs into Java source code, which we then compile using a stock Java compiler. As it is a research prototype, we were able to simplify the implementation far beyond what would be possible for an industrial-strength product.

Much of the complexity of a full Java/MIJAVA compiler involves finding, reading, and processing the various classes in the program. While both languages support compiling one class at a time (barring mutual recursion), type-checking and compiling a class does require loading the signatures of all of the classes that it uses. Supporting this requires the capability to find and load these class definitions from disk, both from source files and from class files (including the standard library), and to resolve the various names within these files while correctly handling mutually-recursive compilation units.

Our prototype implementation, however, only needs to read the class being translated, as a result of the following changes and simplifications:

[5] While, in the description of the translation above, we have assumed that these methods would have the names `fieldinit` and `postinit`, a real implementation would of course have to synthesize fresh names to avoid collisions with methods written by the programmer.

1. We do not allow implicit `this` within postinit blocks, although explicit references to `this` are still legal. So, for example, all field references must have the form `this.fd`, rather than just `fd`.

2. The input syntax requires the programmer to provide two additional annotations in MIJAVA classes: an indication of whether the superclass is written in MIJAVA or in Java, and a list of the superclass's constructors with their signatures and uncaught exception types.

3. The MIJAVA type checker enforces only the following simple rules:

   a) Every return statement in a constructor must have one of the two forms `return preobj { ... }` or `return this(...)`.

   b) Each `preobj` expression must contain a call to a constructor in the superclass and initializers for exactly those fields defined in the containing class.

   c) Each `preobj` expression may appear only as the immediate argument to a `return` statement in the field initialization section of a constructor.

   We leave the rest of the type-checking to the Java compiler, including the following potential errors in MIJAVA constructors:

   · A control path in a field-initialization section that does not end with either a `return` statement or a `throw` statement—that is, it "falls off the end" of the constructor.

     Because the translation preserves control flow outside `preobj` expressions, this results in a `fieldinit` method with a control path that "falls off the end" of the method. Since the method has a non-`void` return type, the Java compiler signals an error at compile-time.

   · A mismatch between the type of a field and its initial value in a `preobj` expression.

     The translation maps the field initialization clauses in `preobj` expressions to field assignments, which require the same relationship between the type of the field and the type of the right-hand side as do field initializations. Therefore, the Java compiler detects and signals any mismatch.

   · A field initialization section that refers to `this`.

     Every expression in a field initialization section appears at some point within the generated `fieldinit` method. Since this method is static, references to `this` are forbidden, and the compiler signals an error.

Delegating the bulk of type-checking to the Java compiler means that we do not have to perform Java name resolution. Name resolution is the process of taking a

dotted sequence of identifiers, like `w.x.y.z`, and determining its precise meaning: is `z` a type within a package `w.x.y`, a member type of class `y` within package `w.x`, a static field in type `w.x.y`, or any one of a number of other possibilities?[6] As a result of this delegation, the error messages issued by the compiler often require detailed information about the translation's output in order to understand and fix. While this is acceptable for a research prototype, a real implementation would of course need to provide error messages in terms of the MIJAVA original.

The restriction on implicit `this` in postinit blocks allows the translation to replace `this` with a reference to a method parameter without detecting implicit occurrences of `this`. Finding all such implicit occurrences would require full name resolution.

[6]While Java convention requires class names to begin with a capital letter, the language does not enforce this, so a name resolver cannot use this as a guide. Additionally, the name's context provides some clues, but it does not always uniquely specify the name's meaning.

Chapter 7

Experience Report

As part of our investigation of the practical impact of MIJAVA's proposed features and of the migration path from Java to MIJAVA, we implemented the translation described in the previous chapter, migrated sections of several existing large Java programs to MIJAVA, compiled the resulting programs, and ran the included test suites to ensure that our changes did not affect program functionality. Our experiences demonstrate that programmers can feasibly use MIJAVA's constructors and `Maybe` type in real programs, and that migrating classes from existing Java programs is possible with only minimal changes to the rest of the program. Several particular cases, however, also demonstrated the need for further developments to MIJAVA, to improve the flexibility of the language and provide programmers with additional abstractions to reduce repeated code introduced during migration.

7.1 Construction

In the first stage of our experiments, we evaluated the practical utility of MIJAVA's construction mechanism in large programs. To investigate this, we migrated selected classes from the Jacksum project,[1] a checksum library that provides Java implementations of various checksum algorithms such as SHA-1 and MD5; from Lucene,[2] a library that provides index-based search capability for collections of text; and from Ant,[3] a build tool for Java programs.

We were particularly interested to explore the consequences of the change in the order in which programs establish a new object's representation invariants, as discussed in section 2.2. Our experience supports our expectation that this difference is not a significant problem in practice, as these multiple invariants never caused difficulties

[1] Version 1.7.0, `http://www.jonelo.de/java/jacksum/`
[2] Version 2.0.0, `http://lucene.apache.org/`
[3] Version 1.7.0, `http://ant.apache.org/`

during the migration. In particular, in every migration attempt, once we fixed the type errors reported by the compiler, the resulting program ran with the same behavior and results as the original code without requiring any additional changes. One facet of MIJAVA's design that contributed to the ease of migration is the order in which it executes postinit blocks. Because we chose to execute these blocks in essentially the same order that Java does, programmers do not have to consider the consequences of re-ordering the effects in these blocks.

Although most of the migrations were straightforward, we did encounter several programs whose object initialization protocols are more complex than we expected, suggesting several possibilities for further improvements to MIJAVA. Even in the worst case, though, preserving existing program behavior required no changes larger than the duplication of initialization code across multiple constructors and methods.

Finally, although we did not include the use of `Maybe` types in this experiment, we did attempt to minimize the number of fields that would need `Maybe` types (beyond those fields that clearly represent optional properties). Specifically, we tried to avoid writing constructors that return pre-objects that initialize some fields to `null` and expect either the postinit block or the client to set those fields to the appropriate values after construction through some other mechanism.

Jacksum

For the Jacksum project, we migrated all of the class files in two packages, `jonelo.jacksum.algorithm` and `jonelo.jacksum.adapt.gnu.crypto.hash`, which implement various checksums and hash functions. We were able to complete this migration with no difficulties, although we did have to duplicate some trivial initialization code in a few classes.

Many classes in `jonelo.jacksum.algorithm` contain constructors that invoke one or more instance methods as part of the object initialization process, and some of these instance methods are also available as part of the classes' public interface. More specifically, some of these classes have multiple constructors that abstract repeated code out into a private `init` method. Several classes are also stateful and therefore provide a public `reset` method, which restores an object to its initial state. Constructors often call this `reset` method, again to avoid repeating initialization code. MIJAVA can express all such constructors, but the migration does require either duplicating some initialization code or delaying initialization and changing several fields to have `Maybe` type.

Because the `init` and `reset` methods assign to fields in `this`, we cannot simply call them from the field initialization section of the constructors. (Turning either method into a static method does not avoid this problem, as we would need to pass `this` as an argument in order to access the object's fields.) Calling these methods from the postinit

block is not desirable either, as it delays the initialization of some fields until after the pre-object has been reified. As a result, we would have to give those fields `Maybe` types, which imposes undue overhead on all references to those fields throughout the class.

In the classes in this package, the initializations performed by the `init` and `reset` methods are simple; the right-hand sides of the assignments in those methods are either literals or simple expressions, so the cost of repeating the code is not excessive. Therefore, we simply inlined the computation from these methods into the constructors, with one exception. `CrcGeneric.init`, unlike the other initialization methods in this package, calls a method that performs significant computation to initialize the elements of an array. We could not realistically inline this second method, but it was straightforward to turn this method into a static method that allocates, initializes, and returns the array, and to call this method from within the `preobj` expressions.

One possibility that avoids the need to duplicate code is to turn the `init` methods into static methods that create and return pre-objects, but this is not always practical either. To begin with, our translation from the previous chapter does not support the use of pre-objects as first-class values. Even if it did, however, the original version of `CrcGeneric.init`, for instance, only computes initial values for four out of the class's ten fields, so this transformed `init` method would need six arguments for those fields whose values differ depending on the calling constructor. The length of this argument list suggests that this strategy is not a suitable general solution. Additionally, it may not be possible in all cases, if `init` does not restrict itself to initializing local fields only but, like `CombinedChecksum.init`, also re-initializes fields inherited from the superclass. MIJAVA's `preobj` form provides no support for this sort of operation.

Another possibility would define private static methods that compute the initial values for the various fields, and the constructors and `reset` methods would call these methods to obtain field values. The `init` methods would be unnecessary in this implementation, as their contents would have been distributed across these field computation methods. While this strategy results in very elegant constructor and `reset` definitions, it is a heavyweight solution, as it requires either many small methods (one per field), or the definition of a new "structure" type to contain all of the field values, and a single method to create and return an instance of this structure. The syntactic overhead of either alternative would discourage programmers from using this implementation technique.

This experience suggests that MIJAVA would benefit from a modified construction mechanism that allowed classes to distribute the field initialization section and pre-object creation across multiple methods, each of which would be able to specify initial values for a subset of the class's fields. Placing restrictions on these methods to prevent the use of `this` during object initialization is not difficult, but proving at compile time that the resulting pre-objects contain values for all of the necessary fields at reification time is a much harder problem than that currently solved by MIJAVA's type system.

We succeeded in migrating the other package we investigated, `jonelo.jacksum.adapt.gnu.crypto.hash`, without difficulty, although this package also demonstrates the problems discussed above. In addition to factoring repeated code out of constructors, many of the concrete classes, such as `Has160`, also define a `clone` method, which uses a private "copy constructor." Each such constructor starts by invoking the class's nullary constructor, which initializes most of the fields in the new object to default values. Once the default constructor returns, the copy constructor uses normal Java field assignment to replace many of those defaults with the values from the object being cloned. The call to the default constructor here avoids duplicating code for those fields whose values are *not* copied during cloning.

Although this code structure again suggests that the ability to spread pre-object creation across multiple methods is desirable, we were able to translate the classes without significant trouble. Because all of the copy constructors are private and thus cannot be used during initialization of subclasses, we were able to put the constructors' assignment statements into the postinit blocks, thus preserving the behavior of the original program.

Lucene

The vast majority of classes that we considered for migration in this project were trivial to translate to MIJAVA. For instance, the `org.apache.lucene.analysis` package contains primarily classes with constructors that initialize all fields in their respective objects and perform no computation on `this`. Such constructors map directly to `preobj` expressions in MIJAVA. The `org.apache.lucene.document` package, on the other hand, defines mostly classes that are uninstantiable, either because they implement the singleton pattern, or because they define only static members and thus do not require constructors.

The class `org.apache.lucene.index.CompoundFileReader` provides two interesting use cases that demonstrate the kind of changes that programmers might have to make in order to use MIJAVA's constructors. First, it defines an inner class, `FileEntry`, which contains two fields and no constructors or methods. It is therefore the client's responsibility to initialize the object's fields through direct assignments. While Java does not require the client to assign to both fields, the MIJAVA equivalent would. Because `FileEntry` is an inner class and thus unsupported by our translation, we cannot migrate this class directly. However, we can simulate the effect of the migration on the class's clients by giving it a two-argument constructor that initializes both fields. Since this prevents the compiler from synthesizing the nullary constructor, clients must now provide both field values immediately upon object construction.

```
class CompoundFileReader extends Directory {

    private static final class FileEntry {
        long offset;
        long length;
    }

    private HashMap entries = new HashMap();

    public CompoundFileReader(Directory dir, String name) throws IOException {

        IndexInput stream = dir.openInput(name);
        int count = stream.readVInt();
        FileEntry entry = null;
        for (int i=0; i<count; i++) {
            long offset = stream.readLong();
            String id = stream.readString();

            if (entry != null) {
                // set length of the previous entry
                entry.length = offset - entry.offset;
            }

            entry = new FileEntry();
            entry.offset = offset;
            entries.put(id, entry);
        }

        // set the length of the final entry
        if (entry != null) {
            entry.length = stream.length() - entry.offset;
        }
    }
}
```

Figure 7.1: Creating `FileEntry` instances, original version

Figure 7.1 contains an excerpt of the `CompoundFileReader` class, in its original form, which demonstrates the definition and instantiation of the `FileEntry` inner class. Among other actions, the outer class's constructor opens a specified file and reads a series of records from the file. For each record, the constructor updates the `length` field in the *previous* record's `FileEntry` instance (in the first highlighted statement), creates a new `FileEntry` instance (the second highlighted statement), initializes its `offset` field, and inserts the new entry into a `HashMap`. The initialization of a `FileEntry` object thus spans two iterations of the loop, so we cannot provide values for both fields at construction time without modifying the constructor heavily.

```
class CompoundFileReader extends Directory {

    private static final class FileEntry {
        long offset;
        long length;
        public FileEntry(long offset_in, long length_in) {
            offset = offset_in; length = length_in;
        }

    private HashMap entries;

    public CompoundFileReader(Directory dir, String name) throws IOException {
        IndexInput stream = dir.openInput(name);
        HashMap entries = new HashMap();
        int count = stream.readVInt();

        if (count > 0) {
            long offsets[] = new long[count];
            String ids[] = new String[count];

            for (int i = 0; i < count; i++) {
                offsets[i] = stream.readLong();
                ids[i] = stream.readString();
            }
            for (int i = 0; i < count - 1; i++) {
                entries.put(ids[i],
                            new FileEntry(offsets[i],
                                          offsets[i + 1] - offsets[i]));
            }
            entries.put(ids[count - 1],
                        new FileEntry(offsets[count - 1],
                                      stream.length - offsets[count - 1]));
        }

        return preobj { Directory(); entries <- entries; };
    }
}
```

Figure 7.2: Creating `FileEntry` instances, migrated version

Figure 7.2 contains the rewritten code. Rather than creating the `FileEntry` objects as we read the data from the directory, we preload the information into two temporary arrays, `offsets` and `ids`, then create the `FileEntry` values only after all the necessary information is available. This change increases memory usage during the constructor, but the program cannot otherwise observe the difference. In particular, even though the migration requires significant modifications to the `CompoundFileReader` constructor, the changes are limited to that method and do not “infect” the rest of the program.

Of course, we could have avoided the need to rewrite `CompoundFileReader`'s constructor so drastically by giving `FileEntry` a nullary constructor, shown below in MIJAVA syntax:

```
private static final class FileEntry {
    long offset;
    long length;
    public FileEntry() {
        return preobj { super(); offset <- 0; length <- 0; }
    }
}
```

With this interface, however, the language could not guarantee that the field values of each `FileEntry` in the `HashMap` are consistent with the data read from the stream. (Indeed, in the original logic, this guarantee would be false, since the program inserts each object into the map before initializing its `length` field.) And, if the fields were of object type instead of `long`, this nullary constructor would require us to give them `Maybe` type in the full MIJAVA translation, which would affect all uses of these objects.

Ant

Due to the size of the Ant code base, we migrated only a portion of the program, specifically those classes in the `org.apache.tools.ant` package. Roughly a third of these classes required little to no effort to translate, either because they already provided well-behaved constructors (that initialize all fields and do not otherwise use `this`), or because they define only static fields with initializers. Many of the rest delay initialization of some of their fields,[4] and MIJAVA constructors are trivial once the relevant fields have `Maybe` type as discussed in Chapter 4.

As an example of the translation effort involved in this package, consider the class `org.apache.tools.ant.AntTypeDefinition`. This class contains only the (compiler-synthesized) default constructor, which initializes all fields to `null`. To completely initialize an object of this class, clients must use various setter methods to initialize the object's fields after constructing it. For the most part, these setters are simple methods that perform a single assignment, but some have additional effects. For instance, the `setClass` method initializes the `clazz` field to its argument, but it also computes values for the `classLoader` and `className` fields from its argument, if those fields have not yet been initialized.

Because the use of setter methods represents an instance of (potentially) delayed initialization, we must give each of the six fields `Maybe` type. With those field types,

[4] Several such examples delay field initialization because Ant uses `Class.newInstance` to create new instances of these classes, and this method (effectively) invokes the nullary constructor.

the straightforward migration is a single constructor with six arguments, one for each field of the class. Clients simply pass `null`/`none` for those fields they do not need to initialize—i.e., those fields for which the original code does not call a setter method.[5] This constructor must be careful, however, to leave the new object with the correct field values for the setter invocations in the order in which they appeared in the original code. Because `setClass` only overwrites the `classLoader` and `className` fields if they have not previously been set, it suffices to take `setClass`'s values for these fields only when the client supplies `null` for the corresponding parameters.

7.2 *Maybe*

To investigate the practical implications of MIJAVA's `Maybe` type in production code, we modified existing Java programs to use `Maybe` instead of `null` wherever possible. Since the resulting programs are still Java, we could not rely on MIJAVA's native `Maybe` type or its interpretation of types like `String` as containing only instances of `String`. To make up for these missing features, we defined `Maybe` as a collection of Java classes, and we added assertions and wrapper classes around the Java API. With these minor changes, we could be reasonably certain that the value `null` never arises in our programs (outside the wrappers).

Figure 7.3 contains our implementation of `Maybe`. We ensure that `Some` and `None` are `Maybe`'s only subclasses by making `Maybe`'s only constructor private. The `some` and `none` methods act as MIJAVA's `some` and `none` keywords. For the purposes of this experiment, we include the `inject` method, which we use as a convenience in lifting the return values of method calls to `Maybe` type; a real implementation would not include this and would instead expect clients to use `some` and `none`. The rest of the class definition is the obvious implementation of the interface from figure 3.1.

Chat Server

For our first experiment, we modified a small TCP-based chat server that could serve as a small project in an undergraduate class on networking and concurrency. Most of the changes that we made would be unnecessary for a real MIJAVA implementation; these include adding explicit `Maybe` injections and projections, wrapping Java library classes like `java.util.HashMap` to use `Maybe` types in their interfaces, and inserting assertions throughout the code to ensure that, e.g., constructor parameters are never `null` (although they may be `none`). Other changes that reflect significant modifications needed to switch to explicit `Maybe` types include the following:

[5] We conjecture that the heavyweight nature of this constructor signature and other Java alternatives to optional and keyword arguments is the reason for the original author's imperative initialization protocol.

```
public abstract class Maybe<T> {
    private Maybe() { }

    public static <U> Maybe<U> some(U obj) {
        assert obj != null;
        return new Some<U>(obj);
    }

    public static <U> Maybe<U> none() {
        return new None<U>();
    }

    // convenience method for wrapping library routines
    public static <U> Maybe<U> inject(U obj) {
        if (obj == null) {
            return new None<U>();
        else
            return new Some<U>(obj);
        }
    }

    public abstract boolean isSome();
    public abstract T valOf();

    public static class MaybeException extends RuntimeException { }

    private static class Some<T> extends Maybe<T> {
        private T val;

        public Some(T v) { val = v; }

        public boolean isSome() { return true; }
        public T valOf() { return val; }
    }

    private static class None<T> extends Maybe<T> {
        public None() { }

        public boolean isSome() { return false; }
        public T valOf() { throw new MaybeException(); }
    }
}
```

Figure 7.3: Implementation of Maybe for experimentation

- Because `Maybe` is invariant in its type argument, we needed to add explicit upcasts in certain contexts. For example, given two classes `C` and `D`, where `D` is a subclass of `C`, we cannot write the following:

  ```
  Maybe<C> m() {
      return Maybe.some(new D(...));
  }
  ```

  This fails to compile because `return`'s argument has type `Maybe<D>`, which is not a subclass of `Maybe<C>`. In order to make this code well-typed, it is necessary to insert an explicit upcast:

  ```
  Maybe<C> m() {
      return Maybe.some((C) (new D(...)));
  }
  ```

  Alternatively, assuming an MIJAVA implementation that supports *Maybe* casts as specified in MIJMODEL, one could write the following instead:

  ```
  Maybe<C> m() {
      return (Maybe<C>) (Maybe.some(new D(...)));
  }
  ```

  This alternative is not available during this experiment, however, because Java does not allow casts between the unrelated types `Maybe<C>` and `Maybe<D>`.

- It was necessary to add explicit `isSome` checks in certain places in the program, rather than relying on the `valOf` method to throw an exception. Each `x.isSome()` call corresponds exactly to a `x != null` check in the original program. Further, these `null` checks are required by program logic and not by robustness.

  By "robustness checks," we mean those checks equivalent in principle to

  ```
  assert x != null;
  ```

  in which the programmer is explicitly checking against null for the sole purpose of detecting and reporting the error earlier than would otherwise happen. In the translated version of the program, `x` would not have `Maybe` type, and therefore MIJAVA's type system would statically ensure that `x` is never `none`, making an explicit `isSome` check unnecessary (as well as ill-typed).

  Other `null` checks, however, are required by program logic. For instance, consider an object with a field that represents a truly optional property: at certain points in the program's execution, the property is present, and at other points, it is not. In the original program, then, code that refers to this property must check

to see whether the field is `null` and take appropriate action. (The appropriate action in cases where the field is `null` may sometimes be signaling an error, but not always. Indeed, in one particular situation in the chat server, the program signals an error if the field is *not* `null`.) Since the field represents an optional property, it must have a `Maybe` type in the translated version, and the program must explicitly call the `isSome` method in cases where it is not correct to throw an exception if the field is `none`. (As an alternative, we could have omitted the call to `isSome` and caught the `MaybeException`, but this strategy is very heavyweight, due to the syntactic overhead and the more complex control flow, and we do not expect that most Java programmers would accept this as the only—or even the primary—strategy.)

It was *not* necessary to change local variables to `Maybe` type if the program did not initialize them at point of declaration, as in the following excerpt:

```
public static void main(String[] args) {
    ServerSocket svr;

    try {
        svr = new ServerSocket(12345);
    } catch (IOException e) {
        System.err.println("Couldn't listen on port 12345");
        return;
    }

    // further references to svr follow
}
```

Here, we cannot move `svr`'s initializer to its declaration, because the initializer (and only the initializer) must run in the context of a particular exception handler. It is not necessary to give `svr` the type `Maybe<ServerSocket>` and initialize it to `none` in the translation, because Java's definite assignment analysis [20, Ch. 16] proves that each control path through this function initializes `svr` before referring to it. This proof is not, however, always possible, as our experience with the next program demonstrates. The analysis only succeeds in this case because of the `return` statement in the exception handler.

Jacksum

For our second experiment, we used the code base of the Jacksum project. Specifically, we modified various classes to use `Maybe` types throughout instead of Java's `null` value; of the various classes, `jonelo.sugar.util.Base64` is a representative example.

(As our changes resulted in modifications to `Base64`'s interface, we had to make minor changes to Java classes that interact with `Base64`, primarily the addition of explicit coercions to and from `Maybe` types. We do not discuss those further here.)

For the most part, our change consisted of the explicit upcasts and `isSome` checks described above, which lends further support to our claim that migrating Java classes to MIJAVA is generally straightforward. Beyond this, however, our experience with Jacksum suggested one way in which `Maybe` types can contribute to software specification and exposed another opportunity for future work.

First, in attempting to eradicate `null` from the `Base64` class, we discovered several incompletely-specified methods in the Java API. For instance, the documentation for the following methods does not specify their behavior for *null* arguments:

- `StringBuffer.replace(int, int, String)`
- `java.io.ByteInputArrayStream(byte[])`—i.e., the constructor

Similarly, the specifications for the following methods do not indicate whether they can return `null`:

- `java.io.ObjectInputStream.readObject()`
- `java.io.ByteArrayOutputStream.toByteArray()`

Making `Maybe` types explicit would force library and documentation authors to specify where `null`/`none` is allowed. It would also draw attention to those places where `none` is allowed, to remind documentation authors to specify how a method behaves when passed `none`, or the circumstances under which it can return `none`.[6]

Second, local variables with delayed initializers proved much more challenging in Jacksum than in the chat server. Figure 7.4 contains the original code of a method from the `Base64` class. The highlighted declarations, for `bais` and `ois`, pose a significant challenge. In translating this function to MIJAVA, we would ideally like to avoid giving these two variables `Maybe` types, so we must drop the initializer expressions. If we do this, however, the Java compiler signals an error, because the definite-assignment analysis cannot prove that `ois` is initialized before its use in the `finally` block. Indeed, `ois` *can* be uninitialized here if the computation of `bais`'s value, the call to `ByteArrayInputStream`'s constructor, terminates with an uncaught exception.

[6] As a related note, we conjecture that fields with explicit `Maybe` types should focus programmers' attention on the costs of delayed initialization, specifically the weakened invariants governing those fields' values and the attendant run-time checks imposed by the coercions. This, in turn, should encourage class authors to provide constructors that initialize all fields to non-`none` values wherever possible, to avoid `Maybe` fields.

```
public static Object decodeToObject(String encodedObject) {
    // Decode and gunzip if necessary
    byte[] objBytes = decode(encodedObject);

    ByteArrayInputStream  bais = null;
    ObjectInputStream     ois  = null;
    Object obj = null;

    try {
        bais = new ByteArrayInputStream(objBytes);
        ois  = new ObjectInputStream(bais);

        obj = ois.readObject();
    } catch(IOException e) {
        e.printStackTrace();
        obj = null;
    } catch(java.lang.ClassNotFoundException e) {
        e.printStackTrace();
        obj = null;
    } finally {
        try { bais.close(); } catch (Exception e) { }
        try { ois.close();  } catch (Exception e) { }
    }

    return obj;
}
```

Figure 7.4: `Base64.decodeToObject`, original version

```
public static Maybe<Object> decodeToObject(String encodedObject)
{
    // Decode and gunzip if necessary
    Maybe<byte[]> objBytes = decode(encodedObject);

    try {
        ByteArrayStream bais =
            new ByteArrayInputStream(objBytes.valOf());

        try {
            ObjectInputStream ois = new ObjectInputStream(bais);
            try {
                return Maybe.inject(ois.readObject());
            } finally {
                try {
                    ois.close();
                } catch (Exception e) { }
            }
        } finally {
            try {
                bais.close();
            } catch (Exception e) { }
        }
    } catch (IOException e) {
        e.printStackTrace();
        return Maybe.inject(null);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
        return Maybe.inject(null);
    }
}
```

Figure 7.5: `Base64.decodeToObject`, first migration attempt

We describe three attempts at a solution here, but none is completely satisfactory. Our first solution appears in figure 7.5, which does compile, but it imposes severe costs on the programmer. In this attempt, we adjust the scopes of `bais` and `ois` so we can initialize these variables at their declarations and in the contexts of the correct exception handlers. The deeply nested `try`/`catch`/`finally` structure is necessary because the scope of a variable declared in a `try` block does not include the associated `finally` block. Because the desired scoping complicates the control flow (and the syntactic nesting) of this method so severely, we cannot reasonably ask programmers to use this idiom.

```
public static Maybe<Object> decodeToObject(String encodedObject)
{
    // Decode and gunzip if necessary
    Maybe<byte[]> objBytes = decode(encodedObject);
    java.util.Stack<InputStream> cleanup =
        new java.util.Stack<InputStream>();

    try {
        ByteArrayStream bais = new ByteArrayInputStream(objBytes.valOf());
        cleanup.push(bais);

        ObjectInputStream ois = new ObjectInputStream(bais);
        cleanup.push(ois);

        return Maybe.inject(ois.readObject());
    } catch (IOException e) {
        e.printStackTrace();
        return Maybe.inject(null);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
        return Maybe.inject(null);
    } finally {
        while (!cleanup.empty()) {
            try {
                cleanup.pop().close();
            } catch (Exception e) { /* NOP */ }
        }
    }
}   // end decodeObject
```

Figure 7.6: `Base64.decodeToObject`, second migration attempt

In our second attempt, in figure 7.6, we simplify the code's structure and the control flow, and instead maintain a work list, as a `Stack`, that contains those input streams to be closed upon exit from the function. This solution has two drawbacks, however. First, it requires the overhead of an additional data structure, which affects both run-time memory requirements and program maintainability, as this new data structure comes with additional invariants. Second, this technique does not scale well to the general case. It works well here because all values that require cleanup are instances of the same type, and the cleanup operation requires invocation of the same method on all applicable values. In the general case, however, these properties do not hold, so representing the work list and performing the cleanup is much more complex. Indeed, in the worst case, the work list must be a list of closures, which require significant syntactic overhead in Java.

```
public static Maybe<Object> decodeToObject(String encodedObject)
{
    // Decode and gunzip if necessary
    Maybe<byte[]> objBytes = decode(encodedObject);

    Maybe<ByteArrayInputStream> bais = Maybe.none(null);
    Maybe<ObjectInputStream> ois = Maybe.none(null);

    try {
        bais = Maybe.some(new ByteArrayInputStream(objBytes.valOf()));
        ois = Maybe.some(new ObjectInputStream(bais.valOf()));

        return decodeWork(ois.valOf());
    } catch (IOException e) {
        e.printStackTrace();
        return Maybe.none(null);
    } finally {
        // any MaybeExceptions are discarded along with IOExceptions
        try { ois.valOf().close(); } catch (Exception e) { }
        try { bais.valOf().close(); } catch (Exception e) { }
    }
}

private static Maybe<Object> decodeWork(ObjectInputStream ois)
      throws IOException {
    try {
        return Maybe.inject(ois.readObject());
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
        return Maybe.none(null);
    }
}
```

Figure 7.7: `Base64.decodeToObject`, third migration attempt

In our third attempt, in figure 7.7, we split `decodeToObject` into two methods. The first, still called `decodeToObject`, performs the setup and tear-down logic, and the second, `decodeWork`, performs the computation of interest. Clients of the original method continue to call `decodeToObject`. This strategy has the advantage that the `Maybe` types on `bais` and `ois` are limited to `decodeToObject`; the `decodeWork` method can safely assume that these objects are always present. The primary drawback to this technique, of course, is the significant syntactic overhead of the two method definitions, which is particularly bad in this case because the additional code dwarfs the code that performs the actual computation.

As difficult as this change was, only one method out of all the classes that we migrated presented these problems. Based on our experience with this codebase, then, it appears likely that problems such as this arise only infrequently during migration.

7.3 Conclusion

Our experience with MIJAVA's constructors and `Maybe` types suggests strongly that programmers can migrate existing Java programs to MIJAVA with only minor difficulties. In many common cases, moving to MIJAVA's constructors requires only changes to the constructors being migrated. Programmers must make non-local changes (to other methods in the same class, or to other classes in the program) only in those situations in which constructors dispatch to helper methods in the class to avoid repeated code, or in which the class relies on an initialization protocol that, like Ant's `AntTypeDefinition`, involves more than invoking the constructor. In cases where the cost of migrating to an MIJAVA constructor is extremely high because of the need to provide values for every field, the programmer has the flexibility to give one or more fields `Maybe` types and define a constructor that provides weaker guarantees about field initialization. The primary outstanding drawback to MIJAVA's constructors, as we define them here, is the duplication of code (beyond that required by Java) in certain circumstances, which suggests future work to find abstractions that avoid this while still allowing MIJAVA to prove its guarantees statically.

While the experiments with `Maybe` generally show that introducing these types to existing programs is straightforward (assuming that the program's binary-only libraries, including the Java API, also use `Maybe`), they also suggest future work involving local variables whose scope interacts poorly with the method's control flow, as in the Jacksum example. While, as with constructors, the programmer can replace these uses of `null` with `Maybe` without requiring significant modifications to the rest of the method's body, this does limit how strong the type system's guarantees can be, and we would like to find abstractions that avoid the use of `Maybe` in these situations.

Even without these better abstractions, though, MIJAVA still represents an improvement in these cases, because in such situations the original Java program must initialize the local variable to `null`, even if the variable in question does not truly represent an "optional" property. As a result, it is still possible for the program to refer to this variable in an uninitialized state, and Java can provide no protection against this. MIJAVA still allows the access, but the `Maybe` type serves not only to detect these errors sooner, but also to signal the programmer that greater care is required when dealing with such variables.

Chapter 8

Related Works

The number of object-oriented languages has proliferated greatly in recent years, producing language implementations, models, and informal proposals. Some of these languages attempt to address the problems with *null*. For those languages like Java that do not directly attempt to reduce *null*'s dangers, researchers have also proposed a number of static analyses to attempt to detect and reason about uses of *null*. None of these efforts provide all of MIJAVA's benefits, as most omit either a migration path from an existing language, or a fully general *Maybe* type, or both.

8.1 Object Creation

Object-oriented languages and models demonstrate a wide range of object initialization mechanisms, including several that attempt to prevent access to uninitialized fields through some combination of dynamic and static checks. Most, however, either do not provide (all of) MIJAVA's guarantees or severely weaken the language's expressiveness in order to do so.

General-Purpose Languages

Spec#

In a parallel effort, Spec# [4, 13] has developed constructors that appear to provide all of the guarantees and flexibilities of MIJAVA's constructors. In Spec#, constructors look much like Java's, as they are a normal sequence of statements that initialize fields through standard field assignments. Constructors must also include a call to one of the superclass's constructors, written `base(...)`.[1] The specific restrictions on this `base`

[1] As in Java and C#, Spec# constructors may dispatch to another constructor in the same class by writing `this(...)` instead of calling the superclass's constructor. Neither the Spec# documentation nor the research literature describes how such constructor dispatches interact with the restrictions on constructors or the creation of cyclical objects.

call, however, differ from Java and depend on whether the programmer has declared the constructor to be "delayed" (the default) or not. If the constructor is delayed, then the following restrictions apply:

- Each control path through the constructor must invoke `base` exactly once.
- Each control path through the constructor must initialize all non-inherited fields of non-null type by assigning to them at least once. These initializations may appear either before or after the call to `base`.
- Before calling `base`, references to `this`, including implicit ones, may appear only in contexts of the form `this.fd = expr`.
- After the call to `base`, the program may refer to `this`, but it has "delayed" type, signaling that it is not completely initialized. Additional constraints apply to objects with delayed type:
  - Programs may invoke only delayed methods[2] on delayed objects. (In a delayed method or constructor, `this` has delayed type.)
  - All fields in a delayed object are also delayed and thus subject to the same restrictions. In particular, therefore, a program may not perform any primitive operations that require the value of a delayed field, like arithmetic computation or writing the contents of the field to an output device.
  - Programs may, however, assign to fields in delayed objects, to complete their initialization.

Different restrictions, however, apply if the programmer explicitly marks the constructor as not delayed: each control path through the constructor must initialize all fields of non-null type *before* calling `base`, `this` is otherwise inaccessible before the call to `base`, and after that call, `this` does not have delayed type. As a result, non-delayed Spec# constructors are (it appears) essentially equivalent to MIJAVA's. The portion of the Spec# constructor before the call to `base` corresponds roughly to MIJAVA's field initialization sections, and the remaining part of the constructor is equivalent to MIJAVA's postinit blocks.

In each case, the Spec# type checker verifies that constructors and delayed methods satisfy the above requirements. These requirements appear to be sufficient to guarantee that no program accesses an uninitialized field, although Fähndrich and Xia do not provide a formal proof of this guarantee.

While delayed types, methods, and constructors are primarily part of Spec#'s mechanism for constructing cyclical data structures, and we discuss them further in that

[2] Unlike constructors, methods are not delayed by default; programmers must explicitly mark them as such.

context below, they do affect the language's ability to represent certain construction features. First, by marking individual methods as delayed, programmers can spread object initialization code out across these delayed methods, thereby factoring repeated code out from multiple constructors. Second, because delayed constructors do not allow actions that require access to the object under construction, delayed constructors have no equivalent to MIJAVA's postinit blocks. Instead, class authors must defer such computation to a normal (non-delayed) method, which clients must explicitly call after instantiating the object.[3]

Two primary differences remain between MIJAVA's constructors and Spec#'s non-delayed constructors. First, MIJAVA guarantees that *all* fields are initialized before use; Spec# guarantees this only for fields of non-null type. Second, MIJAVA executes the base class's field initialization section before that of the derived class, whereas Spec# performs these operations in the opposite order. We conjecture that both orders lend themselves equally well to the development of new classes written directly in MIJAVA or Spec#. In migrating classes from either Java or C#, however, the difference can have significant consequences, as it affects the order in which the program executes any side-effecting operations in the constructors. (The order in which the fields are initialized is less important, because, as we have argued above, programs cannot observe partially-initialized objects.) MIJAVA's preservation of Java's initialization order simplifies the migration process, since (if nothing else) programmers do not have to reason about whether side-effecting constructors are independent of this order.

Finally, although the Spec# wiki[4] does mention integration with C#, this discussion is limited to instructions for invoking the compiler correctly, so Spec# cannot be considered an extension of C# in any meaningful sense. MIJAVA, in contrast, demonstrates how to add uninitialized-field guarantees to an existing language and provides a migration path from that language.

OCaml

Object initialization in OCaml [27, 45] is extremely restrictive compared to Java and other mainstream OO languages, as OCaml classes are effectively limited to a single constructor. OCaml does enforce the distinction between the two stages of object initialization, and field initialization expressions may not refer to `this`, ensuring that all fields are accessed before they are initialized. Further, classes may contain a single `initializer` declaration, which is similar to MIJAVA's postinit blocks. Our experience with MzScheme's object system [15, Ch. 5], which uses a similar initialization mechanism, demonstrates that one constructor per class is simply not enough for large-scale

[3] In a non-delayed constructor, on the other hand, the code that appears after the call to `base` is equivalent to a MIJAVA postinit block, but programmers cannot use such a constructor in the (non-assignment-based) creation of object cycles.

[4] `http://channel9.msdn.com/wiki/specsharp/homepage/`, as of December 14, 2008.

```
object NullInit {
    class Test {
        var s : String = _
        def this(s_in : String) { this(); println(this.m()); s = s_in }
        def m() : int = s.length()
    }

    def main(args : Array[String]) {
        val dummy = new Test("Hello, world!")
        println("done");
    }
}
```

Figure 8.1: Uninitialized field reference in Scala

software development. MIJAVA generalizes OCaml's simplistic constructor model to a realistic mechanism such as that found in Java.

Scala

Scala [35] extends OCaml's approach to support multiple constructors, but field initialization code may refer to `this` in Scala. As a result, Scala suffers from all the problems discussed in Chapter 2, as the code in figure 8.1 demonstrates. Upon execution, this program terminates with an uncaught (Java) null-pointer exception at the highlighted expression.[5]

Haskell's Overlooked Object System

Kiselyov and Lämmel describe OOHASKELL [26], a library that supports Java-like class-based object oriented programming in Haskell [40]. OOHASKELL represents objects as (extensible) records of functions; the functions correspond to Java's methods. Fields, in this system, are merely variables bound within the methods; mutable fields have type, e.g., `IORef Int` or `STRef Int`. Classes, however, do not have an explicit representation separate from the functions that create instances.

Figure 8.2 displays an example of object creation in OOHASKELL. The `mk_rectangle` function, which serves as the constructor of the implicit rectangle "class," takes five arguments and produces a new rectangle instance. The first four parameters, `x`, `y`, `width`, and `height`, are initialization parameters for the object, much like Java constructor arguments, and the fifth parameter, `self`, is a reference to the object itself. The function

[5] Scala 2.7.1 final, Java 1.5.0_16, Mac OS 10.5.5.

```
mk_rectangle x y width height self
  = do
    -- invoke "constructor" in superclass (shape)
    super <- shape x y self

    -- declare mutable fields
    w <- newIORef width
    h <- newIORef height

    -- construct & return object (record of methods)
    returnIO $
        getWidth  .=. readIORef w
    .*. getHeight .=. readIORef h
    .*. setWidth  .=. (\neww -> writeIORef w neww)
    .*. setHeight .=. (\newh -> writeIORef h newh)
    -- etc
    .*. super
```

Figure 8.2: OOHASKELL Example constructor

invokes the superclass's "constructor" to obtain the record representing an instance of the superclass, declares various bindings corresponding to Java fields, and then extends the superclass instance record with new slots corresponding to the class's methods. The expression *label* `.=.` *value* is record-component specification, which serves the purpose of method definition, and the `.*.` operator is record extension.

To create an instance of this class, a client writes

```
mfix (mk_rectangle (10 :: Int) (20 :: Int) 5 6),
```

where the `mfix` function, in Haskell's `MonadFix` class, "feeds the object to its own constructor," so to speak, and finds the fixed point implicit in the methods' mutual recursion. Since fields are normal Haskell bindings, programmers cannot create a field without specifying an initial value (even if the value is `Nothing`, for fields of `Maybe` type), so guarantees against uninitialized field access follow directly from the underlying language.

Because the identifier `self` is in scope throughout the constructor, it is possible to write code that invokes a method on `self` before the object has been constructed, as in the highlighted expression in figure 8.3. (Note that, even though Haskell is lazy, the highlighted expression is guaranteed to execute before the record creation, since the entire body of the constructor is in a `do` form.) This premature method call, however, does not observe the uninitialized field as it would in Java or C++, but rather

```
mk_rectangle x y width height self
  = do
    -- declare mutable fields
    w <- newIORef width
    h <- newIORef height

    print (self # getWidth)
    -- construct & return object (record of methods)
    returnIO $
         getWidth  .=. readIORef w
     .*. getHeight .=. readIORef h
     .*. setWidth  .=. (\neww -> writeIORef w neww)
     .*. setHeight .=. (\newh -> writeIORef h newh)
     .*. emptyRecord
```

Figure 8.3: OOHASKELL: Premature computation on an object

```
mk_rectangle x y width height self
  = do
    -- declare mutable fields
    w <- newIORef width
    h <- newIORef height

    -- print (self # getWidth)
    -- construct & return object (record of methods)
    construct self (\self ->
         getWidth  .=. readIORef w
     .*. getHeight .=. readIORef h
     .*. setWidth  .=. (\neww -> writeIORef w neww)
     .*. setHeight .=. (\newh -> writeIORef h newh)
     .*. emptyRecord)
```

Figure 8.4: OOHASKELL: Statically preventing premature computation

```
class Base {
public:
    Base() { m(); }
    virtual void m() {
        std::cout << "Base\n";
    }
}

class Derived : public Base {
public:
    Derived() : Base() { m(); }
    virtual void m() {
        std::cout << "Derived\n";
    }
}

int main() {
    Derived d;
    return 0;
}
```

Figure 8.5: Constructor behavior in C++

diverges. To help programmers avoid this problem, the authors provide an additional construction mechanism that uses OOHASKELL functions `new` and `construct` instead of `mfix` and `returnIO`, respectively, as demonstrated in figure 8.4. With these functions, OOHASKELL uses the type system to ensure that the program cannot invoke methods on `self` before constructing the record and computing the fixed point, and the highlighted expression in this new figure results in a compile-time type error when uncommented. Their strategy for preventing premature computation on an object does not work for Java, however, as it relies both on Haskell's advanced type system and on its lazy evaluation, neither of which are available in traditional OO languages.

C++

C++ [51] attempts to prevent uninitialized field accesses by treating virtual method dispatch in a special way in the dynamic extent of constructors. Programmers may refer to `this` at any point in the execution of a constructor, and the run-time system adjusts the dynamic type of the object `*this` as construction proceeds. As an example, consider the program in figure 8.5. The allocation of the variable `d`, of type

```
#include <iostream>
using namespace std;
class Base {
public:
    Base() { m2(); }
    virtual ~Base() { }
    virtual void m() = 0;
    void m2() { this->m(); }
};

class Derived : public Base {
public:
    Derived() : Base() { }
    virtual ~Derived() { }
    virtual void m() {
        cout << "Derived::m" << endl;
    }
};

int main() {
    Derived d;
    return 0;
}
```

Figure 8.6: Executing a pure virtual method in C++

`Derived`, starts by executing `Derived`'s constructor, which calls `Base`'s. As execution moves through the two constructors, the run-time system updates the dynamic type (i.e., vtable pointer) of the object `*this` to correspond to the constructor currently executing. So, during the dynamic extent of `Base`'s constructor, the object under construction has the dynamic type `Base`, even though it is "really" an instance of `Derived`, and thus the first call to `m` executes `Base`'s implementation of the method. When control returns to `Derived`'s constructor, the object `*this` now has dynamic type `Derived`, and so the second call to `m` executes the implementation in `Derived`. As a result, the above program prints "`Base`" and then "`Derived`" upon execution.

This strategy is, however, an imperfect solution to the uninitialized-field problem, for two reasons. First, this method can only guarantee that the program never accesses an uninitialized field if every constructor for a class `C` initializes every field in `C` before performing any other operations on `this`, but the compiler issues no diagnostics if a

class author neglects one or more fields. Second, this mechanism makes it possible for programs to call pure virtual (i.e., abstract) methods, causing (on one implementation) the run-time system to abort the program. Some compilers can detect this invocation of pure virtual methods under certain circumstances, but not in the general case. The program in figure 8.6, for instance, compiles without error but fails at runtime. As in the first example, the allocation of `d` causes the program to call `Derived`'s constructor, which calls `Base`'s, which in turn calls `Base`'s `m2()` method. This calls the `m()` method, and since execution is in the dynamic extent of `Base`'s constructor, dynamic dispatch requires the program to select `Base`'s implementation of `m()`. Since no implementation exists, the runtime system (of GCC 4.0.1 on Mac OS X 10.5.4) aborts the program.

Cyclone

Cyclone [21, 25],[6] a safe variant of C, provides a static analysis that ensures that all variables of pointer type are initialized before use, but this does not apply to variables of other type. As an example, consider the program in figure 8.7. The compiler signals a warning at the `return` statement in the `mk_s` function, indicating that the returned expression may not be fully initialized, and refuses to compile the program. If, however, we modify the highlighted declaration to give the `y` field the type `int` rather than `int*`, the compiler issues no warning. Any use of `result` in the body of the `mk_s` function, such as passing it as an argument to another function, would also trigger the compiler diagnostic and prevent compilation. Of course, since Cyclone extends C and not C++, it does not have direct support for objects, inheritance, and dynamic dispatch and thus avoids many of the initialization problems of Chapter 2. Its pointer initialization analysis, therefore, is of limited applicability to explicitly object-oriented languages like MIJAVA.

Other Systems

Fortress's object initialization [1, Ch. 11] is a simplification of OCaml's and is thus even more restrictive. Like OCaml, a Fortress class has a single constructor, but Fortress omits initializer blocks, so `this` is completely inaccessible during object construction. Additionally, Fortress classes may only extend traits, which do not have fields, so superclass initialization is not a concern, and the language avoids all the problems that arise therefrom, so we cannot apply Fortress's techniques to Java programs.

The class system in PLT Scheme [15, Ch. 5] uses an initialization mechanism similar to OCaml's in which classes have a single constructor, and fields must have initialization expressions. Unlike OCaml, though, these expressions may invoke methods on the current object and may therefore refer to uninitialized fields.

[6] See also `http://cyclone.thelanguage.org/`.

```
#include <stdio.h>

typedef struct S {
    int x;
    int *y;
} S;

S *mk_s();
void use_s(S *);

int main() {
    S *p = mk_s();
    use_s(p);
    return 0;
}

S *mk_s() {
    S *result = (S*) malloc(sizeof(S));
    result->x = 3;
    return result;
}

void use_s(S *p) {
    printf("%d\n", *(p->y));
}
```

Figure 8.7: Initialization analysis in Cyclone

Theta [9, 29] is a class-based object-oriented language developed for use within an object-oriented database system. Although it provides both a sophisticated construction system that avoids uninitialized field references and a `maybe` type similar to ML's, the authors do not address its use in situations with large existing code bases. They instead propose Theta as an entirely new language, rather than making incremental changes to an existing language, and they fail to provide any sort of migration path for programmers who want to take advantage of Theta's guarantees in existing programs.

MOBY [42] is an ML-like language that includes an object system. Unlike OCaml, MOBY supports multiple constructors (called "makers") in a single class. It requires each maker to initialize its superclass as well as all fields defined locally, and it prohibits references to `this` within the maker body, and this appears to be sufficient to

prevent access to uninitialized fields. As much as it uses ideas from ML, however, MOBY is a completely separate language, not a set of incremental changes to an existing language, and its authors also do not provide a migration path for existing programs.

In Eiffel [31, 10], a class author may specify one or more creation routines that play the role of constructor. These creation routines may execute arbitrary code and may therefore refer to an uninitialized field. This is mitigated somewhat by Eiffel's contract system, which provides a way to detect incompletely-initialized objects upon entry into a method called by a constructor, particularly through the use of class invariants. Contracts can only catch such partial initializations, however, if the class author has written the contracts in question, and even then the language does not signal an error until run-time. MIJAVA detects partial initialization statically with no additional effort by the class constructor.

O'Haskell [34][7] defines another object-oriented extension to Haskell whose initialization mechanism is much like OCaml's: constructors are standard functions which accept arguments and return newly constructed objects, as in section 3.1 of Nordlander's thesis. As in OCaml, all field declarations must have initial value expressions. This use of functions to create objects causes the same problems with subclassing as static factory methods do in Java,[8] but they avoid the problem by providing only structural subtyping without behavioral inheritance [34, §2.7]. (In the terminology typically used in the Java and C++ communities, O'Haskell provides something close to *interface inheritance* but not *implementation inheritance*.) As a result, the notion of a subclass does not exist, so the problem of superclass initialization does not arise.

ML-ART [44], like OOHASKELL, represents traditional class-based object-oriented concepts in a functional language (here, ML) through clever use of features in the host language's type system. Object instantiation in ML-ART, like in OOHASKELL, avoids uninitialized fields through the use of the underlying language's record creation mechanism, which requires initial value expressions for all fields. This strategy is not available in languages based on Java, since Java does not have a notion of records.

Smalltalk [18] and its descendant Objective C [2] rely on an initialization protocol that requires the client to invoke two methods in succession. The first method, a class method named `new`, allocates memory for an instance of the class and, by default, sets all of the instance's fields to `nil`. The second method, an instance method, performs the rest of the initialization. BETA [30] and `gbeta` [12] use a similar mechanism with minor surface differences. In all of these languages, the initialization method has full access to `this` and therefore suffers from the problems in Chapter 2. Additionally, it is entirely the client's responsibility to invoke the second method; the language issues no

[7] See also `http://www.cs.chalmers.se/~nordland/ohaskell/`.

[8] Java classes with static factories typically have only private constructors, to ensure that all instantiation happens through one of the factory methods. As a result, such classes cannot have subclasses, because there is no way for a subclass to invoke its superclass's constructor.

error or warning if the client performs other operations on the object before invoking the initialization method.

Typestate, introduced by Strom and Yemini [50], extends traditional types with additional information about values and the operations legal upon them. While, for instance, Java's type system cannot distinguish between a `java.io.InputStream` instance that is open and one that has been closed, a typestate extension to Java could track how a variable of type `InputStream` passes from one state to another. As a result, this system could statically prevent a program from reading from a closed stream. The authors also provide several examples of using typestate to prevent accesses of uninitialized local variables. Applying typestate to field initialization is an obvious extension, and the J\mask calculus of Qi and Meyers [43] explores precisely this possibility.

J\mask extends the Java type system with field masks that indicate those fields which may (or must) not be initialized. For instance, if `T` is a class with fields `x`, `y`, and `z`, then the type `T\x` describes an object of class `T` in which the field `x` may not yet be initialized but `y` and `z` definitely are; the type system prevents references to `x` (though assignments are of course legal). The J\mask programmer annotates methods and constructors with effect clauses indicating their (de)initialization behavior. While the J\mask type system cannot infer these effects, the language does provide defaults that, according to the authors, suffice in the common cases. A type system similar to these masked types might be useful in reasoning about the creation and manipulation of partial pre-objects, as mentioned in our discussion of the migration of Jacksum to MIJava's constructors in section 7.1. Their type system, however, is dramatically more complex than either Java or MIJava. Further, unlike MIJava, they do not have explicit support for *null*-as-*Maybe*, and they do not discuss a migration path from Java to J\mask. Instead, it appears that a programmer must write the entire program in J\mask to benefit from its stronger guarantees. Finally, we conjecture that the effect annotations on methods and constructors can interfere with abstraction in much the same way that Java's `throws` clauses currently do, as we describe in section 6.5.

Formal Models

Of the three major formal models of Java, only Featherweight Java's constructor mechanism [24] ensures that all fields are initialized. Because Featherweight's constructors require the initial field values (for *all* fields in the class) as arguments, though, they are not suitable for large-scale software development. ClassicJava [14] has no constructors but instead requires the client to initialize fields with explicit field assignments, and Middleweight Java [5] simply models Java's constructors directly.

Additionally, none of the languages discussed above has formal models that include the entire object construction mechanism. OCaml's model due to Rémy [45] is the closest, but it only covers applicative field initialization and omits the `initializer`

section. (The more recent model of OCaml by Owens [36] omits the object system completely.) Supporting multiple constructors, each with its own postinit block, is by far the most complex part of MIJMODEL and MIJAVA's implementation.

Static Analyses

Many people have proposed static analysis techniques to detect and report uses of uninitialized fields, such as Seo et al [46, 47]. Their analysis is a first-order, type-based analysis; for method calls, they take the most conservative approach and consider the effects of the named method as implemented in *all* possible subtypes of the message recipient. It is a conservative analysis in the sense that all uninitialized field references are reported (and they provide a proof of this soundness property), but the report may contain additional false positives. As a result, their error reporting is much coarser than ours: while MIJAVA can identify the particular constructor that fails to initialize a field, they appear to be able to say only that a particular field is read before initialization. Additionally, they provide no discussion of the precision of their analysis, how many false positives they obtain in practice, and whether their users can easily determine which errors represent actual problems and which do not.

Several analysis techniques [38, 39, 54] attempt to discover various properties of programs in Java-like languages using techniques based on CFA [48] or SBA [22]. None of these analyses, however, attempts to conclude anything about the contents of fields beyond their types and so provide no more information about the program than the Java or C# type systems already yield. They specifically do not attempt to detect references to uninitialized fields.

Hovemeyer et al [23] describe a first-order dataflow analysis that is intended to detect null-pointer dereferences in arbitrary Java method definitions. They do not explicitly use their analysis to ensure that a constructor initializes each field before accessing it, and it is not clear if their analysis applies to this question, as they have published only an informal description of the analysis. It seems, however, to be possible to use their technique to ensure that no fields are *null* at any reference to `this` within the constructor or when the constructor returns. This approach, however, has two drawbacks. First, their method is incapable of distinguishing between an uninitialized field and a field that has been initialized to `null`. More seriously, they acknowledge that their analysis is unsound. MIJAVA avoids both of these problems.

8.2 *Maybe* types

Types similar to MIJAVA's `Maybe` are well-known, appearing in Haskell directly and in SML and OCaml as `OPTION`; similar types appear in derived languages, such as MOBYand OOHASKELL. In these languages, however, programmers use the built-in

general-purpose pattern matching mechanism to dispatch on and deconstruct *Maybe* values. Some object-oriented languages, like Theta, also provide a *Maybe* type with pattern matching (although Theta's is less general than ML's and Haskell's, as it does not support nested patterns). For an existing language like Java, decomposition through pattern matching is not a good fit, as it would require significant programmer-visible additions to the language.

Many research object-oriented languages like Spec# and Cyclone address the problem of *null*-as-*Maybe* by adding non-null pointers to a Java- or C-like type system. For instance, in Spec#, the type `T` induces another type `T!` that represents a non-*null* reference to `T`. A programmer may use an expression of type `T` in a context that expects `T!`, and the compiler inserts a run-time check and issues a warning. Alternatively, the program can explicitly check whether the reference is *null*, and the type system uses this check to refine the type within the body of the conditional, as in the following example:

```
T t = ...;
Console.WriteLine(t.name);      // issues "may be null" warning
if (t != null) {
    // t has type T! in this block
    Console.WriteLine(t.name);  // no warning
}
```

This is much less expressive than *Maybe* as found in MIJAVA, Haskell, or ML, because it does not support a representation for `Maybe<Maybe<T>>` and thus methods that return the nullable type `T` are potentially ambiguous, as we discuss above in section 3.1.

Both Spec# and Cyclone support the equivalent of MIJAVA's coercions. If a programmer in either of these languages uses an expression of nullable pointer type `T` in a context expecting the corresponding non-nullable type `T!`, the compiler issues a warning and inserts a run-time check to ensure that the pointer is not *null*. The programmer can disable this warning either by inserting an explicit cast to `T!`, or (in Spec#) explicitly compare the reference to `null`, but the run-time check clearly remains in either case.

Cω [6], an extension of C#, has support for non-null types as a special case of streams. For a type `T`, the three derived types `T*`, `T?`, and `T!` denote streams of `T` of, respectively, arbitrary length, at most one element, and exactly one element. In stream contexts, then, `T!` and `T?` serve the same purposes as MIJAVA's types `T` and `Maybe<T>`, respectively. Because Cω automatically flattens nested streams—that is, the types `T*` and `T**` are equivalent—it conflates the types `T??` and `T?`, so Cω has no equivalent to MIJAVA's `Maybe<Maybe<T>>` types.[9]

[9]The model in the paper avoids access to uninitialized fields of non-nullable type through the same constructor mechanism in Featherweight Java: ignoring details of surface syntax, each class has a single constructor that takes one argument for every field in the class, including those inherited from superclasses.

Fortress has a `Maybe` type, but it provides pattern matching and an `unJust` method, similar to MIJAVA's `valOf`, for programmers to decompose `Maybe` values, rather than MIJAVA's implicit coercions.

Scala has an explicit `Maybe[T]` type, but it is an imperfect solution to the problem of removing *null* from Java. Scala still includes the value `null`, as distinct from `none`. Expressions of object type, including those of type `Maybe[T]`, may evaluate to `null`, so programmers cannot escape the engineering problems associated with the presence of this value. Additionally, Scala requires programmers to use the language's general pattern matching facility to decompose `Maybe` values, and this requires larger additions to Java than MIJAVA's implicit coercions.

Chadwick and Lieberherr propose DemeterF [8], a system in which the programmer defines "traversals" on data structures; these traversals are a generalization of the visitor pattern [17]. Their language for describing data structures includes a parameterized `Option` type, similar to MIJAVA's `Maybe`. In his forthcoming thesis proposal, Chadwick describes DemeterF's type system, which statically guarantees that programs never attempt to extract the value from `none` and thus never generate the equivalent of MIJAVA's valof-none exception or Java's null-pointer exception. As a consequence, whenever the program applies a traversal to an object graph that includes values of `Option` type, the programmer must supply a visitor that explicitly includes code to handle both `some` and `none` cases. This requirement is simply an example of the tradeoffs associated with static typing: DemeterF provides stronger static guarantees than MIJAVA, but at the cost of rejecting more programs that would complete without a type error. We claim that these additional rejections have a significant impact, requiring the traversal author to specify behavior for both `some` and `none`, even in those (relatively common) cases where the programmer can prove (outside the language) that `none` cannot occur. In contrast, MIJAVA's implicit coercions defer much of the checking until runtime but allow the programmer to omit code for the `none` case when appropriate.[10]

In their formalization of traversal specifications [37], Palsberg, Xiao, and Lieberherr define the notion of object graph conformance (to a specified class graph). Among other restrictions, a conformant object graph cannot contain *null*. Although they do not address this in the paper, programmers can still represent optional properties in this system through the use of a distinguished "sentinel" object that represents an absence of the parameter; the alternative representation of *Maybe* discussed in section 3.3 is one such mechanism. In addition to requiring the programmer to define the sentinel object, such strategies also suffer from many of the same problems discussed in the earlier section, especially those involving subclassing and `final` classes. Further, the authors do not discuss extending these non-*null* guarantees to local variables, method

[10] In the implementation of DemeterF, the programmer may choose not to run the type checker. In that case, of course, the programmer does not have to provide complete traversals, but DemeterF cannot guarantee the absence of `none`.

parameters, method results, and array elements. Finally, they do not address the object creation process and thus provide neither a default value for uninitialized fields nor an argument that such a default is unnecessary.

8.3 Delayed Initialization

Most object-oriented languages do not attempt to provide any support for delayed initialization beyond that of Java. All of the languages we have surveyed require the use of an initial placeholder (and thus a `Maybe`-like type, when possible) and subsequent field assignment for initialization delayed because the necessary values are unavailable. Only J\mask attempts to give the programmer additional support for this, through its types that reflect which fields have been initialized and which have not. Many of these languages do provide support for the creation of (statically known) cycles; we discuss those further below. And finally, only those languages that have general support for laziness (OOHASKELL, O'Haskell, OCaml, and PLT Scheme) support lazy field initialization directly; all others require the "lazy load" pattern of section 4.3.

Cycle Creation

Most of the languages surveyed here have no support for the creation of cyclical data structures without the use of side effects; cycle creation in these languages therefore requires (the equivalent of) *null* as a temporary value, regardless of whether the cycle's structure is known at compile-time. As a consequence, those languages like Fortress that have a *Maybe* or non-null reference type require the programmer to give fields that participate in the cycle the more general type (i.e., nullable reference or `Maybe<T>`) and, where appropriate, mark the field as mutable. The explicit *Maybe* type imposes syntactic overhead on all references to the field, although (as discussed previously) this signals the programmer that it is unsafe to assume that these fields are initialized before use. More seriously, these type and mutability requirements greatly restrict the class author's ability to state invariants about the fields in the type system: the fields in the cycle must have *Maybe* type, even if they do not actually represent optional properties of the containing object, and they must be mutable, even if the programmer wishes to forbid mutation after creating the cycle.

Spec#

As we discussed in Chapter 4, Spec# provides a mechanism for constructing cyclical data structures without explicit side effects, as described in Fähndrich and Xia [13]. Like our proposed mechanism, objects in a cycle have "delayed" type (corresponding to our "interim" type) until all such objects have been fully constructed. However,

```
class List {
    Node! head;
    public List() {
        this.head = new Node(this);
    }
}
class Node {
    List! list;
    Node! prev;
    Node! next;
    public Node([Microsoft.Contracts.Delayed] List! list) {
        this.list = list;
        this.prev = this;
        this.next = this;
    }
}
```

Figure 8.8: Cycle creation in Spec#

Spec#'s mechanism (or the documentation thereof) appears to be very limited in the kinds of cycles it can construct. For instance, in the code in figure 8.8 (from the Spec# wiki) the `List` constructor creates a cycle containing two objects, a `List` instance and a `Node` instance. However, the language does not appear to support the creation of a cycle of `Nodes` outside the context of a `List`. That is, Spec# does not appear to have an equivalent of the proposed `cyclical` declaration form from Chapter 4, so languages like MIJAVA and OCaml appear to be more flexible. Additionally, since the creation of such cycles requires the use of "delayed" constructors, the classes cannot perform actions that require access to the objects as non-delayed values during construction (as is possible in MIJAVA's postinit blocks in the proposal of section 4.2. Instead, the class authors must defer such computation to a normal method, which clients must explicitly call after instantiating the various objects.

We cannot make this comparison with certainty, however, because Fähndrich and Xia's paper concentrates on the formal model of delayed types and does not provide sufficient details of the concrete syntax in the Spec# implementation, and the Spec# documentation is completely silent on this point. Spec# also appears to have no support for creating cycles whose structure is not determined until runtime.

```
using System;

public class ExnContainer {
    public ExnItem item;

    public ExnContainer() {
        base();
        try {
            foo();
        } catch (TestException) {
            Console.WriteLine("caught");
        }
    }

    [Microsoft.Contracts.Delayed]
    public void foo() {
        new ExnItem(this, "x");
    }

    static void Main() {
        ExnContainer e = new ExnContainer();
        Console.WriteLine( e.item.msg.Length );
    }
}

public class ExnItem {
    public string! msg;

    public ExnItem([Microsoft.Contracts.Delayed] ExnContainer b, string! s) {
        base();
        b.item = this;
        throw(new TestException());
        msg = s;
    }
}

public class TestException : Exception {
    public TestException() {
        base();
    }
}
```

Figure 8.9: Exposing partially-initialized objects in Spec#

Fähndrich and Xia also acknowledge the potential for error in the interaction between exception handling and delayed object initialization. In particular, the program in figure 8.9 demonstrates how a delayed object can "escape" before being fully initialized. This program compiles with two warnings about potentially null receivers (`e.item` in the first highlighted expression, and `b` in the second) but no other compiler diagnostics. Upon execution, the program performs the following actions:

1. `Main` begins by instantiating `ExnContainer`; this invokes the class's sole constructor, which is delayed.

2. `ExnContainer`'s constructor calls `base`, and then establishes an exception handler. Within this handler, it calls the `foo` method.

3. The `foo` method creates a new instance of `ExnItem`, supplying a reference to the `ExnContainer` object as an argument. (This is legal only because `ExnItem`'s constructor declares the corresponding formal argument to be delayed.)

4. The `ExnItem` constructor stores a reference to itself in the `ExnContainer`'s `item` field. Then, before it completes its initialization, it throws an exception, which leaves the `msg` field *null*—even though it has non-*null* type.

5. Control returns to the `catch` block in `ExnContainer`'s constructor, which prints `caught`.

6. `ExnContainer`'s constructor returns to `Main`, which attempts to refer to a field of `e.item.msg`, only to fail with a null-reference exception. (This demonstrates that Spec#'s type system is unsound, as `msg` has non-null type!)[11]

They propose to address this problem by prohibiting exception handlers within constructors (and, presumably, other delayed methods), arguing that this restriction is sufficient to ensure that no delayed object "escapes" before becoming completely initialized. In the example above, for instance, the handler for the `TestException` would have to appear in `Main`, which would not have a reference to the `ExnContainer` instance, let alone the `ExnItem` instance. Further, the exception object could not contain a reference to either instance, because Spec# does not permit fields to have delayed type, and the type system forbids programs from storing delayed references into non-delayed fields. The authors, however, provide only an informal argument that this restriction is sufficient to avoid this problem and do not investigate the question in detail. Further, they do not explore the impact of this restriction and its potential loss of expressiveness on programmers.

[11] Note further that the warnings do not accurately suggest the source of the error, as neither of the indicated receivers, `e.item` or `b`, is in fact *null*.

OCaml and PLT Scheme

As discussed in Chapter 4, OCaml allows programmers to create cyclical data structures without explicit side effects or laziness, by extending the recursive local binding form to allow certain forms of expressions beyond functions on the right-hand side of the bindings. The statically-enforced restrictions on the form of these expressions guarantee that the program cannot refer to any of the values being created until it has completed the cyclical structure. This support, however, does not extend to the creation of class instances, nor does it support the creation of cycles whose structure is not known at compile time.

To create cycles involving objects, therefore, an OCaml programmer must use one of two techniques. In addition to the strategy described above that uses post-construction mutation to create the cycle, an OCaml programmer can also use explicit laziness, as in the following example:

```
class circlist (datum : int) (next : circlist Lazy.t) =
  object
    method get_datum = datum
    method get_next = Lazy.force next
  end

(* let rec allows "lazy (...)" but not "new C ..." *)
let cycle = (let rec a = lazy (new circlist 1 b)
                 and b = lazy (new circlist 2 c)
                 and c = lazy (new circlist 3 a)
             in (Lazy.force a))
```

With this implementation, all references to the `next` field within the class must explicitly force the delayed computation, as in the highlighted expression above. Both alternatives, therefore, impose a significant burden on the class author. Further, only the first alternative supports the creation of cycles whose structure is not known until runtime.

PLT Scheme's support for cyclical objects is similar to OCaml's. MzScheme's `shared` construction does not permit the direct construction of cycles involving objects. Programmers have the option of using the language's `delay` form, similar to OCaml's `lazy`, but at the cost of having to force all references to the affected fields.

J\mask

J\mask's type system grants programmers significant flexibility in delaying the initialization of certain fields without the presence of a distinguished *null* value, because the type system prevents access to uninitialized fields. This extends even to the creation

of cyclical data structures, through their use of conditional masks, which denote types of objects whose initialization status depends on that of other objects. For instance, the type `T\f[x.g]` denotes the type `T` in which the field `f` is initialized but refers to a partially initialized object. After the program initializes `x.g`, then the object in the field `f` is fully initialized, and the mask is removed. Complex rules for the cancellation of cyclical conditional masks allow programmers to use these types, along with explicit field assignments, to create cyclical data structures and to have static guarantees that programs never refer to uninitialized fields, even those in cycles. Unlike MIJAVA, J\mask's support for delayed initialization applies even to those cycles whose structure is not known until runtime, but at the cost of a great deal of additional complexity in the type system.

OOHASKELL

Although Kiselyov and Lämmel do not discuss the creation of object cycles explicitly, it is possible to create such cycles in OOHASKELL without using field mutation, although the programmer must make two changes to the object creation mechanism discussed previously. First, as mutually-recursive scope is necessary to construct cycles in Haskell, and as the object creation operation inhabits the `IO` monad,[12] the programmer must use the `mdo` form instead of the more typical `do` to bind the newly-created objects to identifiers.[13] Second, because the type inferred by Haskell for an object includes the signatures of all the object's methods, the class author must exercise special care when dealing with method arguments or results that have the same type as the containing object. Such methods frequently arise in objects that participate in cycles; for example, an object that represents a node in a circular list likely has a `getNext` method that returns the next object (of the same "class") in the list. If `getNext` simply returns the next object, then Haskell will attempt to infer a recursive type for the object and signal a compile-time error, because Haskell does not support equi-recursive types. The same problem arises if an object contains a method that takes an argument of the same type as the containing object, like `setNext` in the circular list.[14] To address these problems, the class author must add a level of indirection to give the object an iso-recursive type, as discussed in section 5.8 of Kiselyov and Lämmel's paper.

[12] Programmers can in special cases write object creations functions that do not have monadic type, but the use of the safer `construct` and `new` forms force the use of the `IO` monad.

[13] The `mdo` form is not part of Haskell 98 but is instead provided as part of the Glasgow extensions. Since, however, OOHASKELL itself requires these extensions, this does not require any language facilities not already available.

[14] Since the problem arises whether the object type appears on the left or right of the arrow in the method's type, the problem with recursive types appears even in objects that require the client to use field mutation to establish cycles.

O'Haskell

The object creation in O'Haskell is also a monadic operation and thus suffers from the limits of `do` bindings as discussed above. However, O'Haskell is available only as a customized build of Hugs,[15] so we suspect that the Glasgow extensions, including `mdo`, are not available in this system, and as a result, effect-free creation of object cycles is not possible. Since, however, O'Haskell was last released in January, 2001, it fails to compile or run on modern versions of Linux, Windows XP, and Mac OS, and we therefore cannot test this claim.

[15] `http://haskell.org/hugs`

Chapter 9

Lessons for the Future

In addition to supporting our claim that removing *null* from Java can have significant engineering benefits with limited impact on existing programs, our work makes several additional contributions for future researchers and language designers. Our experiences with MIJAVA suggest many avenues for future work on Java-like object-oriented languages. Additionally, several of the design decisions we made in this effort and the practical consequences of those decisions provide valuable lessons for other language designers, even those working outside the realm of object-oriented programming languages.

9.1 Future Work

Our work thus far, especially our experience using MIJAVA with existing code bases, suggests several directions for further research. This includes extending the implementation of MIJAVA to support additional Java features like inner classes and extending MIJAVA's design with additional features for programmer convenience. Additionally, some open questions remain, particularly involving the initialization of static fields.

Translation

The translation as described in Chapter 6 lacks support for several important features of Java and MIJAVA, and additional effort is required to make it usable on arbitrarily complex Java programs. Three main points remain to be addressed: support for nested and local classes, removing or mitigating the restrictions on MIJAVA classes that extend Java classes, and making MIJAVA's initialization system more flexible (while preserving safety) to allow cleaner solutions to some of the problems discussed in Chapter 7.

The translation does not currently support nested or local classes. Specifically, it can translate an MIJAVA class that contains nested classes, but it cannot translate a

nested class written in MIJAVA. In addition to implementation work, we need to explore the interaction between MIJAVA's initialization mechanism and Java's initialization mechanism for nested classes to be sure that no violations of MIJAVA's guarantees can arise as a result; extending MIJMODEL is a necessary first step.

We conjecture that the restriction on constructors in Java superclasses of MIJAVA classes is not, in fact, a significant problem during program migration. Consider the case where a programmer migrates a class *C* from Java to MIJAVA, where *C*'s superclass *D* is still written in Java. Because Java requires each of *C*'s constructors to begin with a call to *D*'s constructor, then in the obvious migration, each of *C*'s constructors will begin by invoking *D*'s constructor without performing significant computation first, so supplying the correct arguments is straightforward. To improve expressiveness and programmer flexibility, though, we must investigate other implementation strategies that avoid this constraint.

The largest problem is to find extensions of MIJAVA's initialization mechanism to regain some of Java's flexibility, without weakening MIJAVA's guarantees about uninitialized field accesses. There are several promising possibilities that deserve investigation. To allow classes to support `reset` methods without duplicating code from the constructor, we could add another operation on pre-objects. In MIJAVA as described here, programs may extend pre-objects with additional fields to create a subclass pre-object, and they may reify complete pre-objects into full instances. If MIJAVA allowed programs to *replace* the state in an existing object with that in a pre-object of the corresponding type, MIJAVA programmers could define `reset` methods in terms of the constructors, rather than the other way around. This would avoid repeating the re-initialization code. Further, for `reset` methods that only need to reset some of the object's fields, MIJAVA could allow programmers to create a pre-object based on an existing instance, and then use a functional-update mechanism to adjust fields as needed.

Our experience migrating programs from Java to MIJAVA also suggests that programmers would benefit if MIJAVA allowed them to create pre-objects incrementally, by specifying only some of a class's fields and then using a record-extension mechanism to add the remaining fields. This would be particularly useful to avoid repeated code in classes that support multiple constructors, all of which initialize certain fields in the same way but differ on the remaining fields. The primary difficulty with this proposal, of course, is statically proving that these incrementally-constructed pre-objects contain values for *all* of the class's fields before allowing the program to reify them into full objects. It may be possible to extend the type language to indicate which of a pre-object's fields have been initialized, but the consequences of this added complexity are not clear. While the type system appears to be similar to those for functional languages that allow "breadth" subtyping on records, we need to investigate this proposal's interaction with Java's nominal subtyping and method overloading to be sure it is safe.

Automating the Migration

While Java constructors are largely unconstrained by the language, "well-behaved" constructors, those which invoke a superclass's constructor, initialize all fields, and do not otherwise refer to `this`, are an important special case (not least because many textbooks and programming teachers encourage students to write constructors in this fashion). We can easily automate the migration of such constructors to MIJAVA as follows:

1. Declare a local variable for each field defined within the class.
2. Bind a fresh variable of pre-object type to the result of the superclass constructor.
3. Replace all assignments to class fields with assignments to the corresponding local variable.
4. Replace each `return` statement, including the implicit return at the end of the constructor body, with a `return` statement that uses the superclass pre-object and the local variables to construct a pre-object.

To aid programmers in migrating large projects to MIJAVA, it should be possible to implement this automatic transformation and to provide a static analysis (perhaps based on one of the techniques discussed in the previous chapter) to recognize cases where we can migrate classes automatically.

Automatically Synthesized Constructors

The proposed initialization mechanism of Chapter 2 effectively prevents the compiler from synthesizing default constructors, as Java compilers currently do. As a matter of programmer convenience, MIJAVA could support some degree of compiler generation, but not to the extent possible in Java.

In the face of the *Maybe* type of Chapter 3, the MIJAVA compiler can synthesize a default initial value only for fields of *Maybe* type.[1] So, a default nullary constructor is possible only if the superclass has a nullary constructor, and if every field has *Maybe* type.

As an aid to programmer convenience, though, we could relax MIJAVA's restriction that `preobj` forms provide field initialization clauses for every field defined directly within the containing class and allow the programmer to omit fields of `Maybe` type. For such omitted fields, MIJAVA could synthesize a clause initializing the field to *none*. The remaining question is whether the convenience of this synthesis outweighs the potential for error caused by programmers who inadvertently omit such clauses.

[1] We ignore fields of primitive type (`int`, `boolean`, etc.) here.

```
public class C {
    // singleton pattern
    public static final C theInstance = new C();

    private C() { }

    private static final Boolean FLAG = false;

    private final Boolean b = FLAG;

    public static void main(String[] args) {
        if ( theInstance.b ) {
            System.out.println("it's true!");
        } else {
            System.out.println("it's not!");
        }
    }
}
```

Figure 9.1: Example of difficult static initialization

Static Fields

Static field initialization and its interaction with instance field initialization present difficult problems. As Bloch and Pugh showed [7], programs can exploit this interaction to refer to uninitialized static fields, even with MIJAVA's construction mechanism. As an example, consider the Java 5 program in figure 9.1.

Executing this program produces a null-pointer exception at the highlighted reference to `theInstance.b`, due to the interleaving of static and instance field initialization and the auto-coercions between `Boolean` and `boolean`. Specifically, when this program is run, the following actions take place, in order:

1. The JVM loads the class `C` and initializes all of `C`'s static fields to *null*.[2]

2. The JVM evaluates the initializer expressions for `C`'s static fields and mutates the fields accordingly, beginning with `theInstance` and moving down the class definition.

[2]The JLS specifies that final static fields whose initializers are compile-time constant expressions are initialized to the specified values rather than to *null*. This exception does not apply here, even to the `FLAG` field, because `false` is *not* a compile-time constant expression when it appears in a context that expects a value of type `Boolean`, due to the inserted implicit coercion.

```
public class A {
    public static final Integer s1 = 1;
    public static final Integer b1 = B.s1;
    public static final Integer s2 = 2;

    public static void main(String[] args) {
        B.print();
    }
}

public class B {
    public static final Integer s1 = 3;
    public static final Integer a2 = A.s2;

    public static void print() {
        System.out.println("B.a2 = " + B.a2);
    }
}
```

Figure 9.2: Interleaving static initialization in multiple classes

3. The initializer for `theInstance` creates an instance of `C`, so the JVM now initializes the new object's instance fields, as per Java's semantics. Specifically, the program initializes `b` to the value of `FLAG`, which is still *null*.

4. Now that the initialization of `theInstance` is complete, the JVM continues initializing the rest of `C`'s static fields, starting with `FLAG`.

5. Once the class `C` is completely loaded, the JVM begins to execute `C.main`'s body. At this point, `theInstance.b`, of type `Boolean`, has the value *null*. Because it appears in a context that expects a (primitive) `boolean`, the program attempts to coerce *null* to a `boolean` value and raises a null-pointer exception.

MIJAVA's constructors alone are not sufficient to avoid this problem. Had we given `C` an MIJAVA-style constructor, the program would still have crashed at the same point. The problem arises from the interleaving of static and instance initialization.

We can also cause these symptoms by interleaving the static initialization of two mutually recursive classes. See figure 9.2 for an example. Running this program produces the output `B.a2 = null` because the program causes the JVM to load class `B` during the initialization of `A.b1`. `B`'s initialization refers to `A.s2`, which has not yet been initialized, so the final print statement indicates that `B.a2` is *null*. Since the original program does not instantiate either `A` or `B`, MIJAVA-style constructors do not help here.

Arbitrarily complex variations on this theme are possible, including two classes that create an initialization cycle: static field `D.f` is initialized to the value of static field `E.f`, which is initialized to the value of `D.f`, and so on. We leave a solution to this problem for future research.

9.2 Advice for Language Designers

While we have concentrated in this dissertation on addressing shortcomings in the design of Java, we believe that some of the design decisions and principles that we applied to MIJAVA are relevant to the design of programming languages in general. In particular, MIJAVA's ideas about object initialization, optimization for safety (i.e., non-*null*-ness), and disjoint unions can inform other language design efforts.

Object Initialization

We introduce the discussion of MIJAVA's initialization mechanism in Chapter 2 by demonstrating how Java programs suffer from the ability to refer to fields before initializing them. This ability arises, as we have argued, because programmers can interleave the computation that produces initial values for the fields of an object with general computation on that object. Other languages suffer from equivalent problems, as our survey in the previous chapter shows. The only languages that avoid these problems are those that enforce a rigid separation between these two kinds of computation, like Fortress, OCaml, and MOBY. This suggests that object-oriented language designers should consider a "two-phase initialization" similar to MIJAVA's in order to avoid these problems.[3]

Non-Null by Default

As described above, MIJAVA assumes that references are non-*null* by default and requires the programmer to write `Maybe` types for those which can be `none`. Languages like Spec# and Cyclone, on the other hand, that support non-*null* types require the programmer to mark those types that do not admit *null*. As a result of this design choice, the potentially unsafe case is explicitly marked in MIJAVA, just as it is in Fortress, OCaml, and the other OO languages based on ML and Haskell surveyed in the previous chapter. We hope that the syntactic cost of `Maybe` and the implied run-time overhead encourage programmers to use the simpler, safer types wherever possible.

This use of syntax to encourage safety is a specific instance of a general phenomenon in language design. Choices about the type system, concrete syntax, or inter-

[3] Similar problems plague even functional languages that include mutually-recursive bindings combined with assignment-based initialization, as in Scheme's `letrec` form [49].

face to the standard library can dramatically influence the kind of programs written in the language by facilitating certain paradigms and making others much harder. Other languages have used similar techniques to encourage their users to follow a specific program design philosophy, such as OCaml's assumption that fields are immutable unless explicitly marked otherwise, or Haskell's prohibition against any sort of effects unless the programmer uses a monad (and the accompanying complexity and syntactic overhead). Generalizing MIJAVA's choice to use `Maybe` types, then, we recommend that language designers optimize for the *safe* case, while continuing to make riskier alternatives possible.

Disjoint Unions

Disjoint unions are a central idea in programming, and the specific disjoint union represented by *Maybe* is a particularly important instance of this general idea. Following the argument of the previous section, it is therefore crucial that languages make it possible and easy for programmers to define and use disjoint unions like *Maybe*. While Java does allow programmers to define disjoint unions by using the technique of section 3.3 that involves an abstract base class and one subclass for each branch of the union, it requires a significant amount of overhead. As a result, therefore, programmers prefer the short-term convenience of using the `null`-based representation of *Maybe* instead of the long-term benefits of the explicit union, with the practical engineering drawbacks that we describe in Chapter 3.

9.3 Conclusion

Although several open problems and opportunities for future research remain, our work has demonstrated that, by adding an explicit *Maybe* type and a new constructor mechanism to Java, we can almost completely remove *null* from the language. With these changes, programmers can realize significant engineering benefits, including early error detection, better diagnostics, and simpler interfaces. Further, our translation from MIJAVA to Java not only demonstrates the practical feasibility of the proposal but also allows programmers to gradually adapt existing Java programs to MIJAVA so they can advantage of MIJAVA's features with minimal effort.

Bibliography

[1] Eric Allen, David Chase, et al. The Fortress language specification, version 1.0. Technical report, Sun Microsystems, 2008. http://projectfortress.sun.com/.

[2] Apple Computer, Inc. The Objective-C 2.0 programming language. http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/index.html, 2008.

[3] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, Amsterdam, 1984.

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop CASSIS 2004*, volume 3662 of *Springer Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.

[5] G. M. Bierman, M. J. Parkinson, and A. M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report UCAM-CL-TR-563, Cambridge University, April 2003.

[6] Gavin Bierman, Erik Meijer, and Wolfram Schute. The essence of data access in Cω. In *ECOOP '05—Object Oriented Programming: 19th European Conference*, volume 3586 of *Springer Lecture Notes in Computer Science*, pages 287–311. Springer-Verlag, 2005.

[7] Joshua Bloch and William Pugh. Java puzzlers, episode VI: The phantom-reference menace/attack of the clone/revenge of the shift. Talk TS-2707 at the 2007 JavaOne Conference. Slides available at http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-2707.pdf.

[8] Bryan Chadwick and Karl Lieberherr. Functional adaptive programming. Technical Report NU-CCIS-08-75, Northeastern University College of Computer and Information Science, October 2008.

[9] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: constraining parametric polymorphism. In *OOPSLA '95: Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 156–168, New York, NY, USA, 1995. ACM.

[10] ECMA. Standard ECMA-367: Eiffel analysis, design and programming language, second edition. Available from `http://www.ecma-international.org/publications/standards/Ecma-367.htm`, June 2006.

[11] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 6(9):455–475, October 2007. `http://www.jot.fm/issues/issues_2007_10/paper23`.

[12] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[13] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *OOPSLA '07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications*, pages 337–350, New York, NY, USA, 2007. ACM.

[14] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, volume 1523 of *Springer Lecture Notes in Computer Science*, pages 241–269. Springer-Verlag, 1999.

[15] Matthew Flatt and PLT Scheme. Reference: PLT Scheme. Reference Manual PLT-TR2008-reference-v4.1, PLT Scheme Inc., August 2008. `http://plt-scheme.org/techreports/`.

[16] Martin Fowler and David Rice. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Reading, MA, 2003.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[18] Adele Goldberg and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java(TM) Language Specification*. Addison-Wesley, second edition, 2000.

[20] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java(TM) Language Specification*. Addison-Wesley, third edition, 2005.

[21] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. Cyclone: A type-safe dialect of C. *The C/C++ User's Journal*, 23(1), January 2005.

[22] Nevin Charles Heintze. *Set based program analysis*. PhD thesis, Carnegie Mellon University, 1992.

[23] David Hovemeyer, Jaime Spacco, and William Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, pages 13–19, New York, NY, USA, 2005. ACM Press.

[24] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[25] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A type-safe dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*. The USENIX Association, 2002.

[26] Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. Draft; Submitted for journal publication; online since 30 Sep. 2004; Full version released 10 September 2005, 2005.

[27] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml System: release 3.10*, May 2007. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

[28] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification*. Prentice Hall, second edition, 1999.

[29] Barbara Liskov, Dorothy Curtis, Mark Day, et al. Theta reference manual: Preliminary version. Accessed on May 12, 2008, at `http://www.pmg.lcs.mit.edu/Theta.html`.

[30] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Reading, MA, 1993.

[31] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, New York, 1992.

[32] Microsoft. C# language specification. Available on the web as `http://msdn.microsoft.com/en-us/library/ms228593.aspx` (as of October 2008).

[33] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, 1997.

[34] Johan Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 1999.

[35] Martin Odersky. The Scala language specification, version 2.7. Available from `http;//www.scala-lang.org/`, 2008.

[36] Scott Owens. A sound semantics for OCaml$_{light}$. In Sophia Drossopoulou, editor, *Programming Languages and Systems: 17th European Symposium on Programming ESOP 2008*, volume 4960 of *Springer Lecture Notes in Computer Science*, pages 1–15, 2008.

[37] Jens Palsberg. Efficient inference of object types. *Information & Computation*, 123(2):198–209, 1995.

[38] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *OOPSLA '91: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161, New York, NY, USA, 1991. ACM Press.

[39] Jens Palsberg and Michael I. Schwartzbach. Static typing for object-oriented programming. *Science of Computer Programming*, 23(1):19–53, 1994.

[40] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, Cambridge, U.K., 2003.

[41] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[42] The Moby Project. Report on the Moby programming language, version 0.9.18. Available from `http://moby.cs.uchicago.edu/`, 2003.

[43] Xin Qi and Andrew C. Meyers. Masked types for sound object initialization. In *POPL '09: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Princples of Programming Languages*, 2009. To appear.

[44] Didier Rémy. Programming objects with ML-ART: An extension to ML with abstract and record types. In *Proceedings of the Second International Symposium on Theoretical Aspects of Computer Software*, volume 789 of *Springer Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[45] Didier Rémy and Jérôme Vouillon. Objective ML: an effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[46] Sunae Seo, Youil Kim, Hyun-Goo Kang, and Taisook Han. A static bug detector for uninitialized field reference in Java programs. Technical Report CS-TR-2006-264, Division of Computer Science, Korea Advanced Institute of Science and Technology, December 2006.

[47] Sunae Seo, Youil Kim, Hyun-Goo Kang, and Taisook Han. A static bug detector for uninitialized field references in Java programs. *IEICE Transactions on Information and Systems*, E90-D(10):1663–1671, October 2007.

[48] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[49] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, et al. Revised^6 report on the algorithmic language Scheme. `http://www.r6rs.org/`, 2007.

[50] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, January 1986.

[51] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, special edition, 2000.

[52] Sun Microsystems. Java 2 platform, standard edition, v1.4.2 API specification. `http://java.sun.com/j2se/1.4.2/docs/api/`, 2003.

[53] Sun Microsystems. Java platform, standard edition 6, API specification. `http://java.sun.com/javase/6/docs/api/`, 2006.

[54] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *ECOOP 2001—Object Oriented Programming: 15th European Conference*, volume 2072 of *Springer Lecture Notes in Computer Science*, pages 99–117, Berlin, 2001. Springer-Verlag.

[55] Alessandro Warth. LazyJ: Seamless lazy evaluation in Java. In *Proceedings of the 2007 International Workshop on Foundations and Developments of Object Oriented Languages (FOOL/WOOD '07)*, 2007.

[56] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.