Abstract

A Type-Preserving Compiler Infrastructure

Christopher Adam League

Many kinds of networked devices receive and execute new programs from various sources. Since we may not fully trust the producers of these programs, we must take measures to ensure that such code does not misbehave. Currently deployed mobile code formats can be checked for memory safety and other security properties, but they are relatively high-level. A *type-preserving compiler* generates lower-level, more optimized code that is still verifiable. This increases assurance by reducing the trusted computing base; we need not trust the compiler anymore. Moreover, lower-level representations naturally support a wider variety of source languages.

Previous research on type-preserving compilation focused on functional languages or safe subsets of C. How to adapt this technology to more widely-used object-oriented languages was unknown. This dissertation explores techniques that enable a single strongly-typed intermediate language to certify programs in two very different programming languages: Java and ML.

The major contribution is an efficient new encoding of object-oriented constructs into a typed intermediate language. I give a complete formal translation of a Java-like source calculus into a typed lambda calculus. I prove that both languages are sound and decidable, and that the translation preserves types.

I also address many practical concerns, moving beyond the formal model to include most features of the Java language. To stage the translation, I developed lambda JVM, a novel representation of Java bytecode that is simpler to verify. I describe a prototype compiler that supports both Java and ML, sharing the same typed intermediate language, optimizers, code generator, and runtime system.

A Type-Preserving Compiler Infrastructure

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

By
Christopher Adam League
Dissertation Director: Zhong Shao

December 2002

©2003 Christopher Adam League. All Rights Reserved.

Contents

Contents i

List of Figures iv

Acknowledgments vii

1 Compiler Support for Safe Systems 1

1.1 Safety mechanisms 2

1.2 Type-preserving compilers 5

1.3 Contributions 7

1.4 Structure of this dissertation 8

2 Object Encoding 9

2.1 Why we need encodings 10

2.2 Encodings must enforce safety 11

2.3 Classic encodings 13

2.4 Efficient encoding 15

2.5 Another approach 16

3 Source Language: Featherweight Java 21

3.1 Syntax 22

3.2 Semantics 24

4 Intermediate Language: Mini JFlint 31
4.1 Syntax . . . 32
4.2 Semantics . . . 40
4.3 Properties . . . 44

5 A Type-Preserving Translation 53
5.1 Self application . . . 53
5.2 Type translation . . . 58
5.3 Expression translation . . . 62
5.4 Class encoding . . . 66
5.5 Class translation . . . 69
5.6 Linking . . . 72
5.7 Separate compilation . . . 73
5.8 Properties . . . 74
5.9 Related work . . . 83

6 Beyond Featherweight: the rest of Java 87
6.1 Private fields . . . 89
6.2 Interfaces . . . 91

7 Functional Java Bytecode 93
7.1 Design . . . 94
7.2 Translation . . . 97
7.3 Verification . . . 103
7.4 Implementation . . . 106
7.5 Related work . . . 107

8 A Prototype Compiler for Java and ML 109
8.1 Design . . . 109

8.2 Synergy 115
8.3 Implementation 116

9 Future Directions **119**

9.1 More inclusive encodings 119
9.2 More substantial implementations 120

Bibliography **123**

List of Figures

1.1 Do you trust Microsoft? 3

1.2 Recent CERT advisories 5

2.1 Expanding method invocation to low-level code 11

2.2 Using an arbitrary integer as a pointer 12

2.3 Predicate constructors in Special J 17

3.1 Abstract syntax of Featherweight Java 22

3.2 Sample FJ program with integers, arithmetic, and conditional expressions . . 23

3.3 Auxiliary functions for field and method lookup 24

3.4 Definition of the subtyping relation 25

3.5 Computation rules 25

3.6 Congruence rules 26

3.7 Typing rules for expressions 27

3.8 Well-typed classes and methods 28

4.1 Abstract syntax of Mini JFlint 32

4.2 Derived forms (syntactic sugar) 32

4.3 Formation rules for kinds 34

4.4 Formation rules for environments 35

4.5 Type formation rules 36

4.6 Type formation rules, continued 37
4.7 Term formation rules 39
4.8 Term formation rules, continued 41
4.9 Type equivalence rules 42
4.10 Type equivalence rules, continued 43
4.11 Values and primitive reductions 45
4.12 Congruence rules 46

5.1 Casting q from Point to Object 57
5.2 Field and method layouts for object types 58
5.3 Definition of rows 60
5.4 Macros for object types 60
5.5 Rows for Point and ScaledPoint 62
5.6 Definitions of pack and upcast transformations 62
5.7 Type-directed translation of FJ expressions 63
5.8 Macros for dictionary, constructor, and class types 67
5.9 Translation of class declarations 70
5.10 Translation of method declarations 71
5.11 Program translation and linking 72

7.1 Method syntax in λJVM 95
7.2 A sample Java method with a loop 97
7.3 The same method compiled to JVML 98
7.4 The same method translated to λJVM 98
7.5 A complex example with subroutines 101
7.6 Translation involving subroutines 101
7.7 Selected typing rules 105
7.8 The need for set types 106

8.1 Importing FLINT code from the SML/NJ static environment 110
8.2 Signature for JFlint code 111
8.3 Constructing representations of object types 112
8.4 Compiling invokevirtual 112
8.5 Compiling and running a Java program in SML/NJ 113
8.6 A trivial Java program 113
8.7 Abstract interface for JFlint type representation 117

Acknowledgments

My work was funded in part by a Yale University fellowship, NSF grants CCR-9901011 and CCR-0081590 and a DARPA OASIS grant F30602-99-1-0519.

My committee—Arvind Krishnamurthy, Carsten Schürmann, and Kim Bruce—offered many insightful remarks and asked plenty of tough questions. This work is stronger thanks to their commitment. Thanks also to many anonymous referees for comments on all my papers, accepted or not. Andrew Appel's enthusiasm for my work has been reassuring. Phil Wadler first suggested formulating our ideas in the context of FJ. Amy Zwarico introduced me to programming languages research…a decade ago!

Two students, John Garvin and Daniel Dormont assisted with the implementation, especially by finding and fixing my bugs. Dachuan Yu defined the operational semantics for Mini JFlint and supplied many details for the soundness proofs. Stefan Monnier participated in countless hours of worthy discussion and welcome distraction. Valery Trifonov contributed considerable insight and industry, without which this work would not have succeeded.

When I started graduate school, I dared not dream of finding an advisor half as capable, energetic, and dedicated as Zhong Shao. His willingness to work long hours at the white board with us was remarkable. His insistence on finding ever simpler solutions was critical. He knew when I needed to be pushed. And he knew, somehow, when I was ready to work on my own. It is thanks to his grand vision and tireless recruiting that Yale was such a stimulating place to work.

On a personal note, thanks to Adam, Antony, Bill, Manish, Marg, Mike, Patrick, Rupa, Shamez, Sydney, Tara, Walid, *et alii* for your friendship. To Neeraja, for helping me prove that *the good life* is not incompatible with graduate school. To older friends that I almost lost, I hope I can make it up to you. To my parents for your love and support, and for not asking when I would get a real job. And finally to Art, for countering my complaints with encouragement and emotional support. May we have a long and beautiful life together.

Chapter 1

Compiler Support for Safe Systems

Thesis: A strongly-typed compiler intermediate language can safely and efficiently accommodate very different programming languages. To underscore the significance of this statement, I will broaden my focus for a few pages to explain the motivation for work on type-preserving compilers.

We are entering a world where many kinds of networked devices receive and execute new programs from various sources. A handheld organizer (or even a cellular phone) loads code from a PC conduit, from an infrared link with another device, or from a wireless network. I can donate the idle time on my personal computer to a growing number of projects that use widely distributed computation to crack cryptographic protocols,[1] model protein folding,[2] or search for space aliens.[3] Scientists upload new programs to satellites or space exploration devices. My web browser downloads and runs Java™ applets to provide specialized user interfaces.

In all of these applications, we may not fully *trust* the producers of the programs we receive and run. We must therefore take measures to ensure their code does not misbehave, either intentionally or accidentally. Foreign code should not crash or hang the

[1] `http://www.distributed.net/rc5/`
[2] `http://folding.stanford.edu/`
[3] `http://setiathome.ssl.berkeley.edu/`

device, exhaust precious resources, or interfere with other programs or data. Imagine 'SETI @home' uploading a portion of my email archive each time it exchanges data with its server, or a pointer error in a scientist's program causing the computer on the Mars Explorer to freeze. In space, no one can press RESET.

These examples are about preventing programs from doing *bad* things. Ideally, we might also want to ensure that programs we run do the *right* thing—that they compute what we *intend.* Verifying the correctness of some small procedures is possible, but the technique does not scale to realistic programs. We do not usually *know* what we intend for a program to compute. Indeed, I sometimes run programs just to discover what they do! Even when our intentions are clear, specifying them formally is difficult and prone to error—just like programming. For these reasons, I focus just on *safety,* and ignore correctness.

1.1 Safety mechanisms

Currently deployed tools that attempt to address the safety of foreign code include digital signatures and reference monitors. A *digital signature* identifies and authenticates the sender of a message using public-key cryptography. Perhaps you have seen a dialog similar to the one in figure 1.1 on the facing page. In this case, Microsoft digitally signed the code for their *Internet Explorer Service Pack.* The dialog assures me (the user) that the code I am about to install did indeed come from Microsoft, and not from some unknown third party. (Although I should perhaps be suspicious that the signature's authenticity is verified by the Microsoft certification authority.)

Digital signatures can enforce trust relationships between real-world entities, but they do not directly address *safety.* I must still trust Microsoft's assertion that the code is safe. In effect, digital signatures ensure only that I know who to *blame* when something goes wrong; this is why the next dialog is typically a license agreement!

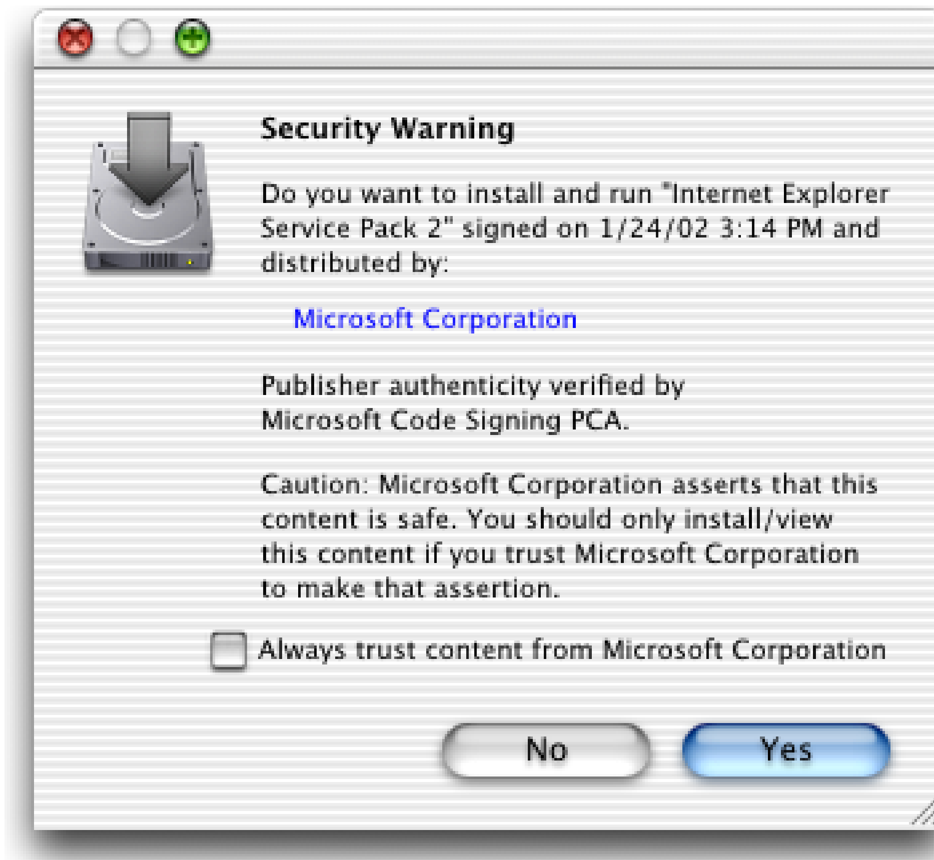

Figure 1.1: Do you trust Microsoft?

Another general tool is a *reference monitor.* Untrusted code is confined to a *sand box,* wherein it can do whatever it pleases. The boundaries of the sand box are enforced by the monitor—it can deny, regulate, or mediate untrusted code's access to the outside world. On modern workstations, the memory management hardware supports the operating system's reference monitor. A typical OS distinguishes between a privileged *kernel* mode and an untrusted *user* mode. User programs run in a sand box, and must perform a *context switch* into the kernel to access the machine's resources.

Because context switches are relatively expensive (compared to a normal function call), reference monitors typically implement fairly coarse-grained controls. Using the technology to, for example, enforce encapsulation or access control between the mod-

ules within one program would probably be overkill. Moreover, some very small devices (watches or cellular phones, for example) may not have the requisite hardware support. Implementing a reference monitor via software rewriting is possible, but this yields even higher overhead since dynamic checks must guard every load and store.

In recent years, programming language support for safe systems is finally drawing much-deserved attention. *Array bounds checking,* for example, ensures that programs do not (accidentally or intentionally) use an array pointer to access arbitrary memory. Similarly, *garbage collection* eliminates a large class of unsafe memory management errors. Languages that enforce *encapsulation* ensure that a rogue module cannot arbitrarily corrupt the private data of another. *Exceptions* encourage safe programming because uncaught exceptions rise immediately to the top level and kill the program. The default behavior, in other words, is to stop on failure. In a language without exceptions, the default behavior is usually to ignore the error code and continue merrily onward.

The most fundamental language feature for ensuring safety is a strong *type system.* I must assure readers accustomed only to the C language that there is much more to modern type systems than distinguishing between `int` and `float`. A type system, just like a formal logic, can encode complex properties that, for example, enforce abstract data types or access control. Wallach, Appel, and Felten (2000) showed that the Java stack inspection mechanism—used upon accessing a privileged resource to ensure the proper chain of authorization—could be formulated within a type system.

The term *type safety* refers to a collection of basic properties that are somewhat stronger than the *memory safety* implemented by memory management hardware. Type safety precludes segmentation faults and stack trashing, but also ensures that program modules respect their interfaces. Critically, a type-safe program cannot corrupt its underlying runtime system. A type system is *sound* if static checking admits type-safe programs only.

Type safety is not just *necessary* for a secure system, but already represents sig-

Date	**System**	**Vulnerability**
1/24/02	AOL ICQ	remotely exploitable buffer overflow
1/14/02	Solaris CDE	buffer overflow vulnerability
12/20/01	MS u-PNP	buffer overflow vulnerability
12/12/01	SysV 'login'	remotely exploitable buffer overflow
11/29/01	WU ftpd	format string vulnerability; `free()` on unallocated pointer
11/21/01	HP-UX lpd	remotely exploitable buffer overflow
10/25/01	Oracle9i AS	remotely exploitable buffer overflow
10/05/01	CDE ToolTalk	format string vulnerability

Figure 1.2: Recent CERT advisories

nificant progress from the status quo. Figure 1.2 shows a sample of recent security advisories from the Computer Emergency Response Team (CERT). A great many of the reported vulnerabilities are buffer overflows, memory management problems, or format string bugs (referring to C's unsafe `printf` routine). All such vulnerabilities could be prevented by using type-safe languages.

1.2 Type-preserving compilers

To ensure safe execution of untrusted code, is it sufficient, then, to program in type-safe languages? Although such a revolution would enormously improve the status quo, I must answer *no.* First, companies like Microsoft are unlikely to ship the source code for users to type-check and compile. Even if vendors distribute something *close enough* to source code that type safety can still be verified—Java bytecode, for example—we must still *trust the compiler.* As a large and complicated program, a compiler can contain serious bugs—or even Trojan horses (Thompson 1984)—that compromise safety. Handing carefully type-checked code to an untrusted compiler is still a serious risk.

The "orange book" (Department of Defense 1985) identifies the *trusted computing base* (TCB) as the elements of a system responsible for supporting the security policy. Identifying and minimizing code in the TCB increases assurance in the entire system. Even when using a type-safe programming language, the compiler is part of the TCB.

An exciting new line of research aims to enable higher assurance systems with a *minimal* TCB (Appel 2001). Raw machine code, annotated with the right types and invariants, can be verified as type-safe by an extremely small verifier. Even the type system for the machine code can be proved sound in some machine-checkable logic.

An essential component of this system is a *type-preserving* or *certifying* compiler. Rather than discard the source language types, such a compiler transforms them along with the program into a strongly-typed intermediate language, and finally into a typed assembly language. Such a compiler *need not be trusted* since, along with the object code, it generates evidence that the code is safe. Type preservation is challenging, both in theory and in practice. Lower-level code always needs more sophisticated types to justify its safety. Unfortunately, such type systems are difficult to implement, and can—if we are not vigilant—hinder efficient execution.

Until now, research in this area has concentrated on compiling either functional languages or a safe subset of C. Tarditi et al. (1996) introduced TIL, a compiler for the polymorphically-typed functional language Standard ML (Milner et al. 1997). Shao (1997) implemented FLINT, a typed intermediate language for the Standard ML of New Jersey compiler (Appel and MacQueen 1991). Necula and Lee (1998) pioneered the idea of *proof-carrying code* and built a certifying compiler for a safe subset of C. Finally, Morrisett et al. (1999a) created two distinct type-preserving compilers: one for Popcorn, a safe C-like language, and another (written in Popcorn) for a subset of Scheme (Clinger and Rees 1991), the dynamically-typed functional language.

If this technology is to succeed in the *real world,* certifying compilers must support real-world programming languages and development models.

1.3 Contributions

This work shows that, with carefully designed encodings, a single typed intermediate language can safely and efficiently accommodate not only a typical functional language (Standard ML) but a typical object-oriented language (Java) as well.

Specifically, we developed a formal translation of Featherweight Java (Igarashi, Pierce, and Wadler 2001) into a typed λ-calculus. At run time, method calls have precisely the same operational behavior as a standard untyped implementation. Classes inherit and override methods from super classes with no overhead. We support mutually recursive classes while maintaining separate compilation. Dynamic casts are implemented as polymorphic methods using tags generated at link-time. The target of this translation (Mini JFlint) is sound and decidable, but really quite conventional—rooted in decades of type theory research. It is a minor extension of the kind of calculus typically used to represent functional languages in compilers (Peyton Jones et al. 1992; Shao and Appel 1995; Morrisett et al. 1996).

This work includes a formal proof that well-formed Featherweight Java programs map to well-formed Mini JFlint programs. Since we also proved that Mini JFlint is sound, translated programs do not become stuck at run time.

We supplement these significant theoretical results with work to address practical concerns. We developed informal extensions to support many Java features that are not included in Featherweight: interfaces, constructors, super calls, privacy, and exceptions, for example. Our prototype compiler is based on Standard ML of New Jersey (Appel and MacQueen 1991). It reads Java *class files* (bytecode for the Java virtual machine) and compiles them to low level JFlint code, with type information preserved. To stage the translation from class files to JFlint, we designed λJVM, a novel representation of Java bytecode that is more explicit and simpler to verify. After JFlint, the compiler calls MLRISC (George 1997) to generate machine code for a variety of architectures.

The ML and Java front ends share optimizations and back ends. Programs from either language run together in the same interactive runtime system with the same garbage collector. The design of JFlint itself supports a pleasing synergy between the encodings of Java and ML. JFlint does not, for example, treat Java classes or ML modules as primitives. Rather, it provides a low-level abstract machine model and sophisticated types that are general enough to prove the safety of a variety of implementations.

1.4 Structure of this dissertation

This chapter sketched the motivation for the work and outlined our contributions. The next chapter reviews the idea of *object encoding,* explaining why type-safe encodings are so important and challenging. Chapters 3 through 5 are the core theory, expanded from our journal article, “Type-Preserving Compilation of Featherweight Java” (League, Shao, and Trifonov 2002b). We formulate models of the source and intermediate languages, give a formal translation between them, and prove several important properties.

The remaining chapters supplement the theory with discussions of various practical issues. Chapter 6 addresses some of the Java features that are not covered in the formal presentation. Chapter 7 is an expansion of a paper called “Functional Java Bytecode” (League, Trifonov, and Shao 2001a) describing the λJVM intermediate language and the niche it was designed to occupy. Chapter 8 summarizes the implementation of the prototype compiler. Finally, chapter 9 concludes with some exciting ideas for future research.

Comparisons to previous work are made throughout the dissertation. Other object encodings are discussed in sections 2.3 and 5.9. Alternative representations of Java bytecode are mentioned in 7.5. Certified implementations of object-oriented languages are covered in sections 2.5 and 8.2.

Chapter 2

Object Encoding

Booch (1994, page 38) gives the following definition of object-oriented programming:

> [It] is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

His book describes techniques for analyzing requirements and designing software using classes of objects as the unifying model.

I do not make any particular claims about the merits of object-oriented programming, analysis, or design. It is clear, however, that objects represent a different way of structuring software, as compared to functional or procedural programming. Furthermore, there is no denying that object-oriented technology remains popular, from the new safe languages Java (Gosling et al. 2000) and C# (Liberty 2002) to scripting languages Python (Lutz 2001) and Ruby (Thomas and Hunt 2000) to that old standby, C++ (Stroustrup 1997). Therefore, for certifying compiler technology to be viable, we must find an efficient way to support object-oriented programming languages.

This chapter is about the art and science of *object encoding*—representing object-oriented features in languages (or models of computation) that lack them. I will keep

the formal semantics and type theory to a minimum (there is plenty of space for that in subsequent chapters), so that any student of computer science can appreciate the significance of this work.

2.1 Why we need encodings

The von Neumann architecture—the model on which virtually all electronic computers are based—has no notion of methods, objects, classes, or inheritance. To implement these features, we must express them using simple instructions that load and store words in a sequential memory. This is not surprising; such is the job of a compiler for *any* high-level programming language. Object-oriented features seem particularly high-level because they naturally decompose into operations on records and functions—the abstractions of procedural languages.

Consider the operation to invoke some *method* of an object. The definition that opened this chapter emphasized classes and inheritance, but one other feature is widely considered essential for object-oriented programming: *dynamic* (or *late*) *binding* of methods. Booch (1994, page 116) distinguishes it this way:

> Inheritance without polymorphism is possible, but it is certainly not very useful. This is the situation in Ada, in which one can declare derived types, but because the language is monomorphic, the actual operation being called is always known at the time of compilation.

With dynamic binding, the operation is *not* known at compile time. Rather, the intuition is that we send a *message* to the object to request some operation, but the object itself chooses (usually by virtue of the class that created it) which *method* gets invoked.

Normally, the object includes some data structure for mapping messages to function pointers. In the case of single inheritance class-based languages (such as Java and C#), this data structure is just a simple record, traditionally called the *virtual function table,*

```
public static void example (Object x, Object y)
{   x.toString ( )
    // virtual method call expands to:
    if (x == null) throw NullPointerException;
    r1 = x.vtbl;
    r2 = r1.toString;
    call r2(x);
}
```

Figure 2.1: Expanding method invocation to low-level code

or *vtable* for short. The typical syntax (after C++) even suggests that method invocation is some combination a record access and a function call: `obj.meth (args)`.

Indeed, methods are just standard functions with an implicit *self* parameter (called **this** in Java) referring to the current object. All instances of the same class share the same vtable. Subclasses append new methods and fields to the vtable and object layout, but do not rearrange the members of super classes. This way, we always know where to find a field in an object, even if the object was created by some unknown subclass.

To invoke a virtual method, we simply load the vtable pointer from the object, load the function pointer from the vtable, and then call the function, providing the object itself as the now-explicit *self* argument. Figure 2.1 shows Java method invocation expanded into lower-level code. The identifiers r_1 and r_2 denote registers.

2.2 Encodings must enforce safety

A certifying compiler must justify that the indirect call to r_2 is safe; this is not at all obvious. If x is an instance of a *subclass* of Object, then the method in r_2 might require of x additional fields and methods that are unknown to the caller. Self-application works thanks to a rather subtle invariant. One way to upset that invariant is to select a method from one object and pass it *another* object as the self argument. For example, replace just the last instruction above with **call** $r_2(y)$.

```
class Ref extends Object
{   public byte[] vec;
    public String toString ( )
    {   vec[13] = 42;
        return "Ha ha!";
    }
}

class Int extends Object
{   public int n;
}

public static void deviant (Object x, Object y)
{   // as low-level code
    if (x == null) throw NullPointerException;
    r1 = x.vtbl;      // fetch method from x
    r2 = r1.toString;
    call r2(y);       // pass y as self argument
}

deviant(new Ref (...), new Int (...));
```

Figure 2.2: Using an arbitrary integer as a pointer

This might seem harmless; after all, both x and y are instances of Object. It is unsound, however, and any unsoundness can be exploited. Figure 2.2 contains a complete example that exploits such code to use an arbitrary integer as the address of a byte vector. Once we can do that, all bets are off.

Class Ref extends Object with a byte vector and overrides toString to write to the byte vector before returning an innocuous string. Class Int extends Object with just one integer field. Importantly, in the representations of Ref and Int objects, the byte vector and the integer occupy the same slot.

The deviant method uses low-level code to fetch the toString method from x, and pass it y as the self argument **this**. Finally, the main program calls deviant with some Ref object as x and some Int as y. What happens? The call in deviant will jump to Ref.toString with **this** bound to the Int object. That method attempts to write to the

byte vector, but finds an integer there instead. This is not type safe!

Of course, it is unlikely that a legitimate compiler would ever generate such code—but that is no consolation. Remember, we do not trust the producer; perhaps he is writing malicious code in assembly language. If so, we must detect it.

2.3 Classic encodings

How can a type system ensure that low-level operations properly encode objects, so that erroneous or malicious code is detected? There is significant precedent for modeling objects in typed λ-calculi (Barendregt 1992). Bruce, Cardelli, and Pierce (1999) summarize several such results in a uniform framework.

We briefly demonstrate how one of these encodings properly rejects the code in figure 2.2. Pierce and Turner (1994) represent objects in $F_\omega^\leq$ (a higher-order typed λ-calculus with subtyping) using an *existential type* to hide the private representations of objects. That is, instances of class Object have type

$$\exists s{::}\mathsf{Type}.\ \{\mathsf{state}: s,\ \mathsf{vtbl}: \{\mathsf{toString}: s \to \mathsf{String}\}\}$$

where s is a *type variable* that masks the types of fields. The underlying representation is a pair containing the values of the fields (i.e., the state of the object) and the method table (vtbl). Each method expects to receive the object state (not the whole object) as its first argument.

The method invocation sequence for this encoding starts with an **open** instruction to eliminate the existential type:

$\langle s_1, r_1 \rangle$ = **open** x;

$r_2 = r_1$.vtbl.toString;

call $r_2(r_1$.state);

The **open** introduces a fresh type variable s_1 to represent the abstract type locally. Then, r_2 gets a function of type $s_1 \to \mathsf{String}$, and r_1.state is a value of type s_1, so the call is safe. What happens if, as in figure 2.2, we try to pass y's state to x's method? First, we must **open** y, yielding a record r_3 and a fresh type variable s_2.

> $\langle s_1, r_1 \rangle$ = **open** x;
>
> $\langle s_2, r_3 \rangle$ = **open** y;
>
> $r_2 = r_1$.vtbl.toString;
>
> **call** $r_2(r_3$.state$)$; // type error

Since r_3.state has type s_2, attempting to pass it to r_2 is a type error.

These object encodings detect low-level errors by embedding objects, classes, and methods in foundational calculi that are known to be sound. They are successful *models* of these features, and helpful for comparing the expressive power of object calculi with λ-calculi. They are not, unfortunately, directly applicable for *compiling* modern object-oriented languages. First, in the early object models (Abadi and Cardelli 1996) classes and method overriding were afterthoughts, not essential features as in Java. Fisher and Mitchell (1998) made the relationship clear by modeling classes as extensible objects, but that does not bring us any closer to a von Neumann machine.

Second, the classic encodings have various inefficiencies, making them unsuitable for use in compilation. For example, the aforementioned existential encoding of Pierce and Turner (1994) seems reasonably efficient until we examine inheritance (section 6 in their paper). To permit subclasses to extend the object representation, they introduce function arguments `get` and `put` to coerce between the final representation and the current representation at some level in the class hierarchy. Methods must call these coercion functions before reading or writing the internal representation. Moreover, the coercions grow with the depth of the hierarchy. Compiler analyses may alleviate some

of the penalty, but removing the coercions entirely would require analyzing the whole program or duplicating the inherited code in each subclass.

The recursive bounded existential of Abadi, Cardelli, and Viswanathan (1996) has a superfluous *self pointer* to dereference on method invocation. The simpler recursive record encodings are not suitable for Java: Cardelli (1988) does not handle dynamic binding; Fisher, Honsell, and Mitchell (1994) support dynamic binding, but selecting a method and substituting the self argument remains an atomic operation.

2.4 Efficient encoding

The fundamental contribution of our work is a simple, efficient encoding that is suitable for compilation of modern object-oriented languages such as Java. Unlike recursive record encodings, it supports dynamic binding with low-level primitives. Unlike Abadi, Cardelli, and Viswanathan (1996), it needs no extra pointers. Unlike Pierce and Turner (1994), methods can be reused in subclasses with no overhead. Moreover, the ambient type theory is quite simple; we do not need subtypes or bounded quantification.

The classic encodings all assumed that *subsumption* was necessary. This is what allows an instance of a subclass (Hexagon) to be substituted wherever a super class (Shape) is expected. This is certainly a desirable (and nearly universal) property in object-oriented languages at the source level. For a compiler intermediate language, it is acceptable to require explicit upward casts instead, as long as they cost nothing at runtime. Type manipulations (such as the **open** in previous examples) guide the type checker, but are erased before the code is run. Therefore, the implicit subsumption in previous models can be replaced with explicit type manipulations. The details and proofs about our encoding will be explained in the next three chapters.

Concurrently with our work, two other researchers developed encodings that seem to have similar properties. Glew (2000a) translates a class-based object calculus using a

special kind of existential quantifier. Crary (1999) encodes the object calculus of Abadi and Cardelli (1996) using an existential and an *intersection* type:

$$\exists\alpha\text{::Type. } \alpha \wedge \{\mathsf{vtbl}:\{\mathsf{toString}:\alpha\to\mathsf{String}\},\ldots\}$$

In words, an object is *both* abstract (having type α) and a record containing a vtable whose methods expect a self argument of type α. Chapter 5 ends with a detailed comparison of these three efficient encodings.

2.5 Another approach

Rather than *encode* object-oriented features in a lower-level type-safe language, some systems try to guarantee safety using the abstractions of the source language directly. Two years ago, Colby et al. (2000) of Cedilla Systems presented initial results about Special J, a certifying compiler for Java. They described the design, defined some of the predicates used in verification conditions, explained their approach to exceptional control flow, and gave some experimental results. Their running example included a loop, an array field, and an exception handler.

The Special J system uses predicate constructors to express proof obligations and properties about the object layout and class hierarchy. For reference, we reprint some of the key constructors from Colby et al. (2000) in figure 2.3 on the facing page. As a simple demonstration, consider the rules to determine whether a field access is safe. From the source program, we would derive $(\mathsf{ifield}\ C\ O\ T)$ for some class C, offset O, and type T. Suppose then that $(\mathsf{type}\ E\ (\mathsf{jinstof}\ C))$ for some E. With these and $(\mathsf{nonnull}\ E)$ as pre-conditions, a particular rule in the system derives $(\mathsf{type}\ (\mathsf{add}\ E\ O)\ (\mathsf{ptr}\ T))$. In words, adding offset O to the object reference yields a pointer to a field of type T. Next, using the axiom $(\mathsf{size}\ (\mathsf{ptr}\ _)\ 4)$, another rule concludes $(\mathsf{saferd4}\ (\mathsf{add}\ E\ O))$.

constructor	**meaning**
jint, jfloat, ...	Java primitive types
(jinstof C)	the type of an instance of class C (or some subclass)
(jvtbl C)	type of the virtual function table of class C
(jimplof C S)	type of a method from class C with signature S
(ptr T)	type of a pointer to a value of type T
(add E O)	denotes the addition of offset O to address E
(size T B)	a value of type T occupies B bytes
(type E T)	E denotes a value of type T
(jextends C D)	encodes the subclass relationship
(vmethod C O S)	class C's vtable contains a method with signature S at offset O
(ifield C O T)	instances of class C have a field of type T at offset O
(nonnull E)	E is not null
(saferd4 E)	it is safe to read four bytes beginning at address E.

Figure 2.3: Predicate constructors in Special J

Due to limited space, Colby et al. (2000) did not address virtual method calls in their paper. We contacted the authors to learn the details, particularly in the context of the malicious code we gave in figure 2.2 on page 12. From the signature of the deviant method, their certifier establish the argument types:

(type x (jinstof Object))   (type y (jinstof Object))

A rule in the system permits x and y to be treated also as read-only pointers to the vtable of Object (since the vtable is at offset zero). After the assignment to r_1,

(type r_1 (jvtbl Object))

From the class definition, we know the offset O_{ts} of the toString method:

(vmethod Object O_{ts} '( )String')

The signature '( )String' means that the method takes no additional arguments and

returns a String. After the assignment to r_2,

(type r_2 (jimplof Object '( )String'))

With this knowledge of the type of r_2, the certifier determines that a call to r_2 is safe as long as the self argument has type (jinstof Object). Since both x and y have this type, the malicious code in figure 2.2 is *accepted.* Necula (2001) confirmed that our example exposed an unsoundness in the inference rules of Special J. The validity of the code certification relies critically on the assumption that the inference rules are sound—they are part of the trusted computing base. Colby et al. (2000) did not prove a soundness theorem for their system.

Necula (2001) proposed a solution where the predicates jvtbl and jimplof mention the *identity* of the object from which they came. Then, the virtual call is safe only if the identity of the self argument matches that of the jimplof type. It is interesting to relate this strategy to the object encodings discussed earlier in this chapter. We believe it bears some resemblance to the encoding of Crary (1999). Since loading the virtual table from x introduces a type containing the identity of x, it is as if we implicitly opened a package to introduce some abstract type. The usual rules still permit fetching methods from x (whose types also mention x). In addition, the revised rule governing the call has us treat x as a value of the new abstract type. Similarly, the intersection type in Crary's encoding permits x to be treated both as abstract and as a table of methods (that expect x as the self argument).

Of course, resemblance to a foundational encoding does not imply soundness. A rigorous soundness proof for a real implementation like Special J is extremely difficult. A more reasonable approach—the one we take in this dissertation—is to formulate a model and prove soundness for a significant subset of the real system. Having no proof at all is dangerous.

We expect that a soundness proof for Special J—even for a subset—will be complex, for a couple of reasons. First, the Special J inference rules encode much of the semantics of Java, giving meaning to features like inheritance and virtual methods. Therefore, the soundness of Special J subsumes the soundness of Java itself. In contrast, the calculi used for object encoding—including Mini JFlint from chapter 4—are not at all Java-specific. Their soundness proofs are completely independent of the soundness of whatever object-oriented languages they support.

Another potential difficulty is the difference in granularity between the predicate constructors and the machine instructions they describe. Many predicates deal with high-level abstractions such as classes and virtual methods. The machine model, on the other hand, deals with code pointers and blocks of memory. The types of Mini JFlint, in contrast, encode the requisite invariants with precisely the granularity at which the code operates—that of functions and records.

Chapter 3

Source Language: Featherweight Java

Our goal is a type-preserving compiler that supports Java. To ensure that the techniques we use in this effort are sound, it is critical to study them in the context of a formal system. By working with an idealization of the actual compiler, we can develop precise semantics, perspicuous translations, and rigorous proofs of important properties. A compiler is nothing more than a translator from some source language into some target language. We must therefore formally specify these two languages; such is the aim of this and the next chapter.

The most important property we want to prove is that our compiler maps well-typed programs in the source language (Java) to well-typed programs in the target language (JFlint). To prove this, we need a formalization of Java with a notion of well-typed programs. Specifically, we need a calculus with a type system, an operational semantics, and a soundness proof.

Concurrently with the start of our work on this project, many researchers worked on provably type-safe idealizations of Java (Drossopoulou and Eisenbach 1999; Syme 1999; Qian 1999). The ClassicJava language by Flatt, Krishnamurthi, and Felleisen (1999) features classes, interfaces, fields with shadowing, dynamic method binding, abstract methods, object creation, casts, and local variables. Although our techniques

$$
\begin{aligned}
\mathsf{CL} &::= \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{C}'\ \{\ \mathsf{C}_1\ \mathsf{f}_1; \ldots \mathsf{C}_n\ \mathsf{f}_n;\ \mathsf{K}\ \ \mathsf{M}_1 \ldots \mathsf{M}_m\ \} \\
\mathsf{K} &::= \mathsf{C}\ (\mathsf{C}_1\ \mathsf{f}_1 \ldots \mathsf{C}_n\ \mathsf{f}_n)\ \{\ \textbf{super}(\mathsf{f}_1 \ldots \mathsf{f}_k);\ \textbf{this}.\mathsf{f}_{k+1} = \mathsf{f}_{k+1}; \ldots \textbf{this}.\mathsf{f}_n = \mathsf{f}_n;\ \} \\
\mathsf{M} &::= \mathsf{C}\ \mathsf{m}\ (\mathsf{C}_1\ \mathsf{x}_1 \ldots \mathsf{C}_n\ \mathsf{x}_n)\ \{\ \textbf{return}\ \mathsf{e};\ \} \\
\mathsf{e} &::= \mathsf{x} \mid \mathsf{e.f} \mid \mathsf{e.m}\ (\mathsf{e}_1, \ldots, \mathsf{e}_n) \mid \textbf{new}\ \mathsf{C}\ (\mathsf{e}_1, \ldots, \mathsf{e}_n) \mid (\mathsf{C})\ \mathsf{e}
\end{aligned}
$$

Figure 3.1: Abstract syntax of Featherweight Java

support all those features (League, Shao, and Trifonov 1999), the complex semantics of ClassicJava caused difficulty in proving essential properties of our translation.

Featherweight Java (FJ) is a particularly small calculus by Igarashi, Pierce, and Wadler (2001). Its semantics and soundness proof are easy to understand, yet it still features classes, inheritance, immutable fields, dynamic method binding, simple object creation, and dynamic casts. Most importantly, with FJ we can demonstrate the major techniques of our translation in a reasonably clean and comprehensible manner. The rest of this chapter formally defines FJ. We made a few minor adaptations to the typing rules; otherwise, the presentation follows very closely that of Igarashi, Pierce, and Wadler.

3.1 Syntax

Figure 3.1 contains a BNF grammar for the abstract syntax of Featherweight Java. Keywords are marked in **bold face.** Class declarations (CL) contain the names of the new class and its super class, a sequence of field declarations, a constructor (K), and a sequence of method declarations (M). We use letters A through E to range over class names, f and g to range over field names, m over method names, and x over formal parameter names. There are five forms of expressions: variable references, field selection, method invocation, object creation, and cast. A *program* (CT, e) consists of a fixed *class table* (CT) mapping class names to declarations, and a *main program expression* e.

There are no assignments, interfaces, **super** calls, exceptions, or access control in FJ. Constructors are greatly simplified: there can be only one, and it must take *all* the

```
class  Pt  extends Object
{   int  x;
    Pt ( int  x )         {super ();  this.x = x ; }
    int  getx ( )         {return  this.x ; }
    Pt   move(int  dx )   {return  new Pt( this.x + dx ); }
    Pt   bump( )          {return  this.move (1); }
    Pt   max (Pt  p )     {return  this.x > p.x? this  :  p ; }
 }

class  SPt  extends Pt
{   int  s;
    SPt ( int  x ,  int  s ) { super(x );  this.s = s ; }
    int  gets ( )         {return  this.s ; }
    Pt   move(int  dx )   {return  new SPt ( this.x + this.s * dx,
                                             this.s ); }
    SPt  zoom(int  s )    {return  new SPt ( this.x ,  this.s * s ); }
 }
```

Main program:

```
((SPt ) ( new Pt (2). max( new SPt (1,2). bump ( ) ))). zoom(3).gets ( )
```

Figure 3.2: Sample FJ program with integers, arithmetic, and conditional expressions

fields as arguments, in the same order that they are declared in the class hierarchy. FJ permits recursive class dependencies with the full generality of Java. A class can refer to the name and constructor of *any* other class, including its sub-classes. While this does not complicate the name-based FJ semantics, it is one of the major challenges of our translation.

Featherweight Java is Turing complete; it is easy to embed a λ-calculus in it. Nevertheless, for demonstration purposes, we will usually augment it with integers, arithmetic, and simple conditional expressions. Figure 3.2 contains a sample program that demonstrates many of FJ's features, including dynamic method binding and dynamic cast. We will revisit this example in chapter 5. Although Featherweight Java is very restricted, its syntax is precisely a subset of Java. That is, we can directly compile the classes of figure 3.2 with `javac`. By adding the main program expression to a static `main` method, we can run the program and verify that the result is 6.

$$\frac{}{\mathit{fields}(\mathsf{Object}) = \bullet} \tag{3.1}$$

$$\frac{\begin{array}{l} CT(\mathsf{C}) = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{B}\ \{\ \mathsf{C}_1\ \mathsf{f}_1; \ldots \mathsf{C}_n\ \mathsf{f}_n;\ \mathsf{K} \ldots\ \} \\ \mathit{fields}(\mathsf{B}) = \mathsf{B}_1\ \mathsf{g}_1 \ldots \mathsf{B}_m\ \mathsf{g}_m \end{array}}{\mathit{fields}(\mathsf{C}) = \mathsf{B}_1\ \mathsf{g}_1 \ldots \mathsf{B}_m\ \mathsf{g}_m, \mathsf{C}_1\ \mathsf{f}_1 \ldots \mathsf{C}_n\ \mathsf{f}_n} \tag{3.2}$$

$$\frac{\begin{array}{l} CT(\mathsf{C}) = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{B}\ \{\ \ldots \mathsf{K}\ \mathsf{M}_1 \ldots \mathsf{M}_n\ \} \\ \exists j : \mathsf{M}_j = \mathsf{D}\ \mathsf{m}\ (\mathsf{D}_1\ \mathsf{x}_1 \ldots \mathsf{D}_m\ \mathsf{x}_m)\ \{\ \textbf{return}\ \mathsf{e};\ \} \end{array}}{\begin{array}{l} \mathit{mtype}(\mathsf{m}, \mathsf{C}) = \mathsf{D}_1 \ldots \mathsf{D}_m \to \mathsf{D} \\ \mathit{mbody}(\mathsf{m}, \mathsf{C}) = (\mathsf{x}_1 \ldots \mathsf{x}_m, \mathsf{e}) \end{array}} \tag{3.3}$$

$$\frac{\begin{array}{l} CT(\mathsf{C}) = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{B}\ \{\ \ldots \mathsf{K}\ \mathsf{M}_1 \ldots \mathsf{M}_n\ \} \\ \mathsf{m}\ \text{not defined in}\ \mathsf{M}_1 \ldots \mathsf{M}_n \\ \mathit{mtype}(\mathsf{m}, \mathsf{B}) = \mathsf{D}_1 \ldots \mathsf{D}_m \to \mathsf{D} \\ \mathit{mbody}(\mathsf{m}, \mathsf{B}) = (\mathsf{x}_1 \ldots \mathsf{x}_m, \mathsf{e}) \end{array}}{\begin{array}{l} \mathit{mtype}(\mathsf{m}, \mathsf{C}) = \mathsf{D}_1 \ldots \mathsf{D}_m \to \mathsf{D} \\ \mathit{mbody}(\mathsf{m}, \mathsf{C}) = (\mathsf{x}_1 \ldots \mathsf{x}_m, \mathsf{e}) \end{array}} \tag{3.4}$$

Figure 3.3: Auxiliary functions for field and method lookup

3.2 Semantics

The semantics of Featherweight Java consists of a set of typing rules and a small-step computation relation. To express both of these cleanly, a few auxiliary definitions are in order. Figure 3.3 defines several relations that describe the inheritance of fields and the dynamic lookup of methods. *fields*(C) returns the sequence of all the fields found in objects of class C. The symbol '•' represents the empty sequence—class Object has no fields in FJ. Fields are assumed to have distinct names, so we need not worry about shadowing.

The relation *mtype*(m, C) finds the type signature for method m in class C by searching up the hierarchy. Type signatures have the form $\mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D}_0$. *mbody*(m, C) is the same, but returns the names of the formal parameters and the method body expression. These relations are defined inductively on the class hierarchy, but surprisingly,

$$\frac{}{\mathsf{C} <: \mathsf{C}} \tag{3.5}$$

$$\frac{CT(\mathsf{C}) = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{B}\ \{\ \ldots\ \} \qquad \mathsf{B} <: \mathsf{A}}{\mathsf{C} <: \mathsf{A}} \tag{3.6}$$

Figure 3.4: Definition of the subtyping relation

$$\frac{\mathit{fields}(\mathsf{C}) = \mathsf{D}_1\ \mathsf{f}_1 \ldots \mathsf{D}_n\ \mathsf{f}_n}{(\textbf{new}\ \mathsf{C}\ (\mathsf{e}_1 \ldots \mathsf{e}_n)).\mathsf{f}_i \longrightarrow \mathsf{e}_i} \tag{3.7}$$

$$\frac{\mathit{mbody}(\mathsf{m}, \mathsf{C}) = (\mathsf{x}_1 \ldots \mathsf{x}_n, \mathsf{e}_0)}{\begin{array}{l}(\textbf{new}\ \mathsf{C}\ (\mathsf{e}_1 \ldots \mathsf{e}_m)).\mathsf{m}\ (\mathsf{d}_1 \ldots \mathsf{d}_n) \longrightarrow \\ [\mathsf{d}_1/\mathsf{x}_1, \ldots, \mathsf{d}_n/\mathsf{x}_n, \textbf{new}\ \mathsf{C}\ (\mathsf{e}_1 \ldots \mathsf{e}_m)/\textbf{this}]\ \mathsf{e}_0\end{array}} \tag{3.8}$$

$$\frac{\mathsf{C} <: \mathsf{D}}{(\mathsf{D})\ \textbf{new}\ \mathsf{C}\ (\mathsf{e}_1 \ldots \mathsf{e}_n) \longrightarrow \textbf{new}\ \mathsf{C}\ (\mathsf{e}_1 \ldots \mathsf{e}_n)} \tag{3.9}$$

Figure 3.5: Computation rules

the base case is not Object. Rather, the base case is defined in rule (3.3) as the nearest super class containing a declaration of method m. If m is not a method of class C, the relations are undefined.

The subtype relation <: (figure 3.4) is the reflexive, transitive closure of the relation defined by the super class declarations (**class** C **extends** B). Igarashi, Pierce, and Wadler (2001) add an explicit rule for transitivity; this is unnecessary and would complicate some lemmas in chapter 5.

With these auxiliary definitions, the dynamic semantics of FJ is easily specified; '⟶' is a small-step reduction relation between expressions. Figure 3.5 contains the most important rules. Since fields are immutable, order of evaluation is unimportant and unspecified. Objects are represented simply as **new** expressions with their field initializers. The class name following the **new** keyword represents the dynamic class of the

$$\frac{\mathsf{e} \longrightarrow \mathsf{e}'}{\mathsf{e}.\mathsf{f}_i \longrightarrow \mathsf{e}'.\mathsf{f}_i} \tag{3.10}$$

$$\frac{\mathsf{e} \longrightarrow \mathsf{e}'}{\mathsf{e}.\mathsf{m}\ (\mathsf{d}_1 \ldots \mathsf{d}_n) \longrightarrow \mathsf{e}'.\mathsf{m}\ (\mathsf{d}_1 \ldots \mathsf{d}_n)} \tag{3.11}$$

$$\frac{\mathsf{e}_i \longrightarrow \mathsf{e}_i'}{\mathsf{e}.\mathsf{m}\ (\ldots \mathsf{e}_i \ldots) \longrightarrow \mathsf{e}.\mathsf{m}\ (\ldots \mathsf{e}_i' \ldots)} \tag{3.12}$$

$$\frac{\mathsf{e}_i \longrightarrow \mathsf{e}_i'}{\textbf{new}\ \mathsf{C}\ (\ldots \mathsf{e}_i \ldots) \longrightarrow \textbf{new}\ \mathsf{C}\ (\ldots \mathsf{e}_i' \ldots)} \tag{3.13}$$

$$\frac{\mathsf{e} \longrightarrow \mathsf{e}'}{(\mathsf{C})\ \mathsf{e} \longrightarrow (\mathsf{C})\ \mathsf{e}'} \tag{3.14}$$

Figure 3.6: Congruence rules

object—it does not change, even after a cast.

A field reference on an object (rule 3.7) reduces to the field initializer expression corresponding to the selected field. To invoke a method on an object (rule 3.8), we first find the method in the hierarchy, searching upward from the class (C) that created the object. Then, we substitute the actual parameters for the formal parameters and the object itself for **this** and continue executing the body of the method. Finally, a cast on an object succeeds (rule 3.9) if the dynamic class (C) of the object is a subclass of the requested class (D).

The congruence rules (figure 3.6) are necessary but uninteresting. They enable computation within expressions that are not themselves redexes. Variables are irreducible, so none of the reduction rules apply. The computation and congruence rules comprise the complete operational semantics of Featherweight Java.

The static semantics is mostly covered by the typing rules for expressions (figure 3.7 on the facing page). The judgment $\Gamma \vdash \mathsf{e} \in \mathsf{C}$ means that expression e has type C in the

$$\frac{}{\Gamma \vdash \mathsf{x} \in \Gamma(\mathsf{x})} \tag{3.15}$$

$$\frac{\Gamma \vdash \mathsf{e} \in \mathsf{C} \qquad \mathit{fields}(\mathsf{C}) = \mathsf{D}_1\ \mathsf{f}_1 \ldots \mathsf{D}_n\ \mathsf{f}_n}{\Gamma \vdash \mathsf{e.f}_i \in \mathsf{D}_i} \tag{3.16}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{e} \in \mathsf{C} \qquad \mathit{mtype}(\mathsf{m}, \mathsf{C}) = \mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D} \\ \Gamma \vdash \mathsf{e}_i \in \mathsf{C}_i \quad \mathsf{C}_i <: \mathsf{D}_i \quad (\forall i \in \{1 \ldots n\})\end{array}}{\Gamma \vdash \mathsf{e.m}\ (\mathsf{e}_1 \ldots \mathsf{e}_n) \in \mathsf{D}} \tag{3.17}$$

$$\frac{\begin{array}{l}\mathit{fields}(\mathsf{C}) = \mathsf{D}_1\ \mathsf{f}_1 \ldots \mathsf{D}_n\ \mathsf{f}_n \\ \Gamma \vdash \mathsf{e}_i \in \mathsf{C}_i \quad \mathsf{C}_i <: \mathsf{D}_i \quad (\forall i \in \{1 \ldots n\})\end{array}}{\Gamma \vdash \mathbf{new}\ \mathsf{C}\ (\mathsf{e}_1 \ldots \mathsf{e}_n) \in \mathsf{C}} \tag{3.18}$$

$$\frac{\Gamma \vdash \mathsf{e} \in \mathsf{D} \qquad \mathsf{D} <: \mathsf{C}}{\Gamma \vdash (\mathsf{C})\ \mathsf{e} \in \mathsf{C}} \tag{3.19}$$

$$\frac{\Gamma \vdash \mathsf{e} \in \mathsf{D} \qquad \mathsf{C} <: \mathsf{D} \qquad \mathsf{C} \neq \mathsf{D}}{\Gamma \vdash (\mathsf{C})\ \mathsf{e} \in \mathsf{C}} \tag{3.20}$$

$$\frac{\Gamma \vdash \mathsf{e} \in \mathsf{D} \qquad \mathsf{C} \not<: \mathsf{D} \qquad \mathsf{D} \not<: \mathsf{C}}{\Gamma \vdash (\mathsf{C})\ \mathsf{e} \in \mathsf{C}} \tag{3.21}$$

Figure 3.7: Typing rules for expressions

context of Γ. The environment Γ maps free variables x_i to their types D_i. Since FJ has no local variables or nested methods, the context is either empty at the top level or it contains bindings for the formal parameters of just one method.

The type of variable reference (rule 3.15) comes directly from the environment. The type of a selected field (rule 3.16) is retrieved from the class hierarchy. To check a method invocation (rule 3.17), we look up the type in the hierarchy above the static class (C) of the receiver. All of the argument expressions must have types compatible with the method signature. Note that this rule assumes that method signatures are

$$\frac{\begin{array}{l}\mathsf{K} = \mathsf{C}\ (\mathsf{B}_1\ \mathsf{g}_1 \ldots \mathsf{B}_n\ \mathsf{g}_n, \mathsf{C}_1\ \mathsf{f}_1 \ldots \mathsf{C}_m\ \mathsf{f}_m) \\ \qquad \{\mathbf{super}(\mathsf{g}_1 \ldots \mathsf{g}_n); \\ \qquad\ \ \mathbf{this}.\mathsf{f}_1 = \mathsf{f}_1; \ldots \mathbf{this}.\mathsf{f}_m = \mathsf{f}_m;\} \\ \mathit{fields}(\mathsf{B}) = \mathsf{B}_1\ \mathsf{g}_1 \ldots \mathsf{B}_n\ \mathsf{g}_n \\ \mathsf{M}_i \text{ ok in } \mathsf{C} \quad \forall i \in \{1 \ldots k\}\end{array}}{\mathbf{class}\ \mathsf{C}\ \mathbf{extends}\ \mathsf{B}\ \{\ \mathsf{C}_1\ \mathsf{f}_1; \ldots \mathsf{C}_m\ \mathsf{f}_m;\ \mathsf{K}\ \mathsf{M}_1 \ldots \mathsf{M}_k\ \} \text{ ok}} \tag{3.22}$$

$$\frac{\begin{array}{l}\mathsf{x}_1 : \mathsf{D}_1, \ldots, \mathsf{x}_n : \mathsf{D}_n, \mathbf{this} : \mathsf{C} \vdash \mathsf{e} \in \mathsf{E} \qquad \mathsf{E} <: \mathsf{D} \\ CT(\mathsf{C}) = \mathbf{class}\ \mathsf{C}\ \mathbf{extends}\ \mathsf{B}\ \{\ \ldots\ \} \\ \mathit{override}(\mathsf{m}, \mathsf{B}, \mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D})\end{array}}{\mathsf{D}\ \mathsf{m}\ (\mathsf{D}_1\ \mathsf{x}_1 \ldots \mathsf{D}_n\ \mathsf{x}_n)\ \{\ \mathbf{return}\ \mathsf{e};\ \} \text{ ok in } \mathsf{C}} \tag{3.23}$$

$$\frac{\mathit{mtype}(\mathsf{m}, \mathsf{B}) = \mathsf{C}_1 \ldots \mathsf{C}_n \to \mathsf{C}_0}{\mathit{override}(\mathsf{m}, \mathsf{B}, \mathsf{C}_1 \ldots \mathsf{C}_n \to \mathsf{C}_0)} \tag{3.24}$$

$$\frac{\neg \exists\, T \text{ such that } \mathit{mtype}(\mathsf{m}, \mathsf{B}) = T}{\mathit{override}(\mathsf{m}, \mathsf{B}, \mathsf{C}_1 \ldots \mathsf{C}_n \to \mathsf{C}_0)} \tag{3.25}$$

Figure 3.8: Well-typed classes and methods

invariant up the hierarchy—we will check that property elsewhere.

The three different rules for cast expressions deserve some explanation. The first (rule 3.19) is an upward cast, and is harmless. The second (3.20) is a downward or *dynamic* cast, which might fail at runtime. Of course, if it succeeds then the expression has the requested type C. Finally, rule 3.21 covers the case where the static class and the requested class are incomparable. Igarashi, Pierce, and Wadler (2001) call this a *stupid cast;* such a construct is not valid in top-level Java programs, but may arise during the course of reduction and must be assigned a type. ClassicJava was unsound as published due to the omission of stupid casts. Please refer to Igarashi, Pierce, and Wadler (2001) for further explanation.

The typing rules for expressions are not the whole story. Several additional constraints on classes and methods must be enforced; they are covered in figure 3.8. The *override* relation (rules 3.24 and 3.25) certifies that method signatures are invariant

up the hierarchy. In other words, overriding cannot change the type of a method—FJ does not support *overloading.* Rule 3.23 ensures that the type of the method body is compatible with the signature. Rule 3.22 enforces the rigid format of the constructor: it must pass the initializers for inherited fields along to the super class constructor, and then initialize its own fields, in order. Arbitrary expressions are banned from the constructor.

The judgments on well-typed classes, methods, and expressions are all decidable, and sound with respect to the operational semantics. Igarashi, Pierce, and Wadler (2001) prove the standard pair of preservation and progress theorems. Please see their article for the detailed proofs.

Chapter 4

Intermediate Language: Mini JFlint

Next, we need a sound formalization of an intermediate language. Building on a typed λ-calculus (Barendregt 1992) is an appropriate strategy for several reasons. First, such languages have been successful in type-based compilers for Standard ML (Shao and Appel 1995; Morrisett et al. 1996). Perhaps this is not surprising, since *untyped* λ-calculi had been used in functional language compilers for many years. Second, Morrisett et al. (1999b) developed techniques to compile System F all the way to typed assembly language.

Still, it was not clear when we started this work that a typed λ-calculus would be suitable for compiling Java. There was significant precedent for encoding object-oriented features in variants of F_ω (Bruce, Cardelli, and Pierce 1999; Hofmann and Pierce 1994; Abadi, Cardelli, and Viswanathan 1996), but not in the context of compilers.

In the end, using F_ω as a starting point was clearly a good choice. Extended with the right primitives, and expressed in either continuation-passing style (Appel 1992) or A-normal form (Flanagan et al. 1993), a typed λ-calculus does indeed resemble a low-level compiler intermediate language.

Kinds	$\kappa ::= \mathsf{Type} \mid \mathsf{R}^L \mid \kappa \Rightarrow \kappa' \mid \{(l{::}\kappa)^*\}$
Types	$\tau ::= \alpha \mid \lambda\alpha{::}\kappa.\,\tau \mid \tau\;\tau' \mid \{(l = \tau)^*\} \mid \tau \cdot l \mid \tau \to \tau' \mid \mathsf{Abs}^L \mid l : \tau\,;\tau'$
	$\mid \{\tau\} \mid \langle\!\langle \tau \rangle\!\rangle \mid \mu\alpha{::}\kappa.\,\tau \mid \forall\alpha{::}\kappa.\,\tau \mid \exists\alpha{::}\kappa.\,\tau$
Selectors	$s ::= \circ \mid s \cdot l$
Terms	$e ::= x \mid \lambda x : \tau.\, e \mid e\; e' \mid \Lambda\alpha{::}\kappa.\, e \mid e\;[\tau] \mid \mathbf{inj}_l^\tau\; e$
	$\mid \mathbf{case}\; e\; \mathbf{of}\; (l\; x \Rightarrow e)^*\; \mathbf{else}\; e \mid \{(l = e)^*\} \mid e.l \mid \mathbf{fix}\,[\tau]\; e$
	$\mid \langle \alpha{::}\kappa = \tau,\; e : \tau' \rangle \mid \mathbf{open}\; e\; \mathbf{as}\; \langle \alpha{::}\kappa,\; x : \tau \rangle\; \mathbf{in}\; e'$
	$\mid \mathbf{fold}\; e\; \mathbf{as}\; \mu\alpha{::}\kappa.\,\tau\; \mathbf{at}\; \lambda\gamma{::}\kappa.\, s[\gamma]$
	$\mid \mathbf{unfold}\; e\; \mathbf{as}\; \mu\alpha{::}\kappa.\,\tau\; \mathbf{at}\; \lambda\gamma{::}\kappa.\, s[\gamma]$
	$\mid \mathbf{abort}\,[\tau]$

Figure 4.1: Abstract syntax of Mini JFlint

$$
\begin{aligned}
l_1 : \tau_1, \ldots, l_n : \tau_n &\equiv l_1 : \tau_1\,; \ldots l_n : \tau_n\,; \mathsf{Abs}^{\{l_1 \ldots l_n\}} \\
\mathbf{1} &\equiv \{\mathsf{Abs}^{\varnothing}\} \\
\mathbf{maybe} &\equiv \lambda\alpha{::}\mathsf{Type}.\, \langle\!\langle \mathsf{some} : \alpha, \mathsf{none} : \mathbf{1} \rangle\!\rangle \\
\mathbf{some} &\equiv \Lambda\alpha{::}\mathsf{Type}.\, \lambda x : \alpha.\, \mathbf{inj}_{\mathsf{some}}^{\mathbf{maybe}\;\alpha}\; x \\
\mathbf{none} &\equiv \Lambda\alpha{::}\mathsf{Type}.\, \mathbf{inj}_{\mathsf{none}}^{\mathbf{maybe}\;\alpha}\; \{\} \\
\mathbf{let}\; x : \tau = e\; \mathbf{in}\; e' &\equiv (\lambda x : \tau.\, e')\; e
\end{aligned}
$$

Figure 4.2: Derived forms (syntactic sugar)

4.1 Syntax

Figure 4.1 contains the BNF grammar for the abstract syntax of Mini JFlint. It is based on the higher-order polymorphic λ-calculus F_ω, described independently by Girard (1972) and Reynolds (1974). Like F_ω, Mini JFlint is an explicitly-typed calculus, with annotations on formal parameters and terms for instantiating polymorphic functions. In addition, we include several well-understood features such as existential types (Mitchell and Plotkin 1988), row polymorphism (Rémy 1993), records, sum types, and recursive types. Several convenient syntactic forms are derived in figure 4.2. As with FJ, we augment the language with integers and arithmetic in examples.

For compiler writers, the key features to notice are in the category of terms. $\lambda x : \tau.\, e$

is an anonymous function with formal parameter x of type τ and body e. Functions and other values are bound to lexically scoped variables using the **let** derived form. A function call is expressed as $e\ e'$, a juxtaposition of the function expression and its actual parameter.

Mini JFlint supports ordered, labeled records, initialized on creation: $\{l_1 = e_1, l_2 = e_2\}$. As in C, the memory layout is determined by the type, so that field offsets are known at compile time. The notation $e.l$ selects field l from record e. Recursive records are expressed using a lazy fixed point operator **fix** $[\tau]$ e where τ is a sequence of record labels and their types, and e is a function from records to records. The fixed point is unrolled as needed to satisfy field selection expressions. In an imperative language, it would be implemented using assignment to create a recursive data structure.

The term $\mathbf{inj}_l^\tau\ e$ tags the value of e with the label l, producing a value of type τ. This implements algebraic data types as in functional languages, and is similar to the tagged union idiom in C. The **case** expression checks the tag and provides access to the corresponding tagged value. The default expression **else** e permits the cases to be non-exhaustive.

The term **abort** $[\tau]$ aborts computation, but is otherwise considered to have type τ. We use this to model a failed dynamic cast. In the operational semantics, evaluating **abort** $[\tau]$ produces an infinite loop, so that "progress" is preserved. In an actual system, **abort** would correspond to throwing an exception.

Language mavens are probably more interested in the typing hierarchy. As in F_ω, the type language is itself a simply-typed λ-calculus. So-called *kinds* classify types. Specifically, Type is the base kind of those types that, in turn, classify terms. The arrow kind $\kappa \Rightarrow \kappa'$ classifies type functions. A polymorphic array constructor, for example, would have kind Type⇒Type. The rules for forming kinds are in figure 4.3 on the following page.

The type function $\lambda\alpha{::}\kappa.\,\tau$ introduces the arrow kind, and $\tau\ \tau'$ eliminates it. That

$$\frac{}{\vdash \mathsf{Type}\ \mathit{kind}} \tag{4.1}$$

$$\frac{}{\vdash \mathsf{R}^L\ \mathit{kind}} \tag{4.2}$$

$$\frac{\vdash \kappa\ \mathit{kind} \qquad \vdash \kappa'\ \mathit{kind}}{\vdash \kappa \Rightarrow \kappa'\ \mathit{kind}} \tag{4.3}$$

$$\frac{\begin{array}{l} l_i = l_j \Rightarrow i = j \quad (\forall i, j \in \{1 \dots n\}) \\ \vdash \kappa_i\ \mathit{kind} \quad (\forall i \in \{1 \dots n\}) \end{array}}{\vdash \{l_1 :: \kappa_1 \dots l_n :: \kappa_n\}\ \mathit{kind}} \tag{4.4}$$

Figure 4.3: Formation rules for kinds

is, $(\lambda\alpha{::}\kappa.\,\tau)\ \tau'$ is well-formed if τ' has kind κ. It is equivalent to $\tau[\alpha := \tau']$, which denotes the capture-avoiding substitution of τ' for α in τ. Labeled tuples of types are enclosed in braces $\{l = \tau \dots\}$ and have tuple kinds $\{\tau :: \kappa \dots\}$. The mid-dot syntax $\tau \cdot l$ denotes selection of a type from a tuple.

The single arrow $\tau \to \tau'$ is the type of a function expecting an argument of type τ and returning a result of type τ'. Our implementation supports multi-argument functions, but for the purposes of formal presentation, we simulate them using *curried* arguments (**int**→**int**→**int**). Polymorphic functions are introduced by the capital lambda $(\Lambda\alpha{::}\kappa.\,e)$ which binds α in e. This term has type $\forall\alpha{::}\kappa.\,\tau$, where e has type τ and α may appear in τ. Thus, the polymorphic identity function is written as $\mathsf{id} = \Lambda\alpha{::}\mathsf{Type}.\,\lambda x{::}\alpha.\,x$ and has type $\forall\alpha{::}\mathsf{Type}.\,\alpha \to \alpha$. An application of id to the integer 3 is written id [**int**] 3. The definitions **maybe**, **some**, and **none** in figure 4.2 are good examples of higher-order types and polymorphism.

A *row* is essentially a suffix (or *tail*) of a record type. Intuitively, rows and types are distinct syntactic categories, but it is convenient to collapse them. Otherwise, we would need to distinguish their quantifiers. Rémy (1993) introduced a kind R^L of rows where

$$\frac{}{\vdash \circ \ \textit{kind env}} \tag{4.5}$$

$$\frac{\vdash \Phi \ \textit{kind env} \qquad \vdash \kappa \ \textit{kind}}{\vdash \Phi, \alpha :: \kappa \ \textit{kind env}} \tag{4.6}$$

$$\frac{}{\Phi \vdash \circ \ \textit{type env}} \tag{4.7}$$

$$\frac{\Phi \vdash \Delta \ \textit{type env} \qquad \Phi \vdash \tau :: \mathsf{Type}}{\Phi \vdash \Delta, x : \tau \ \textit{type env}} \tag{4.8}$$

Figure 4.4: Formation rules for environments

L is the set of labels *banned* from the row. Abs^L is an empty row of kind R^L, and $l : \tau \,;\, \tau'$ prepends a field with label l and type τ onto the row τ'. The row formation rules (4.14 and 4.15; see figure 4.5 on page 36) prohibit duplicate labels: a type variable α of kind $\mathsf{R}^{\{\mathsf{m}\}}$ cannot be instantiated with a row in which the label m is already bound. Braces $\{\cdot\}$ denote the type constructor for records; it lifts a complete row type (of kind $\mathsf{R}^{\varnothing}$) to kind Type. The row syntax is reused within triangle brackets $\langle\!|\ \cdot\ |\!\rangle$ to denote sum types.

Record terms are written as a sequence of bindings in braces: $\{l_1 = e, l_2 = e\}$. Permutations of rows are *not* considered equivalent—the labels are used only for readability. This means that record selection $e.l$ can be compiled using offsets that are known at compile-time. We sometimes use commas and omit Abs^L when specifying complete rows (see the derived forms in figure 4.2). We let $\mathbf{1}$ (read 'unit') denote the empty record type.

Row kinds can be used to encode functions that are polymorphic over the *tail* of a record argument. For example, the function $\Lambda\rho{::}\mathsf{R}^{\{l\}}.\, \lambda x : \{l : \mathsf{string} \,;\, \rho\}.\, \mathsf{print}\ x.l$ can be instantiated and applied to any record which contains a string l as its first field.

Existential types ($\exists\alpha{::}\kappa.\, \tau$) support abstraction by *hiding* a witness type (Mitchell

$$\frac{\vdash \Phi\ \textit{kind env} \qquad \Phi(\alpha) = \kappa}{\Phi \vdash \alpha :: \kappa} \tag{4.9}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \kappa'}{\Phi \vdash \lambda\alpha{::}\kappa.\,\tau :: \kappa \Rightarrow \kappa'} \tag{4.10}$$

$$\frac{\Phi \vdash \tau_1 :: \kappa' \Rightarrow \kappa \qquad \Phi \vdash \tau_2 :: \kappa'}{\Phi \vdash \tau_1\ \tau_2 :: \kappa} \tag{4.11}$$

$$\frac{\begin{array}{l} l_i = l_j \Rightarrow i = j \quad (\forall i, j \in \{1 \ldots n\}) \\ \Phi \vdash \tau_i :: \kappa_i \quad (\forall i \in \{1 \ldots n\}) \end{array}}{\Phi \vdash \{l_1 = \tau_1 \ldots l_n = \tau_n\} :: \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\}} \tag{4.12}$$

$$\frac{\Phi \vdash \tau :: \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\}}{\Phi \vdash \tau \cdot l_i :: \kappa_i} \tag{4.13}$$

$$\frac{\vdash \Phi\ \textit{kind env}}{\Phi \vdash \mathsf{Abs}^L :: \mathsf{R}^L} \tag{4.14}$$

$$\frac{\Phi \vdash \tau :: \mathsf{Type} \qquad \Phi \vdash \tau' :: \mathsf{R}^{L \cup \{l\}}}{\Phi \vdash l : \tau \,;\, \tau' :: \mathsf{R}^{L - \{l\}}} \tag{4.15}$$

Figure 4.5: Type formation rules

and Plotkin 1988). They are introduced at the term level by a *package* $\langle \alpha{::}\kappa = \tau,\, e : \tau' \rangle$, where τ is the witness type (of kind κ) and e has type $\tau'[\alpha := \tau]$. The existential is eliminated (within a restricted scope) by **open**; see rules 4.26 and 4.27 in figure 4.7 on page 39.

The following example demonstrates the syntax for creating and opening existential packages. We will package some value with a function that expects a value of the same

$$\frac{\Phi \vdash \tau_1 :: \mathsf{Type} \qquad \Phi \vdash \tau_2 :: \mathsf{Type}}{\Phi \vdash \tau_1 \to \tau_2 :: \mathsf{Type}} \tag{4.16}$$

$$\frac{\Phi \vdash \tau :: \mathsf{R}^{\varnothing}}{\Phi \vdash \{\tau\} :: \mathsf{Type}} \tag{4.17}$$

$$\frac{\Phi \vdash \tau :: \mathsf{R}^{\varnothing}}{\Phi \vdash \langle\!\langle \tau \rangle\!\rangle :: \mathsf{Type}} \tag{4.18}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \kappa}{\Phi \vdash \mu\alpha{::}\kappa.\, \tau :: \kappa} \tag{4.19}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \mathsf{Type}}{\Phi \vdash \forall\alpha{::}\kappa.\, \tau :: \mathsf{Type}} \tag{4.20}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \mathsf{Type}}{\Phi \vdash \exists\alpha{::}\kappa.\, \tau :: \mathsf{Type}} \tag{4.21}$$

Figure 4.6: Type formation rules, continued

type. Then we will hide that type from outsiders.

$$
\begin{array}{l}
\mathbf{let}\ x_1 : (\exists\alpha{::}\mathsf{Type}.\, \{z : \alpha,\ f : \alpha \to \mathbf{string}\}) = \\
\quad \langle \beta{::}\mathsf{Type} = \mathbf{int},\ \{z = 42,\ f = \mathsf{int2string}\} : \{z : \beta,\ f : \beta \to \mathbf{string}\}\rangle \\
\ \mathbf{in\ let}\ x_2 : (\exists\alpha{::}\mathsf{Type}.\, \{z : \alpha,\ f : \alpha \to \mathbf{string}\}) = \\
\qquad \langle \gamma{::}\mathsf{Type} = \mathbf{real},\ \{z = 3.1415,\ f = \mathsf{real2string}\} : \{z : \gamma,\ f : \gamma \to \mathbf{string}\}\rangle \\
\quad \mathbf{in}\ \ldots
\end{array}
$$

Now, the packages x_1 and x_2 have the same (existential) type, even though the values inside them have different types (**int** and **real**). We used β and γ in this example to emphasize the scopes of type variables bound in each existential package. Here is a

function that will accept x_1, x_2, or any similar existentially-typed value.

$$\lambda y : (\exists \alpha{::}\mathsf{Type}.\, \{z : \alpha,\ f : \alpha \to \mathbf{string}\}).$$
$$\quad \mathbf{open}\ y\ \mathbf{as}\ \langle \delta{::}\mathsf{Type},\ g : \{z : \delta,\ f : \delta \to \mathbf{string}\}\rangle$$
$$\qquad \mathbf{in}\ g.f\ g.z$$

Because we packaged a method with a private value, this simple example even has the flavor of object-oriented programming, although further apparatus is needed to support dynamic binding.

Recursive types are mediated by explicit *fold* and *unfold* terms. These so-called *iso-recursive* types—a term first used by Crary, Harper, and Puri (1999)—simplify type checking, but are less flexible than *equi-recursive* types unless the calculus is equipped with a definedness logic for coercions (Abadi and Fiore 1996). Since we use recursive types at higher kinds, the syntax for folding and unfolding them deserves some explanation. Suppose we wish to encode the following mutually recursive type abbreviations:

$$\mathbf{type}\ \mathsf{even} = \mathbf{maybe}\ \{\mathsf{hd} : \mathbf{int}, \mathsf{tl} : \mathsf{odd}\}$$
$$\mathbf{type}\ \mathsf{odd} = \{\mathsf{hd} : \mathbf{int}, \mathsf{tl} : \mathsf{even}\}$$

The solution is expressed as the fixed point over a tuple:

$$t = \mu\alpha{::}\{\mathsf{even} :: \mathsf{Type},\ \mathsf{odd} :: \mathsf{Type}\}.$$
$$\quad \{\mathsf{even} = \mathbf{maybe}\ \{\mathsf{hd} : \mathbf{int}, \mathsf{tl} : \alpha\cdot\mathsf{odd}\},$$
$$\qquad \mathsf{odd} = \{\mathsf{hd} : \mathbf{int}, \mathsf{tl} : \alpha\cdot\mathsf{even}\}\}$$

Now, the two recursive types are expressed as $t\cdot\mathsf{even}$ and $t\cdot\mathsf{odd}$. There are, however, no type equivalence rules for reducing $t\cdot\mathsf{even}$; a term having this type must first be coerced

$$\frac{\Phi \vdash \Delta \ \mathit{type\ env} \qquad \Delta(x) = \tau}{\Phi; \Delta \vdash x : \tau} \tag{4.22}$$

$$\frac{\Phi; \Delta \vdash e : \tau \qquad \Phi \vdash \tau = \tau' :: \mathsf{Type}}{\Phi; \Delta \vdash e : \tau'} \tag{4.23}$$

$$\frac{\Phi \vdash \tau :: \mathsf{Type} \qquad \Phi; \Delta, x : \tau \vdash e : \tau'}{\Phi; \Delta \vdash (\lambda x : \tau.\, e) : \tau \to \tau'} \tag{4.24}$$

$$\frac{\Phi; \Delta \vdash e_1 : \tau' \to \tau \qquad \Phi; \Delta \vdash e_2 : \tau'}{\Phi; \Delta \vdash e_1\ e_2 : \tau} \tag{4.25}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \mathsf{Type} \qquad \Phi \vdash \tau' :: \kappa \qquad \Phi; \Delta \vdash e : \tau[\alpha := \tau']}{\Phi; \Delta \vdash \langle \alpha :: \kappa = \tau', e : \tau \rangle : \exists \alpha :: \kappa.\, \tau} \tag{4.26}$$

$$\frac{\Phi; \Delta \vdash e : \exists \alpha :: \kappa.\, \tau \qquad \Phi \vdash \tau' :: \mathsf{Type} \qquad \Phi, \alpha :: \kappa; \Delta, x : \tau \vdash e' : \tau' \qquad \alpha \notin \mathrm{dom}(\Phi)}{\Phi; \Delta \vdash \mathbf{open}\ e\ \mathbf{as}\ \langle \alpha :: \kappa, x : \tau \rangle\ \mathbf{in}\ e' : \tau'} \tag{4.27}$$

Figure 4.7: Term formation rules

to a type in which t is *unfolded*. We allow unfolding of recursive types within a tuple by specifying a *selector* after the **at** keyword. Selectors are syntactically restricted to a (possibly empty) sequence of labeled selections from a tuple. The syntax $\lambda y :: \kappa.\, s[y]$ allows identity ($\lambda y :: \kappa.\, y$), one selection ($\lambda y :: \kappa.\, y \cdot l_1$), two selections ($\lambda y :: \kappa.\, y \cdot l_1 \cdot l_2$), and so on. The formation rules (4.33 and 4.34—see figure 4.8 on page 41) further restrict the selectors to have a result of kind Type. Thus, if e has type $t \cdot \mathsf{odd}$, then the expression

$$\mathbf{unfold}\ e\ \mathbf{as}\ t\ \mathbf{at}\ \lambda y :: \{\mathsf{even} :: \mathsf{Type},\ \mathsf{odd} :: \mathsf{Type}\}.\, y \cdot \mathsf{odd}$$

has type $\{\mathsf{hd} : \mathbf{int}, \mathsf{tl} : t \cdot \mathsf{even}\}$. For recursive types of kind Type, the only allowed selector is identity, so we omit it. We sometimes also omit the **as** annotation where it can be

readily inferred.

4.2 Semantics

The semantics of Mini JFlint consists of several kinds of static judgments, plus a small-step operational semantics. In describing the syntax, we already referred to several of the typing rules; now we will introduce them properly. Figure 4.3 on page 34 defines the judgment $\vdash \kappa$ *kind* for well-formed kinds. The rules are quite simple; they ensure only that tuple kinds have distinct labels.

Since both types and terms have free variables, we need environments for each. We use Φ for the kind environment (mapping type variables to their kinds) and Δ for the type environment (mapping term variables to their types). The judgments defined in figure 4.4 ensure that both sorts of environments are well-formed.

The judgment $\Phi \vdash \tau :: \kappa$ states that, in the context of Φ, the type τ has kind κ. Any free variables in τ must be in the domain of the environment Φ. The rules for this judgment are in figures 4.5 and 4.6. They are all quite standard, but the previously noted rules for forming rows and records are the most interesting. Because there are tuples and functions at the type level, we need a notion of equivalence beyond syntactic congruence. The equality relation on types $\Phi \vdash \tau_1 = \tau_2 :: \kappa$ is defined in figures 4.9 on page 42 and 4.10 on page 43.

The typing rules for the term language are in figures 4.7 and 4.8. The judgment $\Phi; \Delta \vdash e : \tau$ means that expression e has type τ, assuming that any free variables are in the domain of Δ. The kind environment Φ is needed because some terms introduce new type variables. Most of the rules are standard, but several warrant further explanation.

Rule 4.23 allows the substitution of an equivalent type anywhere in a derivation; it is the only typing rule that is not syntax-directed. To ensure that the term-level fixed point is used only for constructing records, the type annotation in $\mathbf{fix}\,[\tau]\; e$ is really a

$$\frac{l_i = l_j \Rightarrow i = j \quad (\forall i, j \in \{1 \ldots n\}) \qquad \Phi; \Delta \vdash e_i : \tau_i \quad (\forall i \in \{1 \ldots n\})}{\Phi; \Delta \vdash \{l_1 = e_1 \ldots l_n = e_n\} : \{l_1 : \tau_1 \ldots l_n : \tau_n\}} \tag{4.28}$$

$$\frac{\Phi; \Delta \vdash e : \{l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau\}}{\Phi; \Delta \vdash e.l_i : \tau_i} \tag{4.29}$$

$$\frac{\Phi; \Delta \vdash e : \{\tau\} \to \{\tau\} \qquad \Phi \vdash \tau :: \mathsf{R}^{\varnothing}}{\Phi; \Delta \vdash \mathbf{fix}\,[\tau]\; e : \{\tau\}} \tag{4.30}$$

$$\frac{\Phi \vdash \langle\!| l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau |\!\rangle :: \mathsf{Type} \qquad \Phi; \Delta \vdash e : \tau_i}{\Phi; \Delta \vdash \mathbf{inj}_{l_i}^{\langle\!| l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau |\!\rangle}\; e : \langle\!| l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau |\!\rangle} \tag{4.31}$$

$$\frac{l'_j = l'_{j'} \Rightarrow j = j' \quad (\forall j, j' \in \{1 \ldots m\}) \qquad \Phi; \Delta \vdash e : \langle\!| l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau |\!\rangle \qquad \Phi; \Delta \vdash e' : \tau' \qquad \exists i \in \{1 \ldots n\} : l_i = l'_j \text{ and } \Phi; \Delta, x_j : \tau_i \vdash e_j : \tau' \quad (\forall j \in \{1 \ldots m\})}{\Phi; \Delta \vdash \mathbf{case}\; e\; \mathbf{of}\; (l'_j\; x_j \Rightarrow e_j)^{j \in \{1 \ldots m\}}\; \mathbf{else}\; e' : \tau'} \tag{4.32}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \kappa \qquad \Phi \vdash \tau_s :: \kappa \Rightarrow \mathsf{Type} \qquad \Phi; \Delta \vdash e : \tau_s\; (\tau[\alpha := \mu\alpha{::}\kappa.\, \tau])}{\Phi; \Delta \vdash \mathbf{fold}\; e\; \mathbf{as}\; \mu\alpha{::}\kappa.\, \tau\; \mathbf{at}\; \tau_s \; : \tau_s\; (\mu\alpha{::}\kappa.\, \tau)} \tag{4.33}$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau :: \kappa \qquad \Phi \vdash \tau_s :: \kappa \Rightarrow \mathsf{Type} \qquad \Phi; \Delta \vdash e : \tau_s\; (\mu\alpha{::}\kappa.\, \tau)}{\Phi; \Delta \vdash \mathbf{unfold}\; e\; \mathbf{as}\; \mu\alpha{::}\kappa.\, \tau\; \mathbf{at}\; \tau_s : \tau_s\; (\tau[\alpha := \mu\alpha{::}\kappa.\, \tau])} \tag{4.34}$$

$$\frac{\Phi, \alpha :: \kappa; \Delta \vdash e : \tau \qquad \Phi \vdash \Delta\; \textit{type env}}{\Phi; \Delta \vdash (\Lambda\alpha{::}\kappa.\, e) : \forall\alpha{::}\kappa.\, \tau} \tag{4.35}$$

$$\frac{\Phi; \Delta \vdash e : \forall\alpha{::}\kappa.\, \tau \qquad \Phi \vdash \tau' :: \kappa}{\Phi; \Delta \vdash e\; [\tau'] : \tau[\alpha := \tau']} \tag{4.36}$$

$$\frac{\Phi \vdash \tau :: \mathsf{Type} \qquad \Phi \vdash \Delta\; \textit{type env}}{\Phi; \Delta \vdash \mathbf{abort}\,[\tau] : \tau} \tag{4.37}$$

Figure 4.8: Term formation rules, continued

$$\frac{\Phi \vdash \tau_1 :: \kappa_1 \qquad \Phi, \alpha :: \kappa_1 \vdash \tau_2 :: \kappa_2}{\Phi \vdash (\lambda\alpha{::}\kappa_1.\, \tau_2)\ \tau_1 = \tau_2[\alpha := \tau_1] :: \kappa_2} \tag{4.38}$$

$$\frac{\Phi \vdash \tau :: \kappa \Rightarrow \kappa' \quad \alpha \notin \mathrm{dom}(\Phi)}{\Phi \vdash \lambda\alpha{::}\kappa.\, \tau\ \alpha = \tau :: \kappa \Rightarrow \kappa'} \tag{4.39}$$

$$\frac{\begin{array}{l} l_i = l_j \Rightarrow i = j \quad (\forall i, j \in \{1 \ldots n\}) \\ \Phi \vdash \tau_i :: \kappa_i \quad (\forall i \in \{1 \ldots n\}) \end{array}}{\Phi \vdash \{l_1 = \tau_1 \ldots l_n = \tau_n\} \cdot l_i = \tau_i :: \kappa_i} \tag{4.40}$$

$$\frac{\begin{array}{l} l_i = l_j \Rightarrow i = j \quad (\forall i, j \in \{1 \ldots n\}) \\ \Phi \vdash \tau :: \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\} \end{array}}{\Phi \vdash \{l_1 = \tau \cdot l_1 \ldots l_n = \tau \cdot l_n\} = \tau :: \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\}} \tag{4.41}$$

$$\frac{\Phi \vdash \tau :: \kappa}{\Phi \vdash \tau = \tau :: \kappa} \tag{4.42}$$

$$\frac{\Phi \vdash \tau_1 = \tau_2 :: \kappa}{\Phi \vdash \tau_2 = \tau_1 :: \kappa} \tag{4.43}$$

$$\frac{\Phi \vdash \tau_1 = \tau_2 :: \kappa \qquad \Phi \vdash \tau_2 = \tau_3 :: \kappa}{\Phi \vdash \tau_1 = \tau_3 :: \kappa} \tag{4.44}$$

Figure 4.9: Type equivalence rules

row; see rule 4.30.

The rule for **case** expressions (4.32) ensures that the listed labels are all disjoint and members of the sum type being eliminated. The labeled cases need not be exhaustive; we use the **else** clause in the implementation of dynamic cast and class linking in chapter 5.

The operational semantics are defined in figures 4.11 on page 45 and 4.12 on page 46. The grammar for values v defines a subset of the terms that are irreducible; these include abstractions, records of values, tagged values, and packed or folded values. There is just one dynamic judgment: $e \leadsto e'$ means that term e reduces to term e' in

$$\frac{\Phi, \alpha :: \kappa \vdash \tau_1 = \tau_2 :: \kappa'}{\Phi \vdash \lambda\alpha{::}\kappa.\,\tau_1 = \lambda\alpha{::}\kappa.\,\tau_2 :: \kappa \Rightarrow \kappa'} \quad (4.45)$$

$$\frac{\Phi \vdash \tau_1 = \tau_1' :: \kappa' \Rightarrow \kappa \qquad \Phi \vdash \tau_2 = \tau_2' :: \kappa'}{\Phi \vdash \tau_1\ \tau_2 = \tau_1'\ \tau_2' :: \kappa} \quad (4.46)$$

$$\frac{\begin{array}{l} l_i = l_j \Rightarrow i = j \quad (\forall i, j \in \{1 \ldots n\}) \\ \Phi \vdash \tau_i = \tau_i' :: \kappa_i \quad (\forall i \in \{1 \ldots n\}) \end{array}}{\begin{array}{l} \Phi \vdash \{l_1 = \tau_1 \ldots l_n = \tau_n\} = \{l_1 = \tau_1' \ldots l_n = \tau_n'\} \\ \qquad :: \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\} \end{array}} \quad (4.47)$$

$$\frac{\Phi \vdash \tau = \tau' :: \{l_1 :: \kappa_1 \ldots l_n :: \kappa_n\}}{\Phi \vdash \tau \cdot l_i = \tau' \cdot l_i :: \kappa_i} \quad (4.48)$$

$$\frac{\Phi \vdash \tau_1 = \tau_1' :: \mathsf{Type} \qquad \Phi \vdash \tau_2 = \tau_2' :: \mathsf{Type}}{\Phi \vdash \tau_1 \to \tau_2 = \tau_1' \to \tau_2' :: \mathsf{Type}} \quad (4.49)$$

$$\frac{\Phi \vdash \tau_1 = \tau_1' :: \mathsf{Type} \qquad \Phi \vdash \tau_2 = \tau_2' :: \mathsf{R}^{L \cup \{l\}}}{\Phi \vdash l : \tau_1 ; \tau_2 = l : \tau_1' ; \tau_2' :: \mathsf{R}^{L - \{l\}}} \quad (4.50)$$

$$\frac{\Phi \vdash \tau = \tau' :: \mathsf{R}^{\varnothing}}{\Phi \vdash \{\tau\} = \{\tau'\} :: \mathsf{Type}} \quad (4.51)$$

$$\frac{\Phi \vdash \tau = \tau' :: \mathsf{R}^{\varnothing}}{\Phi \vdash \langle\!\langle \tau \rangle\!\rangle = \langle\!\langle \tau' \rangle\!\rangle :: \mathsf{Type}} \quad (4.52)$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau_1 = \tau_2 :: \kappa}{\Phi \vdash \mu\alpha{::}\kappa.\,\tau_1 = \mu\alpha{::}\kappa.\,\tau_2 :: \kappa} \quad (4.53)$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau_1 = \tau_2 :: \mathsf{Type}}{\Phi \vdash \forall\alpha{::}\kappa.\,\tau_1 = \forall\alpha{::}\kappa.\,\tau_2 :: \mathsf{Type}} \quad (4.54)$$

$$\frac{\Phi, \alpha :: \kappa \vdash \tau_1 = \tau_2 :: \mathsf{Type}}{\Phi \vdash \exists\alpha{::}\kappa.\,\tau_1 = \exists\alpha{::}\kappa.\,\tau_2 :: \mathsf{Type}} \quad (4.55)$$

Figure 4.10: Type equivalence rules, continued

one step. All the interesting reduction rules are in figure 4.11.

Surprisingly, the recursive record fixed point (**fix** $[\tau]$ e) is treated as a value. It is unrolled only when it is the subject of a field selection; see rule 4.58. The **abort** primitive is modeled as an infinite loop, as explained previously.

4.3 Properties

This formalization of our intermediate language enjoys several essential properties. First, the static typing judgments are decidable; this is proved with the aid of the following lemmas.

Lemma 1 (Normalization) *Type reductions are strongly normalizing.*

Proof sketch The type equivalence judgments can be read left-to-right as reductions. To demonstrate that these reductions are strongly normalizing, we view the type language as a simply-typed λ-calculus itself, extended with records (tuples), lists with labeled elements (rows), a base type (Type) and several constants ($\to$, $\{\cdot\}$, $\llbracket \cdot \rrbracket$). The binding operators (μ, $\forall$, $\exists$) are also constants, since they are neither introduced nor eliminated by any reduction rule. Standard proofs for strong normalization of the simply-typed λ-calculus—by Goguen (1995), for example—can be adapted to this type language. □

Lemma 2 (Confluence) *Type reductions are confluent.*

Proof sketch As above, we can adapt a standard proof for confluence of the simply-typed λ-calculus. □

Theorem 1 (Decidability) *All static judgments in the previous section are decidable.*

Proof Judgments for the formation of kinds, kind environments, types, and type environments are all syntax-directed and trivially decidable.

Values $v ::= \lambda x{:}\tau.\,e \mid \{(l = v)^*\} \mid \textbf{fix}\,[\tau]\;e \mid \textbf{inj}_l^\tau\;v \mid \Lambda\alpha{::}\kappa.\,e$
$\mid \langle \alpha{::}\kappa = \tau,\, v : \tau' \rangle \mid \textbf{fold}\;v\;\textbf{as}\;\mu\alpha{::}\kappa.\,\tau\;\textbf{at}\;\lambda\gamma{::}\kappa.\,s[\gamma]$

$$\frac{}{(\lambda x{:}\tau.\,e)\;v \leadsto e[x := v]} \qquad (4.56)$$

$$\frac{}{(\{l_1 = v_1 \ldots l_n = v_n\}).l_i \leadsto v_i} \qquad (4.57)$$

$$\frac{}{(\textbf{fix}\,[\tau]\;e).l \leadsto (e\;(\textbf{fix}\,[\tau]\;e)).l} \qquad (4.58)$$

$$\frac{l_i = l'_k}{\begin{array}{l}\textbf{case inj}_{l_i}^{\langle\!|l_1:\tau_1;\ldots l_n:\tau_n;\tau|\!\rangle}\;v\;\textbf{of}\\ (l'_j\;x_j \Rightarrow e_j)^{j\in\{1\ldots m\}}\;\textbf{else}\;e'\\ \quad\leadsto e_k[x_k := v]\end{array}} \qquad (4.59)$$

$$\frac{l_i \neq l'_k \qquad (\forall k \in \{1\ldots m\})}{\begin{array}{l}\textbf{case inj}_{l_i}^{\langle\!|l_1:\tau_1;\ldots l_n:\tau_n;\tau|\!\rangle}\;v\;\textbf{of}\\ (l'_j\;x_j \Rightarrow e_j)^{j\in\{1\ldots m\}}\;\textbf{else}\;e'\\ \quad\leadsto e'\end{array}} \qquad (4.60)$$

$$\frac{}{\textbf{unfold}\;(\textbf{fold}\;v\;\textbf{as}\;\tau\;\textbf{at}\;\tau_s)\;\textbf{as}\;\tau\;\textbf{at}\;\tau_s \leadsto v} \qquad (4.61)$$

$$\frac{}{(\Lambda\alpha{::}\kappa.\,e)\;[\tau] \leadsto e[\alpha := \tau]} \qquad (4.62)$$

$$\frac{}{\begin{array}{l}\textbf{open}\;\langle\alpha{::}\kappa = \tau',\, v:\tau\rangle\;\textbf{as}\;\langle\alpha{::}\kappa,\, x:\tau\rangle\;\textbf{in}\;e'\\ \quad\leadsto e'[\alpha := \tau'][x := v]\end{array}} \qquad (4.63)$$

$$\frac{}{\textbf{abort}\,[\tau] \leadsto \textbf{abort}\,[\tau]} \qquad (4.64)$$

Figure 4.11: Values and primitive reductions

$$\frac{e \leadsto e'}{e\ e_2 \leadsto e'\ e_2} \tag{4.65}$$

$$\frac{e \leadsto e'}{v_1\ e \leadsto v_1\ e'} \tag{4.66}$$

$$\frac{e \leadsto e'}{\begin{array}{l}\{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e, l_{i+1} = e_{i+1}, \ldots, l_n = e_n\} \\ \quad \leadsto \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = e', l_{i+1} = e_{i+1}, \ldots, l_n = e_n\}\end{array}} \tag{4.67}$$

$$\frac{e \leadsto e'}{e.l \leadsto e'.l} \tag{4.68}$$

$$\frac{e \leadsto e'}{\textbf{inj}_l^\tau\ e \leadsto \textbf{inj}_l^\tau\ e'} \tag{4.69}$$

$$\frac{e \leadsto e'}{\begin{array}{l}\textbf{case}\ e\ \textbf{of}\ (l_i\ x_i \Rightarrow e_i)^{i\in\{1\ldots m\}}\ \textbf{else}\ e'' \\ \quad \leadsto \textbf{case}\ e'\ \textbf{of}\ (l_i\ x_i \Rightarrow e_i)^{i\in\{1\ldots m\}}\ \textbf{else}\ e''\end{array}} \tag{4.70}$$

$$\frac{e \leadsto e'}{\textbf{fold}\ e\ \textbf{as}\ \tau\ \textbf{at}\ \tau_s \leadsto \textbf{fold}\ e'\ \textbf{as}\ \tau\ \textbf{at}\ \tau_s} \tag{4.71}$$

$$\frac{e \leadsto e'}{\textbf{unfold}\ e\ \textbf{as}\ \tau\ \textbf{at}\ \tau_s \leadsto \textbf{unfold}\ e'\ \textbf{as}\ \tau\ \textbf{at}\ \tau_s} \tag{4.72}$$

$$\frac{e \leadsto e'}{e\ [\tau] \leadsto e'\ [\tau]} \tag{4.73}$$

$$\frac{e \leadsto e'}{\langle \alpha{::}\kappa = \tau, e : \tau' \rangle \leadsto \langle \alpha{::}\kappa = \tau, e' : \tau' \rangle} \tag{4.74}$$

$$\frac{e \leadsto e'}{\begin{array}{l}\textbf{open}\ e\ \textbf{as}\ \langle \alpha{::}\kappa, x : \tau \rangle\ \textbf{in}\ e_1 \\ \quad \leadsto \textbf{open}\ e'\ \textbf{as}\ \langle \alpha{::}\kappa, x : \tau \rangle\ \textbf{in}\ e_1\end{array}} \tag{4.75}$$

Figure 4.12: Congruence rules

Type equivalence is not syntax-directed. Since reductions are, however, strongly normalizing (lemma 1) we have an algorithm for deciding type equivalence: reduce τ_1 and τ_2 to normal form, then test whether they are syntactically congruent (modulo renaming of bound variables).

Term formation is syntax-directed except for rule (4.23), which accounts for type equivalences. If an algorithm always reduces types to normal forms, then the types of two different expressions can be checked for syntactic congruence, and rule (4.23) is not needed. □

Next, we give a detailed proof that the type system of Mini JFlint is sound with respect to its operational semantics. This is expressed as the usual pair of theorems: subject reduction and progress. The first means that each reduction preserves the type of an expression. The second guarantees that a well-typed expression is either a value already, or it can be further reduced. Note that, because we defined the reduction of **abort** as an infinite loop, even programs which encounter an **abort** still make progress.

Lemma 3 (Substitution of terms) *If* $\Phi;\Delta \vdash e' : \tau'$ *and* $\Phi;\Delta, x:\tau' \vdash e:\tau$, *then* $\Phi;\Delta \vdash e[x := e'] : \tau$.

Proof By induction on the derivation of $\Phi;\Delta, x:\tau' \vdash e:\tau$. □

Lemma 4 (Substitution of types) *If* $\Phi \vdash \tau' :: \kappa$ *and* $\Phi, \alpha :: \kappa; \Delta \vdash e:\tau$, *then* $\Phi;\Delta[\alpha := \tau'] \vdash e[\alpha := \tau'] : \tau[\alpha := \tau']$.

Proof By induction on the derivation of $\Phi, \alpha :: \kappa; \Delta \vdash e:\tau$. □

Theorem 2 (Subject reduction) *If* $e \leadsto e'$ *and* $\Phi;\Delta \vdash e:\tau$ *then* $\Phi;\Delta \vdash e':\tau$.

Proof By induction on the derivation of $e \leadsto e'$.

Case (4.56) $(\lambda x:\tau.\, e)\ v \leadsto e[x := v]$. From antecedent, $\Phi;\Delta \vdash (\lambda x:\tau.\, e)\ v : \tau'$. By inversion on (4.25) and (4.24), $\Phi;\Delta, x:\tau \vdash e:\tau'$, and $\Phi;\Delta \vdash v:\tau$. Finally, $\Phi;\Delta \vdash e[x := v] : \tau'$ using lemma 3.

Case (4.57) $(\{l_1 = v_1 \ldots l_n = v_n\}).l_i \leadsto v_i$. From antecedent, $\Phi;\Delta \vdash \{l_1 = v_1 \ldots l_n = v_n\}.l_i : \tau$. By inversion on (4.29) and (4.28), $\Phi;\Delta \vdash v_i : \tau$.

Case (4.58) $(\mathbf{fix}\,[\tau]\;e).l_i \leadsto (e\;(\mathbf{fix}\,[\tau]\,e)).l_i$. From antecedent, $\Phi;\Delta \vdash (\mathbf{fix}\,[\tau]\;e).l_i : \tau_i$. By inversion on (4.29), $\Phi;\Delta \vdash \mathbf{fix}\,[\tau]\;e : \{l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau'\}$. By inversion on (4.30), $\Phi;\Delta \vdash e : \{\tau\} \to \{\tau\}$, and $\tau = \{l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau'\}$. Using (4.25), $\Phi;\Delta \vdash e\;(\mathbf{fix}\,[\tau]\,e) : \{l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau'\}$. Then, using (4.29), $\Phi;\Delta \vdash (e\;(\mathbf{fix}\,[\tau]\,e)).l_i : \tau_i$.

Case (4.59) $\mathbf{case}\;\mathbf{inj}_{l_i}^{\{\!|l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau|\!\}}\;v\;\mathbf{of}\;(l'_j\;x_j \Rightarrow e_j)^{j \in \{1 \ldots m\}}\;\mathbf{else}\;e' \leadsto e_k[x_k := v]$ where $l_i = l'_k$. From antecedent, $\Phi;\Delta \vdash \mathbf{case}\;\ldots : \tau'$. By inversion on (4.32), $\Phi;\Delta, x_k : \tau_i \vdash e_k : \tau'$ and $\Phi;\Delta \vdash \mathbf{inj}_{l_i}^{\cdots}\;v : \{\!|l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau|\!\}$. By inversion on (4.31) and lemma 3, $\Phi;\Delta \vdash e_k[x_k := v] : \tau'$.

Case (4.60) $\mathbf{case}\;\mathbf{inj}_{l_i}^{\{\!|l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau|\!\}}\;v\;\mathbf{of}\;(l'_j\;x_j \Rightarrow e_j)^{j \in \{1 \ldots m\}}\;\mathbf{else}\;e' \leadsto e'$ where $l_i \neq l'_k, \forall k \in \{1 \ldots m\}$. From antecedent, $\Phi;\Delta \vdash \mathbf{case}\;\ldots : \tau'$. By inversion on (4.32), $\Phi;\Delta \vdash e' : \tau'$.

Case (4.61) $\mathbf{unfold}\;(\mathbf{fold}\;v\;\mathbf{as}\;\tau\;\mathbf{at}\;\tau_s)\;\mathbf{as}\;\tau\;\mathbf{at}\;\tau_s \leadsto v$. From antecedent, $\Phi;\Delta \vdash \mathbf{unfold}\;\ldots : \tau'$. By inversion on (4.34) and (4.33), $\tau \equiv \mu\alpha{::}\kappa.\,\tau_0$, $\tau' \equiv \tau_s\;(\tau_0[\alpha := \tau])$, and $\Phi;\Delta \vdash v : \tau_s\;(\tau_0[\alpha := \tau])$.

Case (4.62) $(\Lambda\alpha{::}\kappa.\,e)\;[\tau] \leadsto e[\alpha := \tau]$. From antecedent, $\Phi;\Delta \vdash (\Lambda\alpha{::}\kappa.\,e)\;[\tau] : \tau'$. By inversion on (4.36) and (4.35), τ' must be in the form of $\tau_1[\alpha := \tau]$, and $\Phi, \alpha :: \kappa;\Delta \vdash e : \tau_1$, and $\Phi \vdash \tau :: \kappa$. Using lemma 4, $\Phi;\Delta \vdash e[\alpha := \tau] : \tau_1[\alpha := \tau]$, i.e. $\Phi;\Delta \vdash e[\alpha := \tau] : \tau'$.

Case (4.63) $\mathbf{open}\;\langle\alpha{::}\kappa = \tau', v : \tau\rangle\;\mathbf{as}\;\langle\alpha{::}\kappa, x : \tau\rangle\;\mathbf{in}\;e' \leadsto e'[\alpha := \tau'][x := v]$. From antecedent, $\Phi;\Delta \vdash \mathbf{open}\;\ldots : \tau_0$. By inversion on (4.27), $\Phi;\Delta \vdash \langle\alpha{::}\kappa = \tau',\; v : \tau\rangle : \exists\alpha{::}\kappa.\,\tau$, $\Phi, \alpha :: \kappa;\Delta, x : \tau \vdash e' : \tau_0$, and

$\Phi \vdash \tau_0 :: \mathsf{Type}$. By inversion on (4.26), $\Phi;\Delta \vdash v : \tau[\alpha := \tau']$. Using lemma 4, $\Phi;\Delta, x : \tau[\alpha := \tau'] \vdash e'[\alpha := \tau'] : \tau_0[\alpha := \tau']$. Using lemma 3, $\Phi;\Delta \vdash e'[\alpha := \tau'][x := v] : \tau_0[\alpha := \tau']$. This is equivalent to τ_0 since α is not free in τ_0.

Case (4.64) $\textbf{abort}\,[\tau] \rightsquigarrow \textbf{abort}\,[\tau]$. Trivial.

Case (4.65) $e\ e_2 \rightsquigarrow e'\ e_2$ where $e \rightsquigarrow e'$. From antecedent, $\Phi;\Delta \vdash e\ e_2 : \tau$. By inversion on (4.25), $\Phi;\Delta \vdash e : \tau' \rightarrow \tau$; and $\Phi;\Delta \vdash e_2 : \tau'$. By induction hypothesis, $\Phi;\Delta \vdash e' : \tau' \rightarrow \tau$. Using (4.25), $\Phi;\Delta \vdash e'\ e_2 : \tau$.

The cases for all the remaining congruence rules (4.66–4.75) follow the same pattern: invert some typing rule, apply induction hypothesis, then apply the same rule. □

Lemma 5 (Canonical forms) *If v is a value and $\Phi;\Delta \vdash v : \tau$ then v has the canonical form given by the following table.*

τ	v
$\tau_1 \rightarrow \tau_2$	$\lambda x : \tau_1 . e$
$\{l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau'\}$	$\{l_1 = v_1, \ldots, l_n = v_n, \ldots\}$
	or $\textbf{fix}\,[l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau']\ e$
$\{\!\vert l_1 : \tau_1 ; \ldots l_n : \tau_n ; \tau' \vert\!\}$	$\textbf{inj}^{\tau}_{l_i}\ v'$
$s[\mu\alpha{::}\kappa . \tau']$	$\textbf{fold}\ v'\ \textbf{as}\ \mu\alpha{::}\kappa.\tau'\ \textbf{at}\ \lambda\gamma{::}\kappa . s[\gamma]$
$\forall\alpha{::}\kappa . \tau'$	$\Lambda\alpha{::}\kappa . e$
$\exists\alpha{::}\kappa . \tau'$	$\langle \alpha{::}\kappa = \tau'', v' : \tau' \rangle$

Proof By inspection, using lemma 2. □

Theorem 3 (Progress) *If $\Phi;\circ \vdash e : \tau$ then either e is a value or there exists an expression e' such that $e \rightsquigarrow e'$.*

Proof By induction on the derivation of $\Phi; \circ \vdash e : \tau$.

Case (4.22) impossible, since environment is empty.

Case (4.23) direct application of induction hypothesis.

Case (4.24) $\lambda x : \tau . e$ is a value.

Case (4.25) $\Phi; \Delta \vdash e_1\ e_2 : \tau$ where $\Phi; \Delta \vdash e_1 : \tau' \to \tau$. By induction hypothesis, there are three cases: (1) e_1 and e_2 are both values. Using lemma 5, e_1 must have the form $\lambda x : \tau . e$. Using (4.56), $(\lambda x : \tau . e)\ v \leadsto e[x := v]$. (2) e_1 is a value and $e_2 \leadsto e_2'$; use (4.66). (3) $e_1 \leadsto e_1'$; use (4.65).

Case (4.26) $\Phi; \Delta \vdash \langle \alpha :: \kappa = \tau',\ e : \tau \rangle : \exists \alpha :: \kappa . \tau$. By induction hypothesis, there are two cases: (1) e is a value; then $\langle \alpha :: \kappa = \tau',\ e : \tau \rangle$ is a value. (2) $e \leadsto e'$; then use (4.74).

Case (4.27) $\Phi; \Delta \vdash \textsf{open}\ e\ \textsf{as}\ \langle \alpha :: \kappa,\ x : \tau \rangle\ \textsf{in}\ e' : \tau'$. By induction hypothesis, there are two cases: (1) e is a value of type $\exists \alpha :: \kappa . \tau$. By lemma 5, it has the form $\langle \alpha :: \kappa = \tau',\ e : \tau \rangle$; use (4.63). (2) $e \leadsto e'$; use (4.75).

Case (4.28) $\Phi; \Delta \vdash \{l_1 = e_1 \ldots l_n = e_n\} : \{l_1 : \tau_1 \ldots l_n : \tau_n\}$. By induction hypothesis, there are two cases: (1) $e_1 \ldots e_n$ are all values; then $\{l_1 = e_1 \ldots l_n = e_n\}$ is a value. (2) $e_i \leadsto e_i'$ for some i; use (4.67).

Case (4.29) $\Phi; \Delta \vdash e.l_i : \tau_i$. By induction hypothesis, there are three cases: (1) e is a value, and by lemma 5, it has the form $\{l_1 = v_1 \ldots l_n = v_n\}$. Then, progress can be made using rule (4.57). (2) e is a value, and by lemma 5, it has the form $\textsf{fix}\,[\tau]\ e$; then use rule (4.58). (3) $e \leadsto e'$; use (4.68).

Case (4.30) $\textsf{fix}\,[\tau]\ e$ is a value.

Case (4.31) $\Phi; \Delta \vdash \textsf{inj}_{l_i}^{\tau}\ e : \tau$. By induction hypothesis, there are two cases: (1) e is a value; thus $\textsf{inj}_{l_i}^{\tau}\ e$ is a value. (2) $e \leadsto e'$; then, use (4.69).

Case (4.32) $\Phi;\Delta \vdash \textbf{case}\ e\ \textbf{of}\ (l'_j\ x_j \Rightarrow e_j)^{j\in\{1\ldots m\}}\ \textbf{else}\ e' : \tau'$. By induction hypothesis, there are two cases: (1) e is a value. According to lemma 5, it has the form $\textbf{inj}^{\tau}_{l_i}\ v$. Thus, either (4.59) or (4.60) applies. (2) $e \rightsquigarrow e'$; use (4.70).

Case (4.33) $\Phi;\Delta \vdash \textbf{fold}\ e\ \textbf{as}\ \tau\ \textbf{at}\ \tau_s : \tau_s\ \tau$. By induction hypothesis, there are two cases: (1) e is a value; then $\textbf{fold}\ e\ \textbf{as}\ \tau\ \textbf{at}\ \tau_s$ is a value. (2) $e \rightsquigarrow e'$; then use (4.71).

Case (4.34) $\Phi;\Delta \vdash \textbf{unfold}\ e\ \textbf{as}\ \mu\alpha{::}\kappa.\,\tau\ \textbf{at}\ \tau_s : \tau'$. By induction hypothesis, there are two cases: (1) e is a value of type $\tau_s\ (\mu\alpha{::}\kappa.\,\tau)$. By lemma 5 it has the form $\textbf{fold}\ v\ \textbf{as}\ \mu\alpha{::}\kappa.\,\tau\ \textbf{at}\ \tau_s$—use (4.61). (2) $e \rightsquigarrow e'$; then use (4.72).

Case (4.35) $\Lambda\alpha{::}\kappa.\,e$ is a value.

Case (4.36) $\Phi;\Delta \vdash e\ [\tau'] : \tau[\alpha := \tau']$, where $\Phi;\Delta \vdash e : \forall\alpha{::}\kappa.\,\tau$. By induction hypothesis, there are two cases: (1) e is a value. By lemma 5, it must have the form $\Lambda\alpha{::}\kappa.\,e'$; use (4.62). (2) $e \rightsquigarrow e'$; use (4.73).

Case (4.37) $\Phi;\Delta \vdash \textbf{abort}\,[\tau] : \tau$. Evaluates to $\textbf{abort}\,[\tau]$ using (4.64). □

Chapter 5

A Type-Preserving Translation

In the preceding two chapters, we formally defined a source language (Featherweight Java) and an intermediate language (Mini JFlint). This chapter is devoted to translating one to the other.

We will begin by describing and formalizing our basic object encoding in sections 5.1 and 5.2. In section 5.3, we give a type-directed translation of FJ expressions. Inheritance, overriding, and constructors are examined as part of the class encoding in section 5.4, formalized in section 5.5. Next, section 5.6 covers linking and section 5.7 discusses separate compilation. Many aspects of the translation are mutually dependent, but we believe that this ordering yields a reasonably coherent explanation. Finally, section 5.8 contains proofs of important properties, and section 5.9 considers related work.

5.1 Self application

The standard implementation of method invocation using records functions is called *self-application* (Kamin 1988). In a class-based language, the object record contains values for all the fields of the object plus a pointer to a record of functions, traditionally called the *vtable* or *method suite*. The vtable of a class is created once and shared

among all objects of the class. The functions in the vtable expect the object itself as an argument. Suppose class Point has one integer field x and one method getx to retrieve it. Ignoring types for the moment, the term $p_0 = \{\mathsf{vtab} = \{\mathsf{getx} = \lambda\mathsf{self}.\,(\mathsf{self.x})\},\ \mathsf{x} = 42\}$ could be an instance of class Point. The self-application term $p_0.\mathsf{vtab.getx}\ p_0$ invokes the method.

What type should we assign to the self argument? The typing derivation for the self-application term forces it to match the type of the object record itself.

$$\dfrac{\dfrac{\dfrac{\Phi;\Delta \vdash p_0 : \{\mathsf{vtab} : \{\mathsf{getx} : \tau \to \mathbf{int}\},\ \mathsf{x} : \mathbf{int}\}}{\Phi;\Delta \vdash p_0.\mathsf{vtab} : \{\mathsf{getx} : \tau \to \mathbf{int}\}}}{\Phi;\Delta \vdash p_0.\mathsf{vtab.getx} : \tau \to \mathbf{int}} \qquad \Phi;\Delta \vdash p_0 : \tau}{\Phi;\Delta \vdash (p_0.\mathsf{vtab.getx}\ p_0) : \mathbf{int}}$$

That is, well-typed self-application requires that p_0 have type τ where

$$\tau = \{\mathsf{vtab} : \{\mathsf{getx} : \tau \to \mathbf{int}\},\ \mathsf{x} : \mathbf{int}\}$$

Because τ appears in its own definition, the solution must involve a fixed point. The recursive types of Mini JFlint are sufficient because augmenting the code with fold and unfold annotations enables a proper typing derivation. Let the type of self in this example be

$$\tau_{pt} = \mu\mathsf{self}{::}\mathsf{Type}.\ \{\mathsf{vtab} : \{\mathsf{getx} : \mathsf{self} \to \mathbf{int}\},\ \mathsf{x} : \mathbf{int}\}$$

Then, we need to (1) insert an unfold inside the method, before accessing x, and (2) fold

entire object record.

$$p_1 = \textbf{fold}\ \{\mathsf{vtab} = \{\mathsf{getx} = \lambda \mathsf{self}{:}\tau_{pt}.\ (\textbf{unfold}\ \mathsf{self}).\mathsf{x}\},\ \mathsf{x} = 42\}$$
$$\textbf{as}\ \tau_{pt}$$

This object term is indeed well-typed. The new self-application term must unfold the object before fetching its vtable: $(\textbf{unfold}\ p_1).\mathsf{vtab.getx}\ p_1$.

This is a promising start, but now suppose that class ScaledPoint extends Point with an additional field and method. The type of an object of class ScaledPoint would be:

$$\tau_{sp} = \mu \mathsf{self}{::}\mathsf{Type}.\ \{\mathsf{vtab}{:}\{\mathsf{getx}{:}\mathsf{self}{\to}\textbf{int},\ \mathsf{gets}{:}\mathsf{self}{\to}\textbf{int}\},\ \mathsf{x}{:}\textbf{int},\ \mathsf{s}{:}\textbf{int}\}$$

How can we relate the types for objects of these two classes? More to the point, how can we make a function that expects a Point also accept a ScaledPoint? Traditional models employ *subsumption,* but (1) τ_{sp} is not a subtype of τ_{pt}, so some rearrangement is necessary; and (2) we decided to exclude subtyping from our intermediate language—we would prefer to use explicit (but erasable) type manipulations.

Java programmers distinguish the *static* and *dynamic* classes of an object. The type annotations on formal parameters, local variables, and fields all indicate the static classes of the referenced objects. The dynamic class is the one named in the **new** expression where each object is created. Static classes of a given object differ at different program points; dynamic classes are unchanging. Static classes are known at compile-time; dynamic classes are revealed at run-time only by reflection and dynamic casts.

We can implement precisely this distinction in Mini JFlint. Essentially, some prefix of the object record—corresponding to the static class—is known, while the rest of the record is hidden. As we have seen, *rows* allow one to name the suffixes of records, and *existential types* are useful for data hiding. Consider this static type of a Point object; it

uses a pair of existentially-quantified rows.

$$\tau'_{pt} = \exists \mathsf{tail}{::}\{\mathsf{f} :: \mathsf{R}^{\{\mathsf{vtab},\mathsf{x}\}},\ \mathsf{m} :: \mathsf{Type}{\Rightarrow}\mathsf{R}^{\{\mathsf{getx}\}}\}.$$
$$\mu\mathsf{self}.\ \{\mathsf{vtab} : \{\mathsf{getx} : \mathsf{self}{\rightarrow}\mathbf{int} ; \mathsf{tail}{\cdot}\mathsf{m}\ \mathsf{self}\} ; \mathsf{x} : \mathbf{int} ; \mathsf{tail}{\cdot}\mathsf{f}\}$$

The f component of the tail tuple denotes a hidden row *missing* the labels vtab and x. Subclasses of Point append new fields by packaging non-empty rows into the witness type. Similarly, tail contains a component m for appending new methods onto the vtable. In this case, the hidden component is a type function expecting the recursive self type, so that it can be propagated to method types in the dynamic class, further down the hierarchy.

Now, we can disguise objects of sub-classes to look like objects of any super class. The Point object p_1 is packaged into a term of type τ'_{pt} using the trivial witness type

$$\{\mathsf{f} = \mathsf{Abs}^{\{\mathsf{vtab},\mathsf{x}\}},\ \mathsf{m} = \lambda \mathsf{s}{::}\mathsf{Type}.\ \mathsf{Abs}^{\{\mathsf{getx}\}}\}$$

To package an object of dynamic class ScaledPoint into type τ'_{pt} we hide a non-trivial witness type, containing the new field and method:

$$\{\mathsf{f} = (\mathsf{s} : \mathbf{int} ; \mathsf{Abs}^{\{\mathsf{vtab},\mathsf{x},\mathsf{s}\}}),$$
$$\mathsf{m} = \lambda \mathsf{self}{::}\mathsf{Type}.\ (\mathsf{gets} : \mathsf{self}{\rightarrow}\mathbf{int} ; \mathsf{Abs}^{\{\mathsf{getx},\mathsf{gets}\}})\}$$

A packaged object can also be *repackaged* to match the static type of some super class—this is simply an upward cast. The code in figure 5.1 on the facing page explicitly casts q from Point to Object. Establishing the typing derivation (using the rules from chapter 4) is a good way to understand this code.

This is, in essence, the object encoding we use to compile Java. Before embarking on the formal translation, we must explore one more aspect: recursive references. Suppose

$$
\begin{array}{l}
q_o = \textbf{open}\ q\ \textbf{as}\ \langle \mathsf{tail}{::}\{\mathsf{f} :: \mathsf{R}^{\{\mathsf{vtab},\mathsf{x}\}},\ \mathsf{m} :: \mathsf{Type}{\Rightarrow}\mathsf{R}^{\{\mathsf{getx}\}}\},\ \mathsf{obj} : \ldots\rangle \\
\qquad\qquad \textbf{in}\ \langle \mathsf{tail}'{::}\{\mathsf{f} :: \mathsf{R}^{\{\mathsf{vtab}\}},\ \mathsf{m} :: \mathsf{Type}{\Rightarrow}\mathsf{R}^{\varnothing}\} = \\
\qquad\qquad\qquad \{\mathsf{f} = (\mathsf{x} : \textbf{int} ; \mathsf{tail}\cdot\mathsf{f}),\ \mathsf{m} = \lambda \mathsf{s}{::}\mathsf{Type}.\,(\mathsf{getx} : \mathsf{s}{\to}\textbf{int} ; \mathsf{tail}\cdot\mathsf{m}\ \mathsf{s})\}, \\
\qquad\qquad\qquad \mathsf{obj} : \mu \mathsf{self}{::}\mathsf{Type}.\ \{\mathsf{vtab} : \{\mathsf{tail}'\cdot\mathsf{m}\ \mathsf{self}\} ; \mathsf{tail}'\cdot\mathsf{f}\}\rangle
\end{array}
$$

Figure 5.1: Casting q from Point to Object

the Point class has also a method bump which returns a new Point. The type of objects of class Point must then refer to the type of objects of class Point. This recursive reference calls for another fixed point, *outside* the existential:

$$
\begin{array}{l}
\mu \mathsf{twin}.\ \exists \mathsf{tail}.\ \mu \mathsf{self}.\ \{\mathsf{vtab} : \{\mathsf{getx} : \mathsf{self}{\to}\textbf{int} ; \mathsf{bump} : \mathsf{self}{\to}\mathsf{twin} ; \mathsf{tail}\cdot\mathsf{m}\ \mathsf{self}\}; \\
\qquad\qquad\qquad\qquad \mathsf{x} : \textbf{int};\ \mathsf{tail}\cdot\mathsf{f}\}
\end{array}
$$

Using self as the return type would overly constrain implementations of bump, forcing them to return objects of the same dynamic class as the receiver. In Java, type signatures constrain static classes only. Because twin is outside the existential, its witness type can be distinct from that of self.

This technique explains self-references, but Java supports mutually recursive references as well. Suppose class A defines a method returning an object of class B, and vice versa; ignoring fields entirely for a moment, define the following type:

$$
\begin{array}{l}
\mathsf{AB} \equiv \mu \mathsf{w}{::}\{\mathsf{A} :: \mathsf{Type},\ \mathsf{B} :: \mathsf{Type}\}. \\
\qquad\qquad \{\mathsf{A} = \exists \mathsf{tail}{::}\mathsf{Type}{\Rightarrow}\mathsf{R}^{\{\mathsf{getb}\}}.\ \mu \mathsf{self}{::}\mathsf{Type}.\ \{\mathsf{getb} : \mathsf{self}{\to}\mathsf{w}\cdot\mathsf{B} ; \mathsf{tail}\ \mathsf{self}\}, \\
\qquad\qquad\ \ \mathsf{B} = \exists \mathsf{tail}{::}\mathsf{Type}{\Rightarrow}\mathsf{R}^{\{\mathsf{geta}\}}.\ \mu \mathsf{self}{::}\mathsf{Type}.\ \{\mathsf{geta} : \mathsf{self}{\to}\mathsf{w}\cdot\mathsf{A} ; \mathsf{tail}\ \mathsf{self}\}\}
\end{array}
$$

Using the contextual fold and unfold described earlier, objects of class A can be folded into the type $\mathsf{AB}\cdot\mathsf{A}$. This is the natural generalization of the twin fixed point. In the most general case, any class can refer to any other, so w must expand to include all classes; this is the technique we use in the formal translation. In an actual compiler, we would

$$\frac{}{\textit{fieldvec}(\mathsf{Object}) = [(\mathsf{vtab}, vt)]} \tag{5.1}$$

$$\frac{CT(\mathsf{C}) = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{B}\,\{\ \mathsf{D}_1\ \mathsf{f}_1; \ldots \mathsf{D}_m\ \mathsf{f}_m;\ \mathsf{K} \ldots\ \}}{\textit{fieldvec}(\mathsf{C}) = \textit{fieldvec}(\mathsf{B}) \mathbin{+\!\!+} [(\mathsf{f}_1, \mathsf{D}_1) \ldots (\mathsf{f}_m, \mathsf{D}_m)]} \tag{5.2}$$

$$\frac{}{\textit{methvec}(\mathsf{Object}) = [(\mathsf{dyncast}, dc)]} \tag{5.3}$$

$$\frac{CT(\mathsf{C}) = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{B}\,\{\ \ldots \mathsf{K}\ \mathsf{M}_1 \ldots \mathsf{M}_m\ \}}{\textit{methvec}(\mathsf{C}) = \textit{methvec}(\mathsf{B}) \mathbin{+\!\!+} \textit{addmeth}(\mathsf{B}, [\mathsf{M}_1 \ldots \mathsf{M}_m])} \tag{5.4}$$

$$\frac{(\mathsf{m}, _) \in \textit{methvec}(\mathsf{B})}{\begin{array}{l}\textit{addmeth}(\mathsf{B}, [\mathsf{D}\ \mathsf{m}\ (\mathsf{D}_1\ \mathsf{x}_1 \ldots \mathsf{D}_k\ \mathsf{x}_k)\,\{\ \textbf{return}\ \mathsf{e};\ \}\ \mathsf{M}_2 \ldots \mathsf{M}_m]) = \\ \quad \textit{addmeth}(\mathsf{B}, [\mathsf{M}_2 \ldots \mathsf{M}_m])\end{array}} \tag{5.5}$$

$$\frac{(\mathsf{m}, _) \notin \textit{methvec}(\mathsf{B})}{\begin{array}{l}\textit{addmeth}(\mathsf{B}, [\mathsf{D}\ \mathsf{m}\ (\mathsf{D}_1\ \mathsf{x}_1 \ldots \mathsf{D}_k\ \mathsf{x}_k)\,\{\ \textbf{return}\ \mathsf{e};\ \}\ \mathsf{M}_2 \ldots \mathsf{M}_m]) = \\ \quad [(\mathsf{m}, \mathsf{D}_1 \ldots \mathsf{D}_k \to \mathsf{D})] \mathbin{+\!\!+} \textit{addmeth}(\mathsf{B}, [\mathsf{M}_2 \ldots \mathsf{M}_m])\end{array}} \tag{5.6}$$

$$\frac{}{\textit{addmeth}(\mathsf{B}, [\,]) = [\,]} \tag{5.7}$$

Figure 5.2: Field and method layouts for object types

analyze the reference graph and cluster the strongly-connected classes only. Note that this only addresses the typing aspect; mutual recursion has term-level implications also—any class can cast to or construct objects of any other; see section 5.4.

5.2 Type translation

This completes our informal account of self application; we now turn to a formal translation of FJ types. Figure 5.2 defines several functions which govern the layout of fields and methods in object types. Square brackets $[\cdot]$ denote sequences. The sequence

$s_1 +\!\!+\, s_2$ is the concatenation of sequences s_1 and s_2. $|s|$ denotes the number of elements in s. The domain of a sequence of pairs, $\mathrm{dom}(s)$, is a set consisting of the first elements of each pair in s.

The function *fieldvec* maps a class name C to a sequence of tuples of the form (f, D), indicating a field of type D named f—except for the first tuple in the sequence, which is always (vtab, vt), a placeholder for the vtable. Each class simply appends its own fields onto the sequence of fields from its super class. (In FJ, the fields of a class are assumed to be distinct from those of its super classes.)

The layout of methods in an object type is somewhat trickier. Methods that appear in a class definition are either *new* or they *override* methods in the super class. Overriding methods do not deserve a new slot in the vtable. The function *methvec* maps a class name C to a sequence of tuples of the form (m, T), indicating a method named m with signature T. Signatures have the form $\mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D}$. The function *addmeth* iterates through all the methods defined in the class C, adding only those methods that are *new* (not just overridden). The first tuple in *methvec* is always $(\mathsf{dyncast}, dc)$, a placeholder for the special polymorphic method used to implement dynamic casts.

Let *cn* denote the set of class names (including Object) in some program of interest. For the purpose of presentation, we abbreviate the kind of a tuple of all object types as *kcn*. The tuple of row kinds for class C is abbreviated $\mathit{ktail}[\mathsf{C}]$.

$$\begin{aligned} kcn &\equiv \{(\mathsf{E} :: \mathsf{Type})^{\,\mathsf{E} \in cn}\} \\ \mathit{ktail}[\mathsf{C}] &\equiv \{\mathsf{m} :: \mathsf{Type} \Rightarrow \mathsf{R}^{\mathrm{dom}(\mathit{methvec}(\mathsf{C}))}, \mathsf{f} :: \mathsf{R}^{\mathrm{dom}(\mathit{fieldvec}(\mathsf{C}))}\} \end{aligned}$$

For brevity, we sometimes omit kind annotations. By convention, certain named type variables are bound by particular kinds—w has kind *kcn*, self and u have kind Type, and tail has kind $\mathit{ktail}[\mathsf{C}]$, where C should be evident from the context. This is just a convention of notation, and does not imply that the names of type variable are special.

$$\frac{}{\mathit{Rows}[\mathsf{C},\mathsf{C}] = \lambda \mathsf{w}{::}\mathit{kcn}.\, \lambda \mathsf{u}{::}\mathsf{Type}.\, \lambda \mathsf{tail}{::}\mathit{ktail}[\mathsf{C}].\, \mathsf{tail}} \tag{5.8}$$

$$\frac{}{\begin{array}{l}\mathit{Rows}[\mathsf{Object},\top] = \lambda \mathsf{w}{::}\mathit{kcn}.\, \lambda \mathsf{u}{::}\mathsf{Type}.\, \lambda \mathsf{tail}{::}\mathit{ktail}[\mathsf{Object}].\\ \quad \{\mathsf{m} = \lambda \mathsf{self}{::}\mathsf{Type}.\, (\mathsf{dyncast} : \mathsf{self} \to \forall \alpha{::}\mathsf{Type}.\, (\mathsf{u} \to \mathbf{maybe}\ \alpha) \to \mathbf{maybe}\ \alpha\,;\\ \qquad\qquad\qquad \mathsf{tail}\cdot\mathsf{m}\ \mathsf{self})\\ \quad\ \ \mathsf{f}\ \ = \mathsf{tail}\cdot\mathsf{f}\}\end{array}} \tag{5.9}$$

$$\frac{\begin{array}{l}\mathit{CT}(\mathsf{C}) = \mathbf{class}\ \mathsf{C}\ \mathbf{extends}\ \mathsf{B}\,\{\ \mathsf{D}_1\ \mathsf{f}_1 \ldots \mathsf{D}_n\ \mathsf{f}_n\ \mathsf{K}\ \mathsf{M}_1 \ldots \mathsf{M}_m\ \}\\ \mathit{Rows}[\mathsf{B},\mathsf{A}] = \tau \qquad \mathit{addmeth}(\mathsf{B}, [\mathsf{M}_1 \ldots \mathsf{M}_m]) = [(l_1, T_1) \ldots (l_m, T_m)]\end{array}}{\begin{array}{l}\mathit{Rows}[\mathsf{C},\mathsf{A}] = \lambda \mathsf{w}{::}\mathit{kcn}.\, \lambda \mathsf{u}{::}\mathsf{Type}.\, \lambda \mathsf{tail}{::}\mathit{ktail}[\mathsf{C}].\\ \quad \tau\ \mathsf{w}\ \mathsf{u}\ \{\mathsf{m} = \lambda \mathsf{self}{::}\mathsf{Type}.\, (l_1 : \mathit{Ty}[\mathsf{self};\mathsf{w};T_1]\,;\ \ldots l_m : \mathit{Ty}[\mathsf{self};\mathsf{w};T_m]\,;\\ \qquad\qquad\qquad\qquad \mathsf{tail}\cdot\mathsf{m}\ \mathsf{self}),\\ \qquad\qquad \mathsf{f}\ \ = (\mathsf{f}_1 : \mathsf{w}\cdot\mathsf{D}_1\,;\ \ldots \mathsf{f}_n : \mathsf{w}\cdot\mathsf{D}_n\,; \mathsf{tail}\cdot\mathsf{f})\}\end{array}} \tag{5.10}$$

$$\frac{}{\mathit{Ty}[\mathsf{self};\mathsf{w};\mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D}] = \mathsf{self} \to \mathsf{w}\cdot\mathsf{D}_1 \to \ldots \mathsf{w}\cdot\mathsf{D}_n \to \mathsf{w}\cdot\mathsf{D}} \tag{5.11}$$

Figure 5.3: Definition of rows

$$\begin{array}{lll}\mathit{Empty}[\mathsf{C}] & \equiv & \{\mathsf{m} = \lambda \mathsf{self}{::}\mathsf{Type}.\, \mathsf{Abs}^{\mathrm{dom}(\mathit{methvec}(\mathsf{C}))}, \mathsf{f} = \mathsf{Abs}^{\mathrm{dom}(\mathit{fieldvec}(\mathsf{C}))}\}\\ \mathit{ObjRcd}[\mathsf{C}] & \equiv & \lambda \mathsf{w}{::}\mathit{kcn}.\, \lambda \mathsf{u}{::}\mathsf{Type}.\, \lambda \mathsf{tail}{::}\mathit{ktail}[\mathsf{C}].\, \lambda \mathsf{self}{::}\mathsf{Type}.\\ & & \quad \{\mathsf{vtab} : \{(\mathit{Rows}[\mathsf{C},\top]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail})\cdot\mathsf{m}\ \mathsf{self}\}\,;\ (\mathit{Rows}[\mathsf{C},\top]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail})\cdot\mathsf{f}\,\}\\ \mathit{SelfTy}[\mathsf{C}] & \equiv & \lambda \mathsf{w}{::}\mathit{kcn}.\, \lambda \mathsf{u}{::}\mathsf{Type}.\, \lambda \mathsf{tail}{::}\mathit{ktail}[\mathsf{C}].\, \mu \mathsf{self}{::}\mathsf{Type}.\, \mathit{ObjRcd}[\mathsf{C}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}\ \mathsf{self}\\ \mathit{ObjTy}[\mathsf{C}] & \equiv & \lambda \mathsf{w}{::}\mathit{kcn}.\, \lambda \mathsf{u}{::}\mathsf{Type}.\, \exists \mathsf{tail}{::}\mathit{ktail}[\mathsf{C}].\, \mathit{SelfTy}[\mathsf{C}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}\\ \mathit{World} & \equiv & \lambda \mathsf{u}{::}\mathsf{Type}.\, \mu \mathsf{w}{::}\mathit{kcn}.\, \{(\mathsf{E} = \mathit{ObjTy}[\mathsf{E}]\ \mathsf{w}\ \mathsf{u})^{\,\mathsf{E}\in\mathit{cn}}\,\}\end{array}$$

Figure 5.4: Macros for object types

In figure 5.3 we define *Rows*, a type operator that produces rows containing the fields and methods introduced *between* two classes in a subclass relationship. Intuitively, $\mathit{Rows}[\mathsf{C},\mathsf{A}]$ includes fields and methods in class C but *not* in its ancestor class A. Earlier we described how to package dynamic classes into static classes; the witness type was a tuple of rows containing the fields and methods in the dynamic class but not in the static class. This is just one use of the *Rows* operator.

The type operator $\mathit{Rows}[\mathsf{C},\mathsf{A}]$ has kind $\mathit{kcn} \Rightarrow \mathsf{Type} \Rightarrow \mathit{ktail}[\mathsf{C}] \Rightarrow \mathit{ktail}[\mathsf{A}]$; recall that $\Rightarrow$

is the kind-level arrow in Mini JFlint. The operator's first argument, w::*kcn*, is a tuple containing object types for all classes in the compilation unit. The next argument, u::Type, is a *universal* type used to implement dynamic casts. This will be explained in section 5.4; for now, we only observe that the definitions in figure 5.3 simply propagate u so that it can appear in the type of the dyncast pseudo-method. The final argument, tail::*ktail*[C], contains the rows for some subclass of C.

Rows[C, A] is defined by three cases. First, if C and A are the same class, then the result is just the tail—those members in subclasses of C. Second, if C is Object (the root of the class hierarchy) and A is the special symbol $\top$ then the result is the members declared in Object. Treating $\top$ as the trivial super class of Object permits more uniform specifications (since Object contains members of its own). Finally, in the inductive case (where C <: A) we look to C's super class—let's call it B. *Rows*[B, A] produces a type operator for the members between B and A; we need only append the *new* members of C. Conveniently, *Rows*[B, A] has a tail parameter specifically for appending new members. The formal definition of *Rows* is in figure 5.3.

The new fields in C are precisely those listed in the declaration of C; we fetch their types from w and append tail·f. The new methods in C are found using *addmeth*, and their type signatures $D_1 \ldots D_n \to D$ are translated to arrow types: self$\to$ w·$D_1 \to \ldots$ w·$D_n \to$ w·D. We use curried arguments for convenience; an actual implementation would use multi-argument functions instead. As shown in the informal examples, the row for methods is parameterized by the type of self. As a concrete example, the rows for Point and ScaledPoint are in figure 5.5 on the next page.

In figure 5.4, we use the *Rows* operator to define macros for several variants of the object type for any given class. *Empty*[C] denotes the tuple of empty field and method rows of kind *ktail*[C]. *ObjRcd*[C] assembles the rows into records, leaving the subclass rows and self type open. *SelfTy*[C] closes self with a fixed point, and *ObjTy*[C] hides the sublass rows with an existential. Each of these variants is used in our term

$$
\begin{aligned}
\mathit{Rows}[\mathbf{P},\mathsf{Object}] &= \lambda \mathsf{w}.\, \lambda \mathsf{u}.\, \lambda \mathsf{tail}.\, \{\mathsf{m} = \lambda \mathsf{self}.\, (\mathsf{getx} : \mathsf{self} \to \mathbf{int} ; \mathsf{tail} \cdot \mathsf{m}\ \mathsf{self}), \\
&\qquad\qquad\qquad\quad \mathsf{f} = (\mathsf{x} : \mathbf{int} ; \mathsf{tail} \cdot \mathsf{f})\} \\
\mathit{Rows}[\mathbf{SP},\mathbf{P}] &= \lambda \mathsf{w}.\, \lambda \mathsf{u}.\, \lambda \mathsf{tail}.\, \{\mathsf{m} = \lambda \mathsf{self}.\, (\mathsf{gets} : \mathsf{self} \to \mathbf{int} ; \mathsf{tail} \cdot \mathsf{m}\ \mathsf{self}), \\
&\qquad\qquad\qquad\quad \mathsf{f} = (\mathsf{s} : \mathbf{int} ; \mathsf{tail} \cdot \mathsf{f})\} \\
\mathit{Rows}[\mathbf{SP},\mathsf{Object}] &= \lambda \mathsf{w}.\, \lambda \mathsf{u}.\, \lambda \mathsf{tail}.\, \mathit{Rows}[\mathbf{P},\mathsf{Object}]\ \mathsf{w}\ (\mathit{Rows}[\mathbf{SP},\mathbf{P}]\ \mathsf{w}\ \mathsf{tail}) \\
&= \lambda \mathsf{w}.\, \lambda \mathsf{u}.\, \lambda \mathsf{tail}.\, \{\mathsf{m} = \lambda \mathsf{self}.\, (\mathsf{getx} : \mathsf{self} \to \mathbf{int} ; \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{gets} : \mathsf{self} \to \mathbf{int} ; \mathsf{tail} \cdot \mathsf{m}\ \mathsf{self}), \\
&\qquad\qquad\qquad\quad \mathsf{f} = (\mathsf{x} : \mathbf{int} ; \mathsf{s} : \mathbf{int} ; \mathsf{tail} \cdot \mathsf{f})\}
\end{aligned}
$$

Figure 5.5: Rows for Point and ScaledPoint

$$
\begin{aligned}
&\textsc{pack}[\mathsf{C}; \mathsf{u}; \mathsf{tail}; e] = \\
&\quad \mathbf{fold}\ \langle \mathsf{tail}' :: \mathit{ktail}[\mathsf{C}] = \mathsf{tail},\ e : \mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}' \rangle \\
&\qquad \mathbf{as}\ \mathit{World}\ \mathsf{u}\ \mathbf{at}\ \lambda \gamma :: \mathit{kcn}.\, \gamma \cdot \mathsf{C}
\end{aligned}
$$

$$
\begin{aligned}
&\textsc{upcast}[\mathsf{C}; \mathsf{A}; \mathsf{u}; e] = \\
&\quad \mathbf{open}\ (\mathbf{unfold}\ e\ \mathbf{as}\ \mathit{World}\ \mathsf{u}\ \mathbf{at}\ \lambda \gamma :: \mathit{kcn}.\, \gamma \cdot \mathsf{C}) \\
&\qquad \mathbf{as}\ \langle \mathsf{tail} :: \mathit{ktail}[\mathsf{C}],\ x : \mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail} \rangle \\
&\qquad \mathbf{in}\ \textsc{pack}[\mathsf{A}; \mathsf{u}; \mathit{Rows}[\mathsf{C},\mathsf{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}; x]
\end{aligned}
$$

Figure 5.6: Definitions of pack and upcast transformations

translation. All of them remain abstracted over both w (the types of other objects) and u (the universal type, which is simply propagated into the type of dyncast). Finally, *World* constructs a package of the types of objects of all classes, given the universal type u; as we will see later, the actual universal type is a labeled sum of object types, and is defined recursively using *World*.

5.3 Expression translation

Equipped with an efficient object encoding and several type operators for describing it, we now examine the type-directed translation of FJ expressions. Figures 5.6 and 5.7 contain definitions of PACK and UPCAST, and six rules governing the judgment $\textsc{exp}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathsf{e}] = e$ for term translation. Here, Γ is the FJ type environment, u is the

$$\frac{}{\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathsf{x}] = \mathsf{x}} \tag{5.12}$$

$$\frac{(\mathsf{f},_) \in \mathit{fieldvec}(\mathsf{C}) \qquad \Gamma \vdash \mathsf{e} \in \mathsf{C} \qquad \textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathsf{e}] = e}{\begin{array}{l}\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathsf{e.f}] = \\ \quad \mathbf{open}\ (\mathbf{unfold}\ e\ \mathbf{as}\ \mathit{World}\ \mathsf{u}\ \mathbf{at}\ \lambda y{::}kcn.\ y\cdot\mathsf{C}) \\ \quad\quad \mathbf{as}\ \langle \mathsf{tail}{::}\mathit{ktail}[\mathsf{C}],\ x : \mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}\rangle \\ \quad\quad \mathbf{in}\ (\mathbf{unfold}\ x).\mathsf{f}\end{array}} \tag{5.13}$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathsf{e} \in \mathsf{C} \qquad (\mathsf{m}, \mathsf{B}_1 \ldots \mathsf{B}_n \to \mathsf{B}) \in \mathit{methvec}(\mathsf{C}) \qquad \textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathsf{e}] = e \\ \Gamma \vdash \mathsf{e}_i \in \mathsf{D}_i \qquad \mathsf{D}_i <: \mathsf{B}_i \\ \textsc{upcast}[\mathsf{D}_i;\mathsf{B}_i;\mathsf{u};\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathsf{e}_i]] = e_i, \quad i \in \{1..n\}\end{array}}{\begin{array}{l}\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathsf{e.m}\ (\mathsf{e}_1 \ldots \mathsf{e}_n)] = \\ \quad \mathbf{open}\ (\mathbf{unfold}\ e\ \mathbf{as}\ \mathit{World}\ \mathsf{u}\ \mathbf{at}\ \lambda y{::}kcn.\ y\cdot\mathsf{C}) \\ \quad\quad \mathbf{as}\ \langle \mathsf{tail}{::}\mathit{ktail}[\mathsf{C}],\ x : \mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}\rangle \\ \quad\quad \mathbf{in}\ (\mathbf{unfold}\ x).\mathsf{vtab.m}\ x\ e_1\ \ldots\ e_n\end{array}} \tag{5.14}$$

$$\frac{\begin{array}{l}\mathit{fields}(\mathsf{C}) = \mathsf{B}_1\ \mathsf{f}_1 \ldots \mathsf{B}_n\ \mathsf{f}_n \\ \Gamma \vdash \mathsf{e}_i \in \mathsf{D}_i \qquad \mathsf{D}_i <: \mathsf{B}_i \\ \textsc{upcast}[\mathsf{D}_i;\mathsf{B}_i;\mathsf{u};\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathsf{e}_i]] = e_i, \quad i \in \{1..n\}\end{array}}{\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathbf{new}\ \mathsf{C}\ (\mathsf{e}_1 \ldots \mathsf{e}_n)] = (\mathsf{classes.C}\ \{\}).\mathsf{new}\ e_1\ \ldots\ e_n} \tag{5.15}$$

$$\frac{\Gamma \vdash \mathsf{e} \in \mathsf{D} \qquad \mathsf{D} <: \mathsf{C}}{\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};(\mathsf{C})\ \mathsf{e}] = \textsc{upcast}[\mathsf{D};\mathsf{C};\mathsf{u};\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathsf{e}]]} \tag{5.16}$$

$$\frac{\Gamma \vdash \mathsf{e} \in \mathsf{C} \qquad \mathsf{D} <: \mathsf{C} \qquad \textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathsf{e}] = e}{\begin{array}{l}\textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};(\mathsf{D})\ \mathsf{e}] = \\ \quad \mathbf{open}\ (\mathbf{unfold}\ e\ \mathbf{as}\ \mathit{World}\ \mathsf{u}\ \mathbf{at}\ \lambda y{::}kcn.\ y\cdot\mathsf{C}) \\ \quad\quad \mathbf{as}\ \langle \mathsf{tail}{::}\mathit{ktail}[\mathsf{C}],\ x : \mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}\rangle \\ \quad\quad \mathbf{in\ case}\ (\mathbf{unfold}\ x).\mathsf{vtab.dyncast}\ x\ [(\mathit{World}\ \mathsf{u})\cdot\mathsf{D}]\ (\mathsf{classes.D}\ \{\}).\mathsf{proj} \\ \quad\quad\quad \mathbf{of}\ \mathsf{some}\ y \Rightarrow y\ \mathbf{else\ abort}\ [(\mathit{World}\ \mathsf{u})\cdot\mathsf{D}]\end{array}} \tag{5.17}$$

Figure 5.7: Type-directed translation of FJ expressions

universal sum type, classes is a record containing the runtime representations of each class, e is an FJ expression, and e is its corresponding term in the target language. If e has type C, then its translation e has type $(World\ \mathsf{u})\cdot\mathsf{C}$ (see theorem 4 in section 5.8).

The PACK operation packages and folds a recursive record term into a closed, complete object whose type is selected from a mutual fixed point of the types of objects of all classes. Suppose that tail is some row tuple in $ktail[\mathsf{C}]$ and e has type:

$$SelfTy[\mathsf{C}]\ (World\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}$$

Then, the term $\text{PACK}[\mathsf{C};\mathsf{u};\mathsf{tail};e]$ has type $(World\ \mathsf{u})\cdot\mathsf{C}$. Since *unpacking* an object binds a type variable to the hidden witness type, it is not as convenient to define as a macro, and we perform it inline instead.

The UPCAST operation opens a term representing an object of class C and repackages it as a term representing an object of some super class A. The object term e has type $(World\ \mathsf{u})\cdot\mathsf{C}$ where $\mathsf{C} <: \mathsf{A}$, but dynamically it might belong to some subclass $\mathsf{D} <: \mathsf{C}$. The **open** binds the type variable tail to the hidden row types that represent members in D but not in C. The UPCAST macro then uses *Rows* to prefix tail with the types of members in C but not in A. Finally, UPCAST uses PACK to hide the new tail, yielding an object term of type $(World\ \mathsf{u})\cdot\mathsf{A}$.

These definitions simply and effectively formalize the encoding techniques demonstrated in the previous section. Importantly, they use type manipulations only (**fold**, **unfold**, **open**). Since these operations are erased before runtime, the PACK and UPCAST transformations have no impact on performance.

We now explain each of the translation rules in figure 5.7, beginning with (5.12). Variables in FJ are bound as method arguments. Methods are translated as curried abstractions binding the *same* variable names. Therefore, variable translation (5.12) is trivial. An upcast expression (C) e (where $\Gamma \vdash \mathsf{e} \in \mathsf{D}$ and $\mathsf{D} <: \mathsf{C}$) is also trivial; the rule

(5.16) delegates its task to the macro of the same name.

The field selection expression e.f translates to an unfold-open-unfold-select idiom in the target language (5.13). In this sequence, the select alone has runtime effect. Method invocation e.m $(e_1 \ldots e_n)$ augments the idiom with applications to self and the other arguments, but there is one complication. The FJ typing rule permits the actual arguments to have types that are subclasses of the types in the method signature. Since our encoding does not utilize subtyping, the function selected from the vtable expects arguments of precisely the types in the method signature. Therefore, we must explicitly upcast all arguments. Rule (5.14) formalizes the self-application technique demonstrated earlier.

The code to create a new object of class C essentially selects and applies C's constructor from the classes record. Until we explain class encoding and linking, the type of classes will be difficult to justify. Presently it will suffice to say that classes.C applied to the unit value {} returns a record which contains a field new—the constructor for class C. The translation (5.15) upcasts all the arguments, then fetches and applies the constructor.

The final case, dynamic casts, may appear quite magical until we reveal the implementation of the dyncast pseudo-method in the next section. For now it is enough to treat dyncast as a function of type $\mathsf{self} \to \forall \alpha.\, (\mathsf{u} \to \mathbf{maybe}\ \alpha) \to \mathbf{maybe}\ \alpha$, where self is the type of the unfolded unpacked object bound to x. The argument of

$$(\mathbf{unfold}\ x).\mathsf{vtab.dyncast}\ x\ [\tau]$$

is a *projection* function, attempting to convert a value of type u to an object of type τ. The record classes.C {} contains, in addition to the field new, a proj field of type $\mathsf{u} \to \mathbf{maybe}\ ((\mathit{World}\ \mathsf{u}) \cdot \mathsf{C})$. Thus if we select the dyncast method from an object, instantiate it with the object type for some class C, then pass it the projection for class C, it will return **some** C object if the cast succeeds, or **none** if it fails. In case of failure,

evaluation aborts. In full Java, we would throw a ClassCast exception.

Note that Featherweight Java's *stupid* casts (Igarashi, Pierce, and Wadler 2001) are not compiled at all. They arise in intermediate results during evaluation, but should not appear in valid source-level programs.

The expression translation judgment EXP *preserves types*. Informally, if e has type C, then its translation has type $(\mathit{World}\ \mathsf{u})\cdot\mathsf{C}$, for some type u. The type preservation theorem is stated formally and proved in section 5.8.

5.4 Class encoding

Apart from defining types, classes in FJ serve three other roles: they are extended, invoked to create new objects, and specified as targets of dynamic casts. In our translation, each class declaration is separately compiled into a module exporting a record with three elements—one to address each of these roles. We informally explain our techniques for implementing inheritance, constructors, and dynamic casts, then give the formal translation of class declarations.

In a class-based language, each vtable is constructed once and shared among all objects of the same class. In addition, the code of each inherited method should be shared by all inheritors. How might we implement the Point methods so that they can be packaged with a ScaledPoint? We make the method record polymorphic over the tail of the self type:

$$\mathsf{dictPT} = \Lambda\mathsf{tail}{::}\mathit{ktail}[\mathsf{PT}].\ \{\mathsf{getx} = \lambda\mathsf{self}{:}s_{pt}.\ (\mathbf{unfold}\ \mathsf{self}).\mathsf{x}\}$$

$$\text{where } s_{pt} = \mu\alpha.\ \{\mathsf{vtab}{:}\{\mathsf{getx}{:}\alpha{\rightarrow}\mathbf{int};\mathsf{tail}\cdot\mathsf{m}\ \alpha\};\ \mathsf{x}{:}\mathbf{int};\mathsf{tail}\cdot\mathsf{f}\}$$

We call this polymorphic record a *dictionary*. By instantiating it with different tails, we can directly package its contents into objects of subclasses. Instantiated with empty tails ($\mathit{Empty}[\mathsf{PT}]$, for example), this dictionary becomes a vtable for class Point. Suppose

$$
\begin{array}{lll}
\mathit{Dict}[\mathsf{C}] & \equiv & \lambda \mathsf{w}{::}\mathit{kcn}.\, \lambda \mathsf{u}{::}\mathsf{Type}.\, \lambda \mathsf{self}{::}\mathsf{Type}.\, \{(\mathit{Rows}[\mathsf{C}, \top]\ \mathsf{w}\ \mathsf{u}\ \mathit{Empty}[\mathsf{C}]) \cdot \mathsf{m}\ \mathsf{self}\} \\
\mathit{Ctor}[\mathsf{C}] & \equiv & \lambda \mathsf{w}{::}\mathit{kcn}.\, \mathsf{w}\cdot\mathsf{D}_1 \rightarrow \ldots \mathsf{w}\cdot\mathsf{D}_n \rightarrow \mathsf{w}\cdot\mathsf{C} \\
 & & \text{where } \mathit{fields}(\mathsf{C}) = \mathsf{D}_1\ \mathsf{f}_1 \ldots \mathsf{D}_n\ \mathsf{f}_n \\
\mathit{Proj}[\mathsf{C}] & \equiv & \lambda \mathsf{w}{::}\mathit{kcn}.\, \lambda \mathsf{u}{::}\mathsf{Type}.\, \mathsf{u} \rightarrow \mathbf{maybe}\ \mathsf{w}\cdot\mathsf{C} \\
\mathit{Inj}[\mathsf{C}] & \equiv & \lambda \mathsf{w}{::}\mathit{kcn}.\, \lambda \mathsf{u}{::}\mathsf{Type}.\, \mathsf{w}\cdot\mathsf{C} \rightarrow \mathsf{u} \\
\mathit{Class}[\mathsf{C}] & \equiv & \lambda \mathsf{w}{::}\mathit{kcn}.\, \lambda \mathsf{u}{::}\mathsf{Type}.\, \{\mathsf{dict} : \forall \mathsf{tail}{::}\mathit{ktail}[\mathsf{C}].\, \mathit{Dict}[\mathsf{C}]\ \mathsf{w}\ \mathsf{u}\ (\mathit{SelfTy}[\mathsf{C}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}), \\
 & & \qquad \mathsf{proj} : \mathit{Proj}[\mathsf{C}]\ \mathsf{w}\ \mathsf{u},\ \ \mathsf{new} : \mathit{Ctor}[\mathsf{C}]\ \mathsf{w}\,\} \\
 & & \\
\mathit{Classes} & \equiv & \lambda \mathsf{w}{::}\mathit{kcn}.\, \lambda \mathsf{u}{::}\mathsf{Type}.\, ((\mathsf{E} : \mathbf{1} \rightarrow \mathit{Class}[\mathsf{E}]\ \mathsf{w}\ \mathsf{u}\,;)^{\mathsf{E} \in \mathit{cn}}\ \mathsf{Abs}^{\mathit{cn}}) \\
\mathit{ClassF}[\mathsf{C}] & \equiv & \forall \mathsf{u}{::}\mathsf{Type}.\, \mathit{Inj}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u} \rightarrow \mathit{Proj}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u} \rightarrow \\
 & & \qquad \{\mathit{Classes}\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\} \rightarrow \mathbf{1} \rightarrow \mathit{Class}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}
\end{array}
$$

Figure 5.8: Macros for dictionary, constructor, and class types

the ScaledPoint subclass inherits getx and adds a method of its own. Its dictionary would be:

$$
\begin{array}{l}
\mathsf{dictSP} = \Lambda \mathsf{tail}{::}\mathit{ktail}[\mathsf{SP}].\, \{\mathsf{getx} = (\mathsf{dictPT}\ [r_{sp}]).\mathsf{getx}, \\
\qquad\qquad\qquad\qquad \mathsf{gets} = \lambda \mathsf{self} : s_{sp}.\, (\mathbf{unfold}\ \mathsf{self}).\mathsf{s}\} \\
\text{where } r_{sp} = \mathit{Rows}[\mathsf{SP}, \mathsf{PT}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathit{Empty}[\mathsf{SP}] \\
\quad \text{and } s_{sp} = \mu\alpha.\, \{\mathsf{vtab} : \{\mathsf{getx} : \alpha \rightarrow \mathbf{int}\,; \mathsf{gets} : \alpha \rightarrow \mathbf{int}\,; \mathsf{tail}\cdot\mathsf{m}\ \alpha\}\,; \\
\qquad\qquad\qquad \mathsf{x} : \mathbf{int}\,; \mathsf{s} : \mathbf{int}\,; \mathsf{tail}\cdot\mathsf{f}\}
\end{array}
$$

This dictionary can be instantiated with empty tails to produce the ScaledPoint vtable. With other instantiations, further subclasses can inherit either of these methods. The dictionary is labeled dict in the record exported by the class translation.

Constructors in FJ are quite simple; they take all the fields as arguments in the correct order. Fields declared in the super class are immediately passed to the super initializer. We translate the constructor as a function which takes the fields as curried arguments, places them directly into a record with the vtable, and then folds and packages the object. The constructor function is labeled new in the class record. In section 6, we describe how to implement more realistic constructors.

Implementing dynamic cast in a strongly-typed language is challenging. Somehow we must determine whether an arbitrary, abstractly-typed object belongs to a particular class. If it does belong, we must somehow refine its type to reflect this new information. Exception matching in SML poses a similar problem. To address these issues, Harper and Stone (1998) introduce *tags*—values which track type information at runtime. If a tag of abstract type $\mathsf{Tag}\ \alpha$ equals another tag of known type $\mathsf{Tag}\ \tau$, then we update the context to reflect that $\alpha = \tau$. Note that this differs from intensional type analysis (Harper and Morrisett 1995), which performs structural comparison and does not distinguish named types.

Tags work well with our encoding; in an implementation that supports assignment and an SML front-end, it may be a good choice. In this formal presentation, however, type refinement complicates the soundness proof and the imperative nature of maketag constrains the operational semantics, which is otherwise free of side effects. maketag implements a dynamically extensible sum, which is needed for SML exceptions, but is overkill for classes in FJ.

We propose a simpler approach, which co-opts the dynamic dispatch mechanism. The vtable itself provides a kind of runtime class information. A designated method, if overridden in *every* class, could return the receiver at its dynamic class or any super class. We just need a runtime representation of the target class of the cast, and some way to connect that representation to the corresponding object type. For this, we can use the standard sum type and a 'one-armed' case. Let u be a sum type with a variant for each class in the class table. The function

$$\begin{aligned}&\lambda x : u.\, \textbf{case}\ x\ \textbf{of}\ \mathsf{C}\ y \Rightarrow \textbf{some}\ [\mathit{ObjTy}[\mathsf{C}]\ (\mathit{World}\ u)\ u]\ y\\ &\qquad\quad \textbf{else none}\ [\mathit{ObjTy}[\mathsf{C}]\ (\mathit{World}\ u)\ u]\end{aligned}$$

could dynamically represent class C. To connect it to the object type, we make the

dyncast method polymorphic, with the type

$$\mathsf{self} \to \forall \alpha.\, (u \to \textbf{maybe}\ \alpha) \to \textbf{maybe}\ \alpha$$

This method can check its own class against the target class by injecting self and applying the function argument. If the result is **none**, then it tries again by injecting as the super class, and so on up the hierarchy.

With this solution, we must be careful to preserve separate compilation—the universal type u includes a variant for every class in the program. Fortunately, in a particular class declaration we need only inject objects of that class. Class declarations can treat u as an abstract type and take the injection function as an argument. Then only the linker needs to know the concrete u type.

5.5 Class translation

We now explore the formal translation of class declarations and construction of their method dictionaries. In figure 5.8 we define several macros for describing dictionary and class types. Figure 5.9 on the following page gives translations for each component of the class declaration.

Each class is separately compiled to code that resembles an SML *functor*—a set of definitions parameterized by both types and terms. Linking—the process of instantiating the separate functors and combining them into single coherent program—will be addressed in the next section. Our compilation model is the subject of section 5.7.

CDEC[C] produces the functor corresponding to class C; see the definition at the top of figure 5.9. The code has one type parameter: u, the universal type used for dynamic casts. Following it are two function parameters for injecting and projecting objects of class C. The next parameter is classes, a record containing definitions for other classes

Class declaration translation:

$$\begin{array}{l}
\textsc{cdec}[\mathsf{C}] = \\
\quad \Lambda \mathsf{u}{::}\mathsf{Type}.\, \lambda \mathsf{inj} : \mathit{Inj}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}.\, \lambda \mathsf{proj} : \mathit{Proj}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}. \\
\quad \lambda \mathsf{classes} : \{\mathit{Classes}\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\}.\, \lambda _ : \mathbf{1}. \\
\quad \mathbf{let}\ \mathsf{dict} : \forall \mathsf{tail}{::}\mathit{ktail}[\mathsf{C}].\, \mathit{Dict}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ (\mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}) \\
\qquad\qquad = \textsc{dict}[\mathsf{C}; \mathsf{u}; \mathsf{inj}; \mathsf{classes}] \\
\quad\ \mathbf{in}\ \mathbf{let}\ \mathsf{vtab} = \mathsf{dict}\ [\mathit{Empty}[\mathsf{C}]] \\
\qquad\quad \mathbf{in}\ \{\mathsf{dict} = \mathsf{dict}, \mathsf{proj} = \mathsf{proj}, \mathsf{new} = \textsc{new}[\mathsf{C}; \mathsf{u}; \mathsf{vtab}]\}
\end{array} \tag{5.18}$$

Dictionary construction:

$$\begin{array}{l}
\textsc{dict}[\mathsf{Object}; \mathsf{u}; \mathsf{inj}; \mathsf{classes}] = \Lambda \mathsf{tail}{::}\mathit{ktail}[\mathsf{Object}]. \\
\quad \{\mathsf{dyncast} = \lambda \mathsf{self} : \mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}. \\
\qquad\qquad\qquad \Lambda \alpha{::}\mathsf{Type}.\, \lambda \mathsf{proj} : \mathsf{u} \to \mathbf{maybe}\ \alpha. \\
\qquad\qquad\qquad\quad \mathsf{proj}\ (\mathsf{inj}\ \textsc{pack}[\mathsf{Object}; \mathsf{u}; \mathsf{tail}; \mathsf{self}])\}
\end{array} \tag{5.19}$$

$$\frac{CT(\mathsf{C}) = \mathbf{class}\ \mathsf{C}\ \mathbf{extends}\ \mathsf{B}\ \{\ \ldots\ \} \qquad \mathrm{dom}(\mathit{methvec}(\mathsf{C})) = [l_1 \ldots l_n]}{\begin{array}{l}
\textsc{dict}[\mathsf{C}; \mathsf{u}; \mathsf{inj}; \mathsf{classes}] = \Lambda \mathsf{tail}{::}\mathit{ktail}[\mathsf{C}]. \\
\quad \mathbf{let}\ \mathsf{super} : \mathit{Dict}[\mathsf{B}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ (\mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}) \\
\qquad\qquad = (\mathsf{classes.B}\ \{\}).\mathsf{dict}\ [\mathit{Rows}[\mathsf{C}, \mathsf{B}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}] \\
\quad \mathbf{in}\ \{l_1 = \textsc{meth}[\mathsf{C}; l_1; \mathsf{u}; \mathsf{tail}; \mathsf{inj}; \mathsf{classes}; \mathsf{super}], \ldots, \\
\qquad l_n = \textsc{meth}[\mathsf{C}; l_n; \mathsf{u}; \mathsf{tail}; \mathsf{inj}; \mathsf{classes}; \mathsf{super}]\}
\end{array}} \tag{5.20}$$

Constructor code:

$$\frac{\mathit{fields}(\mathsf{C}) = \mathsf{D}_1\ \mathsf{f}_1 \ldots \mathsf{D}_n\ \mathsf{f}_n}{\begin{array}{l}
\textsc{new}[\mathsf{C}; \mathsf{u}; \mathsf{vtab}] = \lambda \mathsf{f}_1 : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{D}_1. \ldots \lambda \mathsf{f}_n : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{D}_n. \\
\qquad \mathbf{let}\ x = \mathbf{fold}\ \{\mathsf{vtab} = \mathsf{vtab}, \mathsf{f}_1 = \mathsf{f}_1, \ldots, \mathsf{f}_n = \mathsf{f}_n\} \\
\qquad\qquad \mathbf{as}\ \mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathit{Empty}[\mathsf{C}] \\
\qquad \mathbf{in}\ \textsc{pack}[\mathsf{C}; \mathsf{u}; \mathit{Empty}[\mathsf{C}]; x]
\end{array}} \tag{5.21}$$

Figure 5.9: Translation of class declarations

Method code:

$$\frac{}{\begin{array}{l}\textsc{meth}[\mathsf{C};\mathsf{dyncast};\mathsf{u};\mathsf{tail};\mathsf{inj};\mathsf{classes};\mathsf{super}] = \\ \quad \lambda\mathsf{self}:\mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}.\ \Lambda\alpha::\mathsf{Type}.\ \lambda\mathsf{proj}:\mathsf{u}\to\mathbf{maybe}\ \alpha. \\ \quad\quad \mathbf{case}\ \mathsf{proj}\ (\mathsf{inj}\ \textsc{pack}[\mathsf{C};\mathsf{u};\mathsf{tail};\mathsf{self}]) \\ \quad\quad\quad \mathbf{of}\ \mathsf{some}\ x \Rightarrow \mathbf{some}\ [\alpha]\ x\ \ \mathbf{else}\ \mathsf{super.dyncast}\ \mathsf{self}\ [\alpha]\ \mathsf{proj}\end{array}} \tag{5.22}$$

$$\frac{\begin{array}{l}CT(\mathsf{C}) = \mathbf{class}\ \mathsf{C}\ \mathbf{extends}\ \mathsf{B}\{\ \ldots\mathsf{K}\ \mathsf{M}_1\ldots\mathsf{M}_n\ \} \\ \mathsf{m}\ \text{not defined in}\ \mathsf{M}_1\ldots\mathsf{M}_n\end{array}}{\textsc{meth}[\mathsf{C};\mathsf{m};\mathsf{u};\mathsf{tail};\mathsf{inj};\mathsf{classes};\mathsf{super}] = \mathsf{super.m}} \tag{5.23}$$

$$\frac{\begin{array}{l}CT(\mathsf{C}) = \mathbf{class}\ \mathsf{C}\ \mathbf{extends}\ \mathsf{B}\{\ \ldots\mathsf{K}\ \mathsf{M}_1\ldots\mathsf{M}_n\ \} \\ \exists j:\mathsf{M}_j = \mathsf{A}\ \mathsf{m}\ (\mathsf{A}_1\ \mathsf{x}_1\ldots\mathsf{A}_m\ \mathsf{x}_m)\{\ \mathbf{return}\ \mathsf{e};\ \} \\ \Gamma = \mathsf{x}_1{:}\mathsf{A}_1,\ldots,\mathsf{x}_m{:}\mathsf{A}_m,\mathbf{this}{:}\mathsf{C} \qquad \Gamma\vdash\mathsf{e}\in\mathsf{D} \qquad \mathsf{D} <: \mathsf{A} \\ \textsc{exp}[\Gamma;\mathsf{u};\mathsf{classes};\mathsf{e}] = e\end{array}}{\begin{array}{l}\textsc{meth}[\mathsf{C};\mathsf{m};\mathsf{u};\mathsf{tail};\mathsf{inj};\mathsf{classes};\mathsf{super}] = \\ \quad \lambda\mathsf{self}:\mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}. \\ \quad\quad \lambda\mathsf{x}_1:(\mathit{World}\ \mathsf{u})\cdot\mathsf{A}_1.\ \ldots\lambda\mathsf{x}_m:(\mathit{World}\ \mathsf{u})\cdot\mathsf{A}_m. \\ \quad\quad\quad \mathbf{let\ this}:(\mathit{World}\ \mathsf{u})\cdot\mathsf{C} = \textsc{pack}[\mathsf{C};\mathsf{u};\mathsf{tail};\mathsf{self}] \\ \quad\quad\quad \mathbf{in}\ \textsc{upcast}[\mathsf{D};\mathsf{A};\mathsf{u};e]\end{array}} \tag{5.24}$$

Figure 5.10: Translation of method declarations

that are mutually recursive with C (for convenience, we assume that each class refers to all the others). The final parameter is of unit type; it simply delays references to classes so that linking terminates.

In the functor body, we define dict (using the macro DICT) and vtab (the trivial instantiation of dict). dict is placed in the class record (so subclasses can inherit its methods); vtab is passed to the NEW macro which creates the constructor code. The constructor is exported so that other classes can create C objects; and, finally, the projection function proj (a functor parameter) is exported so other classes can dynamically cast to C.

The dictionary for class Object is hard-coded as DICT[Object; . . .]. It has a special method dyncast that injects self at class Object, passes this to the proj argument and returns the result. If the class tags do not match, dyncast indicates failure by returning **none**; there is no super class to test. For all other classes, DICT fetches the super class

$$Tagged = \lambda \mathsf{u}{::}\mathsf{Type}.\ \langle\!|\, (\mathsf{C} : (World\ \mathsf{u})\cdot\mathsf{C})^{\,\mathsf{C}\in cn} \,|\!\rangle$$
$$Univ = \mu \mathsf{u}{::}\mathsf{Type}.\ Tagged\ \mathsf{u}$$

$$\textsc{prog}[\mathsf{e}] = \textbf{let}\ x_{cn} = \textsc{link}\ \{(\mathsf{C} = \textsc{cdec}[\mathsf{C}])^{\,\mathsf{C}\in cn}\}\ \textbf{in}\ \textsc{exp}[\circ; Univ; x_{cn}; \mathsf{e}] \tag{5.25}$$

$$\begin{aligned}
\textsc{link} = {} & \lambda x : \{(\mathsf{C} : ClassF[\mathsf{C}])^{\,\mathsf{C}\in cn}\}. \\
& \quad \textbf{fix}\,[Classes\ (World\ Univ)\ Univ] \\
& \qquad (\lambda \mathsf{classes} : \{Classes\ (World\ Univ)\ Univ\}. \\
& \qquad\quad \{(\mathsf{C} = x.\mathsf{C}\ [Univ]\ \mathsf{inj}_{\mathsf{C}}\ \mathsf{proj}_{\mathsf{C}}\ \mathsf{classes})^{\,\mathsf{C}\in cn}\}) \\
& \text{where} \\
& \quad \mathsf{inj}_{\mathsf{C}} = \lambda x : (World\ Univ)\cdot\mathsf{C}.\ \textbf{fold}\ \textbf{inj}_{\mathsf{C}}^{Tagged\ Univ}\ x\ \textbf{as}\ Univ \\
& \quad \mathsf{proj}_{\mathsf{C}} = \lambda x : Univ.\ \textbf{case}\ \textbf{unfold}\ x\ \textbf{of}\ \mathsf{C}\ y \Rightarrow \textbf{some}\ [(World\ Univ)\cdot\mathsf{C}]\ y \\
& \qquad\qquad\qquad\qquad\qquad \textbf{else}\ \textbf{none}\ [(World\ Univ)\cdot\mathsf{C}]
\end{aligned} \tag{5.26}$$

Figure 5.11: Program translation and linking

dictionary from classes and instantiates it as super. It then uses METH to construct code for each method label in *methvec*.

METH is defined in figure 5.10 on the page before. There are three cases: it produces the special dyncast method, which must be overridden in every class (5.22); it inherits a method from the super class (5.23); or it constructs a new method body by translating FJ code (5.24). Note that the inherited method has no overhead; the function pointer is simply copied from the super class dictionary.

Proofs that the translated class declarations are well-typed are in section 5.8.

5.6 Linking

Finally, we must instantiate and link the separate class modules together into a single program. Figure 5.11 gives the translation for a complete FJ program (definition 5.25). The functors that result from translating each class declaration are collected into a record and passed to the LINK function. The result, bound to x_{cn}, is a record of classes. It is used as the classes parameter in translating the main program expression e.

At the type level, we build the universal sum (*Univ*) with a variant for the object type of each class in the class table. This type is propagated into the translation of the main program expression, and it is used to instantiate the functor corresponding to each class. Unfolded, it is written as *Tagged Univ*.

LINK uses **fix** to create a fixed point of the record of classes. Each class functor in the argument x has one type parameter and three value parameters. For each class, we define injection and projection functions, inj_C and proj_C. The former tags an object and folds to create a value of type *Univ*. The latter unfolds and removes the tag, where applicable. We define these functions, and the sum type itself, at the outer level so that classes can be compiled separately, and yet still use the universal sum type to implement dynamic cast. The final argument to x.C is the classes record itself. Section 5.8.5 on page 83 includes a theorem that linkage is well-typed.

5.7 Separate compilation

Our translation supports separate compilation, but the formal presentation does not make this clear. In this section, we describe our compilation model and justify that claim.

What must be known to compile a Java class C to native code? At a minimum, we must know the fields and methods of all super classes, to ensure that the layout of C's vtable and objects are consistent. Next, it is helpful to know enough about classes referenced by C so that the offsets of their fields and methods can be embedded in the code. These principles do not mean that all referenced classes must be *compiled* together. Indeed, as long as the above information is known, classes can be compiled separately, in any order.

In our translation, we need not just offsets but the *full* type information for super classes and referenced classes. If C refers to field x from class D, we need to know all

about the type of `x` as well. This clearly involves extracting type information from more classes, although not necessarily *every* class in the program. Even so, each class can still be compiled separately, in any order, assuming the requisite types are available.

A reasonable compilation strategy starts with some *root set* of classes and builds a dependence graph. For a given program, the root set contains just the class with the `main` method; for a library, it includes all exported classes. Next, traverse the graph bottom-up. Compile each class separately, but propagate the necessary information from `C` to all those classes that depend on it. Of course, there may be cyclic dependencies, represented by strongly-connected components (clusters) in the graph. In these cases, we extract type information from all members of the cluster before compiling any of them. Still, each class in the cluster is compiled separately.

A hallmark of whole-program compilation is that library code must be compiled along with application code. This is clearly not necessary in our model. Library classes would never depend on application classes, so they can be compiled in advance. The reason that our formal translation uses the macro *World* (containing object types for every class in the program) is that, in the most general case, every class in an FJ program refers to every other class. Thus, our translation assumes that the entire program falls within one strongly-connected cluster. In practice, *World* would include just the classes in the same cluster as the class being compiled.

5.8 Properties

This section contains formal statements and proofs of several properties of our translation. The most important is theorem 4 on page 80; it says that well-typed Featherweight Java expressions are translated to well-typed Mini JFlint expressions. Its proof is straightforward if we first factor out and prove several key properties as lemmas.

First, in lemma 6 we establish a correspondence between the *mtype* used in the FJ

semantics and the *methvec* relation used for object layout (likewise between *fields* and *fieldvec*; see lemma 7). Next, in lemmas 10 and 11, we establish the correspondence between pairs in *methvec*/*fieldvec* and elements in *Rows*. These correspondences are proved by induction on the class hierarchy. Finally, we show in lemmas 13 and 14 that the PACK and UPCAST macros return expressions of the expected type. These can be proved by inspection, but the latter argument requires a non-trivial coherence property for *Rows* (lemma 12 on page 78). Specifically, $Rows[\mathsf{A}, \top]\ \mathsf{w}\ \mathsf{u}\ (Rows[\mathsf{C}, \mathsf{A}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail})$ must be equivalent to $Rows[\mathsf{C}, \top]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}$. This is proved by induction on the derivation of $\mathsf{C} <: \mathsf{A}$.

5.8.1 Contents of field/method vectors

Lemma 6 (Method vector) *$mtype(\mathsf{m}, \mathsf{C}) = \mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D}_0$ if and only if $(\mathsf{m}, \mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D}_0) \in methvec(\mathsf{C})$.*

Proof By induction on the derivation of $\mathsf{C} <: \mathsf{Object}$. In the base case, the implication holds trivially. Otherwise, let $CT(\mathsf{C}) =$ **class** C **extends** B { ...K $\mathsf{M}_1 \ldots \mathsf{M}_n$ }. We distinguish two cases:

1. m is not defined in $\mathsf{M}_1 \ldots \mathsf{M}_n$. ($\Longrightarrow$) Then, $mtype(\mathsf{m}, \mathsf{C}) = \mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D}_0 = mtype(\mathsf{m}, \mathsf{B})$. Using the inductive hypothesis, $(\mathsf{m}, \mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D}_0)$ is in $methvec(\mathsf{B})$ and thus it is also in $methvec(\mathsf{C})$. ($\Longleftarrow$) $addmeth(\mathsf{B}, [\mathsf{M}_1 \ldots \mathsf{M}_n])$ could not have added m, so it must be that $(\mathsf{m}, \mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D}_0) \in methvec(\mathsf{B})$. Using the inductive hypothesis, $mtype(\mathsf{m}, \mathsf{B}) = \mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D}_0$ and, in this case, $mtype(\mathsf{m}, \mathsf{C}) = mtype(\mathsf{m}, \mathsf{B})$.

2. $\exists j$ such that $\mathsf{M}_j = \mathsf{D}_0\ \mathsf{m}\ (\mathsf{D}_1\ \mathsf{x}_1 \ldots \mathsf{D}_n\ \mathsf{x}_n)$ { **return** e; }. In this case, $mtype(\mathsf{m}, \mathsf{C})$ is directly defined as $\mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D}_0$.

   ($\Longrightarrow$) **case** $mtype(\mathbf{m}, \mathbf{B}) = \mathbf{C}_1 \ldots \mathbf{C}_n \to \mathbf{C}_0$ Then, from class well-formedness we conclude that $\mathsf{C}_i = \mathsf{D}_i$ for $i \in \{0 \ldots n\}$. From the inductive hypothesis,

we find that $(\mathsf{m}, \mathsf{C}_1 \ldots \mathsf{C}_n \to \mathsf{C}_0) \in \mathit{methvec}(\mathsf{B})$. Thus, $(\mathsf{m}, \mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D}_0) \in \mathit{methvec}(\mathsf{C})$.

$(\Longrightarrow)$ **case $\neg\exists\, T$ such that $\mathit{mtype}(\mathbf{m}, \mathbf{B}) = T$** From the inductive hypothesis (in the reverse direction), $\neg\exists\, T$ such that $(\mathsf{m}, T) \in \mathit{methvec}(\mathsf{B})$. Given this, we can show (by induction on j) that *addmeth* adds $(\mathsf{m}, \mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D}_0)$ to $\mathit{methvec}(\mathsf{C})$.

$(\Longleftarrow)$ **case $(\mathbf{m}, \mathbf{C}_1 \ldots \mathbf{C}_n \to \mathbf{C}_0) \in \mathit{methvec}(\mathbf{B})$** Therefore, by definition, $(\mathsf{m}, \mathsf{C}_1 \ldots \mathsf{C}_n \to \mathsf{C}_0) \in \mathit{methvec}(\mathsf{C})$. From class well-formedness, we argue that $\mathsf{C}_i = \mathsf{D}_i$ for $i \in \{0 \ldots n\}$.

$(\Longleftarrow)$ **case $\neg\exists\, T$ such that $(\mathbf{m}, T) \in \mathit{methvec}(\mathbf{B})$** Then, *addmeth* and $\mathit{mtype}(\mathsf{m}, \mathsf{C})$ both assign m the signature $\mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{D}_0$. □

Lemma 7 (Field vector) *If* $\mathsf{D}\ \mathsf{f} \in \mathit{fields}(\mathsf{C})$*, then* $(\mathsf{f}, \mathsf{D}) \in \mathit{fieldvec}(\mathsf{C})$*.*

Proof By induction on the derivation of $\mathsf{C} <: \mathsf{Object}$. □

5.8.2 Object layout

Lemma 8 (Well-kinded rows) *If* $\mathsf{C} <: \mathsf{A}$*, then* $\Phi \vdash \mathit{Rows}[\mathsf{C}, \mathsf{A}] :: \mathit{kcn} \Rightarrow \mathsf{Type} \Rightarrow \mathit{ktail}[\mathsf{C}] \Rightarrow \mathit{ktail}[\mathsf{A}]$*.*

Proof By induction on the derivation of $\mathsf{C} <: \mathsf{A}$. Observe that $\vdash \mathit{kcn}\ \mathit{kind}$ and, for any $\mathsf{D} \in \mathit{cn}$, $\vdash \mathit{ktail}[\mathsf{D}]\ \mathit{kind}$. Then, the base case ($\mathsf{C} = \mathsf{A}$) holds trivially. Now, let $\mathit{CT}(\mathsf{C}) =$ **class** C **extends** B $\{\ \mathsf{D}_1\ \mathsf{f}_1; \ldots \mathsf{D}_n\ \mathsf{f}_n;\ \mathsf{K} \ldots\ \}$ and $\mathsf{B} <: \mathsf{A}$. Using the inductive hypothesis, $\mathit{Rows}[\mathsf{B}, \mathsf{A}]$ has kind $\mathit{kcn} \Rightarrow \mathsf{Type} \Rightarrow \mathit{ktail}[\mathsf{B}] \Rightarrow \mathit{ktail}[\mathsf{A}]$ in kind environment Φ. The rule (5.10) constructs a tuple $\mathsf{tail}' = \{\mathsf{m} = \ldots, \mathsf{f} = \ldots\}$. Let $\Phi' = \Phi, \mathsf{w} :: \mathit{kcn}, \mathsf{u} :: \mathsf{Type}, \mathsf{tail} :: \mathit{ktail}[\mathsf{C}]$. It remains to be shown that tail' has kind $\mathit{ktail}[\mathsf{B}]$ in kind environment Φ'. Consider the f component; the argument for m is similar. Using the definition of $\mathit{ktail}[\mathsf{C}]$ and the tuple selection rule (4.13), $\Phi' \vdash \mathsf{tail} \cdot \mathsf{f} :: \mathsf{R}^{\mathrm{dom}(\mathit{fieldvec}(\mathsf{C}))}$. Using the

definition of *kcn* and class table well-formedness, $\Phi' \vdash \mathsf{w}\cdot\mathsf{D}_n :: \mathsf{Type}$. Finally, the row formation rule (4.15) assigns kind $\mathsf{R}^{\mathrm{dom}(\mathit{fieldvec}(\mathsf{C}))-\{\mathsf{f}_n\}}$ to the row $(\mathsf{f}_n : \mathsf{w}\cdot\mathsf{D}_n ; \mathsf{tail}\cdot\mathsf{f})$. Iterate for each label; the resulting row has kind $\mathsf{R}^{\mathrm{dom}(\mathit{fieldvec}(\mathsf{C}))-\{\mathsf{C},\mathsf{f}_1\ldots\mathsf{f}_n\}}$ which is the same as $\mathsf{R}^{\mathrm{dom}(\mathit{fieldvec}(\mathsf{B}))}$. □

Lemma 9 (Tail position) *If* $\mathsf{C} <: \mathsf{B}$, $\Phi \vdash \mathsf{w} :: \mathit{kcn}$, $\Phi \vdash \mathsf{tail} :: \mathit{ktail}[\mathsf{C}]$, *and* $\Phi \vdash \mathsf{self} :: \mathsf{Type}$, *then* $(\mathit{Rows}[\mathsf{C},\mathsf{B}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail})\cdot\mathsf{m}\ \mathsf{self}$ *has the form* $(\ldots ; \mathsf{tail}\cdot\mathsf{m}\ \mathsf{self})$ *and* $(\mathit{Rows}[\mathsf{C},\mathsf{B}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail})\cdot\mathsf{f}$ *has the form* $(\ldots ; \mathsf{tail}\cdot\mathsf{f})$.

Proof By inspection. □

Lemma 10 (Method layout) *If* $\Phi \vdash \mathsf{w} :: \mathit{kcn}$, $\Phi \vdash \mathsf{tail} :: \mathit{ktail}[\mathsf{C}]$, $\Phi \vdash \mathsf{self} :: \mathsf{Type}$, *and* $(\mathsf{m}, T) \in \mathit{methvec}(\mathsf{C})$, *then*
$(\mathit{Rows}[\mathsf{C},\mathsf{Object}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail})\cdot\mathsf{m}\ \mathsf{self} = (\ldots ; \mathsf{m} : \mathsf{self} \to \mathit{Ty}[\mathsf{self}; \mathsf{w}; T] ; \ldots ; \mathsf{tail}\cdot\mathsf{m}\ \mathsf{self})$.

Proof By induction on derivation of $\mathsf{C} <: \mathsf{Object}$. $\mathit{methvec}(\mathsf{Object})$ is empty, so the base case holds trivially. Otherwise, let $\mathit{CT}(\mathsf{C}) =$ **class** C **extends** B { ... }.

> **Case** $(\mathbf{m}, T) \in \mathit{methvec}(\mathbf{B})$ Let $\mathsf{tail}' = \{\mathsf{m} = \lambda\mathsf{self}{::}\mathsf{Type}.\ \ldots, \mathsf{f} = \ldots\}$, as given in rule (5.10); according to lemma 8, this has kind $\mathit{ktail}[\mathsf{B}]$. Invoking the inductive hypothesis (with tail′) we find that
>
> $$(\mathit{Rows}[\mathsf{B},\mathsf{Object}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}')\cdot\mathsf{m}\ \mathsf{self}$$
> $$= (\ldots ; \mathsf{m} : \mathsf{self} \to \mathit{Ty}[\mathsf{self}; \mathsf{w}; T] ; \ldots ; \mathsf{tail}'\cdot\mathsf{m}\ \mathsf{self})$$
>
> Then, expanding the definition we get
>
> $$(\mathit{Rows}[\mathsf{C},\mathsf{Object}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail})\cdot\mathsf{m}\ \mathsf{self}$$
> $$= (\ldots ; \mathsf{m} : \mathsf{self} \to \mathit{Ty}[\mathsf{self}; \mathsf{w}; T] ; \ldots\ldots ; \mathsf{tail}\cdot\mathsf{m}\ \mathsf{self})$$
>
> **Case** $(\mathbf{m}, T) \notin \mathit{methvec}(\mathbf{B})$ Then, m must be one of the names $\mathsf{m}_1 \ldots \mathsf{m}_n$ enumerated in the definition. In this case, the row $\mathsf{tail}'\cdot\mathsf{m}\ \mathsf{self}$ will contain an

element m of type $\mathsf{self}{\rightarrow}Ty[\mathsf{self};\mathsf{w};T]$. This tail' is passed to $Rows[\mathsf{B},\mathsf{Object}]$, but according to lemma 9, it will still appear in the result. □

Lemma 11 (Field layout) *If* $\mathsf{C} <: \mathsf{Object}$, $\Phi \vdash \mathsf{w} :: kcn$, $\Phi \vdash \mathsf{tail} :: ktail[\mathsf{C}]$, *and* $fieldvec(\mathsf{C}) = fieldvec(\mathsf{Object}) +\!\!+ [(l_1,\mathsf{D}_1)\ldots(l_n,\mathsf{D}_n)]$, *then* $Rows[\mathsf{C},\mathsf{Object}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail} = l_1 \mathbf{:} \mathsf{w}{\cdot}\mathsf{D}_1\,\mathbf{;}\ldots l_n \mathbf{:} \mathsf{w}{\cdot}\mathsf{D}_n\,\mathbf{;}\mathsf{tail}{\cdot}\mathsf{f}$.

Proof By induction on the derivation of $\mathsf{C} <: \mathsf{Object}$. Similar to the proof of lemma 10. □

Lemma 12 (Rows coherence) *If* $\mathsf{C} <: \mathsf{A}$, $\Phi \vdash \mathsf{u} :: \mathsf{Type}$, $\Phi \vdash \mathsf{w} :: kcn$, *and* $\Phi \vdash \mathsf{tail} :: ktail[\mathsf{C}]$, *then* $Rows[\mathsf{A},\mathsf{Object}]\ \mathsf{w}\ \mathsf{u}\ (Rows[\mathsf{C},\mathsf{A}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}) = Rows[\mathsf{C},\mathsf{Object}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}$.

Proof By induction on the derivation of $\mathsf{C} <: \mathsf{A}$. The base case ($\mathsf{C} = \mathsf{A}$) holds trivially. Now, let $CT(\mathsf{C}) = \textbf{class}\ \mathsf{C}\ \textbf{extends}\ \mathsf{B}\ \{\ \ldots\ \}$ where $\mathsf{B} <: \mathsf{A}$. The rule for $Rows[\mathsf{C},\mathsf{A}]$ defines a tuple $\{\mathsf{f} = \ldots, \mathsf{m} = \ldots\}$ which we will call tail'. Specifically, $Rows[\mathsf{C},\mathsf{A}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail} = Rows[\mathsf{B},\mathsf{A}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}'$. Now, using tail' in the inductive hypothesis, we find that $Rows[\mathsf{A},\mathsf{Object}]\ \mathsf{w}\ \mathsf{u}\ (Rows[\mathsf{B},\mathsf{A}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}') = Rows[\mathsf{B},\mathsf{Object}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}'$. According to the definition, $Rows[\mathsf{B},\mathsf{Object}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}' = Rows[\mathsf{C},\mathsf{Object}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}$, where tail' is the same as above. Substituting equals for equals (twice) yields

$$Rows[\mathsf{A},\mathsf{Object}]\ \mathsf{w}\ \mathsf{u}\ (Rows[\mathsf{C},\mathsf{A}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}) = Rows[\mathsf{C},\mathsf{Object}]\ \mathsf{w}\ \mathsf{u}\ \mathsf{tail}$$

□

5.8.3 Object transformations

Lemma 13 (Well-typed pack) *If* $\Phi \vdash \mathsf{tail} :: ktail[\mathsf{C}]$ *and* $\Phi;\Delta \vdash e : SelfTy[\mathsf{C}]\ (World\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}$, *then* $\Phi;\Delta \vdash \textsc{pack}[\mathsf{C};\mathsf{u};\mathsf{tail};e] : (World\ \mathsf{u}){\cdot}\mathsf{C}$.

Proof By inspection of the definitions, using the term formation rules for fold (4.33) and pack (4.26). □

Lemma 14 (Well-typed upcast) *If* $\Phi; \Delta \vdash e : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}$ *and* $\mathsf{C} <: \mathsf{A}$*, then* $\Phi; \Delta \vdash \textsc{upcast}[\mathsf{C}; \mathsf{A}; \mathsf{u}; e] : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{A}$.

Proof By inspection of the definitions, using the term formation rules for open (4.27) and unfold (4.34) and lemmas 8, 12, and 13. Unfolding e produces a term of type $\mathit{ObjTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}$. Opening this introduces type variable $\mathsf{tail} :: \mathit{ktail}[\mathsf{C}]$ and term variable $x : \mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}$; call this new environment $\Phi'; \Delta'$. The body of the open contains a PACK expression, but in order to use lemma 13, we must establish the following:

1. $\Phi' \vdash \mathit{Rows}[\mathsf{C}, \mathsf{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail} :: \mathit{ktail}[\mathsf{A}]$, and

2. $\Phi'; \Delta' \vdash x : \mathit{SelfTy}[\mathsf{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ (\mathit{Rows}[\mathsf{C}, \mathsf{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail})$.

The first follows from lemma 8. The second reduces to

$$\begin{aligned} \Phi' \vdash\ & \mathit{SelfTy}[\mathsf{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ (\mathit{Rows}[\mathsf{C}, \mathsf{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}) = \\ & \mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail} :: \mathsf{Type} \end{aligned}$$

By expanding the definition of $\mathit{SelfTy}[\cdot]$ and applying equivalence rules, it reduces again to

$$\begin{aligned} \Phi' \vdash\ & \mathit{Rows}[\mathsf{A}, \mathsf{Object}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ (\mathit{Rows}[\mathsf{C}, \mathsf{A}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}) = \\ & \mathit{Rows}[\mathsf{C}, \mathsf{Object}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail} :: \mathit{ktail}[\mathsf{Object}] \end{aligned}$$

which follows from lemma 12. Finally, lemma 13 can be invoked to show that the result of the upcast has type $(\mathit{World}\ \mathsf{u}) \cdot \mathsf{A}$. □

5.8.4 Type preservation for expressions

FJ contexts are translated to type environments as follows:

$$\text{ENV}[\mathsf{u}; \Gamma, \mathsf{x} : \mathsf{D}] = \text{ENV}[\mathsf{u}; \Gamma], \mathsf{x} : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{D}$$

$$\text{ENV}[\mathsf{u}; \circ] = \circ$$

Lemma 15 (Context translation) *If* $\Phi \vdash \mathsf{u} :: \mathsf{Type}$ *and* $\mathrm{range}(\Gamma) \subseteq cn$*, then* $\Phi \vdash \text{ENV}[\mathsf{u}; \Gamma]$ *type env.*

Proof By inspection. □

Theorem 4 (Type preservation) *If* $\Phi \vdash \mathsf{u} :: \mathsf{Type}$*,* $\Phi; \Delta \vdash \mathsf{classes} : \{\mathit{Classes}\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\}$ *and* $\Gamma \vdash \mathsf{e} \in \mathsf{C}$*, then* $\Phi; \Delta, \text{ENV}[\mathsf{u}; \Gamma] \vdash \text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathsf{e}] : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}$*.*

Proof By induction on the structure of e. We use the following abbreviations: Δ_Γ for $\text{ENV}[\mathsf{u}; \Gamma]$; Δ'_Γ for Δ, Δ_Γ; and e for $\text{EXP}[\Gamma; \mathsf{u}; \mathsf{classes}; \mathsf{e}]$.

Case 5.12 $\mathsf{e} = \mathsf{x}$ and, from (3.15), $\mathsf{C} = \Gamma(\mathsf{x})$. Thus, $\Delta_\Gamma(\mathsf{x}) = (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}$ and $\Phi; \Delta'_\Gamma \vdash e : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}$.

Case 5.13 $\mathsf{e} = \mathsf{e}_0.\mathsf{f}_i$ and $\mathsf{C} = \mathsf{C}_i$, where $\Gamma \vdash \mathsf{e}_0 \in \mathsf{C}_0$ and $\mathit{fields}(\mathsf{C}_0) = \mathsf{C}_1\ \mathsf{f}_1 \ldots \mathsf{C}_n\ \mathsf{f}_n$. By inductive hypothesis, $\Phi; \Delta'_\Gamma \vdash e_0 : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}_0$. The code in (5.13) unfolds and opens e_0. Using the same argument as in the proof of lemma 14, this introduces the type variable $\mathsf{tail} :: \mathit{ktail}[\mathsf{C}_0]$ and term variable $x : \mathit{SelfTy}[\mathsf{C}_0]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}$; call this new environment $\Phi'; \Delta''_\Gamma$. Unfolding x yields a term of type

$$\{\mathsf{vtab} : \ldots ; (\mathit{Rows}[\mathsf{C}_0, \mathsf{Object}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}) \cdot \mathsf{f}\}$$

Using lemma 7, $(\mathsf{f}_i, \mathsf{C}_i) \in \mathit{fieldvec}(\mathsf{C}_0)$. Using lemma 11, we find that the row $(\mathit{Rows}[\mathsf{C}_0, \mathsf{Object}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}) \cdot \mathsf{f}$ contains a binding $\mathsf{f}_i : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}_i$. Using

record selection, $\Phi; \Delta''_\Gamma \vdash (\textbf{unfold}\, x \ldots).\mathsf{f}_i : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}_i$. Exiting the scope of the open, we conclude $\Phi; \Delta'_\Gamma \vdash e : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}_i$.

Case 5.14 $\mathsf{e} = \mathsf{e}_0.\mathsf{m}\ (\mathsf{e}_1 \ldots \mathsf{e}_n)$, where $\Gamma \vdash \mathsf{e}_0 \in \mathsf{C}_0$, $\mathit{mtype}(\mathsf{m}, \mathsf{C}_0) = \mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{C}$, $\Gamma \vdash \mathsf{e}_i \in \mathsf{C}_i$, and $\mathsf{C}_i <: \mathsf{D}_i$, for all $i \in \{1 \ldots n\}$. We use the inductive hypothesis on e_0, and the same unfold-open-unfold argument as in the previous case. Selecting vtab yields a term of type
$\{(\mathit{Rows}[\mathsf{C}_0, \mathsf{Object}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}) \cdot \mathsf{m}\ (\mathit{SelfTy}[\mathsf{C}_0]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail})\}$ Using lemma 6, $(\mathsf{m}, \mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{C}) \in \mathit{methvec}(\mathsf{C}_0)$. Using lemma 10, the above record contains a binding

$$\begin{aligned} &\mathsf{m} : (\mathit{SelfTy}[\mathsf{C}_0]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}) \to \mathit{Ty}[\mathsf{self}; \mathit{World}\ \mathsf{u}; \mathsf{D}_1 \ldots \mathsf{D}_n \to \mathsf{C}] \\ = &\mathsf{m} : (\mathit{SelfTy}[\mathsf{C}_0]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail}) \to (\mathit{World}\ \mathsf{u}) \cdot \mathsf{D}_1 \to \ldots \\ &\qquad (\mathit{World}\ \mathsf{u}) \cdot \mathsf{D}_n \to (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C} \end{aligned}$$

Thus, selecting m and applying it to x yields a term of type

$$(\mathit{World}\ \mathsf{u}) \cdot \mathsf{D}_1 \to \ldots (\mathit{World}\ \mathsf{u}) \cdot \mathsf{D}_n \to (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}$$

Now, for each i in $1 \ldots n$, we use the inductive hypothesis on e_i, concluding that $\Phi; \Delta'_\Gamma \vdash e_i : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}_i$. Using this and $\mathsf{C}_i <: \mathsf{D}_i$, lemma 14 l tells us that $\Phi; \Delta'_\Gamma \vdash \textsc{upcast}[\mathsf{C}_i; \mathsf{D}_i; \mathsf{u}; e_i] : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{D}_i$ Finally, using the application formation rule n times, $\Phi; \Delta'_\Gamma \vdash e : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}$.

Case 5.15 $\mathsf{e} = \textbf{new}\ \mathsf{C}\ (\mathsf{e}_1 \ldots \mathsf{e}_n)$, where $\Gamma \vdash \mathsf{e}_i \in \mathsf{C}_i$, $\mathit{fields}(\mathsf{C}) = \mathsf{D}_1\ \mathsf{f}_1 \ldots \mathsf{D}_n\ \mathsf{f}_n$, and $\mathsf{C}_i <: \mathsf{D}_i$ for all i in $1 \ldots n$. From the premise $\Phi; \Delta \vdash \mathsf{classes} : \{\mathit{Classes}\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\}$ using the rules for selection (of C), application, and selection (of new), the new component has type $(\mathit{World}\ \mathsf{u}) \cdot \mathsf{D}_1 \to \ldots (\mathit{World}\ \mathsf{u}) \cdot \mathsf{D}_n \to (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}$. Just as in the previous case, we use the inductive hypothesis and lemma 14 on each e_i. Again,

using the application formation rule n times yields $\Phi;\Delta'_\Gamma \vdash e : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}$.

Case 5.16 follows from inductive hypothesis and lemma 14.

Case 5.17 $\mathsf{e} = (\mathsf{C})\ \mathsf{e}_0$ where $\Gamma \vdash \mathsf{e}_0 \in \mathsf{D}$. We use the inductive hypothesis on e_0 and the usual unfold-open-unfold sequence. We select dyncast from the vtab and self-apply; this produces a polymorphic function of type

$$\forall \alpha.\ (\mathsf{u} \to \mathbf{maybe}\ \alpha) \to \mathbf{maybe}\ \alpha$$

Next we instantiate α with $(\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}$ and apply to the class tag, which the correct type: $\mathsf{u} \to \mathbf{maybe}\ (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}$. The result has type $\mathbf{maybe}\ (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}$, and using the case formation rule, the first branch has type $(\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}$. The other branch aborts evaluation, but is regarded as having the same type. So, finally, $\Phi;\Delta'_\Gamma \vdash e : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C}$. □

5.8.5 Class components

Lemma 16 (Well-typed constructor) *If* $\Phi \vdash \mathsf{u} :: \mathsf{Type}$ *and* $\Phi;\Delta \vdash \mathsf{vtab} : \mathit{Dict}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ (\mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathit{Empty}[\mathsf{C}])$, *then* $\Phi;\Delta \vdash \textsc{new}[\mathsf{C};\mathsf{u};\mathsf{vtab}] : \mathit{Ctor}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})$.

Proof By inspection, using lemma 11. □

Lemma 17 (Well-typed dictionary) *If* $\Phi \vdash \mathsf{u} :: \mathsf{Type}$, $\Phi;\Delta \vdash \mathsf{inj} : (\mathit{World}\ \mathsf{u}) \cdot \mathsf{C} \to \mathsf{u}$, *and* $\Phi;\Delta \vdash \mathsf{classes} : \{\mathit{Classes}\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\}$, *then* $\Phi;\Delta \vdash \textsc{dict}[\mathsf{C};\mathsf{u};\mathsf{inj};\mathsf{classes}] : \forall \mathsf{tail}.\ \mathit{Dict}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ (\mathit{SelfTy}[\mathsf{C}]\ (\mathit{World}\ \mathsf{u})\ \mathsf{u}\ \mathsf{tail})$.

Proof By inspection, using lemma 10. □

Theorem 5 (Well-typed class declaration) $\Phi;\Delta \vdash \textsc{cdec}[\mathsf{C}] : \mathit{ClassF}[\mathsf{C}]$

Proof By inspection, using lemmas 16 and 17 for the non-trivial class components.□

Theorem 6 (Well-typed linkage)

$\Phi; \Delta \vdash \textsc{link} : \{(\mathsf{C} : \mathit{ClassF}[\mathsf{C}])^{\mathsf{C} \in cn}\} \rightarrow \{\mathit{Classes}\ (\mathit{World}\ \mathit{Univ})\ \mathit{Univ}\}$

Proof By inspection. □

5.9 Related work

Fisher and Mitchell (1998) use extensible objects to model Java-like class constructs. Our encoding does not rely on extensible objects as primitives, but it may be viewed as an implementation of some of their properties in terms of simpler constructs. Rémy and Vouillon (1997) use row polymorphism in Objective ML for both class types and type inference on unordered records. Our calculus is explicitly typed, but we use ordered rows to represent the open type of `self`.

Because it uses standard existential and recursive types, our object representation is superficially similar to several of the classic encodings in F_ω-based languages (Bruce, Cardelli, and Pierce 1999; Pierce and Turner 1994). As in the work of Abadi, Cardelli, and Viswanathan (1996), method invocation uses self-application; however, we hide the actual class of the receiver using existential quantification over row variables instead of splitting the object into a known interface and a hidden implementation. This allows reuse of methods in subclasses without any overhead. We use an analog of the recursive-existential encoding due to Bruce (1994) to give types to other arguments or results belonging to the same class or a subclass, as needed in Java, without over-restricting the type to be the same as the receiver's.

Several other researchers describe type-preserving compilation of object-oriented languages. Wright et al. (1998) compile a Java subset to a typed intermediate language, but they use unordered records and resort to dynamic type checks because their system is too weak to type self application. Crary (1999) encodes the object calculus of Abadi and Cardelli (1996) using existential and intersection types in a calculus of coer-

cions. Glew (2000a) translates a simple class-based object calculus into an intermediate language with F-bounded polymorphism (Canning et al. 1989; Eifrig et al. 1995) and a special 'self' quantifier.

Comparing object encodings

A more detailed comparison with the work of Glew and Crary is worthwhile. The three encodings share many similarities, and appear to be different ways of expressing the same underlying idea. In this section, we will attempt to clarify the connections between them. Following Bruce, Cardelli, and Pierce (1999), we can specify object interfaces as type operators, so that the type of the self argument can be plugged in. The `Point` interface, for example, would be represented as $I_P = \lambda\alpha$::Type. $\{\mathsf{getx} : \alpha \to \mathbf{int}\}$.

An *F-bounded* quantifier (Canning et al. 1989) permits a quantified type variable to appear in its own bound. Glew used a twist on F-bounded polymorphism to encode method tables that could be reused in subclasses. This leads naturally to an object encoding using an F-bounded existential (*FBE*): $\exists\alpha \leq I(\alpha).\alpha$, which Glew writes as $\mathsf{self}\ \alpha.I(\alpha)$. Typically, the witness type is recursive; it is a subtype of its unrolling.

The connection between self and the F-bounded existential was recognized independently by Glew (2000c) and ourselves. We can derive the rules governing self from those for F-bounded existentials. Glew uses equi-recursive types in (2000a); a restriction to iso-recursive types is possible, though awkward (Glew 2000b). The rules for packing and opening self types must simultaneously fold and unfold in precisely the right places.

Self application is typable in *FBE* because the object, via subsumption, enjoys two types: the abstract type α and the interface type $I(\alpha)$. Crary (1999) encodes precisely the same property as an intersection type: $\exists\alpha.\,\alpha \wedge I(\alpha)$. Again, the witness type is recursive. With equi-recursive types, a value of type μI also has type $I\,(\mu I)$; it could be packaged as $\alpha \wedge I(\alpha)$. Crary makes this encoding practical using a calculus of *coercions*—explicit retyping annotations. Coercions can drop fields from the end of a record, fold

and unfold recursive types, mediate intersection types, and instantiate quantified types. All coercions are erasable.

We will now show how our own encoding, based on row polymorphism, relates to these. A known technique for eliminating an F-bound is to replace it with a higher-order bound and a recursive type. That is, we could represent $\exists\alpha \le I(\alpha).\alpha$ as $\exists\delta \le I.\mu\,\delta$. Using a point-wise subtyping rule, the interface type operators themselves enjoy a subtyping relationship. Iso-recursive types can be used directly with this technique because the fixed point is separate from the existential.

Next, though it is less efficient, we can implement the higher-order subtyping with a coercion function:

$$\exists\delta :: \mathsf{Type}{\Rightarrow}\mathsf{Type}.\{c : \delta\,(\mu\,\delta) \to I\,(\mu\,\delta),\ o : \mu\,\delta\}$$

To select a method from an object, we first open the package, select the coercion c, and apply it to the unfolding of o. This yields an interface whose methods are then directly applicable to o. Now we no longer require subtyping.

Using a general function for this coercion yields more flexibility than we require to implement Java. All the function ever needs to do is drop fields from records. With row polymorphism, we can express the result of pre-applying the coercions at all levels. Now we no longer require inefficient coercions. The encodings of Crary and Glew work by supplying two distinct views of the object: an abstract subtype of a concrete interface type. With row polymorphism, that distinction is unnecessary; we can hide just the unknown portion of the interface directly.

All three of these encodings appear to be efficient. In an untyped dynamic semantics, their object representations are precisely the same. The major differences among them are in the strength of the underlying type theory: Glew uses subtyping and F-bounded quantifiers; Crary uses intersection types; we use row polymorphism. In our experience,

row polymorphism is the most conservative extension to plain F_ω—they hardly affect the soundness proof at all. Subtyping and F-bounded quantifiers are a more drastic extension to F_ω. To our knowledge, ours is the only one of these encodings to have been implemented in a real system.

Chapter 6

Beyond Featherweight: the rest of Java

Chapters 3 through 5 presented the major theoretical results of this dissertation: an object-oriented source language, a sound and decidable target language, a formal type-preserving translation, and proofs of many important properties. The remaining chapters supplement these theoretical results by discussing how they extend to a full system and by examining many implementation issues.

Featherweight Java, of course, is only the most rudimentary fragment of the full Java language. It includes classes, inheritance, dynamic dispatch, and dynamic casts. It was ideal for the formal part of our work because it made the translation and proofs reasonably comprehensible. Our implementation, however, supports many Java features that are not part of FJ: interfaces, constructors, static members, null objects, mutable fields, super calls, exceptions, and privacy. Some advanced features of Java are not currently supported, but are the subject of ongoing research: protected and package scopes, native code, dynamic class loading, the reflection API, and concurrency.

This chapter informally describes some significant extensions to our translation. We believe that, given a Java calculus with these extensions, the type preservation theorem would still hold. Unfortunately, Java formalizations with these additional features are easily an order of magnitude more complex than FJ—see Drossopoulou and Eisenbach

(1999) or Flatt, Krishnamurthi, and Felleisen (1999), for example. We have not found a way to manage such additional complexity in our translation while still maintaining readability and detailed proofs.

We begin with features that are handled relatively easily in our translation, even though some of them are tough to formalize. Static members, interface fields, and multiple parameterized constructors can be added to the class record, along with the dictionary and tag. Mutable fields are easily modeled using mutable records. As required by the JVM, the new function allocates the object record with a default 'zero' value for each field. Then any public constructor can be invoked to assign new values to the fields. Super invocations select the method statically from the super class dictionary (as is currently used in dyncast). Java exceptions work similarly to those of SML. Java's instanceof uses the same mechanism as dynamic cast, but is simpler since it just returns a boolean value.

Private methods are defined along with the other methods. Since they can neither be called from subclasses nor overridden, we simply omit them from the vtable and dictionary. We do not yet support protected and package scopes, however, because they transcend compilation unit boundaries. In MOBY, Fisher and Reppy (1999) use two distinct views of classes, a class view and an object view. These correspond roughly to the dict and new fields of our class encoding. If we export a class outside its definitional package, all protected methods and fields should be hidden from the object view but not the class view while those of package scope should be hidden from both.

Null references are easily encoded by lifting all external object types to sum types with a null alternate (just like the maybe type). Then, all object operations must verify that the object pointer is not null. Our target calculus, unlike JVML, can express that an object is non-null, so null pointer checks can be safely hoisted.

6.1 Private fields

Private fields can be hidden from outsiders using existential types. For convenience, assume that the private fields of each class in the hierarchy are collected into separate records. Suppose that Point has private fields x and y, and public field z; and ScaledPoint has private field s. The layout for a ScaledPoint object would be $\{\mathsf{vtab}, \mathsf{Pt}: \{\mathsf{x}, \mathsf{y}\}, \mathsf{z}, \mathsf{SPt}: \{\mathsf{s}\}\}$. With the private fields separated like this, it is easy to hide their types separately. (Using a flat representation is possible, but this separation allows a simpler, more orthogonal presentation.) We embed each class functor in an existential package, where the witness type includes the types of the private fields of that class:

$$C_{\mathsf{Pt}} = \langle \mathsf{priv}{::}\mathsf{Type} = \{\mathsf{x}, \mathsf{y}\}, \textsc{cdec}[\mathsf{Pt}]: \ldots \rangle$$

From inside class ScaledPoint, we **open** the Point package, binding a type variable α to represent the private fields of Point:

$$\begin{aligned} C_{\mathsf{SPt}} = \ & \textbf{open } C_{\mathsf{Pt}} \textbf{ as } \langle \alpha{::}\mathsf{Type}, \mathsf{super}: \ldots \rangle \\ & \quad \textbf{in } \langle \mathsf{priv}{::}\mathsf{Type} = \{\mathsf{s}\}, \textsc{cdec}[\mathsf{SPt}]: \ldots \rangle \end{aligned}$$

Then, our local view of the object from within the subclass is

$$\{\mathsf{vtab}, \mathsf{Pt}: \alpha, \mathsf{z}, \mathsf{SPt}: \{\mathsf{s}\}\}$$

As required, the private fields of the super class are hidden. Using *dot notation* (Cardelli and Leroy 1990) for existential types (instead of **open**) makes this encoding more convenient, but is not necessary.

Unfortunately, privacy interacts with mutual recursion. Suppose that A has a private field b of class B and that B has a method getA that returns an object of class A. From within class A, accessing this.b is allowed, as is invoking this.b.getA( ). It is more

difficult to design an encoding that also allows this.b.getA( ).b. Using the existential interpretation of privacy described above, each class has its own *view* of the types of all other objects. From within class A, private fields of other objects of class A are visible. Private fields of objects of other classes are hidden, represented by type variables. In our example, this.b would have a type something like "B with private fields β" where β is the abstract type. Likewise, from within class B, the type of method getA might be self→("A with private fields α"). The challenge is to allow class A to see that the α in the type of getA is actually the known type of its own private fields.

Propagating this information is especially tricky given the limitations of the iso-recursive types used in our target calculus. We found a solution that does not require extending the language. We parameterize everything (including the hidden type itself) by the types of objects of other classes. Then, each class can instantiate the types of the rest of the world using concrete types for its own private fields (wherever they may lurk in other classes) and abstract types for the rest. The issues are subtle, and a formal treatment is outside the scope of my thesis.

Extending Featherweight Java with privacy would help elucidate the technique, but this in itself is non-trivial. Were we to extend FJ and formalize a translation with privacy, we would then like to prove that privacy is preserved in the target language. That is, we would expect the type system to ensure that another module (even if it is not translated from Java) cannot access the private fields. Unfortunately, this would *not* be a corollary of the type preservation theorem. Rather, it is related to a property called *full abstraction,* meaning that abstraction properties in the source language are also protected in the target language. It is still unclear whether our encoding—or the encodings of Crary (1999) or Glew (2000a), for that matter—enjoys full abstraction.

6.2 Interfaces

Given an object of interface type, we know nothing about the shape of its vtable. There are various ways of locating methods in interface objects. Proebsting et al. (1997) construct a per-class dictionary that maps method names to offsets in the vtable. Krall and Grafl (1997) construct a separate method table (called an *itable*) for each declared interface, storing them all somewhere in the vtable. Although they are not clear on how to *use* the itable, there appear to be two choices. First, we can search for the appropriate itable in the vtable, which amounts to lookup of interface names rather than method names. Second, when casting an object from class type to interface type, we can select the itable and then pair it with the object itself. This avoids name lookup entirely but requires minor coercions when casting to and between interface types.

Our translation can be extended to support both strategies. For the first strategy, all we need is to introduce *unordered* records into our target language with a primitive for dictionary lookup. This just means that records with permuted fields are considered equivalent, and that selecting a field from a record is expensive because offsets must be computed at run time. All the itables for a class would be collected into a separate unordered record, itself an element of the still ordered vtable. Then, casting an object to an interface type only requires repackaging (a runtime no-op) to hide those entries not exported by the current interface.

We can also follow the latter strategy, representing interface objects as a pair where the type of the underlying object is concealed by an existential type. For example, an object which implements the Runnable interface includes a method run( ) which can be invoked to start a new thread. In our target language, a Runnable object r is represented as $\exists\alpha{::}\mathsf{Type}.\,\{\mathsf{itab}: \{\mathsf{run}: \alpha\to\mathbf{1}\},\ \mathsf{obj}: \alpha\}$. To invoke the method, we open the existential, select the method from the itab, select the obj, and apply. With this representation,

interface method invocations are about the same cost as normal method invocations:

$$\textbf{open}\ r\ \textbf{as}\ \langle \beta{::}\mathsf{Type},\ z : \{\mathsf{itab} : \{\mathsf{run} : \beta \to \mathbf{1}\},\ \mathsf{obj} : \beta\}\rangle$$
$$\textbf{in}\ z.\mathsf{itab}.\mathsf{run}\ (z.\mathsf{obj})$$

The caveat is that upward casts to interface types are no longer free. To perform the cast, we must select the target interface's itable from the object and pair it with the object itself.

Chapter 7

Functional Java Bytecode

We have presented a formal translation of a Java calculus, proved that it preserves types, and argued that it can be extended to most of Java. Now, we turn to more detailed implementation concerns. The first design decision we encountered was whether to accept Java source code as input! An interesting alternative is to accept the output of `javac`: class files containing Java bytecode, also known as the Java Virtual Machine Language (JVML).

Although JVML uses untyped local variables and is difficult to analyze, in some ways it contains *more* information than Java source code. Identifiers are already resolved, and the types of all fields and methods mentioned in the code are included—even if they are defined in other classes. We decided to let `javac` perform these menial tasks, so that we could concentrate on compiling JVML to a typed intermediate language. This has been a fruitful path.

In compiling JVML, there are two orthogonal sets of issues. The first set, which we addressed in the formal translation, concerns encoding Java features like method invocation and dynamic cast. The second set of features has to do with control and data flow. Java bytecode is a linear instruction stream that uses an implicit operand stack, untyped local variables, and jumps.

To realize this separation of concerns in our implementation, we designed λJVM as an intermediate language that sits *between* JVML and JFlint. Its control and data flow are just like a λ-calculus, but it retains the types and primitive instructions of JVML.

Apart from its role in our system, λJVM is a good alternative to Java bytecode for virtual machines or compilers which optimize methods and produce native code. It is already in a form that, like static single assignment (Alpern, Wegman, and Zadeck 1988), makes data flow explicit. In addition, λJVM is cleaner to specify and simpler to verify than JVML.

7.1 Design

λJVM is a simply-typed λ-calculus expressed in A-normal form (Flanagan et al. 1993) and extended with the types and primitive instructions of the Java virtual machine. The syntax is given in figure 7.1 on the next page. We use terms e in place of the bytecode for method bodies; otherwise the class file format remains the same. A-normal form ensures that functions and primitives are applied to values only; the `let` syntax binds intermediate values to names. A nested algebraic expression such as $(3 + 4) \times 5$ is expressed in A-normal form as `let` $x = 3 + 4$`;` `let` $y = x \times 5$`;` `return` y.

By *simply-typed,* we mean that there are primitive and function types, but no polymorphic or user-defined type constructors. Types include integers `I`, floats `F`, and the rest of Java's primitive types. `V` is the void type, used for methods or functions which do not return a value. Class or interface names c also serve as types. c^0 indicates an *uninitialized* object of class c. We describe our strategy for verifying proper object initialization in section 7.3.

The set type $\{\overline{c}\}$ represents a union. Normally we can treat $\{a, b, c\}$ as equivalent to the name of the class or interface which is the *least common ancestor* of a, b, and c in the class hierarchy. For interfaces, however, a usable ancestor does not always exist;

$$
\begin{array}{lrl}
\text{Types} & \tau ::= & \texttt{I} \mid \texttt{F} \mid \texttt{...} \mid \texttt{V} \mid c \mid \tau\texttt{[]} \\
 & & \mid c^0 \mid \{\overline{c}\} \mid (\overline{\tau}) \to \tau \\
\text{Values} & v ::= & x \mid i \mid r \mid s \mid \texttt{null}[\tau] \mid \lambda\,(\,\overline{x : \tau}\,)\,e \\
\text{Terms} & e ::= & \texttt{letrec}\ \overline{x = v\,\texttt{.}}\ e \mid \texttt{let}\ x = g\texttt{;}\ e \mid g\texttt{;}\ e \\
 & & \mid \texttt{if}\ br[\tau]\ v\ v\ \texttt{then}\ e\ \texttt{else}\ e \\
 & & \mid \texttt{return} \mid \texttt{return}\ v \mid v\,(\,\overline{v}\,) \mid \texttt{throw}\ v \\
\text{Guards} & g ::= & p \mid p\ \texttt{handle}\ v\,(\,\overline{v}\,) \\
\text{Primops} & p ::= & \texttt{new}\ c \mid \texttt{chkcast}\ c\ v \mid \texttt{instanceof}\ c\ v \\
 & & \mid \texttt{getfield}\ fd\ v_o \mid \texttt{putfield}\ fd\ v_o\ v \\
 & & \mid \texttt{getstatic}\ fd \mid \texttt{putstatic}\ fd\ v \\
 & & \mid \texttt{invokevirtual}\ md\ v_o\,(\,\overline{v}\,) \\
 & & \mid \texttt{invokeinterface}\ md\ v_o\,(\,\overline{v}\,) \\
 & & \mid \texttt{invokespecial}\ md\ v_o\,(\,\overline{v}\,) \\
 & & \mid \texttt{invokestatic}\ md\,(\,\overline{v}\,) \\
 & & \mid bo[\tau]\ v\ v \mid \texttt{neg}[\tau]\ v \mid \texttt{convert}[\tau_0, \tau_1]\ v \\
 & & \mid \texttt{newarray}[\tau]\ v_n \mid \texttt{arraylength}[\tau]\ v_a \\
 & & \mid \texttt{aload}[\tau]\ v_a\ v_i \mid \texttt{astore}[\tau]\ v_a\ v_i\ v_o \\
\text{Branches} & br ::= & \texttt{eq} \mid \texttt{ne} \mid \texttt{lt} \mid \texttt{le} \mid \texttt{gt} \mid \texttt{ge} \\
\text{Binops} & bo ::= & br \mid \texttt{add} \mid \texttt{mul} \mid \texttt{div} \mid \texttt{and} \mid \texttt{or} \mid \texttt{...} \\
\text{Field descriptor} & fd ::= & \tau\ c.f \\
\text{Method descriptor} & md ::= & c.m(\overline{\tau})\tau
\end{array}
$$

Figure 7.1: Method syntax in λJVM

see section 7.3. Finally, $(\overline{\tau}) \to \tau$ is the type of a λJVM function with multiple arguments.

Values include names x (introduced by `let`), constants of various types, the null constant ($\texttt{null}[\tau]$ for a given array or object type τ), and, finally, anonymous functions $\lambda\,(\,\overline{x : \tau}\,)\,e$. The names and types of arguments are written inside the parentheses, and followed by e, the function body.

Terms include two binding forms: `letrec` binds a set of mutually recursive functions; $\texttt{let}\ x = g\texttt{;}\ e$ executes the (possibly guarded) primitive operation g, binds the result to x, and continues executing e. If we are uninterested in the result of a primop (or it does not produce a result), the sequencing form $g\texttt{;}\ e$ may be used instead of `let`. The guard $\texttt{handle}\ v\,(\,\overline{v}\,)$ on a primitive indicates where to jump if the operation

throws an exception. Conditional branches are used for numeric comparisons and for testing whether reference values are null. For brevity, we omit the `lookupswitch` and `tableswitch` of the Java virtual machine. Finally, the base cases can return (with an optional value), call a function, or throw an exception.

The primitive operations cover those JVML instructions that are not for control flow or stack manipulation. They may be grouped into three categories: object, numeric, and array. Object primops include `new`, the dynamic cast and `instanceof` predicate, field accesses, and method calls. Field and method descriptors include the class name and type, just as they do in the JVM. Numeric primops are the usual arithmetic and conversions. Branches, when used as primops, return boolean values. Array primops create, subscript, update, and return the length of an array. We use square brackets for type parameters which, in JVML, are part of the instruction. Thus, `iadd` is expressed as `add[I]`, `fmul` is `mul[F]`, and `i2f` is `convert[I,F]`. We omit multi-dimensional arrays, `monitorenter` and `monitorexit` for brevity.

There are three important things to note about λJVM. First, it is *functional*. There is no operand stack, no local variable assignment, and all data flow is explicit. Second, it is impossible to call a function and continue executing after it returns: `let` $y = f(x)$`; ...` is not valid syntax. Therefore all function calls can be implemented as jumps. (Tail call optimization is standard practice in compilers for functional languages.) This makes λJVM functions very lightweight; more akin to basic blocks than to functions in C.

Third, functions are first class and lexically scoped. We use higher-order functions to implement subroutines and exception handlers. Importantly, functions in λJVM cannot escape from the method in which they are declared. Except for the entry point of a method, call sites of all λ-functions are known. This means a compiler is free to use the most efficient calling convention it can find. Typically, each higher-order function is represented as a closure—a pair of a function pointer and an environment containing values for the free variables (Landin 1964). This representation is convenient, consistent, and

```
public static void m (int i) {
   Pt p = new IntPt(i);
   for (int j = 1; j < i; j *= 2) {
      p = new ColorPt(j);
   }
   p.draw();
   return;
}
```

Figure 7.2: A sample Java method with a loop

compatible with separate compilation, but many other techniques are available. Our implementation currently uses *defunctionalization* (Reynolds 1972), but that is just a detail.

Kelsey (1995) and Appel (1998) have observed that A-normal form for functional programs is equivalent to the static single assignment (SSA) form used in many optimizing compilers to make analyses clean and efficient. This is why λJVM is preferable to stack-based Java bytecode for virtual machines and compilers that optimize methods and produce native code—it is already in a format suitable for analysis, optimization, and code generation. Furthermore, as we discuss in section 7.3, type checking for λJVM is far simpler than standard class file verification.

7.2 Translation

In this section, we describe informally how to translate JVM bytecode to λJVM. Figure 7.2 contains a simple Java method which creates objects, invokes a method, and updates a loop counter. Suppose that `IntPt` and `ColorPt` are both subclasses of `Pt`. With this example, we will demonstrate set types and mutable variable elimination.

Figure 7.3 shows the bytecode produced by the Sun Java compiler. The first step in transforming the bytecode to λJVM is to find the basic blocks. This method begins with a block that allocates and initializes an `IntPt`, initializes `j`, then jumps directly

```
public static m(I)V
   new IntPt
   dup
   iload_0
   invokespecial IntPt.<init>(I)V
   astore_1         ; p = new IntPt(i)
   iconst_1
   istore_2         ; j = 1
   goto C
B: new ColorPt
   dup
   iload_2
   invokespecial ColorPt.<init>(I)V
   astore_1         ; p = new ColorPt(j)
   iload_2
   iconst_2
   imul
   istore_2         ; j *= 2
C: iload_2
   iload_0
   if_icmplt B      ; goto B if j < i
   aload_1          ; p.draw()
   invokevirtual Pt.draw()V
   return
```

Figure 7.3: The same method compiled to JVML

```
public static m(I)V = λ(i : I)
letrec C = λ(p : {IntPt, ColorPt}, j : I)
           if lt[I] j i then B(p, j)
           else invokevirtual Pt.draw()V p ();
                return.
       B = λ(p : {IntPt, ColorPt}, j : I)
           let q = new ColorPt;
           invokespecial ColorPt.<init>(I)V q (j);
           let k = mul[I] j 2;
           C(q, k).
let r = new IntPt;
invokespecial IntPt.<init>(I)V r (i);
C(r, 1)
```

Figure 7.4: The same method translated to λJVM

to block C. C does the loop test and either jumps to B (the loop body) or falls through and returns. The loop body creates a `ColorPt`, updates the loop counter, and then falls through to the loop test.

Next, data flow analysis must infer types for the stack and local variables at each program point. This analysis is also needed during bytecode verification. In the beginning, we know that local variable 0 contains the method argument `i` and the stack is empty. (For virtual methods, local 0 contains `this`.) Symbolic execution of the first block reveals that, upon jumping to C, local 1 contains an `IntPt` and local 2 contains an `int`. We propagate these types into block C, and from there into block B. During symbolic execution of B, we store a `ColorPt` into local 1. Since the current type of local 1 is `IntPt`, we must unify these. Fortunately, we can unify these in λJVM without even knowing where they fit into the class hierarchy—we simply place them into a set type. Now local 1 has type `{IntPt,ColorPt}`. If two types cannot be unified (`int` and `IntPt`, for example), then the variable is marked as unusable (`void`). Block C is a successor of B, and since the type of local 1 has changed, we must check it again. Nothing else changes, so the data flow is complete and we know the types of the locals and the stack at the start of each block.

Next we use symbolic execution to translate each block to a λ-function. The type annotations within each λ-binding come directly from the type inference. For each instruction which pushes a value onto the operand stack, we push a value (either a fresh name or a constant) onto the symbolic stack. For each instruction which fetches its operands from the stack, we harvest the values from the symbolic stack and emit the corresponding primop. Figure 7.4 shows the resulting code. The method is a λ-function with an argument i. B and C are functions implementing the basic blocks of the same name, and the code of the first block follows. The loop counter is updated by passing a new value to function C each time around the loop. Since the argument `i` is unchanged in the method, we have lifted its binding so that the other two blocks are within its

scope.

We used this rather simple example to illustrate the basic principles, but two JVML features prove quite challenging: subroutines and exception handlers.

7.2.1 Subroutines

The Java compiler uses *subroutines* to implement `finally` blocks (Lindholm and Yellin 1999). Other compilers that target JVML could, of course, use them for other reasons. The `jsr` instruction pushes a return address onto the stack and transfers control to the specified label. The `ret` instruction jumps back to the return address in the specified local variable.

Subroutines pose three major challenges. First, they are "polymorphic over the types of the locations they do not touch" (Stata and Abadi 1998). As long as a subroutine ignores local 2, say, it could contain an integer at one call site and a float at another. Second, since return addresses can be stored in local variables, subroutine calls and returns need not obey a stack discipline. Indeed, they need not return at all. In Java, we need only to place a `break` or `continue` inside a `finally` block to produce a subroutine which ignores its return address and jumps elsewhere. Finally, a subroutine might update a local variable. Since locals are not mutable in λJVM, the subroutine must explicitly pass the new value back to the caller.

We solve these problems using the *continuation-passing* idiom from functional programming. The subroutine takes a higher-order function (called the *return continuation*) in place of a return address. Any values the subroutine might change are passed to the return continuation as arguments; any free variables in the continuation are preserved across the call.

An example is worthwhile; see figure 7.5 on the facing page. The subroutine `S` has two call sites. In the first, local 1 is uninitialized; in the second, it contains a string. The subroutine either updates local 0 and returns normally or jumps directly to the end of

```
public static f(I)V
   jsr S
   ldc "Hello"
   astore_1
L: jsr S
   aload_1
   invoke println
   goto L
S: astore_2  ; ret addr
   iload_0
   ifeq R
   iinc 0 -1
   ret 2
R: return
```

Figure 7.5: A complex example with subroutines

```
public static f(I)V = λ(n:I)
letrec S = λ(i:I, r:(I) → V)
           if eq[I] i 0 then return
           else let j = add[I] i -1;
                r(j).
       L = λ(i:I, s:String)
           S(i, λ(j:I) invoke println s; L(j, s)).
S(n, λ(j:I) L(j, "Hello"))
```

Figure 7.6: Translation involving subroutines

the method. Bytecode verification is much trickier in the presence of subroutines, and our type inference phase is no different. We must unify the types of locals at different call sites, and decide which are passed to the subroutine, which are passed back to the caller, and which are otherwise preserved across the call.

A translation of the example appears in figure 7.6. The subroutine S takes an argument r of type (I) → V; this is the return continuation. In one branch, it returns from the method, in the other, it jumps to the continuation, passing the new value of local 0. Now consider the two call sites of S. Inside L the string s is a free variable of the functional argument, so it is preserved across the call.

This solution works quite well. We used Jasmin, a JVML assembler (Meyer and Down-

ing 1997), to generate a series of convoluted test cases that do not arise from typical Java compilers. Our code translated all of them correctly. We emphasize again that these higher-order functions can be compiled away quite efficiently in λJVM since all call sites are known.

7.2.2 Exception handlers

The `throw` expression should only be used to signal exceptions that exit the current method. For exceptions handled within the current method, we jump directly to a block implementing the handler. The `handle` suffix on primitive operations is inspired by the *also-unwinds-to* feature of C–– (Ramsey and Peyton Jones 2000), and it serves two distinct purposes.

Consider a primitive (such as `getfield`) that might raise an exception, but that will later be expanded to low-level code. The code will first perform a null check. If it succeeds, the offset corresponding to the field is dereferenced. If the check fails, then an exception is thrown. In this case, once the null check is expanded, the failure branch jumps directly to the handler and the `handle` annotation disappears.

Primitives, such as method invocation, that involve out-of-method function calls work differently. In these cases, the exception could be thrown further down the call stack. The `handle` annotation indicates to the compiler that the function has an abnormal return path. The `throw`, then, is just an abnormal return. Ramsey and Peyton Jones (2000) describe a code generation trick due to Atkinson, Liskov, and Scheifler (1978) where a table of continuation branches follows the call instruction. By adding an offset to the return address, the callee can select which return path to take.

In contrast to Java, λJVM does not specify a series of handlers based on the exception sub-class. The single handler must explicitly query the dynamic class of the exception and dispatch accordingly.

7.3 Verification

The JVM specification (Lindholm and Yellin 1999) defines a conservative static analysis for verifying the safety of a class file. Code which passes verification should not, among other things, be able to corrupt the virtual machine which executes it. One of the primary benefits of λJVM is that verification reduces to simple type checking. Most of the analyses required for verification are performed during translation to λJVM. The results are then preserved in type annotations, so type checking can be done in one pass. Our type checker is less than 260 lines of ML code, excluding the λJVM data structure definitions.

Two of the most complex aspects of class file verification are subroutines (Stata and Abadi 1998) and object initialization (Freund and Mitchell 1999). We have already seen how subroutines disappear, but let us explore in detail the problem of object initialization.

7.3.1 Object initialization

Our explanation of the problem follows that of Freund and Mitchell (1999). In Java source code, the `new` syntax allocates and initializes an object simultaneously:

```
Pt p = new Pt(i);   p.draw();
```

In bytecode, however, these are separate instructions:

```
new Pt                        ; alloc
dup
iload_0
invokespecial Pt.<init>(I)V   ; init
invokevirtual Pt.draw()V      ; use
```

Between allocation and initialization, the pointer can be duplicated, swapped, stored in local variables, etc. Once we invoke the initializer, all instances of the pointer become safe to use. We must track these instances with some form of alias analysis. The

following code creates two points; the verifier must determine whether the drawn point is properly initialized.

```
1: new Pt
2: dup
3: new Pt
4: swap
5: invokespecial Pt.<init>()V
6: pop
7: invokevirtual Pt.draw()V
```

This code would be incorrect without the `pop`.

Lindholm and Yellin (1999) describe the conservative alias analysis used by the Sun verifier. The effect of the `new` instruction is modeled by pushing the `Pt` type onto the stack along with an ‘uninitialized’ tag and the offset of the instruction which created it. To model the effect of the initializer, update all types with the same instruction offset, marking them as initialized. Finally, uninitialized objects must not exist anywhere in memory during a backward branch.

In λJVM, many aliases disappear with the local variable and stack manipulations. Every value has a name and a type. The `new` primop introduces a name with uninitialized object type c^0. The initializer then updates the type of its named argument in the environment. After translating the previous example to λJVM, it is clear that the drawn object is initialized:

```
let x = new Pt;
let y = new Pt;
invokespecial Pt.<init>()V x;
invokevirtual Pt.draw()V x;
```

After `invokespecial`, the type environment contains $x \mapsto$ `Pt` and $y \mapsto$ `Pt`^{0}.

Aliases can also occur in λJVM when the same pointer is passed as two arguments to a basic block. Translating the Java statement `new Pt (f? x : y)` to JVML introduces a branch between the `new` and the `invokespecial`. A naïve translation to λJVM might introduce an alias because the same pointer exists in two locations across basic blocks.

$$\frac{c \in \{\overline{c}\}}{c \leqslant \texttt{\{}\overline{c}\texttt{\}}} \qquad \frac{\{\overline{c_1}\} \subseteq \{\overline{c_2}\}}{\texttt{\{}\overline{c_1}\texttt{\}} \leqslant \texttt{\{}\overline{c_2}\texttt{\}}}$$

$$\frac{\tau \leqslant \tau'}{\tau\texttt{[]} \leqslant \tau'\texttt{[]}} \qquad \frac{c \leqslant c' \quad \forall c \in \{\overline{c}\}}{\texttt{\{}\overline{c}\texttt{\}} \leqslant c'}\ (*)$$

Figure 7.7: Selected typing rules

Our inference algorithm employs the same technique as the Sun verifier—mark the uninitialized object arguments with the offset of the `new` instruction. Then, we recognize arguments that are aliases and coalesce them.

7.3.2 Subtyping and set types

Two other interesting aspects of the λJVM type system are the subtype relation and the set types. The subtype relation ($\tau \leqslant \tau'$) handles numeric promotions such as `I` $\leqslant$ `F`. On class and interface names it mirrors the class hierarchy. The rules for other types are in figure 7.7. The curly typewriter braces `{ · }` are the λJVM set types and the Roman braces $\{\cdot\}$ are standard set notation.

The set elimination rule $(*)$ is required when a value of set type is used in a primop with a field or method descriptor. In function C of figure 7.4, for example, p is used as the self argument for method `draw` in class `Pt`. The type of p is `{IntPt,ColorPt}`, so the type checker requires `IntPt` $\leqslant$ `Pt` and `ColorPt` $\leqslant$ `Pt`.

In our example, we could have used the super class type `Pt` in place of the set `{IntPt,ColorPt}`, but with interfaces and multiple inheritance, this is not always possible. Both Goldberg (1998) and Qian (1999) have observed this problem; the example in figure 7.8 on the next page is from Knoblock and Rehof (2000). What is the type of `x` after the join? The only common super type of `A` and `B` is `Object`. But then the method invocations would not be correct. We must assign to `x` the set type `{A,B}`. For the first method invocation, the type checker requires that `{A,B}` $\leqslant$ `SA`. For the second invo-

```
interface SA { void saMeth(); }
interface SB { void sbMeth(); }
interface A extends SA, SB {…}
interface B extends SA, SB {…}

public static void (boolean f, A a, B b) {
    if (f) { x = a; }
    else { x = b; }
    x.saMeth();
    x.sbMeth();
}
```

Figure 7.8: The need for set types

cation, $\{\mathsf{A}, \mathsf{B}\} \leqslant \mathsf{SB}$. These subtyping judgments are easily derived from the interface hierarchy ($\mathsf{A} \leqslant \mathsf{SA}$, $\mathsf{B} \leqslant \mathsf{SA}$, $\mathsf{A} \leqslant \mathsf{SB}$, and $\mathsf{B} \leqslant \mathsf{SB}$) using the set elimination rule $(*)$.

We utilize subtypes either by subsumption (if v has type τ and $\tau \leqslant \tau'$ then v also has type τ') or as explicit coercions ($\texttt{let}\ x = \texttt{convert}[\tau, \tau']\ v\texttt{;}\ \ldots$ where $\tau \leqslant \tau'$). Our type checker accepts the former but can automatically insert explicit coercions as needed.

7.4 Implementation

We have implemented the translation from Java class files to λJVM. After parsing the class file into a more abstract form, methods are split into basic blocks. Next, type inference determines the argument types for each basic block. It determines subroutine calling conventions and eliminates aliased arguments. Finally, the translation phase uses the results of type inference to guide conversion to λJVM. Our type checker verifies that the translation produced something sensible and checks those JVM constraints that were not already handled during parsing and inference. λJVM is a significant component of our prototype; some measurements are given in the next chapter.

Because our application does not require it, we have not implemented serialization

for λJVM programs. We could borrow the byte codes from JVML for primops, and then use relative instruction offsets for representing `let`-bound names. Or we could follow Amme et al. (2001), who describe two innovative encoding techniques—referential integrity and type separation—in which only well-formed programs can be specified. That is, programs are well-formed by virtue of their encoding.

7.5 Related work

Katsumata and Ohori (2001) translate a subset of JVML into a λ-calculus by regarding programs as *proofs* in different systems of propositional logic. JVML programs correspond to proofs of the sequent calculus; λ-programs correspond to natural deductions. Translations between these systems yield translations of the underlying programs. This is a very elegant approach—translated programs are type-correct by construction. Unfortunately, it seems impossible to extend it to include JVML subroutines and exceptions.

Gagnon, Hendren, and Marceau (2000) give an algorithm to infer static types for local variables in JVML. Since they do not use a single-assignment form, they must occasionally split variables into their separate uses. Since they do not support set types, they insert explicit type casts to solve the multiple interface problem described above.

Amme et al. (2001) translate Java to SafeTSA, an alternative mobile code representation based on SSA form. Since they start with Java, they avoid the complications of subroutines as well as the multiple interface problem. Basic blocks must be split wherever exceptions can occur, and control-flow edges are added to the `catch` and `finally` blocks. Otherwise, SafeTSA is similar in spirit to λJVM.

Chapter 8

A Prototype Compiler for Java and ML

Previous chapters of this dissertation defined a flexible typed intermediate language, proposed efficient implementations of many Java features within that language, and described a high-level representation of Java bytecode. To demonstrate these ideas with real programs, we synthesized all of them within a prototype compiler. It compiles both Java and ML programs using the same back end support and runtime system.

8.1 Design

We started with version 110.30 of the Standard ML of New Jersey compiler (Appel and MacQueen 1991), featuring the FLINT typed intermediate language (Shao 1997; Shao, League, and Monnier 1998). Our first step was to develop and export support modules within the compiler that enable regular SML programs to build, compile, and link FLINT code directly. Previously, the FLINT functionality was available only as an integrated part of the SML compiler.

Figure 8.1 on the following page gives the signature of a support module that queries the SML/NJ static environment and provides representations of ML types, constructors, and values that can be linked into a new FLINT program before it is compiled. For

```
signature IMPORTS = sig
   type imports
   val empty      : imports
   val importVal  : imports × string
                    → imports × FLINT.lvar × FLINT.ty
   val importTyc  : string → Access.consig × FLINT.ty
   val importCon  : string → FLINT.dcon
   val close      : imports × JFlint.exp → JFlint.prog
   exception Unbound of string
end
```

Figure 8.1: Importing FLINT code from the SML/NJ static environment

example, to use the input/output library of ML from FLINT code, one will need the type of a text output stream:

```
Imports.importTyc "TextIO.outstream"
```

The imports type tracks all the values that are imported while building a piece of FLINT code, so that the close function can link them into create a closed program, ready to pass to the type checker or back end.

We added row kinds and existential types to FLINT and called this extension JFlint—figure 8.2 on the next page contains part of its signature. Compared to the abstract syntax of figure 4.1, the implemented version is in A-normal form. Expressions ending with id × exp bind their result to identifier id in the scope of exp. We updated the type checker and optimization phases to recognize the new features in JFlint, but left the code generator and runtime system unchanged.

Finally, we implemented the Java front end as a regular SML program. It parses Java class files, analyzes their methods, and translates them to λJVM. At this stage, we could perform class hierarchy analysis or other object-aware optimizations (Dean, Grove, and Chambers 1995; Dean et al. 1996) because the classes and method calls are still explicit.

From the constant pool, the front end determines what other classes must be loaded.

```
signature JFLINT = sig
  datatype value                                        (* identifiers and constants *)
    = Var of id | Int of Int32.int | String ...

  datatype exp
    = Letrec of fundec list × exp
    | Let    of id × exp × exp
    | Switch of value × (id × exp) list
    | Call   of id × value list
    | Return of value
    | Primop of primop × value list × id × exp
    | Record of value list          × id × exp
    | Load   of value × int         × id × exp
    | Store  of value × int × value      × exp
      ...
                                                   (* type manipulation instructions *)
    | Inst   of id × ty list                        × id × exp
    | Fold   of value × ty                          × id × exp
    | Unfold of value                               × id × exp
    | Pack   of ty list × (value × ty) list         × id × exp
    | Open   of value × id list × (id × ty) list    × exp
      ...
  withtype fundec = id × (id × ty) list × exp
end
```

Figure 8.2: Signature for JFlint code

It recursively parses and analyzes the requisite classes, then computes the strongly-connected components of the dependence graph. Finally, each mutually-dependent cluster of classes is translated separately to JFlint.

Intuitively, there are two stages to this translation. First, we analyze the class declaration to determine the layout of the fields and methods, and to generate representations of the object and class types. Figure 8.3 on the following page contains a fragment of ML code for this stage. Although not all the constructors are defined, its resemblance to the object type macros in figure 5.4 (page 60) should be clear.

The next stage does case analysis on the λJVM code and builds the corresponding lower-level JFlint representation. Figure 8.4 on the next page contains a code fragment for this stage. It shows two helper functions that build an **unfold** and **open** term,

```
val objRcd = T.lam ([T.ktyp, T.ksubm nm, T.ksubf nf, T.ktyp],
               T.rcd (T.row (T.rcd (T.app (T.proj ( allInst , 0),
                                           [T.var (1,3)])),
                      T.proj ( allInst , 1))))

val selfFn = T.lam ([T.ktyp, T.ksubm nm, T.ksubf nf],
               T.fix (T.ktyp, T.app (objRcd,
                 [T.var (2,0), T.var (2,1), T.var (2,2), T.var (1,0)])))

val objTy  = T.fix (T.ktyp,
               T.exist ([T.ksubm nm, T.ksubf nf],
                 [T.app ( selfFn , [T.var (2,0), T.var (1,0), T.var (1,1)])]))

val selfTy = T.app ( selfFn , [ objTy , T.var (1,0), T.var (1,1)])
```

Figure 8.3: Constructing representations of object types

```
(* generate unfold and open terms, in continuation passing style: *)
fun unfold (tyc, v, k) =
 let val x = T.mkLvar()
  in F.Unfold (v, tyc, T.id, x, k (F.Var x))
 end

fun unpack (ks, tyc, v, k) =
 let val x = T.mkLvar()
     fun f k = (T.mkLvar (), k)
  in F.Open (v, map f ks, [(x, tyc)], k (F.Var x))
 end

(* from case analysis on lambda JVM terms: *)
  | INVK (m, VIRTUAL, v0, vs) =>
     let val (im, acc) = memberAccess (im, m)
         val {objTy, selfTy, nm, nf, ...} = classInfoM m
      in compileValue (env, v0, fn v0 =>
         valueOf (objTy, v0, fn v0 =>
         unfold (objTy, v0, fn v0 =>
         unpack ([T.ksubm nm, T.ksubf nf], selfTy, v0, fn v0 =>
         unfold ( selfTy, v0, fn v1 =>
         select (SOME v1, acc, fn vm =>
         compileValues (env, vs, fn vs =>
         F.Let ([y], F.Call (vm, v0 :: vs),
         nextExp )))))))))
     end
```

Figure 8.4: Compiling invokevirtual

```
Standard ML of New Jersey v110.30 [JFLINT 1.2]
- Java.classPath := ["/home/league/r/java/tests"];
val it = () : unit
- val main = Java.run "Hello";
[parsing Hello]
[parsing java/lang/Object]
[compiling java/lang/Object]
[compiling Hello]
[initializing java/lang/Object]
[initializing Hello]
val main = fn : string list -> unit
- main ["Duke"];
Hello, Duke
val it = () : unit
- main [];
uncaught exception ArrayIndexOutOfBounds
  raised at: Hello.main([Ljava/lang/String;)V
- ^D
```

Figure 8.5: Compiling and running a Java program in SML/NJ

```
class Hello {
    public static void main(String [] args ) {
        System.out. println ("Hello ,␣" + args [0]);
    }
}
```

Figure 8.6: A trivial Java program

respectively. They each have a continuation argument to receive the new value and continue generating code. Again, some functions are omitted, but the resemblance to the formal method call translation (figure 5.7, rule 5.14, page 63) should be clear.

On the generated JFlint code, we run several contraction optimizations (inlining, common subexpression elimination, and so on), and type-check the code after each pass. We discard the type information before converting to MLRISC (George 1997) for final instruction selection and register allocation. To generate typed machine code, we would need to preserve types throughout the back end. The techniques of Morrisett et al. (1999b) would apply directly, since JFlint is based on System F.

Figure 8.5 on the preceding page shows the SML/JFlint system in action. The *slanted* text represents user input. The top-level loop accepts Standard ML code, as usual. The JFlint front end is controlled via the `Java` structure; its members include:

- `Java.classPath : string list ref` Initialized from the `CLASSPATH` variable, this is a list of directories where the loader will look for class files.
- `Java.load : string -> unit` looks up the named class using `classPath`, resolves and loads any dependencies, then compiles the byte codes and executes the class initializer.
- `Java.run : string -> string list -> unit` ensures that the named class is loaded, then attempts to call its `main` method with the given arguments.
- `Java.flush : unit -> unit` forces the Java subsystem to forget all previously loaded classes. Normally, loaded classes persist across calls to `load` and `run`; after `flush`, they must be loaded and initialized again.

The session in figure 8.5 sets the `classPath`, then loads the `Hello` class, and binds its `main` method, using partial application of `Java.run`. The method is then invoked twice with different arguments. The second invocation erroneously accesses `argv[0]`; this error surfaces as the ML exception `Java.ArrayIndexOutOfBounds`. The source code for the `Hello` class is in figure 8.6 on the page before.

In this prototype, SML code interacts only with a *complete* Java program. Since both run in the same runtime system, very fine-grained interactions are possible, but have not been our focus. Benton and Kennedy (1999) designed extensions to SML to allow seamless interaction with Java code when both are compiled for the Java virtual machine. Their design should work quite well in our setting also—and since JFlint is more expressive than JVML, we do not need to monomorphize functors and polymorphic functions.

This is essentially a *static* Java compiler, as it does not handle dynamic class loading or the `java.lang.reflect` API. These features are more difficult to verify using a static type system, but they are topics of ongoing research. The SML runtime system does not yet support kernel threads, so we have also ignored Java's concurrency features. Also, the runtime system does not, for now, dynamically load *native* code. This is a dubious practice anyway; such code has free reign over the runtime system, thus nullifying any safety guarantees won by verifying the pure Java code. Nevertheless, this restriction is unfortunate because it limits the set of existing Java libraries that we can use.

8.2 Synergy

One of the benefits of our design is the *synergy* between the encodings of Java and ML. JFlint does not have classes, methods, or modules as primitives. Rather, the records in JFlint model Java objects, vtables, classes and interfaces, plus ML records and the value parts of ML modules. Neither Java nor ML has a universal quantifier, but it is useful for encoding both Java inheritance and ML's parametric polymorphism. The existential type is essential for object encoding, but also useful for ML closures and abstract data types.

Contrast this with other common typed intermediate formats. JVM class files are very high-level and quite partial to the Java language. The bytecode language (JVML) includes no facilities for specifying data layouts or expressing many common optimizations. Compiling other languages for the JVM means making foreign constructs look and act like Java classes or objects. That so many translations exist (Tolksdorf) is a testament to the utility of the mobile code concept, and to the ubiquity of the JVM itself.

To some extent, the Microsoft Common Language Infrastructure (CLI) alleviates these problems (Microsoft Corp., *et al.*). It supports user-defined value types, stack allocation, tail calls, and pointer arithmetic (which is outside the verifiable subset). CLI

has the tendency to incorporate the union of all requested features. Its instructions distinguish, for example, between loading functions *vs.* values from objects *vs.* classes. Still, it tends to prefer a single-inheritance class-based language. A recent proposal to extend CLI for functional language interoperability (Syme 2001) added no fewer than 6 new types and 12 new instructions (bringing the total number of `call` instructions to 5) and it still does not support ML's higher-order modules (Harper, Mitchell, and Moggi 1990) or Haskell's constructor classes (Jones 1995).

We believe this synergy speaks well of our approach in general. Still, it does not mean that we can support all type-safe source languages equally well. Java and ML still have much in common; they work well with precise generational garbage collection and their exception models are similar enough. Weakly typed formats, such as C–– (Peyton Jones, Ramsey, and Reig 1999; Ramsey and Peyton Jones 2000), are more ambitious in supporting a wider variety of language features, including different exception and memory models. Practical type systems to support that level of flexibility are challenging, but are interesting for future work.

8.3 Implementation

To support efficient compilation, types are represented differently from code. Figure 8.7 contains part of the abstract interface to our type system. Compared to the syntax of figure 4.1 on page 32, the record labels are missing. We use integer offsets exclusively. Further, we use de Bruijn indices rather than named type variables.

If a type-preserving compiler is to scale well, extreme care must be taken in implementing the type representations and operations. Several years ago, we presented a series of techniques which, taken together, made the FLINT typed IL practical enough to use in a production compiler (Shao, League, and Monnier 1998). Different type structures arise in the Java encodings, but the techniques are as successful as ever.

```
signature JTYPE = sig
  type ty
  val var     : int × int → ty                        (* type variable *)
  val prim    : primtyc → ty
  val arrow   : ty list × ty → ty                     (* function type *)
  val record  : ty → ty                               (* record types *)
  val row     : ty × ty → ty
  val empty   : int → ty
  ...                                                 (* quantified types *)
  val forall  : kind list × ty list → ty
  val exists  : kind list × ty list → ty
  val fixpt   : kind list × ty list → ty
  val lam     : kind list × ty → ty                   (* higher-order *)
  val app     : ty × ty list → ty
end
```

Figure 8.7: Abstract interface for JFlint type representation

We represent types as directed acyclic graphs. Part of what the JTYPE interface hides is automatic hashing to ensure maximal sharing in the graph representation. Type variables are represented as pairs of integers which represent the lexical binding depth and offset. This means that types which differ only in their variable names *share* the same representation. Therefore, the equivalence of two types in normal form is simply pointer equivalence.

A common operation on types is the substitution of types for variables. Replacing variables using assignment is not an option because so much of the graph is shared. We create the instantiated type lazily, memoizing each result so that the next time we need the same substitution it is available immediately.

With these techniques, compile and verification times remain reasonable. A full type-preserving compile of the 12 classes in the CaffeineMark™ 3.0 embedded benchmark series takes 2.4 seconds on a 927 MHz Intel Pentium III Linux workstation. This is about 60% more than `gcj`, the GNU Java compiler (Bothner 1997). Since `gcj` is written in C and our compiler in SML, the gap is easily attributed to linguistic differences. Verifying both the λJVM and the JFlint code adds another half second.

Run times are promising, but can be improved. Our goal, of course, is to preserve type safety; speed is secondary. CaffeineMark runs at about a third the speed in SML/NJ compared to `gcj -O2`. This difference should not be attributed to type preservation; we have already shown that, with types erased, our object layouts and operations are just like the standard untyped implementations. The difference has more to do with many other properties of our compiler. First, many standard optimizations, especially on loops, have not been implemented in JFlint yet. Second, the code generator is still heavily tuned for SML; record representations, for example, are more boxed than they need to be. Finally, the runtime system is also tuned for SML; to support `callcc`, every activation record is heap-allocated and subject to garbage collection.

Chapter 9

Future Directions

We have shown that a strongly-typed compiler intermediate language can safely and efficiently accommodate two very different programming languages. The intermediate language, JFlint, is sound, decidable, and already supports ML. We developed novel, efficient techniques for compiling Java and have shown that well-typed Featherweight Java programs are mapped to well-typed Mini JFlint programs. Moreover, the encodings are synergetic with the translation of ML. In this final chapter, we briefly survey a few interesting avenues for future research.

9.1 More inclusive encodings

We developed our techniques in the context of Java, and they should apply to similar languages, such as C# (Liberty 2002). There are, however, more experimental languages—Moby (Fisher and Reppy 1999) and Loom (Bruce, Fiech, and Petersen 1997), for example—with different object models. They allow classes as module parameters, treat them as *first-class* values, and use different notions of subtyping. It is still not clear how to map these features *efficiently* to a typed λ-calculus (Vanderwaart 1999).

Fisher, Reppy, and Riecke (2000) proposed λinkς (read "links"), an *untyped* calculus

that can express and optimize the object models of a variety of languages, including Java, Moby, Loom, and OCaml. λinkς reasons about *slots* to build method suites either at compile time or link time. It also features *dictionaries* to dynamically map method labels to slots. With this flexibility, designing a sound type system for λinkς seems a daunting task.

Some recent work demonstrates that such reasoning just might be within the purview of a decidable type system. Shao et al. (2002), for example, reason about integers and integer addition at the type level so that array bounds checks can be optimized away. In their system, records and arrays are both defined using an underlying tuple constructor, where the length can be known either statically or dynamically. Perhaps the method suites of λinkς could be typed using similar ideas.

9.2 More substantial implementations

Our prototype shows that the encodings we developed can be implemented in a real compiler. It is not, unfortunately, substantial enough that it could be used as a drop-in replacement for a Java virtual machine. Certainly, there is more to be learned in the realm of efficient and effective implementation of type-preserving compilers.

We recently started to collaborate with a group at Intel Labs on their IA64 just-in-time compiler for the Microsoft .NET platform. The Intel group already found that limited type information preserved in the compiler is useful for optimization (disambiguating memory references, for example) and for accurate garbage collection (Stichnoth, Lueh, and Cierniak 1999; Cierniak, Lueh, and Stichnoth 2000). They understand that a more rigorous typing discipline would bring security benefits, and are curious about the impact type-preservation might have on the performance, reliability, and maintainability of their system.

This project is exciting for several reasons. First, .NET is a more ambitious platform

than Java, somewhat less partial to a single source language. Also, just-in-time compilation imposes new requirements, most notably on the speed of the compiler. Finally, working with researchers and developers in industry would help bring pragmatic concerns to the foreground. The project would certainly yield an exciting new batch of practical and theoretical problems to solve.

Bibliography

Abadi, Martín, and Luca Cardelli. 1996. *A Theory of Objects.* New York: Springer.

Abadi, Martín, Luca Cardelli, and Ramesh Viswanathan. 1996, January. "An Interpretation of Objects and Object Types." *Proc. Symp. on Principles of Programming Languages.* New York: ACM, 396–409.

Abadi, Martín, and Marcelo P. Fiore. 1996, July. "Syntactic Considerations on Recursive Types." *Proc. 11th Annual IEEE Symp. on Logic in Computer Science.* 242–252.

Alpern, Bowen, Mark Wegman, and F. Kenneth Zadeck. 1988, January. "Detecting equality of variables in programs." *Proc. Symp. on Principles of Programming Languages.* 1–11.

Alves-Foss, Jim, ed. 1999. *Formal Syntax and Semantics of Java.* Volume 1523 of *Lecture Notes in Computer Science.* Springer.

Amme, Wolfram, Niall Dalton, Jeffery von Ronne, and Michael Franz. 2001. "SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form." *Proc. Conf. on Programming Language Design and Implementation.* ACM.

Appel, Andrew W. 1992. *Compiling with Continuations.* Cambridge University Press.

———. 1998. "SSA is Functional Programming." *ACM SIGPLAN Notices*, April.

———. 2001, June. "Foundational Proof-Carrying Code." *Proc. IEEE Symp. on Logic in Computer Science (LICS).*

Appel, Andrew W., and David B. MacQueen. 1991, August. "Standard ML of New Jersey." Edited by Martin Wirsing, *3rd Int'l Symp. on Program. Lang. Impl. and Logic Program.*, Volume 528 of *LNCS.* New York: Springer-Verlag, 1–13.

Atkinson, Russell R., Barbara H. Liskov, and Robert W. Scheifler. 1978. "Aspects of implementing CLU." *Proc. ACM Annual Conference.* 123–129.

Barendregt, Henk. 1992. "Typed Lambda Calculi." In *Handbook of Logic in Computer Science*, edited by Samson Abramsky, Dov Gabbay, and Tom Maibaum, Volume 2. Oxford.

Benton, Nick, and Andrew Kennedy. 1999. "Interlanguage Working Without Tears: Blending SML with Java." *Proc. Int'l Conf. Functional Programming.* ACM, Paris, 126–137.

Booch, Grady. 1994. *Object-Oriented Analysis and Design, with Applications.* 2nd edition. Addison-Wesley.

Bothner, Per. 1997. "A GCC-based Java Implementation." *Proc. IEEE Compcon.*

Bruce, Kim B. 1994. "A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics." *J. Functional Programming* 4 (2): 127–206.

Bruce, Kim B., Luca Cardelli, and Benjamin C. Pierce. 1999. "Comparing Object Encodings." *Information and Computation* 155 (1–2): 108–133.

Bruce, Kim B., Adrian Fiech, and Leaf Petersen. 1997. "Subtyping is not a good 'Match' for Object-Oriented Languages." *Proc. European Conf. Object-Oriented Prog.*, Volume 1241 of *LNCS.* Berlin: Springer-Verlag, 104–127.

Canning, Peter, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. 1989, September. "F-bounded polymorphism for object-oriented programming." *Proc. Int'l Conf. on Functional Programming and Computer Architecture.* ACM, 273–280.

Cardelli, Luca. 1988. "A Semantics of Multiple Inheritance." *Information and Computation* 76 (2/3): 138–164 (February/March).

Cardelli, Luca, and Xavier Leroy. 1990, April. "Abstract Types and the Dot Notation." *Proc. IFIP Working Conf. on Programming Concepts and Methods.* Israel, 466–491.

Cierniak, Michał, Guei-Yuan Lueh, and James Stichnoth. 2000, June. "Practicing JUDO: Java Under Dynamic Optimizations." *Proc. Conf. on Programming Language Design and Implementation.* ACM, Vancouver.

Clinger, William, and Jonathan Rees. 1991, November. *Revised^4 Report on the Algorithmic Language Scheme.*

Colby, Christopher, Peter Lee, George C. Necula, Fred Blau, Ken Cline, and Mark Plesko. 2000, June. "A Certifying Compiler for Java." *Proc. Conf. on Programming Language Design and Implementation.* ACM, Vancouver.

Crary, Karl. 1999, January. "Simple, Efficient Object Encoding using Intersection Types." Technical Report CMU-CS-99-100, Carnegie Mellon University, Pittsburgh.

Crary, Karl, Robert Harper, and Sidd Puri. 1999. "What is a Recursive Module?" *Proc. Conf. on Programming Language Design and Implementation.* New York: ACM.

Dean, Jeffrey, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. 1996, October. "Vortex: An Optimizing Compiler for Object-Oriented Languages." *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications.* ACM, San Jose, 83–100.

Dean, Jeffrey, David Grove, and Craig Chambers. 1995. "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis." *Proc. European Conf. Object-Oriented Programming.*

Department of Defense. 1985, December. Trusted Computer System Evaluation Criteria. 5200.28-STD.

Drossopoulou, Sophia, and Susan Eisenbach. 1999. "Describing the Semantics of Java and Proving Type Soundness." In Alves-Foss 1999, 41–82.

Eifrig, Jonathan, Scott Smith, Valery Trifonov, and Amy Zwarico. 1995. "An Interpretation of Typed OOP in a Language with State." *Lisp and Symbolic Comput.* 8 (4): 357–397.

Fisher, Kathleen, Furio Honsell, and John C. Mitchell. 1994. "A Lambda Calculus of Objects and Method Specialization." *Nordic Journal of Computing* 1:3–37.

Fisher, Kathleen, and John C. Mitchell. 1998. "On the Relationship between Classes, Objects and Data Abstraction." *Theory and Practice of Object Systems* 4 (1): 3–25.

Fisher, Kathleen, and John Reppy. 1999. "The design of a class mechanism for MOBY." *Proc. Conf. on Programming Language Design and Implementation.* New York: ACM.

Fisher, Kathleen, John Reppy, and Jon G. Riecke. 2000. "A Calculus for Compiling and Linking Classes." *Proc. European Symp. on Program.* 135–149.

Flanagan, Cormac, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993, June. "The Essence of Compiling with Continuations." *Proc. Conf. on Programming Language Design and Implementation.* Albuquerque, 237–247.

Flatt, Matthew, Shriram Krishnamurthi, and Matthias Felleisen. 1999. "A Programmer's Reduction Semantics for Classes and Mixins." In Alves-Foss 1999, 241–269.

Freund, Stephen N., and John C. Mitchell. 1999. "A Type System for Object Initialization in the Java Bytecode Language." *ACM Trans. on Programming Languages and Systems* 21 (6): 1196–1250.

Gagnon, Etienne, Laurie Hendren, and Guillaume Marceau. 2000. "Efficient Inference of Static Types for Java Bytecode." *Proc. Static Analysis Symp.*

George, Lal. 1997. "Customizable and Reusable Code Generators." Technical Report, Bell Labs.

Girard, J. Y. 1972. "Interpretation Fonctionnelle et Elimination des Coupures dans l'Arithmetique d'Ordre Superieur." Ph.D. diss., University of Paris VII.

Glew, Neal. 2000a, October. "An Efficient Class and Object Encoding." *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications.* ACM.

———. 2000b, January. "Low-Level Type Systems for Modularity and Object-Oriented Constructs." Ph.D. diss., Cornell University.

———. 2000c, August. Personal communication.

Goguen, Healfdene. 1995. "Typed Operational Semantics." In *Typed Lambda Calculi and Applications*, edited by Mariangiola Dezani-Ciancaglini and Gordon Plotkin, Volume 902 of *LNCS*, 186–200. Berlin: Springer-Verlag.

Goldberg, Allen. 1998. "A Specification of Java Loading and Bytecode Verification." *Conf. on Computer and Communications Security.* ACM, 49–58.

Gosling, James, Bill Joy, Guy Steele, and Gilad Bracha. 2000. *The Java Language Specification.* 2nd edition. The Java Series. Reading, Mass.: Addison-Wesley.

Harper, Robert, John C. Mitchell, and Eugenio Moggi. 1990. "Higher-Order Modules and the Phase Distinction." *Proc. Symp. on Principles of Programming Languages.* ACM, 341–354.

Harper, Robert, and Greg Morrisett. 1995, January. "Compiling Polymorphism Using Intensional Type Analysis." *Proc. Symp. on Principles of Programming Languages*. New York: ACM, 130–141.

Harper, Robert, and Chris Stone. 1998. "A Type-Theoretic Interpretation of Standard ML." Chapter 12 of *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, edited by Gordon Plotkin, Colin Stirling, and Mads Tofte, 341–388. Cambridge, Mass.: MIT Press.

Hofmann, Martin, and Benjamin C. Pierce. 1994. "A Unifying Type-Theoretic Framework for Objects." *Journal of Functional Programming*, January.

Igarashi, Atsushi, Benjamin C. Pierce, and Philip Wadler. 2001. "Featherweight Java: A Minimal Core Calculus for Java and GJ." *ACM Trans. on Programming Languages and Systems* 23 (3): 396–450 (May).

Jones, Mark P. 1995. "A system of constructor classes: overloading and implicit higher-order polymorphism." *J. Functional Programming* 5 (1): 1–35.

Kamin, Samuel. 1988, January. "Inheritance in Smalltalk-80: A Denotational Definition." *Proc. Symp. on Principles of Programming Languages*. New York: ACM, 80–87.

Katsumata, Shin-ya, and Atsushi Ohori. 2001, April. "Proof-Directed Decompilation of Low-Level Code." *Proc. 10th European Symp. on Programming (ESOP)*, Volume 2028 of *LNCS*. Geneva.

Kelsey, Richard. 1995, March. "A correspondence between continuation passing style and static single assignment form." *Proc. Workshop on Intermediate Representations*. ACM, 13–22.

Knoblock, Todd, and Jakob Rehof. 2000. "Type Elaboration and Subtype Completion for Java Bytecode." *Proc. Symp. on Principles of Programming Languages*. 228–242.

Krall, Andreas, and Reinhard Grafl. 1997. "CACAO—A 64-bit Java VM Just-In-Time Compiler." *Proc. ACM PPoPP'97 Workshop on Java for Science and Engineering Computation.*

Landin, P. 1964. "The Mechanical Evaluation of Expressions." *Computer J.* 6 (4): 308–320.

League, Christopher, Zhong Shao, and Valery Trifonov. 1999, September. "Representing Java Classes in a Typed Intermediate Language." *Proc. Int'l Conf. Functional Programming*. Paris: ACM, 183–196.

———. 2002a, March. "Precision in Practice: A Type-Preserving Java Compiler." Technical Report 1223, Yale University, New Haven.

———. 2002b. "Type-Preserving Compilation of Featherweight Java." *ACM Trans. on Programming Languages and Systems* 24, no. 2 (March).

League, Christopher, Valery Trifonov, and Zhong Shao. 2001a, July. "Functional Java Bytecode." *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.

———. 2001b, January. "Type-Preserving Compilation of Featherweight Java." *Proc. Int'l Workshop on Foundations of Object-Oriented Languages*. London.

Liberty, Jesse. 2002. *Programming C#*. 2nd edition. O'Reilly.

Lindholm, Tim, and Frank Yellin. 1999. *The Java Virtual Machine Specification*. 2nd edition. Addison-Wesley.

Lutz, Mark. 2001. *Programming Python*. 2nd edition. O'Reilly.

Meyer, Jon, and Troy Downing. 1997. *Java Virtual Machine*. O'Reilly.

Microsoft Corp., *et al.* "Common Language Infrastructure." Drafts of the ECMA TC39/TG3 standardization process. `http://msdn.microsoft.com/net/ecma/`.

Milner, Robin, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The Definition of Standard ML (Revised)*. MIT Press.

Mitchell, John C., and Gordon D. Plotkin. 1988. "Abstract Types have Existential Type." *ACM Transactions on Programming Languages and Systems* 10 (3): 470–502 (July).

Morrisett, Greg, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. 1999a, May. "TALx86: A Realistic Typed Assembly Language." *Proc. Workshop on Compiler Support for Systems Software*. New York: ACM, 25–35.

Morrisett, Greg, David Tarditi, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. 1996. "The TIL/ML Compiler: Performance and Safety Through Types." *Proc. Workshop on Compiler Support for Systems Software*. New York: ACM.

Morrisett, Greg, David Walker, Karl Crary, and Neal Glew. 1999b. "From System F to Typed Assembly Language." *ACM Trans. on Programming Languages and Systems* 21 (3): 528–569 (May).

Necula, George C. 2001, September and October. Personal communication.

Necula, George C., and Peter Lee. 1998, June. "The Design and Implementation of a Certifying Compiler." *Proc. Conf. on Programming Language Design and Implementation*. ACM, Montréal, 333–344.

Peyton Jones, Simon, Norman Ramsey, and Fermin Reig. 1999. "C––: a Portable Assembly Language that Supports Garbage Collection." In *Proc. Conf. on Principles and Practice of Declarative Programming*, edited by Gopalan Nadathur, Volume 1702 of *LNCS*, 1–28. Springer.

Peyton Jones, Simon L., Cordelia Hall, Kevin Hammond, Will Partain, and Philip Wadler. 1992, December. "The Glasgow Haskell Compiler: A Technical Overview." *Proc. UK Joint Framework for Information Technology*.

Pierce, Benjamin C., and David N. Turner. 1994. "Simple Type-Theoretic Foundations for Object-Oriented Programming." *J. Functional Programming* 4 (2): 207–247 (April).

Proebsting, Todd A., Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. 1997. "Toba: Java for Applications: A Way Ahead of Time (WAT) Compiler." *Proc. Third Conf. on Object-Oriented Technologies and Systems (COOTS'97)*.

Qian, Zhenyu. 1999. "A Formal Specification for Java Virtual Machine Instructions for Objects, Methods, and Subroutines." In Alves-Foss 1999, 271–312.

Ramsey, Norman, and Simon Peyton Jones. 2000. "A Single Intermediate Language that Supports Multiple Implementations of Exceptions." *Proc. Conf. on Programming Language Design and Implementation.* ACM, 285–298.

Rémy, Didier. 1993. "Syntactic theories and the algebra of record terms." Technical Report 1869, INRIA.

Rémy, Didier, and Jérôme Vouillon. 1997, January. "Objective ML: A Simple Object-Oriented Extension of ML." *Proc. Symp. on Principles of Programming Languages.* New York: ACM, 40–53.

Reynolds, John C. 1972. "Definitional Interpreters for Higher-Order Programming Languages." *Proc. 25th ACM Nat'l Conf.* Boston, 717–740.

———. 1974. "Towards a Theory of Type Structure." *Proc. Colloque sur la Programmation*, Volume 19 of *LNCS*. Berlin: Springer-Verlag, 408–425.

Shao, Zhong. 1997, June. "An Overview of the FLINT/ML Compiler." *Proc. Int'l Workshop on Types in Compilation.* Amsterdam.

Shao, Zhong, and Andrew W. Appel. 1995, June. "A Type-Based Compiler for Standard ML." *Proc. Conf. on Programming Language Design and Implementation.* La Jolla: ACM, 116–129.

Shao, Zhong, Christopher League, and Stefan Monnier. 1998, September. "Implementing Typed Intermediate Languages." *Proc. Int'l Conf. Functional Programming.* Baltimore: ACM, 313–323.

Shao, Zhong, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. 2002, January. "A Type System for Certified Binaries." *Proc. Symp. on Principles of Programming Languages.* Portland, 217–232.

Stata, Raymie, and Martín Abadi. 1998, January. "A Type System for Java bytecode subroutines." *Proc. Symp. on Principles of Programming Languages.* ACM, San Diego, 149–160.

Stichnoth, James M., Guei-Yuan Lueh, and Michał Cierniak. 1999, May. "Support for Garbage Collection at Every Instruction in a Java Compiler." *Proc. Conf. on Programming Language Design and Implementation.* ACM, Atlanta, 118–127.

Stroustrup, Bjarne. 1997. *The C++ Programming Language.* 3rd edition. Addison-Wesley.

Syme, Don. 1999. "Proving Java Type Soundness." In Alves-Foss 1999, 83–118.

———. 2001. "ILX: Extending the .NET Common IL for Functional Language Interoperability." *Proc. BABEL Workshop on Multi-Language Infrastructure and Interoperability.* ACM.

Tarditi, David, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. 1996. "TIL: A Type-Directed Optimizing Compiler for ML." *Proc. Conf. on Programming Language Design and Implementation.* New York: ACM.

Thomas, David, and Andrew Hunt. 2000. *Programming Ruby: A Pragmatic Programmer's Guide.* Addison-Wesley.

Thompson, Ken. 1984. "Reflections on Trusting Trust." *Communications of the ACM* 27 (8): 761–763. Turing Award lecture.

Tolksdorf, Robert. “Programming Languages for the JVM.” `http://flp.cs.tu-berlin.de/~tolk/vmlanguages.html`.

Vanderwaart, Joseph C. 1999. “Typed Intermediate Representations for Compiling Object-Oriented Languages.” Williams College Senior Honors Thesis.

Wallach, Dan S., Andrew W. Appel, and Edward W. Felten. 2000. “SAFKASI: A Security Mechanism for Language-Based Systems.” *ACM Trans. on Software Engineering and Methodology* 9, no. 4.

Wright, Andrew, Suresh Jagannathan, Cristian Ungureanu, and Aaron Hertzmann. 1998, March. “Compiling Java to a Typed Lambda-Calculus: A Preliminary Report.” *Proc. Int’l Workshop on Types in Compilation*, Volume 1473 of *LNCS*. Berlin: Springer, 1–14.