DESIGNING FLEXIBLE, MODULAR LINGUISTIC ABSTRACTIONS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Yizhou Zhang

August 2019

DESIGNING FLEXIBLE, MODULAR LINGUISTIC ABSTRACTIONS

Yizhou Zhang, Ph.D.

Cornell University 2019

Programming-language design is more active than ever: existing languages are evolving continually and rapidly, and new languages keep springing up. While this constant iteration of language design aims to help programmers manage a growing software complexity, programmers are still frequently frustrated by poor design decisions in even the most essential aspects of modern programming languages. Less than satisfactory solutions to generic programming and exception handling typify this situation: the inadequacy of current solutions has even forced language designers to abandon these problematic language features. This is an unfortunate state of affairs.

Language design does not have to be about abandoning old features or piling on new ones. This dissertation proposes novel linguistic abstractions for the aforementioned design problems, offering ease of use, expressive power, strong guarantees, and good performance all at the same time.

It introduces a new mechanism for generic programming, embodied in the Genus programming language. Genus adds expressive power and strengthens static checking, while handling common usage patterns simply. The power of Genus is then integrated into a second language design, Familia, that unifies several polymorphism mechanisms in a lightweight package. Evaluation suggests the design of Genus and Familia addresses the need for genericity and extensibility in developing large, complex software.

This dissertation also introduces a new mechanism for exception handling. By allowing exceptions to tunnel through handlers, the design offers both the static assurance of checked exceptions and the flexibility of unchecked exceptions.

This tunneling semantics is then generalized to a broader class of control effects to address a fundamental modularity problem: it prevents effect-polymorphic abstractions from handling effects by accident. This claim about abstraction safety is formally accounted for.

We hope that the language-design ideas presented here will make their way into mainstream programming languages and help make it easier to write and reason about software.

BIOGRAPHICAL SKETCH

Yizhou Zhang (Chinese: 张屹洲) grew up in Changzhou, where he attended Changzhou High Schoool. He obtained a Bacholor of Science in Software Engineering from Shanghai Jiao Tong University. Intrigued by research on programming languages, Yizhou applied to and entered the Computer Science Ph.D. program at Cornell University. Yizhou obtained a Masters of Science in Computer Science in 2016, and was awarded his Ph.D. in 2019.

For my family

ACKNOWLEDGMENTS

I am deeply indebted to my advisor, Andrew Myers, by whom my intellectual growth and academic character were heavily influenced. I thank Andrew for his unfailing support and incisive guidance, and also for the latitude in working on things that most interest me. His intellectual depth and breadth, his scientific creativity and curiosity, his taste for research problems, and his enthusiasm for research and teaching have been a constant source of inspiration.

I am grateful to my other special-committee members, Dexter Kozen, Ross Tate, and Bart Selman, for their encouragement and for their useful critiques on the dissertation work.

Barbara Liskov and Guido Salvaneschi have been role models and great collaborators. I benefited immensely from working on the Genus project with Andrew, Barbara, and Guido in my earlier years as a Ph.D. student. I learned how to do research by watching them doing it.

Over the years, I had the great pleasure to work alongside many incredible researchers and students at Cornell. Jed Liu, K. Vikram, Danfeng Zhang, Owen Arden, Chinawat Isradisaikul, Tom Magrino, Isaac Sheff, Matthew Milano, Ethan Cecchetti, Rolph Recto, Drew Zagieboylo, Josh Acay, and Siqiu Yao made the Applied Programming Languages Group a truly enjoyable environment to discuss research and provide feedback. Matthew Loring, Quinn Beightol, Jonathan Chan, Matthew Gharrity, Daniel Weber, and Jacopo Banfi were a delight to work with. Heartfelt thanks to Yang Yuan, Chen Wang, and many others for their friendship.

I thank my parents for providing me with a good education and for being loving and supportive throughout the journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The complexity of computer software is growing constantly. Taming this complexity places an ever growing demand for expressive power and static assurance on programming languages.

As responses to this demand, existing language designs—including Rust [154], Swift [169], JavaScript [62], Java [167], C# [90], Scala [133], and Haskell [67], to name a few—are evolving continually and rapidly, and new languages keep emerging.

While these efforts help address some challenges faced by programmers, awkward design decisions persist in even the most essential aspects of modern programming languages. This situation is typified by the unsatisfactory support for generic programming and for exception handling in existing languages. In fact, the lack of a good solution to generics or exceptions even induced language designers to abandon the problematic language features. For example, C# was designed without support for statically checked exceptions, but only because the language designers did not know how to design such an exception mechanism well [89]. The Go programming language was designed without support for generic programming, only to find itself needing a solution now because it is failing to scale to large software projects [50].

This dissertation examines these challenging problems in language design, and proposes new linguistic abstractions for code genericity and for handling exceptions and other control effects alike. These linguistic abstractions are flexible and modular to use, making it easier to construct and reason about large, complex software. Specifically, the dissertation advances the state of the art in the following two aspects:

- The dissertation presents a new exception mechanism that offers, at the same time, expressive power, static guarantees, and good performance, therefore addressing unsatisfactory tradeoffs in previous approaches that have made exception handling an unattractive language feature. Furthermore, it generalizes the approach to a broader class of control-flow effects (known as algebraic effects), and formally pins down the claim that this new semantics of algebraic effects protects abstraction boundaries.
- The dissertation presents an expressive yet lightweight language mechanism for generic programming, offering better code reuse and enforcing stronger static checking. It also unifies this genericity mechanism with the more common object-oriented (OO) programming paradigm, and further integrates the power of family polymorphism. The result is a compact set of linguistic abstractions that enable powerful forms of polymorphism and extensibility.

The rest of this chapter overviews these contributions.

1.1 Fixing the Tragedy of Exceptions

Since the advent of exception-handling mechanisms in the 1970s, there has been debate about whether exceptions should be subject to static checking. Unhandled exceptions crash programs, so a static checking should in principle make software more reliable. However, checked exceptions are so rigid that programmers often intentionally subvert static checking. This problem has become more apparent as programming with higher-order abstractions has become more common. As a result, new languages are being designed without statically checked exceptions,

simply because the designers do not know how to design such an exception mechanism well. This is an unfortunate state of affairs.

This dissertation proposes a new *tunneling* semantics for exceptions. It breaks with the commonly perceived but rather rigid dichotomy between checked exceptions and unchecked exceptions. Instead, guided by modular thinking, exceptions are statically checked, but only in contexts aware of them. Otherwise, exceptions tunnel through handlers.

Tunneling combines the benefits of static checking with the flexibility of unchecked exceptions. An evaluation of this design on real-world codebases shows it is effective in improving expressivity and safety: tunneled exceptions avoid tedious antipatterns, and more importantly, restore static checking and local reasoning, uncovering bugs including unhandled and accidentally handled exceptions.

The new design also helps implement exceptions efficiently. An implementation of the new mechanism outperforms Java on exception-heavy programs, and achieves comparable performance on standard benchmarks, despite the extra bookkeeping needed for tunneling.

1.2 Protecting Abstraction against Accidental Handling

Algebraic effects [16, 147, 148] are an emerging language abstraction that is quickly gaining popularity among language designers and programmers. They subsume a wide variety of built-in language features for control flow, including exceptions, coroutine iterators, and async–await.

Unfortunately, the safety of algebraic effects is threatened by the possibility that effect-polymorphic abstractions may handle, by accident, effects they are not designed to handle. While the aforementioned tunneling semantics for

exceptions can be adapted to address this problem for algebraic effects, it remains an open problem to account formally and rigorously for what it means for a language to safely prevent accidental handling.

A key insight is that accidental handling is an abstraction violation. Current semantics of algebraic effects allow a client to observe different behaviors about two implementations of the same abstraction, when one of them happens to use effects internally (and can thus handle effects by accident). Implementation details leak through abstraction boundaries!

To formally capture this insight, we develop a logical-relations model for a core language equipped with tunneled algebraic effects, and prove it satisfies an "abstraction theorem". This logical-relations model offers a sound reasoning process for proving the equivalence of pure and effectful program fragments, justifying the claim that tunneling enforces abstraction.

1.3 Redesigning Generics for Object-Oriented Languages

The benefits of generics are evident: more code reuse and more modularity. But it appears that current OO languages still offer an unsatisfactory tradeoff among expressivity, modularity, and usability. Designing a generics mechanism requires answering the questions of how to expose operations of type parameters and how to show type arguments have these operations. Current solutions to these questions fall short in expressivity and safety. For example, languages such as Java and C# expose operations of type parameters using subtyping constraints, making it impossible to use generics with type arguments that are not declared in advance as subtypes. As a result, programmers—and even Java's own collections framework—shy away from using subtyping constraints and resort to unsafe workarounds.

This dissertation provides the design and implementation of a new generics mechanism, embodied in a language called Genus. Genus introduces a pair of language constructs called *constraints* and *models*. Constraints specify operations of type parameters, and models satisfy constraints for type arguments, with no preplanning needed. Constraints and models are inspired by Haskell's type classes and instances [176]. But unlike type classes, constraints can be witnessed in more than one way, and the type system, for modularity, prevents the programmer from confusing these witnesses.

Genus makes generics safer and more expressive, as demonstrated by an evaluation that ports significant Java codebases into Genus. By optimizing generic code for particular type arguments, the Genus compiler can deliver very good performance. The deep integration with OO subtyping and variance poses algorithmic challenges to inference; yet inference remains decidable with generally accepted syntactic restrictions.

1.4 A Deep Unification of Polymorphism Mechanisms

The language design literature has accumulated a rich set of powerful language mechanisms for polymorphism. But it remains an elusive goal to harmoniously integrate these distinct mechanisms in a single language. The design of Genus is a step towards this goal; it integrates constrained parametric polymorphism (in the form of type classes and instances) with object-oriented polymorphism. But the addition of constraints and models burdens an already feature-rich language with entirely new kinds of constructs. Even for Haskell, it has been argued that type classes and instances introduce feature redundancy that confront programmers with added surface complexity.

This dissertation presents a language design called Familia that integrates several polymorphism mechanisms in a deep way. Familia is lightweight—it can be used as an ordinary Java-like OO language. By exploiting a duality between object-oriented polymorphism and type-class-based parametric polymorphism, Familia readily supports expressive generics à la Genus, but without needing additional language constructs. The design further unifies *associated types* found in Haskell type classes with *family polymorphism* approaches found in OO languages, allowing a group of related interfaces, classes, and modules to be extended in a coordinated and type-safe way.

A case study of using Familia to implement a highly reusable program analysis framework suggests that this set of design ideas coheres, and has a high payoff in increasing code extensibility for large, complex software. A formalization of Familia in a core language shows the design is type-safe.

1.5 Roadmap and Published Work

The remainder of this dissertation proceeds as follows. Chapter 2 presents the design and implementation of the new exception mechanism. Chapter 3 generalizes the tunneling semantics to algebraic effects and develops the theoretical underpinning of abstraction-safe effect handlers. Chapter 4 presents the design and implementation of the new generics mechanism embodied in the Genus language. Chapter 5 discusses the design of Familia, a language that unifies Genus-style generics, OO programming, and family polymorphism. Chapter 6 concludes.

These chapters are based on the following published work of the author: Zhang et al. [193], Zhang and Myers [189], Zhang et al. [190], and Zhang and Myers [186].

CHAPTER 2

ACCEPTING BLAME FOR TUNNELED EXCEPTIONS

Exceptions make code more reliable by helping programmers handle abnormal or unusual run-time conditions. The core idea is to transfer control in a nonlocal way to handler code that can be factored out from common-case code. This separation of concerns simplifies code and prompts programmers not to forget about exceptional conditions.

There has been disagreement since the 1970s about how or whether exceptions should be subject to static checking [80, 111]. This disagreement continues to the present day [78]. Some currently popular languages—Java [82] and Swift [169]—offer *checked exceptions* that the compiler statically ensures are handled. However, exceptions are not part of type checking in other popular languages such as C++ [164], C# [90], Scala [133], and Haskell [143].

Proponents of static checking argue that exceptions represent corner cases that are easy to forget. The evidence suggests they have a point. One study of a corpus of C# code [36] determined that 90% of the possible exceptions are undocumented. Undocumented exceptions make it hard to know whether all exceptions are handled, and these unhandled exceptions percolate up through abstraction layers, causing unexpected software failures. Statically checked exceptions help programmers build more robust code [175].

Opponents of static checking argue that the annotation burden of statically checked exceptions does not pay off—that statically checked exceptions are too rigid to support common design patterns and common ways in which software evolves [61, 180]. They, too, have a point. The problems with statically checked exceptions have become more apparent in recent years as object-oriented (OO) languages like C#, Scala, and Java have acquired lambda expressions and the use

of higher-order functions has become more common. As a result, the promise of exceptions to help make software more reliable has been partly lost.

Studies of the effectiveness of exception mechanisms have concluded that existing mechanisms do not satisfy the appropriate design criteria [34, 35]. C# does not statically check exceptions because its designers did not know how to design such an exception mechanism well, saying "more thinking is needed before we put some kind of checked exceptions mechanism in place" [89]. It seems the long-running conflict between checked and unchecked exceptions can be resolved only by a new exception mechanism.

This chapter aims to provide a better exception mechanism, one that combines the benefits of static checking with the flexibility of unchecked exceptions. The new mechanism gives programmers static, compile-time guidance to ensure exceptions are handled, but works well with higher-order functions and design patterns. It adds little programmer burden and even reduces that burden. The run-time overhead of the mechanism is low, because exception handling does not require stack-trace collection in common use cases and avoids the need to wrap checked exceptions inside unchecked ones.

Two main insights underlie the new design. The first is that the distinction between "checked" and "unchecked" should not be a property of the type of the exception being raised, as it is in Java, but rather a property of the context in which the exception propagates. In contexts that are *aware* of an exception, the exception should be checked statically to ensure that it is handled. To handle higher-order functions and design patterns, however, some contexts must be *oblivious* to exceptions propagating through them; exceptions should *tunnel* uncaught through oblivious contexts, effectively *unchecked*.

This principle implies that the same exception may be *both* checked and unchecked at different points during its propagation. To prevent oblivious code from *accidentally* catching exceptions, a second insight is needed: exceptions can be distinguished by expanding the space of exception identifiers with an additional label that describes the exception-aware context in which this exception can be caught. These labels can be viewed as an extension of the notion of *blame labels* found in previous work on gradual typing [178]. Unlike with gradual typing, this sort of "blame" is not a programmer error; it is instead a way to indicate that exceptions should tunnel through the oblivious code until they arrive at the right exception-aware context.

We start the rest of the chapter by exploring requirements for a good exception mechanism in Section 2.1. Sections 2.2–2.5 present our new exception mechanism informally in the context of a Java-like language. Section 2.6 defines a core language whose semantics show more precisely how the mechanism works. Using this core language, we prove the key theorem that all exceptions are handled explicitly. Section 2.7 describes our implementation of the new exception mechanism in the context of the Genus programming language [190]. The effectiveness of the mechanism is evaluated in Section 2.8, using code drawn from real-world codebases. Related work is explored more fully in Section 2.9.

2.1 Design Principles for Exceptions

The goal of an exception mechanism is to simplify and regularize the handling of exceptional events, making programs more reliable. However, there are two quite different classes of exceptional events, with different design goals:

- **Failures.** Some events cannot reasonably be expected to be handled correctly by the program—especially, events that arise because of programmer

mistakes. Other events such as running out of memory also fall into this category.

- **Unusual conditions.** Other events arise during correct functioning of the program, in response to an unusual but planned-for state of the environment, or even just an unusual case of an algorithm.

These two classes place different requirements on the exception mechanism. For failures, efficiency is not a concern because the program is not expected to recover. However, programmers need the ability to debug the (stopped) program to discover why the failure occurred, so it is important to collect a stack trace. Furthermore, since failures imply violation of programmer assumptions, having to declare them as part of method signatures or write handler code for them is undesirable. Nonetheless, there are cases where the ability to catch failure exceptions is useful, such as when building frameworks for executing code that might fail.

For the second class of exceptions, unusual conditions, the design goals are different. Now efficiency matters! Because exceptions are slow in many common languages, programmers have learned to avoid using them. One insight is that because unusual conditions are part of the correct functioning of the program, the overhead of collecting a stack trace is unnecessary.

Unfortunately, existing languages tend not to support the distinction between these two exception classes well. For example, typical Java usage always leads to stack-trace collection, making exceptions very expensive. At the same time, code is cluttered with handlers for impossible exceptions.

2.1.1 Higher-Order Functions and Exceptions

Current exception mechanisms do not work well in code that uses higher-order functions. An example is an ML-style `map` method that applies a function argument to each element of a list, returning a new list. Callers of the higher-order function may wish to provide as an argument a function that produces exceptions to report an unusual condition that was encountered, such as an I/O exception. Of course, we want these exceptions to be handled. But the implementation of `map` knows nothing about these exceptions, so if such exceptions do occur, they should be handled by the caller of `map`. It is unreasonable for `map` either to handle or to declare them—its code should be *oblivious* to the exceptions.

This example illustrates why otherwise statically typed functional programming languages such as ML and Haskell do not try to type-check exceptions statically [104]. The problem has also become more prominent in modern OO languages that have added *lambda expressions* and are increasingly relying on libraries that use them (e.g., JavaFX, Apache Spark). The problem is encountered even without lambda expressions, though; for example, Java's `Iterator` interface declares no exceptions on its methods, which means that implementations of `Iterator` are not allowed to generate exceptions—unless they are made "unchecked".

Our goal is to achieve the notational convenience of the functional programming languages along with the assurance that exceptions are handled, which is offered by languages such as Java, Swift [169], and Modula-3 [128]. We propose that a higher-order function like `map` should be implementable in a way that is *oblivious* to the exceptions possibly generated by its argument. The possible exceptions of the argument should not be declared in the signature of `map`; nor should the code of `map` have to say anything about these exceptions.

A subtle problem arises when a higher-order function like `map` uses exceptions inside its own implementation. If the exceptions of the argument and the internal exceptions collide, the `map` code could then *accidentally* catch exceptions that are not intended for it—an effect we call *exception capture* by analogy with the problem of variable capture in dynamically scoped languages [163]. For modularity, a way is needed to *tunnel* such exceptions through the intermediate calling context. In fact, accidentally-caught exceptions are a real source of serious bugs [52, 70].

An alternative to our oblivious-code approach that has been suggested previously [79, 155, 174] is to parameterize higher-order code like `map` over the unknown exceptions of its argument. This *exception polymorphism* approach requires writing annotations on oblivious code yet still permits accidental exception capture.

2.1.2 Our Approach

Backed by common sense and some empirical evidence, we believe that code is more reliable when compile-time checking guides programmers to handle exceptional cases. It is disappointing that recent language designs such as C# and Scala have backed away from statically declared exceptions.

We propose a new statically checked exception mechanism that addresses the weaknesses of prior exception mechanisms:

- It supports static, local reasoning about exceptions. Local reasoning is efficient, but more importantly, it aids programmer understanding. A code context is required to handle only the exceptions it knows about statically.
- The mechanism is cheap when it needs to be: when exceptions are used for nonlocal control flow rather than failures.

- In the failure case, however, the mechanism collects the stack trace needed for debugging.
- It supports higher-order functions whose arguments are other functions that might throw exceptions to which the higher-order code is oblivious.
- It avoids the exception-capture problem both for higher-order functions and for failures.

2.2 The Exception Mechanism

We use Java as a starting point for our design because it is currently the most popular language with statically checked exceptions. Our design is presented as a version of the Genus language, a variant of Java with an expressive constrained genericity mechanism [190]. The essential ideas should apply equally well to other languages, such as Java, C# [118], Scala [133], and ML [121]. Since exception-oblivious code (like `map`) is often generic, it is important to study how exceptions interact with sophisticated generics.[1]

Previous languages have either had entirely "checked" or "unchecked" exceptions (in Java's terminology), or, as in Java, have assigned exception types to one of these two categories. Our insight is that "checked" vs. "unchecked" is a property of the context of the exception rather than of its type. Any exception should be "checked" in a context that is not oblivious to the exception and can therefore handle it. But in a context that is oblivious to the exception, the exception should be treated as "unchecked".

Genus requires that a method handle or explicitly propagate all exceptions it knows can arise from operations it uses or methods it calls. If the implementa-

[1]Genus uses square brackets rather than angle brackets for generic type arguments: `List[T]` rather than `List<T>`. For brevity, we use Genus syntax and the Genus equivalents of Java core classes without further explanation.

tion of a method explicitly throws an unhandled exception whose type (or the supertype thereof) is not listed in the method's header, the program is rejected.

Like in Java, exceptions can be handled using the `try`–`catch` construct, and a `finally` block that is always executed may be provided. (The `try`-with-resources statement of Java 7 is easily supported; it is orthogonal to the new features.)

A method that wishes merely to propagate an exception to its caller can simply place a `throws` clause in its signature. We say such an exception propagates in *checked mode*. Unlike in Java, exceptions in checked mode do not cause stack-trace collection.

2.2.1 Failures

Unlike Java, our mechanism makes it easy for the programmer to indicate that an exception should not happen. The programmer ordinarily does this by putting a `fails` clause in the method header. Any caller of the method is then oblivious to the exception, meaning that the exception will be treated as unchecked as it propagates further. When code fails because of an exception, the exception propagates in a special mode, the *failure mode*.

For example, a programmer who is certain that the `Object` class can be loaded successfully can write

```
Class loadObject() fails ClassNotFoundException {
  return Class.forName("genus.lang.Object");
}
```

where the method `forName` in `Class` declares `ClassNotFoundException` in its `throws` clause. Note that a `fails` clause is really part of the implementation rather than part of the method signature or specification. We write it in the header just for convenience.

Exceptions propagating in failure mode also differ in what happens at run time. Programmers need detailed information to debug the stopped program to discover how the failure occurred. Therefore, failure exceptions collect a stack trace. This is relatively slow (as slow as most Java exceptions!) but efficiency is not a concern for failures.

2.2.2 Avoiding Exception Capture

Since exceptions propagating in failure mode do not appear in method signatures, it is important to avoid catching them accidentally. For example, consider the following code that calls two functions `g()` and `h()`:

```
void f() {
  try {
    g(); // signature says throws MyExc
    h(); // signature doesn't say throws MyExc
  } catch (MyExc e) { … }
}
```

Suppose that because of a programmer mistake, the call to `h()` unexpectedly fails with exception `MyExc`. If this exception were caught by the `catch` clause, `f()` would execute code intended to compensate for something that happened in `g()`. We prevent this undesirable exception capture by ensuring that failure exceptions cannot be caught by any ordinary `catch` clauses: failure exceptions *tunnel* past all ordinary `catch` clauses.

Although exceptions in failure mode are not normally handled, there may be value in catching them at the top level of the program or at the boundary between components, to allow for more graceful exit. Genus supports this with a rarely used `catch all` construct that catches all exceptions of a given type,

regardless of propagation mode. For example, if the `try` statement above were extended to include a second clause

```
catch all (MyExc e) { … }
```

the first `catch` clause would catch the expected `MyExc`s thrown by `g`, and the second `catch all` clause would catch failure-mode `MyExc`s tunneled through `h`.

2.2.3 Fail-by-Default Exceptions

Java has a commonly accepted set of exceptions that usually correspond to programmer mistakes: the *built-in* subclasses of `RuntimeException` or `Error`. To reduce annotation burden for the programmer, our mechanism does not ordinarily require writing a `fails` clause in order to convert such exceptions to failure mode. We say these exceptions *fail by default*.

Fail-by-default exceptions are different from Java's unchecked exceptions. Unchecked exceptions conflate failures with ordinary exceptions that are tunneling through oblivious code but that still ought to be subject to static checking.

In contrast, fail-by-default exceptions remain in checked mode until they reach a method-call boundary; they convert from checked mode to failure mode only if the current method does not declare the exception in its `throws` clause. A fail-by-default exception collects a stack trace only if and when it does fail, so code can still handle exceptions like `NoSuchElementException` efficiently. It is therefore reasonable and useful to write code that handles such exceptions.

2.3 Higher-Order Abstractions and Tunneling

As discussed in Section 2.1.1, higher-order functions pose a problem for statically checked exception mechanisms. The same problem arises for many common

```
List[R] map[T,R](Function[T,R] f, List[T] src) {
    List[R] dest = new ArrayList[R]();
    for (T t : src) dest.add(f.apply(t));
    return dest;
}
```

Figure 2.1. A higher-order function written in Genus

```
1 List[String] x = ...;
2 List[Class] y;
3 try { y = map(Class::forName, x); }
4 catch (ClassNotFoundException e) { ... }
```

Figure 2.2. Passing a function that throws extra exceptions

object-oriented design patterns, which are essentially higher-order functions. Our solution is to tunnel exceptions through oblivious code.

For example, Genus allows the programmer to pass to the higher-order function `map` (Figure 2.1) an implementation of `Function` that throws exceptions, even though the signature of `Function` does not mention any exceptions. If the passed `Function` throws an exception, that exception is tunneled through the exception-oblivious code of `map` to the caller of the exception-aware code that called `map`.

In the code of Figure 2.2, the method `forName` in class `Class` is passed to `map`. This call is legal even though `forName` is declared to throw an exception `ClassNotFoundException`. Since `map` is oblivious to this exception, it cannot be expected to handle it. By contrast, the caller of `map` is aware that an object that throws exceptions is used at a type (`Function`) that does not declare any

exceptions. Because it is aware of the exception, the caller is responsible for the exception.

If `ClassNotFoundException` arises at run time, it tunnels through the code of `map` and is caught by the `catch` clause. Alternatively, the caller could have explicitly converted the exception to a failure (via a `fails` clause) or explicitly allowed it to propagate (via a `throws` clause). In any case, exception handling is enforced statically.

2.3.1 Exception Tunneling is Safe and Lightweight

In Java, the rigidity of checked exceptions has led to some verbose and dangerous idioms, especially when higher-order functions and design patterns are used. Exception tunneling helps avoid these undesirable programming practices.

In particular, Java programmers often abandon static checking of exceptions to make it possible for their exceptions to pass through exception-oblivious higher-order code. They either define their own exceptions as unchecked, or they cope with preexisting checked exceptions by calling unsafe APIs—e.g., `sun.misc.Unsafe::throwException` [116]—or by wrapping checked-exception objects inside unchecked-exception objects. Wrapping with unchecked exceptions is the safest of these workarounds because it makes exception capture less likely, but wrappers are verbose and expensive.

Figure 2.3 shows an example of this idiom taken from the `javac` compiler [139], which contains a number of visitors for the Java AST. In order to conform to the `Visitor` interface, the `visit` methods in the pretty-printing visitor `Pretty` wrap the checked `IOException` into unchecked wrappers, which are then unwrapped as shown in method `printTree`. This programming pattern is verbose, abandons static checking, and is likely to be slow due to stack-trace collection.

```
interface Visitor {
  void visit(IfTree t);
  …
}

class IfTree implements Tree {
  void accept(Visitor v) { v.visit(this); }
  …
}

class Pretty implements Visitor {
  void visit(IfTree t) {
    try { … } // pretty-print IfTree
    catch (IOException e) { throw new UncheckedIO(e); }
  } // wraps IOException
  …
}

void printTree(Tree t, Pretty v) throws IOException {
  try { t.accept(v); }
  catch (UncheckedIO u) { throw u.getCause(); }
} // unwraps UncheckedIO

class UncheckedIO extends RuntimeException { …}
```

Figure 2.3. The pretty-printing visitor in `javac` (simplified). Code for exception wrapping and unwrapping is highlighted.

When written in Genus, the same Visitor pattern (Figure 2.4) does not require exception wrapping or unwrapping to achieve tunneling. The modifier `weak` on the interface `Visitor` (see Section 2.3.4) makes it legal for its implementations to declare new exceptions (the interface `Function` is annotated similarly). In `printTree`, the call `t.accept(v)` passes a visitor object that throws the additional exception `IOException`. If this exception is thrown by the visitor, it tunnels until

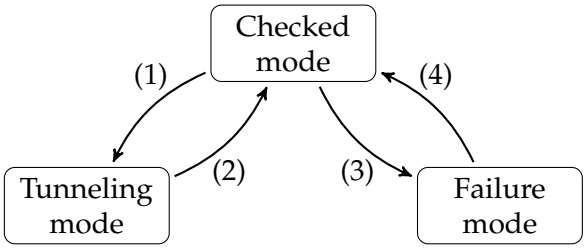
```
weak interface Visitor{
  void visit(IfTree t);
  …
}

class IfTree implements Tree {
  void accept(Visitor v) { v.visit(this); }
  …
}

class Pretty implements Visitor {
  void visit(IfTree t) throws IOException { … } //OK
  …
}
```

```
1 void printTree(Tree t, Pretty v) throws IOException {
2   t.accept(v);
3 }
```

Figure 2.4. Exception tunneling in `javac`, ported to Genus



(1)(2): Sections 2.3.2 and 2.3.3 (3)(4): Sections 2.2.1 and 2.2.3

Figure 2.5. Three exception propagation modes

it reaches `printTree`. Thus, the antipattern of exception wrapping becomes unnecessary.

```
List[R] map[T,R](Function[T,R] f, List[T] src) {
  List[R] dst = new ArrayList[R]();
  mapImpl(f, src, dst);
  return dst;
}
void mapImpl[T,R](Function[T,R] f,
    List[T] src, List[R] dst) {
  if (dst.size() >= src.size()) return;
  dst.add(f.apply(src.get(dst.size())));
  mapImpl(f, src, dst);
}
```

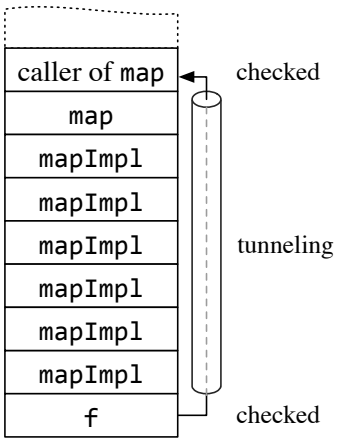


Figure 2.6. A recursive implementation of `map` (left) and a stack snapshot showing propagation of an exception caused by `f` (right). The stack grows downwards.

2.3.2 Tunneling Checked Exceptions

Earlier we discussed two modes of exception propagation: checked mode and failure mode. Tunneling mode is a third mode of propagation. The relationships between the three modes are summarized in Figure 2.5. In tunneling mode, as in checked mode, static checking enforces handling of exceptions. As in failure mode, these exceptions do not need to be declared in signatures of the methods they tunnel through.

A given exception may propagate in more than one mode. Consider the alternative, slightly contrived implementation of `map` in Figure 2.6. It calls a helper method `mapImpl`, which then recursively calls itself to traverse the list. Suppose an exception arises when function `f` is applied to the sixth element in the list. Figure 2.6 shows a snapshot of the current call stack. Since `f` is the place where the exception is generated, the exception first propagates in *checked mode* within the function `f`. Because its caller `mapImpl` is oblivious to the exception, the exception then switches to *tunneling mode* and propagates through all the

`mapImpl` stack frames. Finally, the caller of `map` knows about the exception and thus can handle it. The exception returns to checked mode when it reaches this caller. From there, it can be either caught, rethrown in checked mode, or turned into failure.

2.3.3 Tunneling, Exception Capture, and Blame

Tunneling avoids the phenomenon of exception capture discussed in Section 2.2. In OO languages like Java and C#, exception capture occurs because of an unexpected collision in the space of *exception identifiers*; an exception identifier in these languages is simply the exception type. We avoid exception capture by augmenting the identity of a thrown exception to include a notion of "blame".

To ensure that every exception is eventually either handled or treated as a failure, a method must *discharge* every exception it is aware of statically. There are three ways to discharge an exception:

(1) handle it with a `catch` clause,

(2) propagate it to its caller as a checked exception via a `throws` clause, or

(3) convert it to a failure via a `fails` clause (which is implicit for fail-by-default exceptions).

Each of these three ways discharges exceptions from a set of program points that are *statically* known to give rise to these exceptions. We say these program points can be *blamed* for the exception. With each such program point, we associate a *blame label* that identifies where responsibility lies for the exception.

At run time, then, a thrown exception is identified both by its exception type, and by its blame label. An exception is discharged (e.g., caught by a `catch` clause, or converted to failure by a `fails` clause), only if the blame label of the

exception lies within the lexical scope covered by that particular discharging point. Otherwise, the exception is one that the discharging point is oblivious to.

We use the word "blame" because this mechanism is related to the notion of blame used in work on behavioral contracts [72, 73] and gradual typing [178]. A compiler for a gradually typed language might label program points with unique identifiers wherever there is a *mismatch* between expected and actual types. When a cast failure happens at run time, blame can be attributed to the program point at fault.

Here, mismatch occurs analogously when the type of an actual argument declares exceptions not encompassed by those declared by the formal parameter type. Any exception mismatch in the parameters passed to a method call causes blame to be assigned to the program point of the method call. However, unlike in gradual typing, exceptions arising from a program point assigned blame do not imply mistakes,[2] since the programmer must discharge the exceptions.

For example, in Figure 2.2, the program point where `Class::forName` is used at type `Function` is in the scope of the ensuing `catch` clause at line 4. Because it creates a mismatch with the signature of `Function`, this program point can be blamed for the exception. Similarly, in Figure 2.4, the highlighted program point (line 2) where a `Pretty` object is used at type `Visitor` is in the scope of the clause `throws IOException` (line 1). Because there is a mismatch between `Pretty` and `Visitor`, the highlighted program point can be blamed for the exception. Throughout this chapter , we highlight program points that are assigned blame and their matching discharging points.

[2] Findler et al. [73] use the term "blame" to mean "the programmer should be held accountable for shoddy craftsmanship". At the risk of confusion, we reuse the term to mean there is an exception to be discharged in this context.

```
weak interface Iterator[E] {
    // The exception indicates iteration is over
    E next() throws NoSuchElementException;
    …
}

class Tokenizer implements Iterator[Token] {
    Token next() throws IOException, NoSuchElementException { … }
    …
}
```

Figure 2.7. The `Iterator` interface and an inexact subtype

2.3.4 Weak Types

Some supertypes, usually interfaces, abstract across families of otherwise unrelated subtypes. Such interfaces often arise with design patterns like Iterator and Visitor. The intention of such types is to capture only a fragment of the behavior—a core set of methods—so that various implementations can have a common interface other software components can use.

Frequently there is utility in allowing subtypes of these interfaces to throw new exceptions. For example, suppose a lexer breaks input files into tokens; an `Iterator` might be used to deliver the tokens to code that consumes them. However, reading from a file can cause an `IOException`; the exception cannot reasonably be handled by the iterator, so should be propagated to the client code. In Java, programmers cannot throw a checked exception like `IOException` in such an implementation; they must either resort to unchecked exceptions, abandoning static checking, or define their own interfaces that compose poorly with existing components.

```
Iterator[Token] iter = new Tokenizer(reader);
while (true) {
  Token t;
  try { t = iter.next(); }
  catch (NoSuchElementException e) { break; }
  catch (IOException e) { log.write(…); continue; }
  …
}
```

Figure 2.8. Using a `Tokenizer` as an `Iterator` generates blame

Genus addresses this need for flexibility. Methods that allow their overridings to throw new exceptions are declared with the `weak` modifier. The `weak` modifier can also be applied to a type definition to conveniently indicate that all methods in that type are intended to be `weak`; see the definition of `Iterator` in Figure 2.7 for an example.

A subtype of a weak type can be *inexact*; for example, the `Tokenizer` class in Figure 2.7 is inexact with respect to its weak interface since its `next` method adds `IOException`. A `Tokenizer` can be used as an `Iterator` but this generates blame, forcing `IOException` to be handled, as in Figure 2.8.

Behavioral subtyping and conformance. Behavioral subtyping [109] is based on the idea that the allowed uses of an object should be known based on its apparent type. Therefore, an overriding method cannot add new exceptions to the supertype's signature for the method.

Our mechanism relaxes this requirement for weak types. Methods of an inexact subtype must obey the supertype specification—except that they can throw more exceptions. This implies that their additional exceptional conditions must be signaled with different types than those in the supertype method—

`Tokenizer` indicates an I/O problem by throwing `IOException`, not `NoSuchElementException`—and that the exceptional conditions the supertype defines must not be signaled in other ways—`Tokenizer` cannot issue a failure or return `null` when the iteration has no more elements.

2.4 Generics and Exceptions

We have also used Genus to explore the important interaction between exceptions and mechanisms for constrained parametric polymorphism. Various languages constrain generic type arguments in various ways: for example, Java and C# use subtyping constraints, whereas Haskell and Genus use the more flexible mechanism of type classes [176].

Genus provides constrained parametric polymorphism via *constraints* and *models* [190]. Like type classes, Genus constraints are predicates describing type features required by generic code. Genus models show how types can satisfy constraints, like type class instances in Haskell. Unlike Haskell instances, models are explicitly part of the instantiation of a generic abstraction. For example, the two instantiations `Set[String]` and `Set[String with CaseInsensEq]` are different types distinguished by the use of the model `CaseInsensEq` in place of default string equality. This distinction is helpful for precisely reasoning about exceptions.

As with interfaces, we would like the flexibility to instantiate generic abstractions with types whose operations throw additional exceptions not provided for by the constraint. Thus, similar to interfaces, Genus constraints can be weak; models may be inexact with respect to the weak constraints they witness.

The example in Figure 2.9 shows the utility of this feature, comparing Java and Genus code for an object pool abstraction. The upper half of the figure shows

```
class ObjectPool<T> {
  Factory<T> f;
  T borrow() throws Exception { … f.make() … }
  …
}
interface Factory<T> { T make() throws Exception; }
class ConnFactory implements Factory<Connection> {
  Connection make() throws SQLException { … }
}
```

```
class ObjectPool[T] where Factory[T] {
  T borrow() { … T.make() … }
  …
}
weak constraint Factory[T] { static T make(); }
model ConnFactory for Factory[Connection] {
  static Connection make() throws SQLException { … }
}
```

Figure 2.9. Object pool in Java (top) and in Genus (bottom)

an example adapted from the Apache Commons project [9]. The abstract factory type `Factory` defines a method `make` with a signature that declares "`throws Exception`". This idiom is common in Java libraries, because it permits subtypes to refine the actual exceptions to be thrown.

However, such declarations are a source of frustration for Java programmers [180]. Consider that method `make` is called by the method `borrow` in `ObjectPool` to create a new object in case there is no idle one, so `borrow` must also declare "`throws Exception`". The precise exception (in the example, `SQLException`) is therefore lost. Clients of `ObjectPool<Connection>` must handle `Exception`, which is no better, and perhaps worse, than having an unchecked exception.

Further, client code that handles the overly general exception is more likely to suffer from exception capture.

The lower half of Figure 2.9 shows a reasonable way to implement the example in Genus. A constraint `Factory[T]` is used to express the requirement that objects of type `T` can be made in some way; because it does not declare any exceptions, method `borrow` need not either. However, a model like `ConnFactory` can add its exceptions such as `SQLException` to the method `make`. Client code can then use the type `ObjectPool[Connection with ConnFactory]` to recycle `Connection` objects. Because the model is part of the type, it is statically apparent that the exception `SQLException` may be thrown; the client code will be required to handle this exception—but only this exception.

2.5 Exactness Analysis

The new exception mechanism poses new challenges for type checking. One challenge is that the identity of an exception includes a blame label, so blame should not be allowed to escape its scope. Otherwise, an exception might not be handled.

For example, consider the first definition of the `PeekingIterator` class in Figure 2.10. It decorates a wrapped `Iterator` in field `inner` to support one-element lookahead. Its `next` and `peek` methods call `inner.next()`. Per Section 2.3.4, it is possible to pass an object of some inexact subtype of `Iterator` to the constructor, to be assigned to `inner`. However, `Iterator` is a weak type, so the methods of the actual object being passed may throw exceptions not declared by `Iterator`. If the assignment to `inner` were allowed, the same exceptions would propagate when the `next` method of the `PeekingIterator` object were called. And this call

could be delayed until outside the context that is aware of the mismatch with `Iterator`. In this case, the exceptions would not be guaranteed to be handled.

Therefore, we want to detect statically that the assignment to `inner` lets the blame from the inexact object escape. Storing an inexact object into a data structure at a weak type, or even returning an inexact object from a method, may permit such an escape of blame.

A second challenge for the new exception mechanism, and indeed for exception mechanisms generally, is that programmers should not be forced to write handlers for program points where exceptions cannot happen. To address this challenge, the location of blame should be precise. Figure 2.8 offers an example of this problem. The method call `iter.next()` might appear (to the compiler) not to throw any exceptions because `iter` has type `Iterator[Token]`, yet we know that it may throw an `IOException` because `iter` is initialized to a `Tokenizer`. A safe, conservative solution would be to require all of the code below this initialization to be wrapped in a `try–catch`. But this solution would make it difficult to continue the iteration after an `IOException` is raised.

Genus addresses these two challenges using an intraprocedural program analysis that assigns *exactness* to uses of weak types, with little annotation effort by the programmer.

2.5.1 Exactness Annotations and Exactness Defaults

Indicating intolerance of inexactness at use sites. When a formal parameter or local variable is assigned a weak type, the programmer can annotate the type with `@exact` to indicate that an exact subtype must be used. For example, if the programmer adds `@exact` to the formal parameter of the constructor at line 3 in Figure 2.10,

```
1 class PeekingIterator[E] implements Iterator[E] {
2   Iterator[E] inner;
3   PeekingIterator(Iterator[E] it) { inner = it; … }
4   E peek() throws NoSuchElementException { … inner.next() … }
5   …
6 }
```

```
class PeekingIterator[I extends Iterator[E], E] implements Iterator[E] {
  I inner;
  PeekingIterator(I it) { inner = it; … }
  E peek() throws NoSuchElementException { … inner.next() … }
  …
}
```

Figure 2.10. Two definitions of `PeekingIterator`. The top one allows blame to escape, so a warning is issued for the constructor. The bottom one uses dependent exactness to soundly avoid the warning.

```
PeekingIterator(@exact Iterator[E] it) { inner = it; … }
```

it becomes a static error to pass an inexact implementation of `Iterator` to the constructor. So it is guaranteed that using `inner` will not generate unexpected exceptions. By contrast, without `@exact`, Genus issues a warning about the assignment to the escaping pointer `inner` at line 3. If the exception is actually thrown at run time, it is converted into failure and the usual stack trace is collected.

Although `@exact` might appear to add notational burden, our experience with porting existing Java code into Genus (Section 2.8) suggests that this escape hatch is rarely needed. We have not seen the need to use the `@exact` annotation to dismiss such warnings in existing code ported from Java. It seems that programmers use weak types differently from other types—weak types provide functionality rather than structure. Further, exactness defaults and exactness

inference (Section 2.5.2) reduce annotation overhead, and exactness-dependent types (Section 2.5.3) provide more expressiveness.

Exactness defaults. Exactness defines what exceptions that are not declared by the weak type might nevertheless be generated by the term being typed. Exactness $\mathcal{E}$ is formally a mapping from methods to sets of exceptions. These mappings form a lattice ordered as follows:

$$\mathcal{E}_1 \leq \mathcal{E}_2 \Leftrightarrow \mathit{domain}(\mathcal{E}_1) \subseteq \mathit{domain}(\mathcal{E}_2) \ \wedge \ \forall m.\, \mathcal{E}_1(m) \subseteq \mathcal{E}_2(m)$$

The bottom lattice element is strict exactness, denoted by $\varnothing$.

To avoid the need for programmers to write the annotation `@exact` for most uses of weak types, the compiler determines exactness using a combination of exactness defaults and automatic exactness inference. To see how these mechanisms work, consider the code in Figure 2.11.

1. Weak types used as return types or field types are exact, as these are the channels through which pointers can escape. For these types, we have $\mathcal{E} = \varnothing$.
2. Methods and constructors are implicitly polymorphic with respect to exactness. That is, they can be viewed as parameterized by the exactness of their argument and receiver types.
3. Weak types in a local context are labeled by exactness variables: for example, x and y in Figure 2.11, at lines 2 and 11 respectively.

   A unification-based inference engine solves for these variables, inferring exactness from the local variable uses (Section 2.5.2).
4. For a procedure call, an exactness variable is generated for each argument and/or receiver whose formal parameter type is weak. Exactness variables of this kind in Figure 2.11 are z (line 9) and w (line 12).

```
weak interface Runnable { void run(); }

1  void g[T extends Runnable⟨e⟩](List[T] l) {
2    Runnable⟨x⟩ r0;
3    if (new Random().nextBoolean()) {
4      r0 = new Runnable() { void run() throws IOException { … } };
5    } else {
6      r0 = new Runnable() { void run() throws EOFException { … } };
7    }
8    try {
9      r0.run(); //Runnable⟨z⟩
10   } catch (IOException ex) { … }
11   for (Runnable⟨y⟩ r : l)
12     r.run();  //Runnable⟨w⟩
13 }
```

Figure 2.11. Running example for exactness analysis. Uses of weak types are tagged by exactness variables, to be fed into the solver. (`EOFException` is a subtype of `IOException`.)

5. Recall that Genus supports constrained parametric polymorphism via type constraints [190]. Subtype constraints and where-clause constraints can also specify exactness. Their default exactness is deduced in ways similar to those listed above. For example, in Figure 2.11 the use of `Runnable` in g's signature (line 1) constrains the type parameter `T`, so its exactness is resolved to a fresh name e, with respect to which g is polymorphic.

2.5.2 Solving Exactness Constraints

At each assignment (including variable initialization and argument passing) to a variable with weak type τ, inexact values must not escape into variables with exact types. Therefore, constraints of form $\tau\langle\mathcal{E}_r\rangle \leq \tau\langle\mathcal{E}_l\rangle$ are generated, where $\mathcal{E}_l$

is the exactness of the left-hand side expected type τ in an assignment, resolved as described in Section 2.5.1, and $\mathcal{E}_r$ is the exactness of the right-hand side actual type with respect to τ. For example, the code of Figure 2.11 then generates the following constraints with line numbers attached:

$$\begin{aligned}
\tau\langle\mathcal{E}_r\rangle &\leq \tau\langle\mathcal{E}_l\rangle & \\
\texttt{Runnable}\langle\{\,\texttt{run} \mapsto \texttt{IOException}\,\}\rangle &\leq \texttt{Runnable}\langle x\rangle & \textit{line 4} \\
\texttt{Runnable}\langle\{\,\texttt{run} \mapsto \texttt{EOFException}\,\}\rangle &\leq \texttt{Runnable}\langle x\rangle & \textit{line 6} \\
\texttt{Runnable}\langle x\rangle &\leq \texttt{Runnable}\langle z\rangle & \textit{line 9} \\
\texttt{Runnable}\langle e\rangle &\leq \texttt{Runnable}\langle y\rangle & \textit{line 11} \\
\texttt{Runnable}\langle y\rangle &\leq \texttt{Runnable}\langle w\rangle & \textit{line 12} \\
\texttt{Iterator[T]}\langle\varnothing\rangle &\leq \texttt{Iterator[T]}\langle v\rangle & \textit{line 11}
\end{aligned}$$

The last constraint is due to desugaring of the loop at lines 11–12:

```
for (Iterator[T]⟨v⟩ i = l.iterator(); ; ) {
    Runnable⟨y⟩ r;
    try { r = i.next(); }
    catch (NoSuchElementException ex) { break; }
    r.run();
}
```

Note that enhanced `for` loops are translated differently by Java and Genus. Genus exceptions are fast enough that it is often faster to call `next()` and catch a final exception, rather than calling `hasNext()` on every iteration.

The compiler solves these constraints by finding the least upper bounds of x, y, z, w, and v:

$$x = z = \{\,\texttt{run} \mapsto \{\,\texttt{IOException}, \texttt{EOFException}\,\}\,\}$$

$$y = w = e$$

$$v = \varnothing$$

The fact that a solution exists implies that inexact pointers will not escape to the heap, addressing the first challenge of escaping blame.

The solution also reveals precisely where exception handling is required, addressing the second challenge of blame precision. Specifically, the solution to each variable generated via the fourth case of exactness defaults (Section 2.5.1) tells what exception can be produced by the method call. In our running example, the solution to z is that the method call `r0.run()` may throw `IOException` or `EOFException`, which are handled by the `catch` block. Notice that although mismatches happen at lines 4 and 6, the blame does not take effect until the method call at line 9.

The solution to w is the polymorphic label e, which means that the current context is oblivious to whatever exceptions correspond to e when g is called, as discussed in Section 2.3.

2.5.3 Exactness-Dependent Types

It is possible to write a type-safe `PeekingIterator` using the `@exact` annotation, but we might want a peeking iterator that throws the same exceptions as its underlying iterator does. This expressiveness can be obtained by making `PeekingIterator` talk about the potential mismatch, as in the bottom definition of `PeekingIterator` in Figure 2.10. The class is now parameterized by the type of the iterator it decorates. In the using code below, `PeekingIterator` is instantiated on an inexact subtype of `Iterator`, so the compiler requires the exception to be handled:

```
try {
    PeekingIterator[Tokenizer,Token] pi =
      new PeekingIterator[Tokenizer,Token](...);
    while (pi.hasNext()) { … pi.peek() … }
} catch (IOException e) { … }
```

Parameterized by a weak constraint, the Genus version of `ObjectPool` in Figure 2.9 is similarly exactness-dependent.

The special `this` keyword, when referring to the current object of a weak class, also has an exactness-dependent type—it is dependent on the exactness of the run-time class of the object with respect to the enclosing weak class. Since the run-time class is statically unknown, the compiler must assume that it can add arbitrary exceptions. Thus it results in a compiler warning to use `this` in ways that generate blame. However, we expect this to be rare: most weak types are interfaces, which do not normally use `this`.

2.6 Formalization

We formalize the new exception mechanism using a core calculus, CBC (for Checked Blame Calculus).

2.6.1 Syntax and Notations

Figure 2.12 defines both a *surface* and a *kernel* syntax for CBC. Programs are written in the surface syntax, and rewritten to and evaluated in the kernel calculus. Applications in surface terms are tagged by unique *blame labels* (ranged over by ℓ) representing lexical positions where blame can arise. Rewriting propagates these labels from the surface language to the kernel language, for blame tracking during kernel evaluation.

The surface syntax assumes a fixed set of exception names ranged over by E. The kernel syntax allows them to be labeled to form new names; E and E^{ℓ} are different names.

Surface types (τ) include weak types and strong types. A strong type (σ) is either the base type B or a function type $\tau \xrightarrow{\perp} [\sigma]_{\overline{E}}$ that does not allow mismatch against $\overline{E}$. (An overline denotes a (possibly empty) set.) A weak type $\tau \xrightarrow{\top} [\sigma]_{\overline{E}}$, on the other hand, tolerates mismatch. Notice that function return types must be strong to prevent blame from escaping. The same can be said about kernel types. Since there is an obvious injection (syntactic identity) from surface types to kernel types, we abuse notation by using τ and σ in places where kernel types T and S are expected.

Environments Γ contain mappings from variables to their types and exactness. Exactness is represented by sets of exceptions. Strong types enforce strict exactness, so their exactness is represented by $\varnothing$. The auxiliary function $\mathsf{K}(\cdot)$ returns $\perp$ for strong types and $\top$ for weak types; values of strong types must not escape. $\Gamma_\perp$ retains only the strongly-typed variables in Γ.

2.6.2 Semantics

Surface-to-kernel rewriting. Figure 2.13 defines the rewriting rules. The judgment $\Gamma; K \vdash f \rightsquigarrow e : [\tau]_{\overline{E}}$ translates the surface term f to the kernel term e, assigns f the type τ, and infers the exceptions $\overline{E}$ that evaluating e might raise.

Typing is dependent on K, which indicates whether the term in question is guaranteed not to escape. For example, the left-hand-side term in an application is type-checked with $K = \top$ as in [R-APP-E] and [R-APP-I], while the body of an abstraction is type-checked with $K = \perp$ as in [R-ABS]. [R-VAR-W] denies first-class citizenship to weakly typed variables, and augments the return type if the environment indicates inexactness.

Exception mismatch is computed using the subtyping relation $\lesssim_{\overline{E}}$, as in [R-APP-I] and [R-LET]. But only with [R-APP-I] can blame take effect, and

$$
\begin{array}{lrcl}
\text{surface terms} & f & ::= & b \mid x \mid \lambda x{:}\tau\,.\,f \mid \{f_1\ f_2\}^{\ell} \mid \\
 & & & \textbf{let}\ x{:}\tau \leftarrow f_1\ \textbf{in}\ f_2 \mid \\
 & & & \textbf{throw}\ E \mid \textbf{try}\ f_1\ \textbf{catch}\ E \triangleright f_2 \\
\text{surface types} & \tau & ::= & \sigma \mid \tau \xrightarrow{\top} [\sigma]_{\overline{E}} \\
\text{surface strong types} & \sigma & ::= & B \mid \tau \xrightarrow{\bot} [\sigma]_{\overline{E}} \\
\text{kernel terms} & e & ::= & b \mid x \mid \lambda x{:}T\,.\,e \mid e_1\ e_2 \mid \\
 & & & \textbf{let}\ x{:}T \leftarrow e_1\ \textbf{in}\ e_2 \mid \\
 & & & \textbf{throw}\ U \mid \textbf{try}\ e_1\ \textbf{catch}\ \overline{U \triangleright e_2} \\
\text{kernel types} & T & ::= & S \mid T \xrightarrow{\top} [S]_{\overline{U}} \\
\text{kernel strong types} & S & ::= & B \mid T \xrightarrow{\bot} [S]_{\overline{U}} \\
\text{kernel exceptions} & U & ::= & E \mid E^{\ell} \\
\text{environments (surface)} & \Gamma & ::= & \varnothing \mid \Gamma,\ x{:}\tau\langle\overline{E}\rangle \\
\text{environments (kernel)} & \Delta & ::= & \varnothing \mid \Delta,\ x{:}T\langle\overline{U}\rangle \\
\text{escape kind} & K & ::= & \top \mid \bot
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{K}(\sigma) = \bot & \Gamma_{\bot} = \left\{ x{:}\sigma\langle\varnothing\rangle \;\middle|\; x{:}\sigma\langle\varnothing\rangle \in \Gamma \right\} \\
\mathsf{K}\left(\tau \xrightarrow{\top} [\sigma]_{\overline{E}}\right) = \top & \Gamma_{\top} = \Gamma \\
\mathsf{K}(S) = \bot & \Delta_{\bot} = \left\{ x{:}S\langle\varnothing\rangle \;\middle|\; x{:}S\langle\varnothing\rangle \in \Delta \right\} \\
\mathsf{K}\left(T \xrightarrow{\top} [S]_{\overline{U}}\right) = \top & \Delta_{\top} = \Delta
\end{array}
$$

Figure 2.12. Syntax of CBC

its translation is consequently less obvious. To avoid exception capture, we need to give new names to exceptions involved in the mismatch. Therefore, the translation wraps the argument in an abstraction, which, when applied, catches any exceptions in the mismatch ($\overline{E}$) and rethrows their labeled versions ($\overline{E^{\ell}}$). The caller is aware of the exceptions $\overline{E}$, so it catches the labeled exceptions tunneled to its context and strips off their labels.

$$\boxed{\Gamma; K \vdash f \rightsquigarrow e : [\tau]_{\overline{E}}}$$

[R-Const]
$$\frac{}{\Gamma; K \vdash b \rightsquigarrow b : [B]_{\varnothing}}$$

[R-Var-s]
$$\frac{x : \sigma\langle\varnothing\rangle \in \Gamma_K}{\Gamma; K \vdash x \rightsquigarrow x : [\sigma]_{\varnothing}}$$

[R-Var-w]
$$\frac{x : \tau \xrightarrow{\top} [\sigma]_{\overline{E_1}}\langle\overline{E_2}\rangle \in \Gamma_K}{\Gamma; K \vdash x \rightsquigarrow x : \left[\tau \xrightarrow{\top} [\sigma]_{\overline{E_1} \cup \overline{E_2}}\right]_{\varnothing}}$$

[R-Abs]
$$\frac{\Gamma_K,\ x : \tau\langle\varnothing\rangle; \bot \vdash f \rightsquigarrow e : [\sigma]_{\overline{E}}}{\Gamma; K \vdash \lambda x : \tau\,.\, f \rightsquigarrow \lambda x : \tau\,.\, e : \left[\tau \xrightarrow{\bot} [\sigma]_{\overline{E}}\right]_{\varnothing}}$$

[R-Let]
$$\frac{\Gamma; \mathsf{K}(\tau_1) \vdash f_1 \rightsquigarrow e_1 : [\tau_3]_{\overline{E_1}} \qquad \tau_3 \lesssim_{\overline{E}} \tau_1 \qquad \Gamma,\ x : \tau_1\langle\overline{E}\rangle; K \vdash f_2 \rightsquigarrow e_2 : [\tau_2]_{\overline{E_2}}}{\Gamma; K \vdash \mathbf{let}\ x : \tau_1 \leftarrow f_1\ \mathbf{in}\ f_2 \rightsquigarrow \mathbf{let}\ x : \tau_1 \leftarrow e_1\ \mathbf{in}\ e_2 : [\tau_2]_{\overline{E_1} \cup \overline{E_2}}}$$

[R-App-e]
$$\frac{\Gamma; \top \vdash f_1 \rightsquigarrow e_1 : \left[\tau_1 \xrightarrow{\top} [\sigma]_{\overline{E_3}}\right]_{\overline{E_1}} \qquad \Gamma; \mathsf{K}(\tau_1) \vdash f_2 \rightsquigarrow e_2 : [\tau_2]_{\overline{E_2}} \qquad \tau_2 \lesssim_{\emptyset} \tau_1}{\Gamma; K \vdash \{f_1\ f_2\}^{\ell} \rightsquigarrow e_1\ e_2 : [\sigma]_{\overline{E_1} \cup \overline{E_2} \cup \overline{E_3}}}$$

[R-App-i]
$$\frac{\begin{array}{c}\Gamma; \top \vdash f_1 \rightsquigarrow e_1 : \left[\tau_1 \xrightarrow{\top} [\sigma_1]_{\overline{E_3}}\right]_{\overline{E_1}} \qquad \Gamma; \mathsf{K}(\tau_1) \vdash f_2 \rightsquigarrow e_2 : [\tau_2]_{\overline{E_2}} \\ \tau_2 \lesssim_{\overline{E}} \tau_1 \qquad \tau_2 = \tau_{21} \xrightarrow{\top} [\sigma_{22}]_{\overline{E_4}}\end{array}}{\Gamma; K \vdash \{f_1\ f_2\}^{\ell} \rightsquigarrow \begin{array}{l}\mathbf{try}\ e_1 \begin{pmatrix}\mathbf{let}\ x : \tau_2 \leftarrow e_2\ \mathbf{in} \\ \lambda y : \tau_{21}\,.\,\mathbf{try}\ x\ y\ \mathbf{catch}\ \overline{E \triangleright \mathbf{throw}\ E^{\ell}}\end{pmatrix} \\ \quad \mathbf{catch}\ \overline{E^{\ell} \triangleright \mathbf{throw}\ E}\end{array} : [\sigma_1]_{\overline{E_1} \cup \overline{E_2} \cup \overline{E_3} \cup \overline{E}}}$$

[R-Try-Catch]
$$\frac{\Gamma; K \vdash f_1 \rightsquigarrow e_1 : [\tau]_{\overline{E_1}} \qquad \Gamma; K \vdash f_2 \rightsquigarrow e_2 : [\tau]_{\overline{E_2}} \qquad E \in \overline{E_1}}{\Gamma; K \vdash \mathbf{try}\ f_1\ \mathbf{catch}\ E \triangleright f_2 \rightsquigarrow \mathbf{try}\ e_1\ \mathbf{catch}\ E \triangleright e_2 : [\tau]_{(\overline{E_1} \setminus \{E\}) \cup \overline{E_2}}}$$

[R-Throw]
$$\frac{}{\Gamma; K \vdash \mathbf{throw}\ E \rightsquigarrow \mathbf{throw}\ E : [\sigma]_E}$$

[R-Subsume]
$$\frac{\Gamma; K \vdash f \rightsquigarrow e : [\tau_1]_{\overline{E_1}} \qquad \tau_1 \lesssim_{\varnothing} \tau_2 \qquad \overline{E_1} \subseteq \overline{E_2}}{\Gamma; K \vdash f \rightsquigarrow e : [\tau_2]_{\overline{E_2}}}$$

Figure 2.13. CBC surface-to-kernel rewriting

$$\boxed{\tau_1 \lesssim_{\overline{E}} \tau_2}$$

[SS-B]
$$\frac{}{B \lesssim_{\varnothing} B}$$

[SS-S]
$$\frac{\tau_2 \lesssim_{\varnothing} \tau_1 \qquad \sigma_1 \lesssim_{\varnothing} \sigma_2 \qquad \overline{E_1} \subseteq \overline{E_2}}{\tau_1 \xrightarrow{\perp} [\sigma_1]_{\overline{E_1}} \lesssim_{\varnothing} \tau_2 \xrightarrow{\perp} [\sigma_2]_{\overline{E_2}}}$$

[SS-W]
$$\frac{\tau_2 \lesssim_{\varnothing} \tau_1 \qquad \sigma_1 \lesssim_{\varnothing} \sigma_2}{\tau_1 \xrightarrow{K} [\sigma_1]_{\overline{E_1}} \lesssim_{\overline{E_1} \setminus \overline{E_2}} \tau_2 \xrightarrow{\top} [\sigma_2]_{\overline{E_2}}}$$

Figure 2.14. CBC surface subtyping

Semantics of the kernel calculus. The static semantics of the kernel can be found in the technical report [192]. The static semantics is largely similar to the typing induced by rewriting, except that the kernel need not worry about exception capture. Hence, exception propagation happens in the usual way. Figure 2.15 defines the dynamic semantics of the kernel.

2.6.3 Type Safety

The type system guarantees that if a program can be typed without exceptional effects, it cannot get stuck when evaluated, or terminate in an exception. This guarantee easily follows from two other standard results.

First of all, the kernel type system is sound:

Theorem 1 (KERNEL SOUNDNESS: PRESERVATION AND PROGRESS)**.**

- If $\varnothing; K \vdash e : [T]_{\overline{U}}$ and $e \longrightarrow e'$ then $\varnothing; K \vdash e' : [T]_{\overline{U}}$.
- If $\varnothing; K \vdash e : [T]_{\overline{U}}$ then either
  1. $\exists v.\, e = v$, or
  2. $\exists U_0 \in \overline{U}.\, e =$ **throw** U_0, or

$$
\begin{array}{lrcl}
\text{values} & v & ::= & k \mid \lambda x{:}T\,.\,e \\
\text{evaluation contexts} & \mathcal{E} & ::= & [\cdot] \mid \mathcal{E}\;e \mid v\;\mathcal{E} \mid \mathbf{let}\;x{:}T \leftarrow \mathcal{E}\;\mathbf{in}\;e \mid \\
 & & & \mathbf{try}\;\mathcal{E}\;\mathbf{catch}\;\overline{U \triangleright e}
\end{array}
$$

$$\boxed{e \longrightarrow e'}$$

$$\frac{e \longrightarrow e'}{\mathcal{E}[e] \longrightarrow \mathcal{E}[e']}$$

$$\frac{\mathbf{throw}\;U \curvearrowright \mathcal{E}}{\mathcal{E}[\mathbf{throw}\;U] \longrightarrow \mathbf{throw}\;U}$$

$$(\lambda x{:}T\,.\,e)\;\;v \longrightarrow e\,\{v/x\}$$

$$\mathbf{let}\;x{:}T \leftarrow v\;\mathbf{in}\;e \longrightarrow e\,\{v/x\}$$

$$\mathbf{try}\;v\;\mathbf{catch}\;\overline{U \triangleright e} \longrightarrow v$$

$$\mathbf{try}\;\;(\mathbf{throw}\;U)\;\;\mathbf{catch}\;\ldots\;U \triangleright e\;\ldots \longrightarrow e$$

$$\boxed{\mathbf{throw}\;U \curvearrowright \mathcal{E}}$$

$$\mathbf{throw}\;U \curvearrowright [\cdot]\;e$$

$$\mathbf{throw}\;U \curvearrowright v\;[\cdot]$$

$$\mathbf{throw}\;U \curvearrowright \mathbf{let}\;x{:}T \leftarrow [\cdot]\;\mathbf{in}\;e$$

$$\mathbf{throw}\;U_0 \curvearrowright \mathbf{try}\;[\cdot]\;\mathbf{catch}\;\overline{U \triangleright e}\;\;(U_0 \notin \overline{U})$$

Figure 2.15. Dynamic semantics of the CBC kernel

3. $\exists e'.\, e \longrightarrow e'$.

Proof. This is proved in the usual way, by induction on the kernel typing derivation. See the technical report [192] for details of this proof and of other formal results in this chapter.

Second, the translation from the surface language to the kernel language is type-preserving:

Lemma 1 (REWRITING PRESERVES TYPES)**.**
If $\Gamma; K \vdash f \rightsquigarrow e : [\tau]_{\overline{E}}$ then $\Gamma; K \vdash e : [\tau]_{\overline{E}}$.

Proof. By induction on the derivation of the translation.

The guarantee that well-typed programs handle their exceptions is a direct corollary of these two previous results:

Corollary 1 (NO UNCAUGHT EXCEPTIONS)**.**
If $\varnothing; \bot \vdash f \rightsquigarrow e : [\tau]_{\varnothing}$ and $e \longrightarrow^* e'$, then either $\exists v.\, e' = v$ or $\exists e''.\, e' \longrightarrow e''$.

2.7 Implementation

We have implemented the new exception mechanism for the Genus programming language. The implementation consists of about 5,800 lines of code, extending the compiler for the base Genus language [190]. Genus is implemented using Polyglot [130], so code generation works by translating to Java code, using a Java compiler as a back end.

```
class Blame extends RuntimeException {
  Throwable inner;
  Blame(Throwable t) { inner = t; }
  Throwable fillInStackTrace() { return this; }
} // represents exceptions in tunneling mode

class Failure extends RuntimeException {
  Throwable inner;
  Failure(Throwable t) { inner = t; }
} // represents exceptions in failure mode
```

Figure 2.16. The `Blame` and `Failure` classes in the Genus runtime

The rest of this section focuses on the translation into Java. The translation is guided by two goals: 1) it should prevent accidental capturing of exceptions, and 2) it should add negligible performance overhead to normal control flow.

2.7.1 Representing Exceptions in Non-Checked Modes

Unlike exceptions in checked mode, exceptions traveling in tunneling mode or failure mode must acquire new identities to avoid accidental capturing. These identities are implemented by wrapping exceptions into objects of classes `Blame` and `Failure` (Figure 2.16). Both classes extend `RuntimeException`, but `Blame` does not collect a stack trace.

2.7.2 Translating Exception-Oblivious Code

Methods with weakly typed parameters ignore extra exceptions generated by them. To ensure they are handled in the appropriate context, weakly typed parameters, including the receiver, are accompanied by an additional `Blame`

argument that serves as the blame label. When the actual argument is exact, the `Blame` argument is null. For example, the weak type `Iterator` has the following translation:

```
interface Iterator<E> {
    E next$Iterator(Blame b$) throws NoSuchElementException;
    ...
}
```

It receives a `Blame` object from its caller to accompany the weakly typed receiver. If an implementation of `Iterator` throws a mismatched exception, it is wrapped in this `Blame` object and tunneled through the code oblivious to it.

Exception-oblivious procedures are translated in a similar way. For example, the code generated for `map` (Figure 2.1) looks like the following:

```
<T,R> List<R> map(Function<T,R> f, List<T> src, Blame b$) {
    ... f.apply$Function(t, b$) ...
}
```

The extra `Blame` argument `b$` is intended for potential mismatch in the argument function `f`, and is passed down to the method call `f.apply$Function(...)` so that exceptions from `f` have the right blame label.

2.7.3 Translating Exception-Aware Code

The definition of `Iterator`'s inexact subtype `Tokenizer` is exception-aware. Its translation is shown in the top of Figure 2.17. Per Java's type-checking rules, the overriding method in `Tokenizer` cannot throw extra checked exceptions. Instead, the overriding method `next$Iterator(Blame)` redirects to method `next()` that does the real work, possibly throwing an `IOException`, and then turns that exception into either a `Blame` or a `Failure`, which is unchecked. If the `Blame` argument is *not* null, there must be a program point ready to handle the exception. So

```
// Translation of class Tokenizer from Figure 2.7
class Tokenizer implements Iterator<Token> {
  Token next$Iterator(Blame b$) throws NoSuchElementException {
    try { return next(); }
    catch (IOException e) {
      if (b$ != null) { b$.inner = e; throw b$; }
      else throw new Failure(e);
    }
  }
  Token next() throws IOException, NoSuchElementException { … }
  …
}

// Translation of the using code from Figure 2.8
Blame b$ = null;
try {
  Iterator<Token> iter = new Tokenizer(reader);
  while (true) {
    Token t;
    try { t = iter.next$Iterator(b$ = Thread.borrowBlame$()); }
    catch (NoSuchElementException e) { break; }
    catch (Blame bCaught$) {
      if (bCaught$ == b$) { log.write(…); continue; }
      else throw bCaught$;
    }
  }
} finally { Thread.handbackBlame$(b$); }
```

Figure 2.17. Translating exception-aware code into Java

the `IOException` is wrapped in the `Blame` object and is tunneled to that program point. If the `Blame` object *is* null, a `Failure` object wrapping the `IOException` is created and thrown. This might happen, for example, if the programmer chose to disregard the compiler warning reported for passing a `Tokenizer` (Figure 2.7) into the constructor of `PeekingIterator` (top of Figure 2.10).

The code in Figure 2.8 is also exception-aware, and Figure 2.17 (bottom) shows its translation. Instead of creating a new `Blame` object every time a mismatch happens, each thread maintains a `Blame` object pool that recycles `Blame` objects. A `Blame` object is borrowed from the pool at the blamable program point in the `try` block, and is returned in the `finally` block. The `catch` block catches any `Blame`, but *only* executes the exception handling code if the `Blame` caught is indeed the one associated with the blamable program point; otherwise it rethrows the `Blame`.

Aggressive interweaving of `try`–`catch` in method bodies might preclude certain compiler optimizations. Therefore, the translation uses a simple program analysis to identify existing enclosing `try` blocks onto which these new `catch` blocks can be attached.

2.7.4 Translating Failure Exceptions

A method body is wrapped in a `try` block whose corresponding `catch` block catches all exceptions that will switch to failure mode after exiting the method. A `catch all` block is translated into possibly multiple `catch` blocks to also catch compatible `Failure` and `Blame` objects.

2.8 Evaluation

The aim of our evaluation is to explore the expressiveness of the new exception mechanism, and its overhead with respect to both performance and notational burden.

2.8.1 Porting Java Code to Use Genus Exceptions

To evaluate the expressive power of the new exception mechanism, we ported various existing Java programs and libraries into Genus. Some of this code (`ObjectPool` and `PeekingIterator`) is described earlier, but we examined some larger code samples:

- We ported the Java Collections Framework and found that no `@exact` annotations were needed. In addition, the Genus compiler found unreachable code in `AbstractSequentialList`, thanks to fail-by-default exceptions propagating in checked mode (Section 2.2.3).
- We ported the `javac` visitor of Figure 2.3 into Genus, as mentioned earlier in Section 2.3.1. Conversion to the new exception mechanism allows more than 200 lines of code to be removed from class `Pretty` (~1,000 LOC), and more importantly, restores static checking.
- Using only checked exceptions, we managed to reimplement the `EasyIO` text parsing package[3] that was developed for a Cornell programming course. This codebase (~1,000 LOC) uses exceptions heavily for backtracking.

[3] www.cs.cornell.edu/courses/cs2112/2015fa/#Libraries

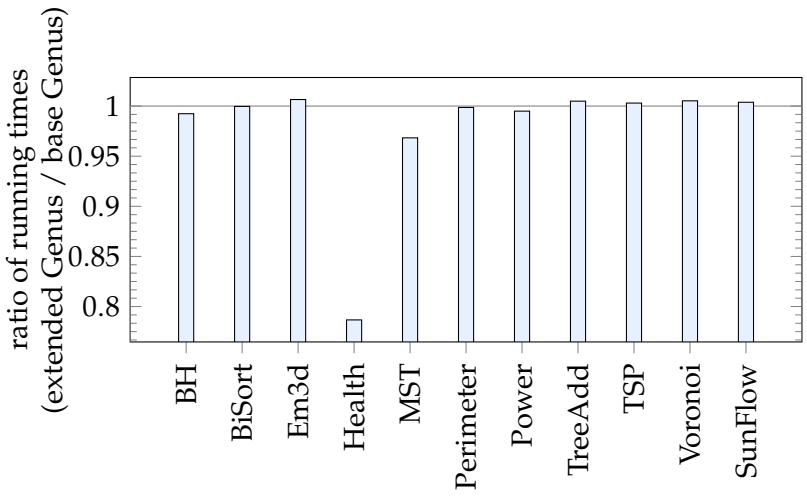


Figure 2.18. Performance of the exception mechanism on the JOlden benchmarks and SunFlow

2.8.2 Performance

The current Genus implementation targets Java. We explored its performance through several experiments. All data were collected using Java 8 on a 3.4GHz Intel Core i7 processor after warming up the HotSpot JVM.

Performance of normal-case code. Perhaps the most important performance consideration for an exception mechanism is whether it slows down normal-case code. To evaluate this, we ported Java code that only uses exceptions lightly—specifically, the JOlden benchmarks [37] and, representing larger applications, the SunFlow benchmark [168] from the DaCapo suite [21]. SunFlow is a ray tracer containing ~200 classes and ~21K LOC.

To evaluate the overhead of the new exception mechanism fairly, we compared the running times of this code between the extended Genus language (Section 2.7) and the base Genus language. Despite support for reified generics,

Table 2.1: Performance with `EasyIO` (s)

Java w/ stack	Genus	Java w/o stack
7.19	1.16	1.16

the performance under base Genus is close to Java: compared to Java, it incurred a slowdown of 0.3%.

Figure 2.18 reports the results of this comparison. Each reported measurement averages at least 20 runs, with a standard error less than 1.2%. Benchmark parameters were set so that each run took more than 30 seconds.

Overall, the extended compiler generates slightly faster code than the original compiler. The average speedup is 2.4%, though performance varies by benchmark. The speedup is caused largely by the exception-based translation of enhanced `for` loops.

Performance of exception-heavy code. To evaluate the performance improvement that Genus obtains by avoiding stack-trace collection on realistic code, we measured the running times of pattern-matching regular expressions using the `EasyIO` package. It makes heavy use of exceptions to control search. The running times are shown in Table 2.1. Each number averages 10 runs, with 6.2M exceptions thrown in each run.

Stack-trace collection in Java causes more than 6× overhead compared to Genus exceptions. Genus targets Java, so it is not surprising that similar performance can be achieved with Java, if stack-trace collection is turned off and exception objects are cached.

Exception tunneling performance. We used a microbenchmark to explore the overhead of exception tunneling, the key difference between Genus and Java. The microbenchmark performs method calls on objects passed down from a

Table 2.2: Exception tunneling microbenchmarks

mechanism	exception objects	time (ns)
Java exceptions	new instance	817.7
	cached	124.7
Java unchecked wrappers	cached	826.0
Genus exceptions (tunneled)	new instance	139.8
	cached	128.6

higher-order function; the method call either throws an exception or returns immediately, and is prevented from being inlined. Since there is no other work done by the method, performance differences are magnified.

The results are shown in Table 2.2. We compare exception tunneling in Genus to two (unsafe) Java workarounds: typical Java exceptions and unchecked-exception wrappers. We also refine the comparison for typical Java exceptions and tunneled Genus exceptions based on whether a new exception object is created for each `throw`; throwing a single cached instance is reasonable for non-failure exceptions carrying no extra information. Each number averages 20 runs, with a standard error less than 0.6% of the mean.

The rightmost column measures time to exit via an exception. Genus exceptions perform well because they do not collect a stack trace. The slowdown compared to the second row is mostly because exception mismatch requires `borrowBlame$` and `handbackBlame$` calls (Section 2.7.3) in every loop iteration.

On the other hand, Genus exceptions significantly outperform unchecked wrappers, the safest way to tunnel in Java. Java's performance is poor here because an unchecked-exception object is created for each raised exception, whereas the implementation of Genus recycles `Blame` objects.

The microbenchmark also measures the time to return from a method normally. The average cost of a call and return in Java was 6.0 ns. In the absence of mismatch, our translation adds the overhead of passing a null pointer to the nor-

mal return path, increasing the cost slightly to 6.3 ns. The results in Figure 2.18 suggest this increase is negligible in practice.

2.9 Related Work

Notable approaches to exceptions. PL/I was the first language with exceptions. It supports user-defined exceptions with exception handlers dynamically bound to exceptions [111]. Goodenough [80] and Liskov and Snyder [107] introduced statically scoped handlers. CLU [107] was the first language to support some static checking of exceptions. Exceptions are declared in function signatures, and thrown exceptions must appear in these signatures. If not explicitly resignaled, propagating exceptions automatically convert to failure. Mesa [122] supports both termination- and resumption-style exceptions but does not check them statically. Ada [3] attaches handlers to blocks, procedures, or packages. Unhandled exceptions propagate automatically, but exceptions are not declared. Eiffel [117] exceptions originate from the violation of assertions and are raised implicitly. Upon exceptions, programmers can *retry* the execution with different parameters; otherwise, exceptions implicitly propagate to callers. Modula-3 [128] introduced fully statically checked exceptions. Black [19] and Garcia et al. [75] present comparative studies of some exception mechanisms.

Empirical findings. An empirical study by Cabral and Marques [35] shows that in Java and .NET exception handlers are not specialized enough to allow effective handling, which we believe is partly attributable to a lack of documentation of exceptions in the .NET case [36] and the rigidity of checked exceptions in the Java case. Robillard and Murphy [153] identify the global reasoning required of programmers as a major reason why exceptions are hard to use.

Exception analysis. Function types in functional languages such as ML and Haskell do not include exceptions because they would interfere with the use of higher-order functions. Exception capture can be avoided in SML [121] because exception types are generative, but other variants of ML lack this feature. Leroy and Pessaux [104] observe that uncaught exceptions are the most common failure mode of large ML applications, motivating them and others [68] to develop program analyses to infer exceptions. Such analyses can be helpful, especially for usage studies [184], but they necessarily involve trading off performance and precision, and entail non-local reasoning that does not aid programmers in reasoning about their code. A benefit of our mechanism is that it is likely to lead to more accurate, scalable static analyses, because precise exceptions largely remove the need to approximate exceptional control flow [27, 161].

Exception polymorphism. Some recent designs attempt to address the rigidity of checked exceptions through exception polymorphism: anchored exceptions [174] as a Java extension, polymorphic effects [155] as a Scala extension, row-polymorphic effect types in the Koka language [102], and the `rethrows` clause introduced in Swift 2 [169]. These approaches add annotation burden to exception-oblivious code; more fundamentally, they do not address exception capture.

Blame tracking. Our notion of blame is related to that introduced by work on contracts [72] and further developed by work on gradual typing [178], in which blame indicates where fault lies in the event of a contract violation (or cast failure). In our setting, an exception mismatch results in static blame being assigned that indicates where "fault" lies should an exception arise at run time,

with the compiler statically checking that the "faulty" program point handles exceptions.

Blame has polarity [72, 178]; in our setting, exception mismatch at covariant (or contravariant) positions gives rise to positive (or negative) blame. Existing mechanisms for checked exceptions only consider positive blame; exceptions with negative blame are the missing piece. By contrast, in Genus both kinds of exceptions are subject to static checking, and our implementation and formalization manifest their difference: exceptions with negative blame acquire new identities to achieve safe tunneling.

CHAPTER 3

ABSTRACTION-SAFE EFFECT HANDLERS

Algebraic effects [16, 147, 148] have developed into a powerful unifying language feature, shown to encompass a wide variety of other important features that include exceptions, dynamically scoped variables, coroutines, and asynchronous computation. Although some type systems make algebraic effects *type-safe* [15, 103, 106], we argue in this chapter that algebraic effects are not yet *abstraction-safe*: details about the use of effects leak through abstraction boundaries.

As an example, consider the higher-order abstraction `map`, which applies the same function to each element in a list:

```
map[X,Y,E](l : List[X], f : X → Y throws E) : List[Y] throws E
```

In general, the computation embodied in the functional argument `f` may be effectful, as indicated by the clause `throws E` in the type of `f`. To make it reusable, `map` is defined to be polymorphic over the latent effects `E` of `f`, and propagates any such effect to its own caller.

The `map` abstraction can be implemented in many different ways; modularity is preserved if clients cannot tell which implementation is hiding behind the abstraction boundary. It would thus be surprising if two implementations of this `map` abstraction behaved differently when used in the same context. However, current semantics of algebraic effects allow a client to observe different behaviors—and to distinguish between the two implementations—when one of the implementations happens to use algebraic effects internally.

For example, suppose an implementation of `map` traverses the list using an iterator object. The iterator throws a `NoSuchElement` exception when it reaches the

end of the list, and the implementation handles it accordingly. If the client function `f` also happens to throw `NoSuchElement`, the implementation may handle—*by accident*—an effect it is not designed to handle. By breaking the implementation of `map` in this way, such a client thereby improperly observes internals of its implementation. This violation of abstraction is also a failure of modularity.

We contend that this failure is a direct consequence of the dynamic semantics of algebraic effect handlers. Intuitively, for Reynolds' Abstraction Theorem [152] (also known as the Parametricity Theorem [177]) to hold for a language with type abstraction (such as System F), polymorphic functions cannot make decisions based on the types instantiating the type parameters. Analogously, parametricity of effect polymorphism demands that an effect-polymorphic function should not make decisions based on the effect it is instantiated with. Yet the dynamic nature of algebraic effects runs afoul of this requirement: an effect is handled by searching the dynamic scope for a handler that can handle the effect. To restore parametricity, we propose to give algebraic effects a new semantics based on *tunneling*:

> *Algebraic effects can be handled only by handlers that are statically aware of them; otherwise, effects* tunnel *through handlers.*

This semantics provides sound modular reasoning about effect handling, while preserving the expressive power of algebraic effects.

For a formal account of abstraction safety, the typical syntactic approach to type soundness no longer suffices, because it is difficult to syntactically track type-system properties that are deeper than subject reduction [17, 56, 120, 185]. By contrast, a semantic approach that gives a relational interpretation of types can be applied to the harder problem of reasoning about program refinement and equivalence. Therefore, a prime result of this chapter is a semantic type-

soundness proof for a core language with tunneled algebraic effects. To this end, we define a step-indexed, biorthogonal logical-relations model for the core language, giving a relational interpretation not just to types, but also to effects. We show this logical-relations model offers a sound and complete reasoning process for proving contextual refinement and equivalence. Effectful program fragments can then be rigorously proved equivalent, supporting reasoning about the soundness of program transformations. We proceed as follows:

- We illustrate the problem of accidentally handled effects in Section 3.1, clarifying the observation that algebraic effect handlers violate abstraction.
- We present tunneled algebraic effects in Section 3.2. Tunneling causes no significant changes to the usual syntax of algebraic effects; it changes the dynamic semantics of effects but does not lose any essential expressive power.
- We define the operational and static semantics of tunneling via a core language (Section 3.3).
- In Section 3.4, we give a logical-relations model for the core language. We establish important properties of the logical relation, including parametricity and soundness with respect to contextual refinement. These results, checked using Coq, make rigorous the claim that tunneled algebraic effects are abstraction-safe.
- We demonstrate the power of the logical relation in Section 3.5 by proving program equivalence. As promised, effect-polymorphic abstractions in the core language hide their use of effects.
- We survey related work in Section 3.6.

3.1 Algebraic Effects and Accidental Handling

Algebraic effects are gaining popularity among language designers because they enable statically checked, programmer-defined control-flow transfer. Legacy language abstractions for control flow, including exceptions, yielding iterators, and async/await, become just instances of algebraic effects.

We illustrate the problems with algebraic effects in the setting of a typical object-oriented language, like Java, C#, and Scala, that has been extended with algebraic effects and effect polymorphism. Despite this object-oriented setting, the problems we identify and the solution we propose are broadly applicable to languages with algebraic effects or with mechanisms subsumed by algebraic effects.

3.1.1 Algebraic Effects and Handlers

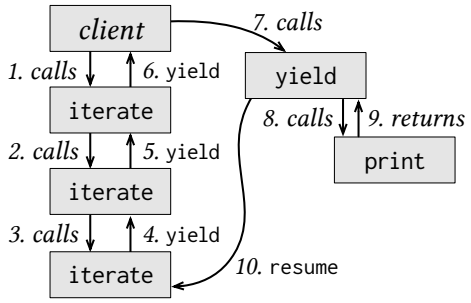
The generality of algebraic effects comes from the ability to define an *effect signature* whose implementations are provided by *effect handlers*. An effect signature defines one or more *effect operations*. For example, the code below

```
effect Yield[X] {
  yield(X) : void
}
```

defines an effect signature named `Yield`, parameterized by a type variable `X`. This signature contains only one operation, `yield`, and invoking this operation requires a value of type `X`. This `Yield` effect can be used for declarative definitions of iterators. For example, the function `iterate` is an in-order iterator for binary trees:

```
val tr : Tree[int] = ...
try { tr.iterate() }
with yield(x) {
    print(x)
    resume()
}
```

(a)



(b)

Figure 3.1. (a) Client code iterating over a binary tree. (b) A stack diagram showing the control flow.

```
class Tree[X] {
    val value : X
    val left, right : Tree[X]
    iterate[X]() : void throws Yield[X] {
        if (left != null) left.iterate()
        yield(value)
        if (right != null) right.iterate()
    }
    ...
}
```

Invoking an effect operation has the corresponding effect. In the example, the `iterate` function invokes the `yield` operation, so it has the effect `Yield[X]`. Static checking of effects requires that this effect be part of the function's type, in its `throws` clause.

Traversing a tree using the effectful `iterate` function uses the help of an effect handler (Figure 3.1a). The effectful computation is surrounded by `try { ... }`, while the handler follows `with` and provides an implementation for each effect operation. In this example, the implementation of `yield` first prints the yielded integer, and resumes the computation in the `try` block.

The implementation of an effect operation has access to the *continuation* of the computation in the corresponding `try` block. This continuation, denoted by the identifier `resume`, takes as an argument the result of the effect operation, and when invoked, resumes the computation at the invocation of the effect operation in the `try` block. Because the result type of `yield` is `void`, the call to `resume` accepts no argument. Figure 3.1b visualizes the control flow under this resumptive semantics using a stack diagram.

The handling code of Figure 3.1a is actually syntactic sugar for code declaring an anonymous handler:

```
try { tr.iterate() }
with new Yield[int]() {
  yield(x : int) : void { … }
}
```

The sugared form in Figure 3.1a requires the name `yield` to be unambiguous in the context. It is also possible to define standalone handlers instead of inlining them. Handlers can also have state. For example, handler `printInt`, defined separately from its using code, stops the iteration after 8 rounds:

```
handler printInt for Yield[int] {
  var cnt = 0 // State of the handler
  yield(x : int) : void {
    if (cnt < 8) {
      print(x)
      ++cnt
      resume()
    }
  }
}
```

```
// Using code allocates a handler object
// with state cnt initialized to 0
try { tr.iterate() }
with new printInt()
```

Effect Polymorphism. Higher-order functions like `map` accept functional arguments that are in general effectful. Such higher-order functions are therefore

polymorphic in the effects of their functional argument. Language designs for effects typically include this kind of polymorphism to allow the definition of reusable generic abstractions [91, 103, 106, 155]. As an example, consider a filtering iterator that yields only those elements satisfying a predicate `f` that has its own effects `E`.

```
fiterate[X,E](tr : Tree[X], f : X → bool / E) : void / Yield[X], E {
  foreach (x : X) in tr
    if (f(x)) { yield(x) }
}
```

Here we introduce "`/`" as a shorthand for `throws`. The higher-order function is parameterized by an *effect variable* `E`, which is the latent effect of the predicate `f`. The implementation iterates over the tree and yields elements that test true with `f`. Because it invokes `yield` and `f`, its effects consist of both `Yield[X]` and `E`.

3.1.2 Accidentally Handled Effects Violate Abstraction

Suppose we want a higher-order abstraction that computes the number of tree elements satisfying some predicate. It can be implemented by counting the elements yielded by `fiterate`, as shown by function `fsize1` in Figure 3.2. The same abstraction can also be implemented in a recursive manner, as shown by function `fsize2` in Figure 3.2. We would hope that these implementations are *contextually equivalent*, meaning that they can be interchanged freely without any client noticing a difference.

Unfortunately, there do exist clients that can distinguish between the two implementations, as shown in Figure 3.3. This client code interacts with the abstraction whose implementation is provided either by `fsize1` or by `fsize2`, and uses a function named `f` as the predicate. But it also does something else with each element that `f` is applied to, using the help of an effect handler: it

```
fsize1[X,E](tr : Tree[X], f : X → bool / E) : int / E {
    val num = 0
    try { fiterate(tr, f) }
    with yield(x : X) : void {
      ++num
      resume()
    }
    return num
}

fsize2[X,E](tr : Tree[X], f : X → bool / E) : int / E {
    val lsize = fsize2(tr.left(), f)
    val rsize = fsize2(tr.right(), f)
    val cur = f(tr.value()) ? 1 : 0
    return lsize + rsize + cur
}
```

Figure 3.2. Two implementations of a higher-order abstraction. The intended behaviors of these two implementations are the same: returning the number of elements satisfying a predicate in a binary tree.

```
val fsize = ... // The right-hand side is either fsize1 or fsize2
val g = fun(x : int) : bool / Yield[int] { yield(x) ; f(x) }
try { fsize(tr, g) }
with yield(x : int) : void {
    ... // do something with x
    resume()
}
```

Figure 3.3. A client that can distinguish between `fsize1` and `fsize2`, two supposedly equivalent implementations of the same abstraction.

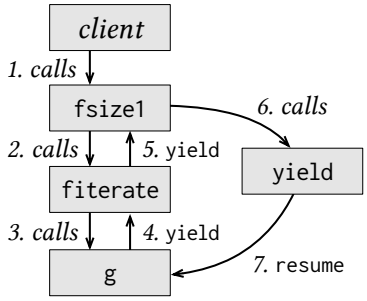


Figure 3.4. Snapshot of the stack when `fsize1` accidentally handles an `Yield` effect raised by applying `g`

wraps `f` in another function `g` (line 2), which, before calling `f`, yields the element to a handler that does the extra work (line 5). The client passes to the abstraction the wrapper `g`, which is eventually applied somewhere down the call chain. This application of `g` raises an `Yield[int]` effect, which the programmer would expect to be propagated back to the client code and handled at lines 4–7.

However, the programmer will be unpleasantly surprised if the client uses the implementation provided by `fsize1`. At the point where the effect arises, the runtime searches the dynamic scope for a handler that can handle the effect. Because the nearest dynamically enclosing handler for `Yield[int]` is the one in `fsize1` (lines 4–7 in Figure 3.2), the effect is unexpectedly intercepted by this handler, incorrectly incrementing the count. Figure 3.4 shows the stack snapshot when this accidental handling happens.

By contrast, the call to `fsize2` behaves as expected. Hence, two well-typed, type-safe, intuitively equivalent implementations of the same abstraction exhibit different behaviors to the same client. Syntactic type soundness is preserved—neither program gets stuck during execution—but the type system is not doing its job of enforcing abstraction.

The above example demonstrates a violation of abstraction from the implementation perspective, but a similar story can also be told from the client perspective: two apparently equivalent clients can make different observations on the same implementation of an abstraction. For example, consider the following two clients of `fsize1`: one looks like Figure 3.3 but with line 5 left empty, and the other is simply `fsize1(tr, f)`.

The handling of the `Yield` effect in the first client ought to amount to a no-op, so the two programs would be equivalent. Yet the equivalence does not hold because of the accidental handling of effects in the first program. This client perspective shows directly that the usual semantics of algebraic effect handling fails to comply with Reynolds' notion of relational parametricity [152], which states that applications of a function to related inputs should produce related results.

Prior efforts based on effect rows and row polymorphism have aimed to prevent client code from meddling with the effect-handling internals of library functions [18, 102]. Notably, recent work by Biernacki et al. [18] has shown relational parametricity for a core calculus with algebraic effects, but the type system compromises on the expressiveness of effect subsumption and relies on extra programmer annotations. For example, under their typing rules, function `fsize1` would not type-check unless (a) its signature mentioned the `Yield` effect, thereby exposing the implementation detail that `fsize1` handles `Yield` internally:

```
fsize1[X,E](tr : Tree[X], f: X → bool / {Yield[X], E} ) : int / E
```

or (b) a special "lift" operator is inserted at the place where `f` is applied in `fiterate`.

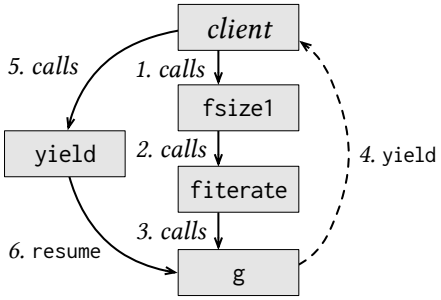


Figure 3.5. Snapshot of the stack when a `Yield` effect raised by applying `g` is tunneled to the client code

3.2 Tunneled Algebraic Effects

Just as algebraic effect handlers arose as a generalization of exception handlers [148], we build on the insight of Zhang et al. [193], who argue that *tunneled exceptions* make exceptions safer through a limited form of exception polymorphism. We show that tunneling can be generalized to algebraic effects broadly along with the general form of effect polymorphism presented in Section 3.1.1.

Tunneled algebraic effects address the problem of accidental handling. Despite this increase in safety, there is no increase in programmer effort. In fact, with the new tunneling semantics in effect, the examples from Section 3.1.2 become free of accidental handling, with *no* syntactic changes required.

Consider the version of Figure 3.3 that resulted in accidental handling of effects (i.e., the version that uses `fsize1`). Under the new semantics, the `Yield` effect raised by applying `g` is tunneled straightaway to the client code, without being intercepted by the intermediary contexts. Figure 3.5 shows the stack snapshot when this tunneling happens.

3.2.1 Tunneling Restores Modularity

This tunneling semantics enforces the modular reasoning principle that handlers should only handle effects they are locally aware of. In the example, the intermediary contexts, `fsize1` and `fiterate`, are polymorphic in an effect variable that represents the latent effects of their functional arguments. So they ought to be *oblivious* to whatever effect applying `g` might raise at run time. The modular reasoning principle hence prohibits handlers in these intermediary contexts from capturing any dynamic instantiations of the effect variable; accidental handling is impossible.

The client code, by contrast, is locally *aware* that applying `fsize1` to `g` manifests the latent effect of `g`. The modular reasoning principle thus requires that the client code provide a handler for this effect in order to maintain type safety.

The lack of modularity in the presence of higher-order functions is an inherent problem of language mechanisms based on some form of dynamic scoping, many of which are subsumed by algebraic effects. Among such effects, the one that most famously conflicts with modular reasoning is perhaps dynamically scoped variables.

Dynamically scoped variables increase code extensibility, as exemplified by the TEX programming language [101], because they act as implicit parameters that can be accessed—and overridden—in their dynamic extents. But their unpredictable semantics prevents wider adoption. In particular, a higher-order function may accidentally override variables that its functional argument expects from the dynamic scope, a phenomenon known in the Lisp community as the "downward funarg problem" [163]. This problem with dynamically scoped variables is an instance of accidental handling.

Fortunately, tunneling offers a solution broadly applicable to all algebraic effects, including dynamically scoped variables and exceptions. We illustrate this solution through an example involving the tunneling of multiple effects.

3.2.2 Tunneling Preserves the Expressivity of Dynamic Scoping

Consider the Visitor design pattern [74], which recursively traverses an abstract syntax tree (AST). Visitors often keep intermediate state in some associated *context*. For example, a type-checking visitor would use a typing environment as the context, while a pretty-printing visitor would use a context to keep track of the current indentation level. The state in such contexts is essentially an instance of dynamic scoping. Moreover, the type-checking visitor may expect the context to handle typing errors, while the pretty-printing visitor needs the context to handle I/O exceptions. A common `Visitor` interface is therefore unable to capture this variability in the notion of context. So either uses of the Visitor pattern are limited to settings that do not need context, or the programmer has to resort to error-prone workarounds.

One such workaround is to capture context information as mutable state. However, recursive calls to the visitor often need to update context information. So side effects need to be carefully undone as each recursive call returns; otherwise, subtrees yet to be visited would not have the right context information.

Tunneled algebraic effects provide the expressive power needed to address this quandary, without incurring the problems of dynamic scoping. Figure 3.6 shows a pretty-printing visitor defined using tunneled algebraic effects. The `Visitor` interface (lines 1–5) is generic with respect to the effects of the visitor methods. AST visitors can all implement this interface but provide their own notions of context. For the pretty-printer, indentation is modeled as an (immutable)

```
interface Visitor[E]  {
  visit(While) : void/E
  visit(Assign) : void/E
  ...
}
interface While extends Stmt {
  cond() : Expr
  body() : Stmt
  accept[E](v : Visitor[E]) : void/E { v.visit(this) }
  ...
}
effect Val[X] { get() : X }                        // Immutable variables
effect Var[X] extends Val[X] { put(X) : void }     // Mutable variables
effect IOExc { throw() : void }
print(s : String) : void / IOExc { ... }
indent(l : int) : void / IOExc { ... }
class pretty for Visitor[{Val[int],IOExc}]{
  visit(w : While) : void / _ {                    // Infers effects
    val l = get()                                  // Current level of indentation
    indent(l)                                      // Print indentation
    print("while ")
    w.cond().accept(this)
    print("\n")
    try { w.body().accept(this) }
    with get() : int { resume(l + 1) }             // Increment indentation level
  }
  ...
}
try { program.accept(new pretty()) }
with {
  get() : int { resume(0) }
  throw() : void { ... }
}
```

Figure 3.6. Using tunneled algebraic effects to provide access to the context for visitors

dynamically scoped variable, whose effect signature is given on line 12. This signature can be extended to support mutability (line 13), though it is not needed by this example. The visitor also uses methods `print` and `indent` (lines 15 and 16), which can raise I/O exceptions.

Pretty-printing `While` loops (lines 18–26) manipulates the dynamic scope. To properly indent, the current indentation level is obtained from the dynamically scoped variable by invoking the effect operation `get` (line 19). The loop body is printed using the same visitor, but with an updated indentation level. This overriding of the dynamically scoped variable is done by providing a new handler for the recursive visit of the loop body (lines 24–25). The initial level of indentation is provided by the client code on line 31.

Figure 3.7 visualizes the propagation of a `Yield[int]` effect and an `IOExc` exception raised when visiting a loop body. Notice that these effects tunnel through the effect-polymorphic `accept` methods. So even if any of the `accept` methods handled effects internally, they would not be able to intercept the effects passing by.

3.2.3 Accomplishing Tunneling by Statically Choosing Handlers

The modular reasoning principle requires that it be possible to reason statically about which handler is used for each invocation of effect operations. Accordingly, the language mechanism for accomplishing tunneling requires that an effect handler be given whenever an effect operation is invoked. As we show below, such a handler can take the form of a concrete definition or of a *handler variable*, and does not have to be provided explicitly in typical usage.

The effect-handling code on the left is actually shorthand for the code on the right, which explicitly names the exception handler to use:

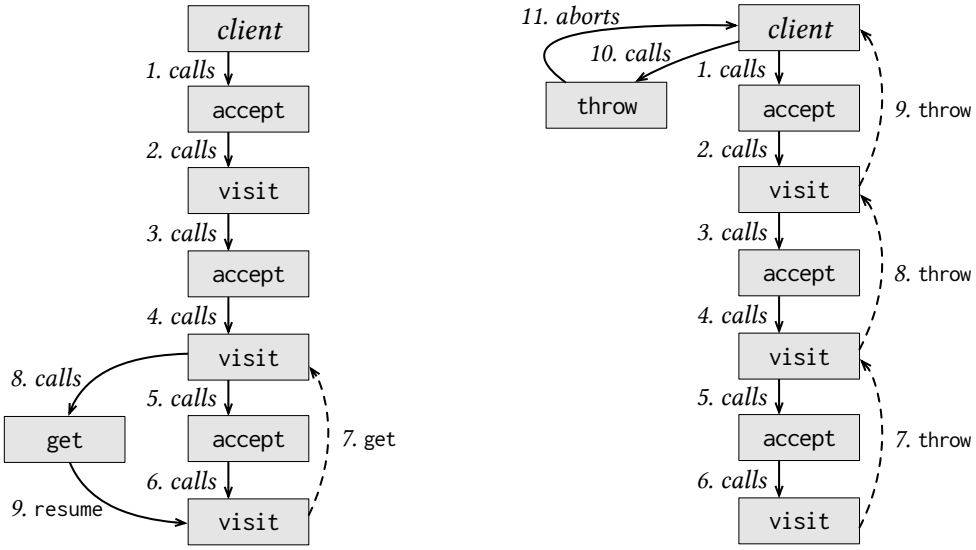


Figure 3.7. Left: stack snapshot at the point when printing the loop body asks for the current indentation level. Right: stack snapshot when an I/O exception is raised while printing the loop body.

```
try { throw() }
with throw() { … }
```

```
try { H.throw() }
with H = new IOExc() {
  throw() : void { … }
}
```

The handler with a concrete definition is given the name `H`, and the invocation `H.throw()` indicates that `H` is chosen explicitly as the handler for the effect operation.

While the `try–with` construct introduces bindings of handlers with concrete definitions, mentions of effect names in method, interface, or class headers introduce bindings of *handler variables*. For example, the `iterate` method from Section 3.1.1 mentions `Yield[X]` in its `throws` clause:

```
iterate[X]() : void / Yield[X] { … }
```

So `iterate` is desugared using explicit parameterization with a handler variable named `h`:

```
iterate[X, h : Yield[X]]() : void / h {
  if (left != null) left.iterate[X, h]()
  h.yield(value)
  if (right != null) right.iterate[X, h]()
} // Uses of the handler variable are highlighted
```

The method is polymorphic over a handler for `Yield[X]`, and the effectful computation in its body is handled by this handler.

Inferring omitted handlers. Naming the handler might seem verbose, but does not create a burden on the programmer: when programs are written using the usual syntax, the choice of handler is obvious, so the language can always figure out what is omitted.

To map a program written in the usual syntax into one in which the choice of handler is explicit, two phases of rewriting are performed: desugaring, and resolving omitted handlers. Desugaring involves

(a) introducing explicit bindings for concrete handler definitions and explicit handler-variable bindings for handler polymorphism, and

(b) identifying where handlers are omitted and must be resolved—namely at invocation sites of effect operations and of handler-polymorphic abstractions.

Once the program is desugared, an omitted handler for some effect signature (or effect operation) is *always* resolved to the nearest lexically enclosing handler binding for that signature (or operation).

In the examples above, the concrete handler definition `H` is the closest lexically enclosing one for `IOExc`, and the handler variable `h` is the closest lexically enclosing one for `Yield[X]`. So when they are omitted in the program text, the language automatically chooses them as handlers for the respective effects.

Tunneling. Tunneling falls out naturally. Performing the rewriting discussed above on the example in Figure 3.3 yields the following program:

```
val fsize = …
val g = fun[h : Yield[int]](x : int) : bool / h { h.yield(x); f(x) }
try { fsize(tr, g[H]) }
with H = new Yield[int]() { yield(x : int) : void { … } }
```

When `g` is passed to the higher-order function, its handler variable is substituted with the locally declared handler `H`, the closest lexically enclosing one for `Yield[int]`. As a result, the invocation of the effect operation in `g` will unequivocally be handled by `H`, rather than being intercepted by some handler declared in an intermediary context.

As another example, class `pretty` in Figure 3.6 is actually parameterized by two handler variables `ind` and `io` representing the dynamically scoped indentation level and the handling of I/O exceptions:

```
class pretty[ind : Val[int], io : IOExc] for Visitor[{ind,io}] {
  visit(w : While) : void / {ind,io} {
    …
    try { w.body().accept[{H,io}](this[H, io]) }
    with H = new Val[int]() { get() : int { resume(l+1) } }
    …
  }
  …
}
```

For the code that visits the loop body (i.e., line 24 of Figure 3.6, whose full form is also shown above), two handlers for `Val[int]` are lexically in scope—the handler variable `ind` and the handler definition named `H`. The closest lexically enclosing one is chosen, so loop bodies are visited using an incremented indentation level. Notice that the `this` keyword is actually a handler-polymorphic value, so it is possible to recursively invoke the visitor while overriding the handler. For the handling of I/O exceptions, the handler variable `io` is the only applicable handler

lexically in scope. Both kinds of effects are guaranteed not to be captured by the effect-polymorphic `accept` methods.

Disambiguating the choice of handler. Although explicitly naming handlers is not necessary in most cases, the ability to specify handlers explicitly adds expressivity. For example, in their recent work on using algebraic effects to encode complex event processing, Braćevac et al. (2018) describe a situation where different invocations of the same effect operation need to be handled by different surrounding handlers. The ability to explicitly specify handlers addresses this need.

3.2.4 Region Capabilities as Computational Effects

With the rewriting described in Section 3.2.3, it may seem superfluous to still statically track the effects of methods like `iterate` and `g` via `throws` clauses. After all, the desugared method signatures explicitly require a handler to be provided—it appears guaranteed that the effect of any call to `iterate` or `g` is properly handled.

However, programs would go wrong if these effects were ignored. Consider the program in the upper half of Figure 3.8, where the type system does not track the effect of `g` other than requiring a handler to be provided. In this example, `g` is passed to the (higher-order) identity function, and the result is stored into a local variable `f`. As with the `fsize` example, the handler to provide for `g` is resolved to the closest enclosing handler `H`. So when a `Yield` effect arises as a result of applying `f` to an integer (line 8), the handling code in `H` is executed. But `H` does not have a computation to resume: the current control state is no longer within a `try` block!

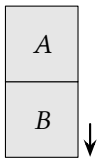
```
val f : int → void
val g = fun[h : Yield[int]](x : int) : void { … h.yield(x) … }
try { f = identity(g[H]) }
with H = new Yield[int]() {
  yield(x : int) : void { … resume() }
}
f(0) // Invokes g[H](0) but causes a run-time error
```

```
val f : int → void
val g = fun[h : IOExc](x : int) : void { … h.throw() … }
try { f = identity(g[H]) }
with H = new IOExc() {
  throw() : void { … }
}
… // Unable to transfer control here when H finishes
return f // Run-time error if f is invoked later
```

Figure 3.8. Both programs would type-check statically but go wrong dynamically if the type system did not tracking the effect of g other than requiring a handler to be provided. Region capabilities (Section 3.2.4) address this issue.

A similar problem happens when handlers do not resume—but rather abort—computations in `try` blocks, such as exception handlers. In the program in the lower half of Figure 3.8, `g` may throw an `IOExc` exception, and the computation in `g[H]` is returned to the caller. When an exception handler finishes, control ought to be transferred to the point immediately following the corresponding `try–with` statement. However, when `g[H]` is invoked later, raising an exception, the computation following `try–with` is no longer available when the exception handler `H` finishes execution, because the stack frame containing the computation has been deallocated.

We can view a `try–with` statement as marking a program point which, at run time, divides the stack into two regions. In the figure to the right, the stack grows downwards, and an effect is raised at the bottom of the stack. The two regions, A and B, represent the possible control-flow transfer targets when the handler finishes handling the effect: the upper region A is the computation to jump to if the handler aborts the computation in the `try` block, and the lower region B is the `try`-block computation possibly to be resumed.



To handle an effect thus requires the *capability* to access the stack regions. A `try–with` statement introduces a unique capability, which the corresponding handler holds within the `try` block. Capabilities must not be able to escape their corresponding `try` blocks; otherwise, they would refer to deallocated stack regions.

To this end, the type system tracks these stack-region capabilities as computational effects. In the example above, applying `g` needs the capability held by the handler variable `h`. So the effect of `g` is this capability, denoted by `h` in the `throws` clause of `g`:

```
val g = fun[h : Yield[int]](x : int) : void / h { … h.yield(x) … }
```

In the `try` block, the handler `H` provided by the enclosing `try–with` is used to substitute for the handler variable, so the expression `identity(g[H])`—and therefore `f`—must have type `int → void / H`, meaning that the capability held by `H` is needed to apply `f`. However, because `f` outlives the `try–with` that introduces this capability, the capability will be unavailable when `f` is applied. Fortunately, since capabilities are tracked statically, the type system rejects this program.

This capability-effect system is more expressive than previous approaches to effect polymorphism that use an escape analysis to prevent accesses to deallo-

```
// An effect-polymorphic data structure
class cachingFun[X,Y,E] for Fun[X,Y,E] {
  val f : X ⟶ Y/E
  cachingFun(f : X ⟶ Y/E) { this.f = f }
  apply(x : X) : Y/E { … f(x) … }
  …
}

// Using code
val g = fun(x : int) : void / Yield { … }
try {
  val f = new cachingFun(g)
  … // apply f
}
with yield(x : int) : void { … }
```

Figure 3.9. An effect-polymorphic data structure and its using code

cated regions [141, 193]. In contrast to these approaches, we allow values with latent polymorphic effects to escape into (effect-polymorphic) data structures, as long as uses of the data structure do not outlive the corresponding stack regions. For example, class `cachingFun` in Figure 3.9 implements a function that caches the result of its application, and is polymorphic over the latent effects of that function. In the using code, the effectful computation in `g` escapes into the newly allocated data structure denoted by `f`. So `f` has type `Fun[int,void,H]`, assuming the handler is named `H`. But since `f` does not outlive the `try–with` that introduces the capability held by `H`, the code is safely accepted.

3.2.5 Implementation

This chapter does not explore the options for implementing the new effect mechanism. However, implementation is largely an orthogonal concern. It appears

entirely feasible to build on ongoing work on efficiently implementing algebraic effects [25, 103]. When algebraic effects are used as a termination-style exception mechanism, it is important that `try`-block computations be cheap; it should be possible to adapt the technique used by Zhang et al. [193], which corresponds to passing (static) capability labels rather than whole continuations.

3.3 A Core Language

To pin down the semantics of tunneled algebraic effects, we formally define a core calculus we call $\lambda_{⇩⇧}$, which captures the key aspects of the language mechanisms introduced in Section 3.2.

3.3.1 Syntax

The language $\lambda_{⇩⇧}$ is a simply typed lambda calculus, extended with language facilities essential to tunneling, including effect polymorphism, handler polymorphism, a way to access effect operations (⇧), and a way to discharge effects (⇩). For simplicity, it is assumed that handlers are always given explicitly for effectful computations (rather than resolving elided handlers to the closest lexically enclosing binding), that effect signatures contain exactly one effect operation, and that effect operations accept exactly one argument. Lifting these restrictions is straightforward, but adds syntactic complexity that obscures the key issues.

Like previous calculi, our formalism omits explicit handler state. But handler state can be encoded within the algebraic-effects framework—and consequently in $\lambda_{⇩⇧}$—as Bauer and Pretnar [16] show. It is also possible to extend the core calculus with handler state and, potentially, existentials to ensure encapsulation of the state. We expect such an extension to be largely orthogonal.

effects	e	::=	$\alpha \mid \ell$
capability labels	ℓ	::=	$\mathsf{L} \mid \mathsf{h}.\mathbf{lbl}$
types	T, S	::=	$\mathbb{1} \mid S \to [T]_{\overline{e}} \mid \forall\alpha.\, T \mid \Pi_{\mathsf{h}:\mathbb{F}}\, [T]_{\overline{e}}$
handlers	h	::=	$\mathsf{h} \mid H^{\mathsf{L}}$
terms	t, s	::=	$() \mid \mathsf{x} \mid \mathbf{let}\ \mathsf{x}:T = t\ \mathbf{in}\ s \mid$
			$\lambda \mathsf{x}:T\,.\, t \mid t\ s \mid \Lambda\alpha.\, t \mid t\ [\overline{e}] \mid$
			$\lambda \mathsf{h}:\mathbb{F}.\, t \mid t\ h \mid \Uparrow h \mid \Downarrow^{\mathsf{L}}_{[T]_{\overline{e}}} t$
handler definitions	H, G	::=	$\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\, t$
effect var. environments	Δ	::=	$\varnothing \mid \Delta, \alpha$
handler var. environments	P	::=	$\varnothing \mid \mathrm{P}, \mathsf{h}:\mathbb{F}$
term var. environments	Γ	::=	$\varnothing \mid \Gamma, \mathsf{x}:T$
label environments	Ξ	::=	$\varnothing \mid \Xi, \mathsf{L}:[T]_{\overline{e}}$
effect names	$\mathbb{F}$		
label identifiers	L		
effect variables	α		
handler variables	h		
term variables	$\mathsf{x}, \mathsf{y}, \mathsf{k}, \ldots$		

Figure 3.10. Syntax of $\lambda_{\Downarrow\Uparrow}$

Figure 3.10 presents the syntax of $\lambda_{\Downarrow\Uparrow}$. An overline denotes a (possibly empty) sequence of syntactic objects. For instance, $\overline{e}$ denotes a list of effects, with an empty sequence denoted by $\varnothing$. The i-th element in a sequence $\overline{\bullet}$ is denoted by $\bullet^{(i)}$. Metavariables standing for identifiers are given a lighter color.

Types. Types include the base type $\mathbb{1}$, function types $S \to [T]_{\overline{e}}$, effect-polymorphic types $\forall\alpha.\, T$, and handler-polymorphic types $\Pi_{\mathsf{h}:\mathbb{F}}\, [T]_{\overline{e}}$. The result type of a function type or that of a handler-polymorphic type can be annotated by effects. For brevity, we omit explicit annotations when there is no effect; for example, the

type $S \to T$ means $S \to [T]_{\varnothing}$. Computations directly quantified by effect variables must be pure, an easily lifted simplification that matches both typical usage and previous formalizations (e.g., [18, 103]). Abstract handlers h implement effect signatures, whose names are ranged over by $\mathbb{F}$. We assume a global mapping from effect names to effect signatures; given an effect name $\mathbb{F}$, the helper function $op(\cdot)$ returns the type of its effect operation.

Terms. Terms consist of the standard ones of the simply typed lambda calculus plus those concerned with effects, including the ⇧- and ⬇-terms, effect-polymorphic abstraction $\Lambda\alpha.\, t$ and its application, and handler-polymorphic abstraction $\lambda \mathsf{h} : \mathbb{F}.\, t$ and its application. The ⇧- and ⬇- terms, which we read as "up" and "down", correspond in the language of Section 3.2 to effect operations and effect handling.

For example, given a handler variable h that implements an effect $\mathbb{F}$ with signature $T_1 \to T_2$, the term ⇧h is an effect operation whose implementation is provided by h, while the term ⇧$\mathsf{h}\ v$ invokes the effect operation (assuming the value v has type T_1), raising an effect.

The `try–with` construct corresponds to terms of form

$$\Downarrow^{\mathsf{L}}_{[T]_{\overline{e}}} (\lambda \mathsf{h} : \mathbb{F}.\, t)\ H^{\mathsf{L}}$$

where the term t corresponds to the computation in the `try` block, and H the handler in the `with` clause. Term t is placed in a handler-polymorphic abstraction, which is then immediately applied to the handler. The handler variable h, occurring free in t, can be thought of as creating a local binding for handler H that t uses to handle its effects.

As discussed in Section 3.2.4, a `try–with` expression implicitly marks a program point, creating a stack-region capability that is in scope within the `try` block. Correspondingly, ⬇-terms in $\lambda_{\Downarrow\Uparrow}$ mark program points that create capabilities.

These capabilities are represented by labels L; terms of form $\blacktriangledown^{\mathsf{L}}_{[T]_{\bar{e}}}\ t$ bind a label L whose scope is t. Subterms of t can then use L to show they possess the region capability. Labels bound by different $\blacktriangledown$-terms are assumed to be unique. To ensure unique typing, a $\blacktriangledown$-term is annotated with the type and effects $[T]_{\bar{e}}$ of the very term; they correspond to the type and effects of a `try–with` expression as a whole. We omit these annotations when they are irrelevant in the context.

To handle an effect requires both the handling code and the capability. Hence, handler definitions H are always tagged by a label in scope, forming pairs of form H^{L}. Our use of $\blacktriangledown$-terms supports pairing different handler definitions with the same program point, a useful feature that is common in programming languages with exception handlers but that does not seem to be captured by previous formalisms. For example, the following term corresponds to associating two handlers with the same `try` block:

$$\blacktriangledown^{\mathsf{L}}_{[T]_{\bar{e}}}\left(\lambda \mathsf{h}_1 : \mathbb{F}_1.\,(\lambda \mathsf{h}_2 : \mathbb{F}_2.\,t)\ H_2^{\mathsf{L}}\right)\ H_1^{\mathsf{L}}$$

Handlers. A handler h is either a handler variable h or a definition–label pair H^{L}. The (statically unknown) label embodied in a handler variable h is denoted by $\mathsf{h}.\mathbf{lbl}$. Substituting a handler of form H^{L} for a handler variable h also replaces any occurrences of $\mathsf{h}.\mathbf{lbl}$ with L.

Handler definitions H are of form $\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\,t$, where $\mathbb{F}$ is the effect signature being implemented and t is the handling code. Variables x and k may occur free in t: x denotes the argument passed to the effect operation, and k the continuation at the point the effect operation is invoked.

Effects. The type system needs to track region capabilities as computational effects. An effect e is either an effect variable α or a capability label ℓ. A capability label is either a label L bound by a $\blacktriangledown$-term, or the label of a handler variable.

With effects being just capabilities, we can handle effect composition simply: effect sequences $\overline{e}$ are essentially sets—the order and multiplicity of effects in a sequence are irrelevant. Substituting an effect sequence $\overline{e}$ for an effect variable α that is part of another effect sequence works by flattening $\overline{e}$ and replacing α with the flattened effects.

3.3.2 Operational Semantics

To give an operational semantics to $\lambda_{⬇⇧}$, terms in Figure 3.10 are extended with a ⇩-construct:

$$\textit{terms} \quad t, s \ ::= \ \ldots \mid ⇩^{\lfloor} t$$

A small-step operational semantics of the core language is given in Figure 3.11. Individual reduction steps take the form $\overline{\lfloor_1} \,;\, t_1 \longrightarrow \overline{\lfloor_2} \,;\, t_2$, meaning that term t_1 steps to term t_2 while the set of *activated* region-capability labels grows from $\overline{\lfloor_1}$ to $\overline{\lfloor_2}$. Per rule [E-DOWN], a label bound by a ⬇-term is activated when the ⬇-term is reduced to a ⇩-term. While ⬇-terms lexically bind labels, ⇩-terms are non-binding constructs; evaluation contexts of form $⇩^{\lfloor} K$ serve as stack delimiters. Closed terms can then mention activated labels, which is useful, for example, in defining a logical relation on closed terms. The transitive, reflexive closure of the small-step transition relation $\longrightarrow$ is denoted by $\longrightarrow^*$. The distinction between ⬇ and ⇩ is not brought up in Zhang and Myers [188, 189], on which this chapter is based; however, it does exist in the accompanying Coq formalization. This chapter makes the distinction explicit so that the paper presentation here matches the Coq formalization more closely.

Rule [E-DOWN-UP] deals with invocations of effect operations. Evaluating an invocation $⇧ H^{\lfloor_0} v$ amounts to evaluating the handling code in H, which requires

values $v, u ::= () \mid \lambda x{:}T\,.\,t \mid \Lambda\alpha\,.\,t \mid \lambda h{:}\mathbb{F}\,.\,t \mid \Uparrow H^{\mathsf{L}}$

evaluation contexts $K ::= [\cdot] \mid K\ t \mid v\ K \mid K\ [\overline{\mathsf{L}}] \mid K\ H^{\mathsf{L}} \mid \mathbf{let}\ x{:}T = K\ \mathbf{in}\ t \mid \Downarrow^{\mathsf{L}} K$

$$\boxed{\overline{\mathsf{L}_1}\,;\, t_1 \longrightarrow \overline{\mathsf{L}_2}\,;\, t_2}$$

[E-KTX] $$\frac{\overline{\mathsf{L}_1}\,;\, t_1 \longrightarrow \overline{\mathsf{L}_2}\,;\, t_2}{\overline{\mathsf{L}_1}\,;\, K[t_1] \longrightarrow \overline{\mathsf{L}_2}\,;\, K[t_2]}$$

[E-APP] $$\overline{\mathsf{L}}\,;\, (\lambda x{:}T\,.\,t)\ v \longrightarrow \overline{\mathsf{L}}\,;\, t\,\{v/x\}$$

[E-EAPP] $$\overline{\mathsf{L}}\,;\, (\Lambda\alpha\,.\,t)\ [\overline{\mathsf{L}_0}] \longrightarrow \overline{\mathsf{L}}\,;\, t\left\{\overline{\mathsf{L}_0}\Big/\alpha\right\}$$

[E-HAPP] $$\overline{\mathsf{L}}\,;\, (\lambda h{:}\mathbb{F}\,.\,t)\ H^{\mathsf{L}_0} \longrightarrow \overline{\mathsf{L}}\,;\, t\left\{H^{\mathsf{L}_0}/h\right\}$$

[E-LET] $$\overline{\mathsf{L}}\,;\, \mathbf{let}\ x{:}T = v\ \mathbf{in}\ t \longrightarrow \overline{\mathsf{L}}\,;\, t\,\{v/x\}$$

[E-DOWN] $$\frac{\mathsf{L}_0 \notin \overline{\mathsf{L}}}{\overline{\mathsf{L}}\,;\, \Downarrow^{\mathsf{L}_0} t \longrightarrow \overline{\mathsf{L}}, \mathsf{L}_0\,;\, \Downarrow^{\mathsf{L}_0} t}$$

[E-DOWN-VAL] $$\frac{\mathsf{L}_0 \in \overline{\mathsf{L}}}{\overline{\mathsf{L}}\,;\, \Downarrow^{\mathsf{L}_0} v \longrightarrow \overline{\mathsf{L}}\,;\, v}$$

[E-DOWN-UP] $$\frac{op(\mathbb{F}) = T_1 \to T_2 \qquad \mathsf{L}_0 \in \overline{\mathsf{L}} \qquad \mathsf{L}_0 \curvearrowright K}{\overline{\mathsf{L}}\,;\, \Downarrow^{\mathsf{L}_0} K\left[\Uparrow\left(\mathbf{handler}^{\mathbb{F}}\ x\ k.\ t\right)^{\mathsf{L}_0} v\right] \longrightarrow \overline{\mathsf{L}}\,;\, t\left\{\lambda y{:}T_2\,.\,\Downarrow^{\mathsf{L}_0} K[y]\Big/k\right\}\{v/x\}}$$

$$\boxed{\mathsf{L} \curvearrowright K}$$

$\mathsf{L} \curvearrowright [\cdot]$ $$\frac{\mathsf{L} \curvearrowright K}{\mathsf{L} \curvearrowright K\ t} \qquad \frac{\mathsf{L} \curvearrowright K}{\mathsf{L} \curvearrowright v\ K} \qquad \frac{\mathsf{L} \curvearrowright K}{\mathsf{L} \curvearrowright K\ [\overline{\mathsf{L}_0}]} \qquad \frac{\mathsf{L} \curvearrowright K}{\mathsf{L} \curvearrowright K\ H^{\mathsf{L}_0}}$$

$$\frac{\mathsf{L} \curvearrowright K}{\mathsf{L} \curvearrowright \mathbf{let}\ x{:}T = K\ \mathbf{in}\ t} \qquad \frac{\mathsf{L} \curvearrowright K \qquad \mathsf{L} \neq \mathsf{L}_0}{\mathsf{L} \curvearrowright \Downarrow^{\mathsf{L}_0} K}$$

Figure 3.11. Operational semantics of $\lambda_{\Downarrow\Uparrow}$

the capability to access the stack regions marked by $\llcorner_0$. Therefore, to reduce $\Uparrow H^{\llcorner_0}\ v$, the dynamic scope is searched for an evaluation context $\Downarrow^{\llcorner_0} K[\cdot]$ where $\llcorner_0 \curvearrowright K$ (that is, K does not contain $\llcorner_0$ as a stack delimiter). This evaluation context K is then passed to the handling code as the resumption continuation. In case the handler chooses to abort the computation in K, evaluation continues with the surrounding evaluation context, as rule [E-KTX] suggests. Notice that K is guarded by $\Downarrow^{\llcorner_0}$ when passed to the handling code, so any invocation of effect operations labeled by $\llcorner_0$ in the resumption continuation can be handled properly.

3.3.3 Static Semantics

The static semantics of $\lambda_{\Downarrow\Uparrow}$ is provided in Figures 3.12–3.14. Term well-formedness rules have form $\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T]_{\overline{e}}$, where Δ, P, Γ and Ξ are environments of free effect variables, handler variables, term variables, and labels, respectively. The judgment form says that under these environments the term t has type T and effects $\overline{e}$.

Rule [T-UP] suggests that an effect operation $\Uparrow h$ is a first-class value with type $T \to [S]_e$, where $T \to S$ is the effect signature and e is the capability held by h.

Rule [T-DOWN] suggests that a term t guarded by $\Downarrow^{\llcorner}$ possesses the capability $\llcorner$: in the premise, t is typed under the label environment augmented with $\llcorner$. Importantly, however, the label $\llcorner$ must not occur free in the result type T and effects $\overline{e}$. Otherwise, $\llcorner$ could outlive its binding scope. For instance, it would then be possible to type the term $\Downarrow^{\llcorner}_{[S_1 \to [S_2]_{\llcorner}]_{\varnothing}} (\Uparrow H^{\llcorner})$ as $S_1 \to [S_2]_{\llcorner}$, assuming H implements effect signature $S_1 \to S_2$. Per evaluation rule [E-DOWN-VAL], the term would then evaluate to $\Uparrow H^{\llcorner}$. But without a corresponding $\Downarrow^{\llcorner}$ in the dynamic context, an invocation of the effect operation $\Uparrow H^{\llcorner}\ t$ would get stuck. Notice that

$$\boxed{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T]_{\overline{e}}}$$

[T-UNIT] $\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash () : [\mathbb{1}]_{\varnothing}$

[T-VAR] $$\frac{\Gamma(\mathsf{x}) = T}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \mathsf{x} : [T]_{\varnothing}}$$

[T-ABS] $$\frac{\Delta \mid \mathrm{P} \mid \Xi \vdash S \qquad \Delta \mid \mathrm{P} \mid \Gamma, \mathsf{x}{:}S \mid \Xi \vdash t : [T]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \lambda \mathsf{x}{:}S \,.\, t : [S \to [T]_{\overline{e}}]_{\varnothing}}$$

[T-APP] $$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [S \to [T]_{\overline{e}}]_{\overline{e}} \qquad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash s : [S]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t\ s : [T]_{\overline{e}}}$$

[T-LET] $$\frac{\Delta \mid \mathrm{P} \mid \Xi \vdash S \qquad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash s : [S]_{\overline{e}} \qquad \Delta \mid \mathrm{P} \mid \Gamma, \mathsf{x}{:}S \mid \Xi \vdash t : [T]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \mathbf{let}\ \mathsf{x}{:}S = s\ \mathbf{in}\ t : [T]_{\overline{e}}}$$

[T-EABS] $$\frac{\Delta, \alpha \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T]_{\varnothing}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \Lambda\alpha \,.\, t : [\forall \alpha \,.\, T]_{\varnothing}}$$

[T-EAPP] $$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [\forall \alpha \,.\, T]_{\overline{e_1}} \qquad (\forall i)\ \Delta \mid \mathrm{P} \mid \Xi \vdash e_2^{(i)}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t\ [\overline{e_2}] : [T\,\{\overline{e_2}/\alpha\}]_{\overline{e_1}}}$$

[T-HABS] $$\frac{\Delta, \mathsf{h}{:}\mathbb{F} \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \lambda \mathsf{h}{:}\mathbb{F} \,.\, t : [\Pi_{\mathsf{h}:\mathbb{F}}\, [T]_{\overline{e}}]_{\varnothing}}$$

[T-HAPP] $$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : \left[\Pi_{\mathsf{h}:\mathbb{F}}\, [T]_{\overline{e_1}}\right]_{\overline{e_2}} \qquad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h : \mathbb{F} \mid \ell}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t\ h : [T\,\{h/\mathsf{h}\}]_{\overline{e_1}\{h/\mathsf{h}\},\, \overline{e_2}}}$$

[T-UP] $$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h : \mathbb{F} \mid \ell \qquad op(\mathbb{F}) = T \to S}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \Uparrow h : [T \to [S]_{\ell}]_{\varnothing}}$$

[T-DOWN] $$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi, \mathsf{L}{:}[T]_{\overline{e}} \vdash t : [T]_{\overline{e}, \mathsf{L}} \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \Downarrow^{\mathsf{L}}_{[T]_{\overline{e}}} t : [T]_{\overline{e}}}$$

[T-SUB] $$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T_1]_{\overline{e_1}} \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T_1 \leq T_2 \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e_1} \leq \overline{e_2}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T_2]_{\overline{e_2}}}$$

Figure 3.12. Term well-formedness rules of $\lambda_{\Downarrow\Uparrow}$

$$\boxed{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h : \mathbb{F} \mid \ell}$$

$$[\text{T-HVAR}]\ \frac{\mathrm{P}(\mathsf{h}) = \mathbb{F}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \mathsf{h} : \mathbb{F} \mid \mathsf{h}.\mathbf{lbl}} \qquad [\text{T-HDEF}]\ \frac{\Xi(\mathsf{L}) = [S]_{\overline{e}} \qquad op(\mathbb{F}) = T_1 \to T_2 \qquad \Delta \mid \mathrm{P} \mid \Gamma,\ \mathsf{x}{:}T_1,\ \mathsf{k}{:}T_2 \to [S]_{\overline{e}} \mid \Xi \vdash t : [S]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \left(\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\ t\right)^{\mathsf{L}} : \mathbb{F} \mid \mathsf{L}}$$

Figure 3.13. Handler well-formedness rules of $\lambda_{\downarrow\uparrow}$

we do not give a typing rule for the auxiliary $\Downarrow$ construct; $\Downarrow$-terms emerge only when evaluating a $\lambda_{\downarrow\uparrow}$ program.

Handler well-formedness rules have form $\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h : \mathbb{F} \mid \ell$, which states that handler h implements the algebraic effect $\mathbb{F}$ and has label ℓ. Rule [T-HDEF] requires that the handling code t of a handler H^{L} be typable using the type and effects $[S]_{\overline{e}}$ prescribed by the label L.

Encoding data structures. For simplicity, $\lambda_{\downarrow\uparrow}$ does not have data structures. However, $\lambda_{\downarrow\uparrow}$ allows their encoding via closures, where the captured variables may have latent polymorphic effects. For example, a simplified *pair* data structure polymorphic over the latent effects of its components can be encoded as follows:

$$\begin{array}{rcll}
T & \stackrel{def}{=} & S_1 \to [S_2]_{\alpha} & S_1 \text{ and } S_2 \text{ can be any closed type} \\
\mathit{pair} & \stackrel{def}{=} & \Lambda\alpha.\,\lambda\mathsf{x}{:}T\,.\,\lambda\mathsf{y}{:}T\,.\,\lambda\mathsf{f}{:}T \to T \to T\,.\,\mathsf{f}\ \mathsf{x}\ \mathsf{y} & \text{construct a pair} \\
\mathit{first} & \stackrel{def}{=} & \Lambda\alpha.\,\lambda\mathsf{p}{:}(T \to T \to T) \to T\,.\,\mathsf{p}\ (\lambda\mathsf{x}{:}T\,.\,\lambda\mathsf{y}{:}T\,.\,\mathsf{x}) & \text{obtain the first component} \\
\mathit{second} & \stackrel{def}{=} & \Lambda\alpha.\,\lambda\mathsf{p}{:}(T \to T \to T) \to T\,.\,\mathsf{p}\ (\lambda\mathsf{x}{:}T\,.\,\lambda\mathsf{y}{:}T\,.\,\mathsf{y}) & \text{obtain the second component}
\end{array}$$

The two components, both having type T, have α as their latent effects. The *pair* constructor is then polymorphic in α.

This example cannot be readily encoded in previous formalisms [141, 193], which support a limited form of effect polymorphism by introducing second-class values that cannot escape their defining scope. In particular, these systems

$$\boxed{\Delta \mid \mathrm{P} \mid \Xi \vdash T}$$

$$\Delta \mid \mathrm{P} \mid \Xi \vdash \mathbb{1} \qquad \frac{\Delta \mid \mathrm{P} \mid \Xi \vdash T \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash S \qquad (\forall i)\ \Delta \mid \mathrm{P} \mid \Xi \vdash e^{(i)}}{\Delta \mid \mathrm{P} \mid \Xi \vdash T \to [S]_{\overline{e}}}$$

$$\frac{\Delta, \alpha \mid \mathrm{P} \mid \Xi \vdash T}{\Delta \mid \mathrm{P} \mid \Xi \vdash \forall \alpha.\, T} \qquad \frac{\Delta \mid \mathrm{P}, \mathsf{h}:\mathbb{F} \mid \Xi \vdash T \qquad (\forall i)\ \Delta \mid \mathrm{P}, \mathsf{h}:\mathbb{F} \mid \Xi \vdash e^{(i)}}{\Delta \mid \mathrm{P} \mid \Xi \vdash \Pi_{\mathsf{h}:\mathbb{F}}\, [T]_{\overline{e}}}$$

$$\boxed{\Delta \mid \mathrm{P} \mid \Xi \vdash e}$$

$$\frac{\alpha \in \Delta}{\Delta \mid \mathrm{P} \mid \Xi \vdash \alpha} \qquad \frac{\mathsf{L} \in \mathit{domain}(\Xi)}{\Delta \mid \mathrm{P} \mid \Xi \vdash \mathsf{L}} \qquad \frac{\mathsf{h} \in \mathit{domain}(\mathrm{P})}{\Delta \mid \mathrm{P} \mid \Xi \vdash \mathsf{h}.\mathbf{lbl}}$$

$$\boxed{\Delta \mid \mathrm{P} \mid \Xi \vdash T \leq S}$$

$$\Delta \mid \mathrm{P} \mid \Xi \vdash \mathbb{1} \leq \mathbb{1} \qquad \frac{\Delta \mid \mathrm{P} \mid \Xi \vdash T_2 \leq T_1 \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash S_1 \leq S_2 \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e_1} \leq \overline{e_2}}{\Delta \mid \mathrm{P} \mid \Xi \vdash T_1 \to [S_1]_{\overline{e_1}} \leq T_2 \to [S_2]_{\overline{e_2}}}$$

$$\frac{\Delta, \alpha \mid \mathrm{P} \mid \Xi \vdash T_1 \leq T_2}{\Delta \mid \mathrm{P} \mid \Xi \vdash \forall \alpha.\, T_1 \leq \forall \alpha.\, T_2} \qquad \frac{\Delta \mid \mathrm{P}, \mathsf{h}:\mathbb{F} \mid \Xi \vdash T_1 \leq T_2 \qquad \Delta \mid \mathrm{P}, \mathsf{h}:\mathbb{F} \mid \Xi \vdash \overline{e_1} \leq \overline{e_2}}{\Delta \mid \mathrm{P} \mid \Xi \vdash \Pi_{\mathsf{h}:\mathbb{F}}\, [T_1]_{\overline{e_1}} \leq \Pi_{\mathsf{h}:\mathbb{F}}\, [T_2]_{\overline{e_2}}}$$

$$\frac{\Delta \mid \mathrm{P} \mid \Xi \vdash T_1 \leq T_2 \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T_2 \leq T_3}{\Delta \mid \mathrm{P} \mid \Xi \vdash T_1 \leq T_3}$$

$$\boxed{\Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e_1} \leq \overline{e_2}}$$

$$\frac{(\forall j,\ \exists i)\ e_1^{(j)} = e_2^{(i)} \qquad (\forall i)\ \Delta \mid \mathrm{P} \mid \Xi \vdash e_2^{(i)}}{\Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e_1} \leq \overline{e_2}}$$

Figure 3.14. Type and effect well-formedness of $\lambda_{\downarrow\uparrow}$

$$
\begin{aligned}
C \quad ::= \quad & [\cdot] \mid C[\lambda x{:}T\,.\,[\cdot]] \mid C[[\cdot]\ t] \mid C[t\ [\cdot]] \mid C[\textbf{let}\ x{:}T = [\cdot]\ \textbf{in}\ t] \mid \\
& C[\textbf{let}\ x{:}T = t\ \textbf{in}\ [\cdot]] \mid C[\Lambda\alpha\,.\,[\cdot]] \mid C[[\cdot]\ [\overline{e}]] \mid C[\lambda h{:}\mathbb{F}\,.\,[\cdot]] \mid C[[\cdot]\ h] \mid \\
& C\left[t\ \left(\textbf{handler}^{\mathbb{F}}\ x\ k.\ [\cdot]\right)^{\llcorner}\right] \mid C\left[\Uparrow\left(\textbf{handler}^{\mathbb{F}}\ x\ k.\ [\cdot]\right)^{\llcorner}\right] \mid C\left[\Downarrow^{\llcorner}[\cdot]\right]
\end{aligned}
$$

Figure 3.15. Program contexts of $\lambda_{\Downarrow\Uparrow}$

do not admit the subterm $\lambda x{:}T\,.\,\lambda y{:}T\,.\,x$ in the definition of *first*, or the subterm $\lambda y{:}T\,.\,y$ in the definition of *second*. Variable x in the first subterm, being second-class because it has a polymorphic latent effect, escapes its defining scope via the closure $\lambda y : T\,.\,x$ capturing it. Similarly, in the second subterm, variable y escapes its defining scope. By contrast, our use of explicit effect polymorphism and capability labels enables the definition of effect-polymorphic data structures.

3.3.4 Contextual Refinement and Equivalence

A *program context* is a program with a hole $[\cdot]$ in it. Figure 3.15 shows the different types of program contexts in $\lambda_{\Downarrow\Uparrow}$. Well-formedness judgments for program contexts have the form

$$\vdash C\ :\ \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T$$

The meaning of this judgment is that if a term t satisfies the typing judgment $\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [S]_{\overline{e}}$, then plugging t into C results in a program that satisfies $\varnothing \mid \varnothing \mid \varnothing \mid \varnothing \vdash C[t] : [T]_{\varnothing}$. These rules can be found in Figure 3.16.

Our goal is to prove that with tunneling, algebraic effects can preserve abstraction. Abstraction is shown by demonstrating that implementations using effects internally cannot be distinguished by external observers. The gold standard of indistinguishability is *contextual equivalence*: two terms are contextually equiv-

$$\boxed{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T}$$

$$\vdash [\cdot] : \varnothing \mid \varnothing \mid \varnothing \mid \varnothing \mid [T]_{\varnothing} \rightsquigarrow T \qquad \frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T \to [S]_{\overline{e}}]_{\varnothing} \rightsquigarrow T' \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T}{\vdash C[\lambda x{:}T\,.\,[\cdot]] : \Delta \mid \mathrm{P} \mid \Gamma, x{:}T \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T' \quad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T]_{\overline{e}}}{\vdash C[[\cdot]\ t] : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T \to [S]_{\overline{e}}]_{\overline{e}} \rightsquigarrow T'} \qquad \frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T' \quad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T \to [S]_{\overline{e}}]_{\overline{e}}}{\vdash C[t\ [\cdot]] : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T]_{\overline{e}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T]_{\overline{e}} \rightsquigarrow T' \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash S \qquad \Delta \mid \mathrm{P} \mid \Gamma, x{:}S \mid \Xi \vdash t : [T]_{\overline{e}}}{\vdash C[\mathbf{let}\ x{:}S = [\cdot]\ \mathbf{in}\ t] : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T]_{\overline{e}} \rightsquigarrow T' \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash S \qquad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash s : [S]_{\overline{e}}}{\vdash C[\mathbf{let}\ x{:}S = s\ \mathbf{in}\ [\cdot]] : \Delta \mid \mathrm{P} \mid \Gamma, x{:}S \mid \Xi \mid [T]_{\overline{e}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [\forall \alpha\,.\,T]_{\varnothing} \rightsquigarrow T'}{\vdash C[\Lambda\alpha\,.\,[\cdot]] : \Delta, \alpha \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T]_{\varnothing} \rightsquigarrow T'} \qquad \frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T\,\{\overline{e_2}/\alpha\}]_{\overline{e_1}} \rightsquigarrow T' \quad (\forall i)\ \Delta \mid \mathrm{P} \mid \Xi \vdash e_2^{(i)}}{\vdash C[[\cdot]\ [\overline{e_2}]] : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [\forall \alpha\,.\,T]_{\overline{e_1}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [\Pi_{h:\mathbb{F}}\,[T]_{\overline{e}}]_{\varnothing} \rightsquigarrow T'}{\vdash C[\lambda h{:}\mathbb{F}\,.\,[\cdot]] : \Delta \mid \mathrm{P}, h{:}\mathbb{F} \mid \Gamma \mid \Xi \mid [T]_{\overline{e}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T\,\{h/h\}]_{\overline{e_1}\{h/h\}, \overline{e_2}} \rightsquigarrow T' \qquad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h : \mathbb{F} \mid \ell}{\vdash C[[\cdot]\ h] : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid \left[\Pi_{h:\mathbb{F}}\,[T]_{\overline{e_1}}\right]_{\overline{e_2}} \rightsquigarrow T'}$$

$$\frac{\begin{array}{c}\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T\,\{\mathsf{L}/h.\mathbf{lbl}\}]_{\overline{e_1}\{\mathsf{L}/h.\mathbf{lbl}\}, \overline{e_2}} \rightsquigarrow T' \\ \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : \left[\Pi_{h:\mathbb{F}}\,[T]_{\overline{e_1}}\right]_{\overline{e_2}} \qquad \Xi(\mathsf{L}) = [S]_{\overline{e_3}} \qquad op(\mathbb{F}) = T_1 \to T_2\end{array}}{\vdash C\left[t\ \left(\mathbf{handler}^{\mathbb{F}}\ x\ k.\ [\cdot]\right)^{\mathsf{L}}\right] : \Delta \mid \mathrm{P} \mid \Gamma, x{:}T_1, k{:}T_2 \to [S]_{\overline{e_3}} \mid \Xi \mid [S]_{\overline{e_3}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T_1 \to [T_2]_{\mathsf{L}}]_{\varnothing} \rightsquigarrow T' \qquad \Xi(\mathsf{L}) = [S]_{\overline{e}} \qquad op(\mathbb{F}) = T_1 \to T_2}{\vdash C\left[\Uparrow\left(\mathbf{handler}^{\mathbb{F}}\ x\ k.\ [\cdot]\right)^{\mathsf{L}}\right] : \Delta \mid \mathrm{P} \mid \Gamma, x{:}T_1, k{:}T_2 \to [S]_{\overline{e}} \mid \Xi \mid [S]_{\overline{e}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T]_{\overline{e}} \rightsquigarrow T' \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e}}{\vdash C\left[\Downarrow^{\mathsf{L}}_{[T]_{\overline{e}}}[\cdot]\right] : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi, \mathsf{L} \mid [T]_{\overline{e},\mathsf{L}} \rightsquigarrow T'}$$

$$\frac{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T_2]_{\overline{e_2}} \rightsquigarrow T' \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T_1 \leq T_2 \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e_1} \leq \overline{e_2}}{\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T_1]_{\overline{e_1}} \rightsquigarrow T'}$$

Figure 3.16. Context well-formedness of $\lambda_{\Downarrow\Uparrow}$

alent if plugging them into an arbitrary well-formed program context always gives two programs whose evaluations yield the same observation [124].

We define contextual equivalence in terms of *contextual refinement*, a weaker, asymmetric relation that requires one term to be able to simulate the behaviors of the other:

Definition 1 (contextual refinement $\preccurlyeq_{ctx}$ and contextual equivalence $\approx_{ctx}$)**.**

$$\begin{aligned}
\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{ctx} t_2 : [T]_{\bar{e}} \quad &\stackrel{def}{=} \quad \forall C.\ \vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T]_{\bar{e}} \rightsquigarrow T' \Rightarrow \\
&\quad \forall \overline{\llcorner_1}, v_1.\ \varnothing\,;\, C[t_1] \longrightarrow^* \overline{\llcorner_1}\,;\, v_1 \Rightarrow \\
&\quad \exists \overline{\llcorner_2}, v_2.\ \varnothing\,;\, C[t_2] \longrightarrow^* \overline{\llcorner_2}\,;\, v_2 \\
\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \approx_{ctx} t_2 : [T]_{\bar{e}} \quad &\stackrel{def}{=} \quad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{ctx} t_2 : [T]_{\bar{e}} \wedge \\
&\quad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_2 \preccurlyeq_{ctx} t_1 : [T]_{\bar{e}}
\end{aligned}$$

For programs to be equivalent in the above definition, they only need to agree on termination, but this seemingly weak observation of program behavior does not weaken the discriminating power of the definition, because of the universal quantification over all possible program contexts and because $\lambda_{\downarrow\uparrow}$ is Turing-complete (see Section 3.4.1). Hence, if two computations that reduce to observably different values, one can always construct a program context that makes the two computations exhibit different termination behavior.

However, the universal quantification over contexts also makes it hard to show equivalence by using the definition directly. We therefore take one of the standard approaches to establishing contextual equivalence: constructing a logical relation that implies contextual equivalence.

3.4 A Sound Logical-Relations Model

We develop a logical-relations model for $\lambda_{\downarrow\uparrow}$ and prove the important property that logically related terms are contextually equivalent. This semantic soundness result guarantees that the language $\lambda_{\downarrow\uparrow}$ is both type-safe and abstraction-safe.

3.4.1 Step Indexing

A logical-relations model gives a relational interpretation of types, traditionally defined inductively on the structure of types. But language features like recursive types require a more sophisticated induction principle. Algebraic effects present a similar challenge because effect signatures can be defined recursively.

Recursively defined effect signatures give rise to programs that diverge, and consequently make the language Turing-complete. For example, suppose effect $\mathbb{F}$ has signature $op(\mathbb{F}) = \mathbb{1} \to \Pi_{\mathsf{h}:\mathbb{F}}\, [T]_{\mathsf{h}.\mathbf{lbl}}$, which recursively mentions $\mathbb{F}$, and that H is defined as follows:

$$H \stackrel{def}{=} \mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\ \mathsf{k}\ \ (\lambda \mathsf{h}:\mathbb{F}.\ \uparrow \mathsf{h}\ ()\ \mathsf{h})$$

Then the evaluation of the program $\downarrow^{\mathsf{L}}_{[T]_\varnothing}\ (\lambda \mathsf{h}:\mathbb{F}.\ \uparrow \mathsf{h}\ ()\ \mathsf{h})\ \ H^{\mathsf{L}}$ does not terminate:

$$\begin{aligned}
&\varnothing\ ;\ \downarrow^{\mathsf{L}}\ \left(\lambda \mathsf{h}:\mathbb{F}.\ \uparrow \mathsf{h}\ ()\ \mathsf{h}\right)\ \ H^{\mathsf{L}} \\
\longrightarrow\ &\mathsf{L}\ ;\ \downarrow^{\mathsf{L}}\ \left(\lambda \mathsf{h}:\mathbb{F}.\ \uparrow \mathsf{h}\ ()\ \mathsf{h}\right)\ \ H^{\mathsf{L}} \longrightarrow \mathsf{L}\ ;\ \downarrow^{\mathsf{L}}\ \left(\uparrow H^{\mathsf{L}}\ ()\ H^{\mathsf{L}}\right) \\
\longrightarrow\ &\mathsf{L}\ ;\ \left(\lambda \mathsf{y}:\Pi_{\mathsf{h}:\mathbb{F}}\, [T]_{\mathsf{h}.\mathbf{lbl}}\ .\ \downarrow^{\mathsf{L}}\ \mathsf{y}\ H^{\mathsf{L}}\right)\ \ (\lambda \mathsf{h}:\mathbb{F}.\ \uparrow \mathsf{h}\ ()\ \mathsf{h}) \\
\longrightarrow\ &\mathsf{L}\ ;\ \downarrow^{\mathsf{L}}\ \left(\lambda \mathsf{h}:\mathbb{F}.\ \uparrow \mathsf{h}\ ()\ \mathsf{h}\right)\ \ H^{\mathsf{L}} \longrightarrow \cdots
\end{aligned}$$

Because of this recursion in the signature of $\mathbb{F}$, structural induction alone is unable to give a well-defined relational interpretation of $\mathbb{F}$.

Step indexing [10] has been successfully applied to cope with recursive types (e.g., by Ahmed [5]). In this approach, the logical relation is defined using a

$$[\text{LÖB}]\quad \frac{P, \triangleright Q \vdash Q}{P \vdash Q} \qquad [\text{MONO}]\quad \frac{P, Q \vdash R}{P, \triangleright Q \vdash \triangleright R}$$

Figure 3.17. Rules for $\triangleright$

double induction, first on a step index, and second on the structure of types. Intuitively, the step index indicates for how many evaluation steps the proposition is true; at step 0 everything is vacuously true, and if a proposition is true for any number of steps then it is true in a non-step-indexed setting.

Our definition is step-indexed. It uses a logic equipped with the modality $\triangleright$, read as "later", which offers a clean abstraction of step indexing [11, 58]. If proposition P holds for n steps, then $\triangleright P$ means P holds for $n - 1$ steps. So P implies $\triangleright P$. Importantly, the $\triangleright$ modality provides the [LÖB] axiom (Figure 3.17), which can be viewed as an induction principle on step indices. The $\triangleright$ modality distributes over other connectives, so rule [MONO] is derivable.

As we shall see in Section 3.4.4, to ensure well-definedness, recursive invocations of the interpretation of effect signatures occur under the $\triangleright$ modality.

3.4.2 World Indexing

The operational semantics presented in Section 3.3.2 evolves a set of activated capability labels, analogous to how calculi supporting reference cells evolve a set of allocated memory locations during evaluation. Therefore, our logical relation is indexed by possible worlds of activated labels, analogous to how a logical relation for reasoning about reference cells is indexed by possible worlds describing allocated memory locations [146]. Our logical relation is binary; a world W is composed of two sets of activated labels, denoted by W_1 and W_2.

World W' is a *future world* of W, written as $W \subseteq W'$, if W' (possibly) adds freshly activated labels to W.

3.4.3 A Biorthogonal Term Relation

We introduce a logical relation for terms, which are closed under the empty variable environments but may use capability labels that are not locally bound. The term relation is defined using the technique of *biorthogonality*, pioneered by Pitts and Stark [146]. Biorthogonality, also known as $\top\top$-closure, lends itself to languages whose operational semantics manipulate evaluation contexts [18, 59, 96]: in a biorthogonal term relation, two terms are related if evaluating them in related evaluation contexts yields related observations.

Figure 3.18 shows that the term relation $\mathcal{T}$ is defined using relation $\mathcal{R}$ providing a notion of relatedness for evaluation contexts and relation O relating observations. Apart from the $\mathcal{S}$ relation, the definitions are standard. We define logical equivalence in terms of a notion of *logical refinement*, in much the same way that we define contextual equivalence in terms of contextual refinement. Rather than requiring the terms to exhibit the same termination behavior, the observation relation O relates two computations where termination of the first computation merely *implies* that of the second one:

$$\begin{aligned} O\,(W,\ t_1,\ t_2) \ \stackrel{def}{=}\ & \left(\exists W', v_1, v_2.\ W_1' = W_1 \ \wedge\ t_1 = v_1 \ \wedge\ W_2 \,;\, t_2 \longrightarrow^* W_2' \,;\, v_2\right) \ \vee \\ & \left(\exists W', t_1'.\ W_2' = W_2 \ \wedge\ W_1 \,;\, t_1 \longrightarrow W_1' \,;\, t_1' \ \wedge\ \triangleright\, O\left(W',\ t_1',\ t_2\right)\right) \end{aligned}$$

The O relation is defined recursively; the use of the $\triangleright$ modality suggests that the definition is implicitly indexed by the number of remaining evaluation steps the first computation can take.

Two evaluation contexts are related by $\mathcal{R}$ if they yield related observations when applied to related values. However, in the presence of algebraic effects,

$$
\begin{aligned}
\mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}(W,\, t_1,\, t_2) &\stackrel{def}{=} \forall K_1, K_2.\, \mathcal{R}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}(W,\, K_1,\, K_2) \Rightarrow \mathcal{O}(W,\, K_1[t_1],\, K_2[t_2]) \\
\mathcal{R}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}(W,\, K_1,\, K_2) &\stackrel{def}{=} \forall W'.\, W \subseteq W' \Rightarrow \\
&\quad \left(\forall v_1, v_2.\, \mathcal{V}[\![T]\!]^{\rho}_{\delta}(W',\, v_1,\, v_2) \Rightarrow \mathcal{O}(W',\, K_1[v_1],\, K_2[v_2])\right) \wedge \\
&\quad \left(\forall t_1, t_2.\, \mathcal{S}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}(W',\, t_1,\, t_2) \Rightarrow \mathcal{O}(W',\, K_1[t_1],\, K_2[t_2])\right) \\
\mathcal{S}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}(W,\, K_1[t_1],\, K_2[t_2]) &\stackrel{def}{=} \exists \psi, \overline{\llcorner_1}, \overline{\llcorner_2}.\, \mathcal{U}[\![\overline{e}]\!]^{\rho}_{\delta}\left(W,\, t_1,\, t_2,\, \psi,\, \overline{\llcorner_1},\, \overline{\llcorner_2}\right) \wedge \\
&\quad \left(\forall j.\, \llcorner_i^{(j)} \curvearrowright K_i\right)_{\,(i\,=\,1,2)} \wedge \\
&\quad \forall W', t_1', t_2'.\, W \subseteq W' \Rightarrow \psi\left(W',\, t_1',\, t_2'\right) \Rightarrow \\
&\quad \triangleright\, \mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}\left(W',\, K_1\left[t_1'\right],\, K_2\left[t_2'\right]\right)
\end{aligned}
$$

Figure 3.18. Biorthogonality

values are not the only kind of irreducible term. Terms of form $K\left[\Uparrow H^{\llcorner} v\right]$ where $\llcorner \curvearrowright K$ are stuck when put into an empty evaluation context.

So we borrow from Biernacki et al. [18] a logical relation $\mathcal{S}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$, which, being a smaller relation than $\mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$, relates two computations that can possibly get stuck by themselves because they raise effects among $\overline{e}$. The definition of the $\mathcal{R}$ relation then requires that two related evaluation contexts yield related observations when applied to not only values related by $\mathcal{V}$ but also terms related by $\mathcal{S}$. The $\mathcal{S}$ relation is discussed further in Section 3.4.4.

Because of the use of biorthogonality, and assuming parametricity is derivable, our term relation is automatically complete with respect to contextual refinement [59, 146]: contextually equivalent terms are always logically related. So the key theorems to prove are parametricity and soundness.

The definitions of the relations $\mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$, $\mathcal{R}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$, and $\mathcal{S}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$ are mutually recursive, and are parameterized by the semantic interpretation of the type ($\mathcal{V}[\![T]\!]^{\rho}_{\delta}$) and that of the effect sequence ($\mathcal{U}[\![\overline{e}]\!]^{\rho}_{\delta}$), defined below.

3.4.4 Semantic Types, Semantic Effect Signatures, and Semantic Effects

The logical relation $\mathcal{V}[\![T]\!]^{\rho}_{\delta}$ (Figure 3.19), defined by structural induction on the type T, interprets T as a binary relation on values. The unit type and function types are interpreted in a standard way, following the contract that the logical relation should be preserved by the elimination (or introduction) forms of the types.

Effect-polymorphic types and handler-polymorphic types bind effect variables and handler variables. Accordingly, environments δ and ρ are introduced to provide substitutions for variables occurring free in the type being interpreted:

$$\delta ::= \varnothing \mid \delta,\ \alpha \mapsto \left\langle \overline{\mathsf{L}_1}, \overline{\mathsf{L}_2}, \phi \right\rangle \qquad \rho ::= \varnothing \mid \rho,\ \mathsf{h} \mapsto \left\langle H_1^{\mathsf{L}_1}, H_2^{\mathsf{L}_2}, \eta \right\rangle$$

We use δ_1 and δ_2 (resp. ρ_1 and ρ_2) to mean the substitution functions for free effect (resp. handler) variables. In addition to these syntactic substitution functions, the environment δ maps each effect variable to a third component that is the semantic interpretation chosen for the effect variable, while the environment ρ maps each handler variable to a third component that is the term relation the computations of the two handlers satisfy. (Metavariables ϕ, η, and ψ range over relation variables.) The definitions in Figure 3.19 are also parameterized by a label environment Ξ; labels in the domain of Ξ may occur free in the types and effects being interpreted. We omit Ξ for brevity.

The definition of $\mathcal{V}[\![\forall\alpha.\, T]\!]^{\rho}_{\delta}$ shows the source of the abstraction guarantees provided by effect-polymorphic abstractions: two effect-polymorphic abstractions are related if their applications are related however the effect variable is interpreted. The definition of $\mathcal{V}[\![\Pi_{\mathsf{h}:\mathbb{F}}\ [T]_{\overline{e}}]\!]^{\rho}_{\delta}$ says that two handler-polymorphic abstractions are related if their applications to any related handlers are related.

Semantic types

$$\mathcal{V}[\![\mathbb{1}]\!]^\rho_\delta(W,\, v_1,\, v_2) \stackrel{def}{=} v_1 = () \;\wedge\; v_2 = ()$$

$$\mathcal{V}[\![T \to [S]_{\overline{e}}]\!]^\rho_\delta(W,\, v_1,\, v_2) \stackrel{def}{=} \forall W', u_1, u_2.\; W \subseteq W' \Rightarrow \mathcal{V}[\![T]\!]^\rho_\delta(W',\, u_1,\, u_2) \Rightarrow \mathcal{T}[\![[S]_{\overline{e}}]\!]^\rho_\delta(W',\, v_1\; u_1,\, v_2\; u_2)$$

$$\mathcal{V}[\![\forall\alpha.\, T]\!]^\rho_\delta(W,\, v_1,\, v_2) \stackrel{def}{=} \forall W', \overline{\mathsf{L}_1}, \overline{\mathsf{L}_2}, \phi.\; W \subseteq W' \Rightarrow \phi \subseteq W' \Rightarrow \mathcal{T}[\![[T]_\varnothing]\!]^\rho_{\delta,\, \alpha \mapsto \langle \overline{\mathsf{L}_1}, \overline{\mathsf{L}_2}, \phi\rangle} \left(W',\, v_1\; [\overline{\mathsf{L}_1}],\, v_2\; [\overline{\mathsf{L}_2}]\right)$$

$$\mathcal{V}[\![\Pi_{\mathsf{h}:\mathbb{F}}\, [T]_{\overline{e}}]\!]^\rho_\delta(W,\, v_1,\, v_2) \stackrel{def}{=} \forall W', H_1^{\mathsf{L}1}, H_2^{\mathsf{L}2}, \eta.\; W \subseteq W' \Rightarrow \mathsf{L}_i \in W'_i\;{}_{(i=1,2)} \Rightarrow \triangleright\mathcal{H}[\![\mathbb{F}]\!](W',\, H_1,\, H_2,\, \eta) \Rightarrow \mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho,\, \mathsf{h} \mapsto \left\langle H_1^{\mathsf{L}1}, H_2^{\mathsf{L}2}, \eta\right\rangle}_\delta \left(W',\, v_1\; H_1^{\mathsf{L}1},\, v_2\; H_2^{\mathsf{L}2}\right)$$

Semantic effect signatures

$$\mathcal{H}[\![\mathbb{F}]\!](W,\, H_1,\, H_2,\, \eta) \stackrel{def}{=} H_i = \mathbf{handler}^{\mathbb{F}}\; \mathsf{x}\; \mathsf{k}.\, t_i\;{}_{(i=1,2)} \;\wedge\; op(\mathbb{F}) = T \to S \;\wedge\; \forall W'.\; W \subseteq W' \Rightarrow \forall v_1, v_2.\; \mathcal{V}[\![T]\!]^\varnothing_\varnothing(W',\, v_1,\, v_2) \Rightarrow \forall u_1, u_2.\; \begin{pmatrix} \forall W''.\; W' \subseteq W'' \Rightarrow \\ \forall w_1, w_2.\; \mathcal{V}[\![S]\!]^\varnothing_\varnothing(W'',\, w_1,\, w_2) \Rightarrow \\ \triangleright\, \eta(W'',\, u_1\; w_1,\, u_2\; w_2) \end{pmatrix} \Rightarrow \eta(W',\, t_1\,\{u_1/\mathsf{k}\}\,\{v_1/\mathsf{x}\},\, t_2\,\{u_2/\mathsf{k}\}\,\{v_2/\mathsf{x}\})$$

Semantic effects

$$\mathcal{U}[\![\alpha]\!]^\rho_\delta\left(W,\, t_1,\, t_2,\, \psi,\, \overline{\mathsf{L}_1},\, \overline{\mathsf{L}_2}\right) \stackrel{def}{=} \delta(\alpha) = \left\langle \overline{\mathsf{L}'_1}, \overline{\mathsf{L}'_2}, \phi\right\rangle \;\wedge\; \phi\left(W,\, t_1,\, t_2,\, \psi,\, \overline{\mathsf{L}_1},\, \overline{\mathsf{L}_2}\right)$$

$$\mathcal{U}[\![\ell]\!]^\rho_\delta(W,\, t_1,\, t_2,\, \psi,\, \mathsf{L}_1,\, \mathsf{L}_2) \stackrel{def}{=} \rho_i\ell = \mathsf{L}_i\;{}_{(i=1,2)} \;\wedge\; t_1 = \Uparrow H_1^{\mathsf{L}1}\, v_1 \;\wedge\; t_2 = \Uparrow H_2^{\mathsf{L}2}\, v_2 \;\wedge\; \triangleright\mathcal{H}[\![\mathbb{F}]\!]\left(W,\, H_1,\, H_2,\, \mathcal{T}[\![\ell]\!]^\rho_\delta\right) \;\wedge\; op(\mathbb{F}) = T \to S \;\wedge\; \triangleright\mathcal{V}[\![T]\!]^\varnothing_\varnothing(W,\, v_1,\, v_2) \;\wedge\; \psi \equiv \triangleright\mathcal{V}[\![S]\!]^\varnothing_\varnothing$$

$$\mathcal{U}[\![\overline{e}]\!]^\rho_\delta(W,\, t_1,\, t_2,\, \psi,\, \mathsf{L}_1,\, \mathsf{L}_2) \stackrel{def}{=} \exists i.\; \mathcal{U}\left[\!\left[e^{(i)}\right]\!\right]^\rho_\delta(W,\, t_1,\, t_2,\, \psi,\, \mathsf{L}_1,\, \mathsf{L}_2)$$

Auxiliary definitions

$$\mathcal{T}[\![\mathsf{h}.\mathbf{lbl}]\!]^\rho_\delta(W,\, t_1,\, t_2) \stackrel{def}{=} \rho(\mathsf{h}) = \left\langle H_1^{\mathsf{L}1}, H_2^{\mathsf{L}2}, \eta\right\rangle \;\wedge\; \eta(W,\, t_1,\, t_2)$$

$$\mathcal{T}[\![\mathsf{L}]\!]^\rho_\delta(W,\, t_1,\, t_2) \stackrel{def}{=} \Xi(\mathsf{L}) = [T]_{\overline{e}} \;\wedge\; \mathcal{T}[\![[T]_{\overline{e}}]\!]^\rho_\delta(W,\, t_1,\, t_2)$$

$$\phi \subseteq W \stackrel{def}{=} \forall W', t_1, t_2, \psi, \overline{\mathsf{L}_1}, \overline{\mathsf{L}_2}.\; \phi\left(W',\, t_1,\, t_2,\, \psi,\, \overline{\mathsf{L}_1},\, \overline{\mathsf{L}_2}\right) \Rightarrow \overline{\mathsf{L}_i} \subseteq W_i\;{}_{(i=1,2)}$$

Figure 3.19. Relational interpretation of types, effect signatures, and effects

Handler-relatedness is defined by the logical relation $\mathcal{H}[\![\mathbb{F}]\!]$, indexed by effect signatures $\mathbb{F}$. As discussed in Section 3.4.1, effect signatures can be recursively defined. Thus $\mathcal{H}[\![\mathbb{F}]\!]$ is invoked here under the $\triangleright$ modality so that the definition is admissible.

The interpretation of an effect signature $\mathbb{F}$ is similar to that of a function type: two handlers are related if their handling code is related under any related substitutions for the free variables. $\mathcal{H}[\![\mathbb{F}]\!]$ relates a third component η that is a term relation; the handler computations are in this relation. $\mathcal{H}[\![\mathbb{F}]\!]$ is not indexed by environments δ and ρ, because effect signatures are closed.

We revisit the definition of the $\mathcal{S}$ relation introduced in Section 3.4.3. As mentioned earlier, $\mathcal{S}$ can relate terms of form $K\left[\Uparrow H^{\mathsf{L}}\ v\right]$ where $\mathsf{L} \curvearrowright K$—although terms in this relation are not necessarily effectful, because it is possible for programs that use effects and those that do not to be equivalent. The operational meaning of these terms depends upon a larger surrounding context marked by label L. Therefore, the relation $\mathcal{S}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$ is defined using the $\mathcal{U}[\![\overline{e}]\!]^{\rho}_{\delta}$ relation, which relates the (possibly) effectful computations t_1 and t_2 and also a term relation ψ specifying the *outcomes* of these computations in a larger context. Given this specification, the definition of $\mathcal{S}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$ checks that plugging any pair of terms t_1' and t_2' related by the outcome specification into the current evaluation contexts yield related terms. Notice that $K_1\left[t_1'\right]$ and $K_2\left[t_2'\right]$ only need to be related in the future as indicated by the use of the $\triangleright$ modality, because it takes evaluation steps to reach t_1'.

Capability effects are interpreted by the $\mathcal{U}[\![e]\!]^{\rho}_{\delta}$ relation. For an effect variable α, the interpretation $\mathcal{U}[\![\alpha]\!]^{\rho}_{\delta}$ is simply the relation mapped to by the environment δ. For a capability label ℓ, the interpretation $\mathcal{U}[\![\ell]\!]^{\rho}_{\delta}$ relates two effect operation invocations: $\Uparrow H_1^{\mathsf{L}^1}\ v_1$ and $\Uparrow H_2^{\mathsf{L}^2}\ v_2$ are related provided the handlers $H_1^{\mathsf{L}^1}$

and $H_2^{\mathsf{L}\cdot 2}$ are related and the arguments v_1 and v_2 are related. The outcome relation ψ in this case is the value relation at the return type of the effect operation. The handler computations must be related at $\mathcal{T}[\![\ell]\!]^\rho_\delta$; this relation is defined at the bottom of Figure 3.19. For a label of form h.**lbl**, this relation is the one that ρ maps h to, while for a label of form L, this relation is $\mathcal{T}[\![[T]_{\overline{e}}]\!]^\rho_\delta$, provided the label environment Ξ maps L to $[T]_{\overline{e}}$.

The interpretation of a sequence of effects $\overline{e}$ is naturally the union of the interpretation of the individual effects in the sequence.

3.4.5 Properties of the Logical Relations

Basic properties. We point out some basic properties of the logical relations. These properties are employed by the proof leading to the soundness theorem and are used frequently in proofs of logical relatedness.

The following lemma applies when the goal is to prove the relatedness of two terms in which the subterms in the evaluation contexts are related:

Lemma 2. Given world W and evaluation contexts K_1 and K_2, if

(a) for any future world W' and for any v_1 and v_2, $\mathcal{V}[\![T]\!]^\rho_\delta(W', v_1, v_2)$ implies $\mathcal{T}[\![[T']_{\overline{e'}}]\!]^\rho_\delta(W', K_1[v_1], K_2[v_2])$, and

(b) for any future world W' and for any s_1 and s_2, $\mathcal{S}[\![[T]_{\overline{e}}]\!]^\rho_\delta(W', s_1, s_2)$ implies $\mathcal{T}[\![[T']_{\overline{e'}}]\!]^\rho_\delta(W', K_1[s_1], K_2[s_2])$,

then for any future world W' and for any t_1 and t_2, $\mathcal{T}[\![[T]_{\overline{e}}]\!]^\rho_\delta(W', t_1, t_2)$ implies $\mathcal{T}[\![[T']_{\overline{e'}}]\!]^\rho_\delta(W', K_1[t_1], K_2[t_2])$.

The lemma says it suffices to show the evaluation contexts K_1 and K_2 satisfy the following conditions: applying K_1 and K_2 to (a) related values and (b) related terms in the $\mathcal{S}[\![[T]_{\overline{e}}]\!]^\rho_\delta$ relation yields related terms in the $\mathcal{T}[\![[T']_{\overline{e'}}]\!]^\rho_\delta$ rela-

tion. We capture the preconditions of Lemma 2 by defining a logical relation $\mathcal{K}[\![[T]_{\overline{e}} \rightsquigarrow [T']_{\overline{e'}}]\!]^{\rho}_{\delta}$: two evaluation contexts K_1 and K_2 are in this relation under world W precisely when they satisfy the preconditions (a) and (b) of Lemma 2.

The following two lemmas show that reduction on either side reflects the term relation:

Lemma 3.

$$W_2 = W_2' \ \wedge \ W_1 \,;\, t_1 \longrightarrow W_1' \,;\, t_1' \ \wedge \ \triangleright\, \mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}\left(W',\, t_1',\, t_2\right) \Rightarrow \mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}(W,\, t_1,\, t_2).$$

Lemma 4.

$$W_1 = W_1' \ \wedge \ W_2 \,;\, t_2 \longrightarrow W_2' \,;\, t_2' \ \wedge \ \mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}\left(W',\, t_1,\, t_2'\right) \Rightarrow \mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}(W,\, t_1,\, t_2).$$

The asymmetry with respect to the use of the $\triangleright$ modality in the preconditions is a result of the asymmetry in the definition of the $\mathcal{O}$ relation.

The following lemma allows proving two terms related by showing that they are in the $\mathcal{V}$ relation or in the $\mathcal{S}$ relation:

Lemma 5. $\mathcal{V}[\![T]\!]^{\rho}_{\delta} \subseteq \mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta} \ \wedge \mathcal{S}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta} \subseteq \mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}$

These basic properties (Lemmas 2 to 5) are a consequence of the biorthogonal, step-indexed term relation defined in Section 3.4.3.

In addition, it can be derived that logical relations including $\mathcal{R}$, $\mathcal{V}$, $\mathcal{H}$, and $\mathcal{K}$ are monotone with respect to world extension, precisely because their definitions are quantified over future worlds. In particular, values related in one world remain related in a future world:

Lemma 6. $W \sqsubseteq W' \ \Rightarrow \ \mathcal{V}[\![T]\!]^{\rho}_{\delta}(W,\, v_1,\, v_2) \ \Rightarrow \ \mathcal{V}[\![T]\!]^{\rho}_{\delta}(W',\, v_1,\, v_2)$

Soundness. Contextual refinement is defined for open terms, so we lift the term relation and the handler relation to open terms and open handlers by quantifying over related closing substitutions for the variable environments, as

$$
\begin{aligned}
\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\overline{e}} \quad &\stackrel{def}{=} \quad \forall W, \delta, \rho, \gamma.\ \mathit{domain}(\Xi) \subseteq W_{i\ (i=1,2)} \Rightarrow \\
&\quad \forall \delta.\ [\![\Delta]\!]\,(W,\ \delta) \Rightarrow \forall \rho.\ [\![\mathrm{P}]\!]\,(W,\ \rho) \Rightarrow \\
&\quad \forall \gamma.\ [\![\Gamma]\!]^{\rho}_{\delta}\,(W,\ \gamma) \Rightarrow \\
&\quad \mathcal{T}[\![[T]_{\overline{e}}]\!]^{\rho}_{\delta}\,(W,\ \delta_1\rho_1\gamma_1 t_1,\ \delta_2\rho_2\gamma_2 t_2) \\
\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h_1 \preccurlyeq_{log} h_2 : \mathbb{F} \mid \ell \quad &\stackrel{def}{=} \quad \forall W, \delta, \rho, \gamma.\ \mathit{domain}(\Xi) \subseteq W_{i\ (i=1,2)} \Rightarrow \\
&\quad \forall \delta.\ [\![\Delta]\!]\,(W,\ \delta) \Rightarrow \forall \rho.\ [\![\mathrm{P}]\!]\,(W,\ \rho) \Rightarrow \\
&\quad \forall \gamma.\ [\![\Gamma]\!]^{\rho}_{\delta}\,(W,\ \gamma) \Rightarrow \\
&\quad \forall H_1, \llcorner_1, H_2, \llcorner_2.\ \delta_i\rho_i\gamma_i h_i = H_i^{\llcorner_i}{}_{\ (i=1,2)} \Rightarrow \\
&\quad \rho_i\ell = \llcorner_{i\ (i=1,2)} \ \wedge\ \mathcal{H}[\![\mathbb{F}]\!]\left(W,\ H_1,\ H_2,\ \mathcal{T}[\![\ell]\!]^{\rho}_{\delta}\right)
\end{aligned}
$$

$$
\begin{aligned}
[\![\varnothing]\!]\,(W,\ \delta) &\stackrel{def}{=} \delta = \varnothing \\
[\![\Delta, \alpha]\!]\,(W,\ \delta) &\stackrel{def}{=} \delta = \delta', \alpha \mapsto \left\langle \overline{\llcorner_1}, \overline{\llcorner_2}, \phi \right\rangle \ \wedge\ \phi \subseteq W \ \wedge\ [\![\Delta]\!]\,(W,\ \delta') \\
[\![\varnothing]\!]\,(W,\ \rho) &\stackrel{def}{=} \rho = \varnothing \\
[\![\mathrm{P}, \mathsf{h} : \mathbb{F}]\!]\,(W,\ \rho) &\stackrel{def}{=} \rho = \rho', \mathsf{h} \mapsto \left\langle H_1^{\llcorner_1}, H_2^{\llcorner_2}, \eta \right\rangle \ \wedge\ \llcorner_i \in W_{\ (i=1,2)} \ \wedge \\
&\qquad \mathcal{H}[\![\mathbb{F}]\!]\,(W,\ H_1,\ H_2,\ \eta) \ \wedge\ [\![\mathrm{P}]\!]\,(W,\ \rho') \\
[\![\varnothing]\!]^{\rho}_{\delta}\,(W,\ \gamma) &\stackrel{def}{=} \gamma = \varnothing \\
[\![\Gamma, \mathsf{x} : T]\!]^{\rho}_{\delta}\,(W,\ \gamma) &\stackrel{def}{=} \gamma = \gamma', \mathsf{x} \mapsto \langle v_1, v_2 \rangle \ \wedge\ \mathcal{V}[\![T]\!]^{\rho}_{\delta}\,(W,\ v_1,\ v_2) \ \wedge\ [\![\Gamma]\!]^{\rho}_{\delta}\,(W,\ \gamma')
\end{aligned}
$$

Figure 3.20. Logical relations for open terms and handlers

shown in Figure 3.20. Here, γ provides substitution functions for term variables: $\gamma ::= \varnothing \mid \gamma,\ \mathsf{x} \mapsto \langle v_1, v_2 \rangle$. The interpretation of variable environments as relations on substitutions, also given in Figure 3.20, is standard.

Central to the proof of soundness are the compatibility lemmas; they show that logical refinement $\preccurlyeq_{log}$ is preserved by the syntactic typing rules. The compatibility lemmas corresponding to the typing rules in Figures 3.12 and 3.13 are stated in Figures 3.21 and 3.22, respectively. The lemmas are written in the style of inference rules so that they can be read in tandem with the corresponding

$$\boxed{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\overline{e}}}$$

$$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T_1]_{\overline{e_1}} \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T_1 \leq T_2 \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e_1} \leq \overline{e_2}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T_2]_{\overline{e_2}}}$$

$$\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash () \preccurlyeq_{log} () : [\mathbb{1}]_{\varnothing} \qquad \frac{\Gamma(\mathsf{x}) = T}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \mathsf{x} \preccurlyeq_{log} \mathsf{x} : [T]_{\varnothing}}$$

$$\frac{\Delta \mid \mathrm{P} \mid \Xi \vdash S \qquad \Delta \mid \mathrm{P} \mid \Gamma, \mathsf{x}{:}S \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [S]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \lambda \mathsf{x}{:}S\,.\,t_1 \preccurlyeq_{log} \lambda \mathsf{x}{:}S\,.\,t_2 : [S \to [T]_{\overline{e}}]_{\varnothing}}$$

$$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [S \to [T]_{\overline{e}}]_{\overline{e}} \qquad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash s_1 \preccurlyeq_{log} s_2 : [S]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1\ s_1 \preccurlyeq_{log} t_2\ s_2 : [T]_{\overline{e}}}$$

$$\frac{\Delta \mid \mathrm{P} \mid \Xi \vdash S \qquad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash s_1 \preccurlyeq_{log} s_2 : [S]_{\overline{e}} \qquad \Delta \mid \mathrm{P} \mid \Gamma, \mathsf{x}{:}S \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \mathbf{let}\ \mathsf{x}{:}S = s_1\ \mathbf{in}\ t_1 \preccurlyeq_{log} \mathbf{let}\ \mathsf{x}{:}S = s_2\ \mathbf{in}\ t_2 : [T]_{\overline{e}}}$$

$$\frac{\Delta, \alpha \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\varnothing}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \Lambda\alpha.\, t_1 \preccurlyeq_{log} \Lambda\alpha.\, t_2 : [\forall\alpha.\, T]_{\varnothing}}$$

$$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [\forall\alpha.\, T]_{\overline{e'}} \qquad (\forall i)\ \Delta \mid \mathrm{P} \mid \Xi \vdash e^{(i)}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1\ [\overline{e}] \preccurlyeq_{log} t_2\ [\overline{e}] : [T\ \{\overline{e}/\alpha\}]_{\overline{e'}}}$$

$$\frac{\Delta \mid \mathrm{P}, \mathsf{h}{:}\mathbb{F} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \lambda \mathsf{h}{:}\mathbb{F}\,.\,t_1 \preccurlyeq_{log} \lambda \mathsf{h}{:}\mathbb{F}\,.\,t_2 : [\Pi_{\mathsf{h}:\mathbb{F}}\, [T]_{\overline{e}}]_{\varnothing}}$$

$$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : \left[\Pi_{\mathsf{h}:\mathbb{F}}\, [T]_{\overline{e'}}\right]_{\overline{e''}} \qquad \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h_1 \preccurlyeq_{log} h_2 : \mathbb{F} \mid \ell}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1\ [h_1] \preccurlyeq_{log} t_2\ [h_2] : [T\ \{\ell/\mathsf{h}.\mathbf{lbl}\}]_{\overline{e'}\{\ell/\mathsf{h}.\mathbf{lbl}\},\, \overline{e''}}}$$

$$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h_1 \preccurlyeq_{log} h_2 : \mathbb{F} \mid \ell \qquad op(\mathbb{F}) = T \to S}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \Uparrow h_1 \preccurlyeq_{log} \Uparrow h_2 : [T \to [S]_{\ell}]_{\varnothing}}$$

$$\frac{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi, \mathsf{L}{:}[T]_{\overline{e}} \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\overline{e}, \mathsf{L}} \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash T \qquad \Delta \mid \mathrm{P} \mid \Xi \vdash \overline{e}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \Downarrow^{\mathsf{L}}_{[T]_{\overline{e}}} t_1 \preccurlyeq_{log} \Downarrow^{\mathsf{L}}_{[T]_{\overline{e}}} t_2 : [T]_{\overline{e}}}$$

Figure 3.21. Term compatibility lemmas

$$\boxed{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h_1 \preccurlyeq_{log} h_2 : \mathbb{F} \mid \ell}$$

$$\frac{\mathrm{P}(\mathsf{h}) = \mathbb{F}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \mathsf{h} \preccurlyeq_{log} \mathsf{h} : \mathbb{F} \mid \mathsf{h}.\mathbf{lbl}}$$

$$\frac{\Xi(\mathsf{L}) = [S]_{\overline{e}} \qquad op(\mathbb{F}) = T_1 \to T_2 \qquad \Delta \mid \mathrm{P} \mid \Gamma,\ \mathsf{x}{:}T_1,\ \mathsf{k}{:}T_2 \to [S]_{\overline{e}} \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [S]_{\overline{e}}}{\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash \left(\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\ t_1\right)^{\mathsf{L}} \preccurlyeq_{log} \left(\mathbf{handler}^{\mathbb{F}}\ \mathsf{x}\ \mathsf{k}.\ t_2\right)^{\mathsf{L}} : \mathbb{F} \mid \mathsf{L}}$$

Figure 3.22. Handler compatibility lemmas

typing rules. Parametricity, and the fact that well-formed program contexts preserve logical refinement, are direct consequences of the compatibility lemmas:

Theorem 2 (PARAMETRICITY, A.K.A., FUNDAMENTAL PROPERTY, A.K.A., ABSTRACTION THEOREM)**.**

1. $\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t : [T]_{\overline{e}} \Rightarrow \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t \preccurlyeq_{log} t : [T]_{\overline{e}}$
2. $\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h : \mathbb{F} \mid \ell \Rightarrow \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash h \preccurlyeq_{log} h : \mathbb{F} \mid \ell$

Lemma 7 (CONGRUENCY)**.**

$\vdash C : \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \mid [T]_{\overline{e}} \rightsquigarrow T' \ \wedge\ \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\overline{e}} \Rightarrow$

$\varnothing \mid \varnothing \mid \varnothing \mid \varnothing \vdash C[t_1] \preccurlyeq_{log} C[t_2] : [T']_{\varnothing}$

One last step leading to the soundness theorem is to show the logical relation is adequate—two logically related pure terms are observationally related:

Lemma 8 (ADEQUACY)**.**

$\varnothing \mid \varnothing \mid \varnothing \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\varnothing} \Rightarrow domain(\Xi) \subseteq W \Rightarrow O\,(W,\ t_1,\ t_2)$

Type safety, the property that well-typed programs can only evaluate to values or diverge, falls out as an easy corollary of ADEQUACY and PARAMETRICITY, as the O relation only relates terms whose evaluations do not get stuck.

Theorem 3 (TYPE SAFETY)**.**
If $\varnothing \mid \varnothing \mid \varnothing \mid \varnothing \vdash t : [T]_{\varnothing}$ and $\varnothing\,;\, t \longrightarrow^* \overline{\mathsf{L}'}\,;\, t'$, then either there exists v such that $t' = v$ or there exists t'' and $\overline{\mathsf{L}''}$ such that $\overline{\mathsf{L}'}\,;\, t' \longrightarrow \overline{\mathsf{L}''}\,;\, t''$.

The key theorem that logical refinement implies contextual refinement—and therefore logical equivalence implies contextual equivalence—is a result of ADEQUACY and CONGRUENCY:

Theorem 4 (SOUNDNESS)**.**
$\Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{log} t_2 : [T]_{\bar{e}} \Rightarrow \Delta \mid \mathrm{P} \mid \Gamma \mid \Xi \vdash t_1 \preccurlyeq_{ctx} t_2 : [T]_{\bar{e}}$

Formalization in Coq. The definitions and results presented above have also been formalized using the Coq proof assistant [49]. The codebase contains about 7,000 lines of code for defining the language and proving syntactic and operational properties, and another 5,000 lines of code for defining the logical relations and proving properties about them. The logical relations are defined using the IxFree library [149], which is a shallow embedding of Dreyer et al.'s logic LSLR [58] in Coq. It also provides tactics for manipulating inference rules such as [LÖB] and [MONO], as well as a fixpoint operator for functions contractive in the use of the step index. We have extended IxFree to support dependently typed fixpoint functions in the logic; the formalization uses dependently typed de Bruijn indices to model environments of effect variables, handler variables, and term variables, by which the logical-relations definitions are indexed. Bindings of capability labels are modeled using cofinite quantification [14] to make it easy to generate fresh names.

3.5 Proving Example Equivalence

We demonstrate that the logical-relations model allows us to prove refinement and equivalence results that would not hold if algebraic effects were not tunneled. Beyond the usefulness of equivalence for programmer reasoning, such equivalence results could be used to justify the soundness of compiler transformations on effectful programs.

In particular, we show that clients of an effect-polymorphic abstraction cannot cause implementation details of the abstraction to leak out. We assume that $\lambda_{⇩⇧}$ has a second base type $\mathbb{N}$ with the operator $+$.

Let $\mathtt{f}$ be a variable with an effect-polymorphic type $T \stackrel{def}{=} \forall\alpha.\,(\mathbb{N} \to [\mathbb{N}]_\alpha) \to [\mathbb{N}]_\alpha$. Our goal is to prove the following two terms contextually equivalent:

$$
\begin{aligned}
t_1 &\stackrel{def}{=} \mathtt{f}\ [\varnothing]\ (\lambda \mathtt{x}{:}\mathbb{N}\,.\,\mathtt{x} + \mathtt{x}) \\
t_2 &\stackrel{def}{=} \textbf{let}\ \mathtt{g} : \Pi_{\mathtt{h}:\mathbb{F}}\ \mathbb{N} \to [\mathbb{N}]_{\mathtt{h}.\textbf{lbl}} = \lambda \mathtt{h}{:}\mathbb{F}\,.\,\lambda \mathtt{x}{:}\mathbb{N}\,.\,⇧\mathtt{h}\ \mathtt{x}\ \textbf{in} \\
&\quad\ ⇩^{\mathtt{L}}_{[\mathbb{N}]_\varnothing}\ (\lambda \mathtt{h}{:}\mathbb{F}\,.\,\mathtt{f}\ [\mathtt{h}.\textbf{lbl}]\ (\mathtt{g}\ \mathtt{h}))\ H^{\mathtt{L}}
\end{aligned}
$$

where $H \stackrel{def}{=} \textbf{handler}^{\mathbb{F}}\ \mathtt{x}\ \mathtt{k}.\,\mathtt{k}\ (\mathtt{x} + \mathtt{x})$ and $op(\mathbb{F}) = \mathbb{N} \to \mathbb{N}$. The second term t_2 corresponds to the following program written using the `try–with` construct, assuming the effect operation is named `twice`:

```
effect 𝔽 { twice(ℕ) : ℕ }

val g = fun(x : ℕ) : ℕ / 𝔽 { return twice(x) }
try { f(g) }
with twice(x) : ℕ { resume(x + x) }
```

Notice that this equivalence should apply to all possible implementations of $\mathtt{f}$, so even if the implementation handles $\mathbb{F}$ internally, the clients are unable to make different observations. As a result, equivalence results of this kind ensure the correctness of compiler transformations that optimize away uses of effects like that in t_2.

By the SOUNDNESS theorem, it suffices to show that t_1 and t_2 are logically equivalent. Below we show the logical refinement $\varnothing \mid \varnothing \mid \mathtt{f} : T \mid \varnothing \vdash t_1 \preccurlyeq_{log} t_2 : [\mathbb{N}]_\varnothing$ holds; the proof of the other direction is similar. By the definition of logical refinement ($\preccurlyeq_{log}$), we need to show for any world W and for any values f_1 and f_2 in the logical relation $\mathcal{V}[\![T]\!]^\varnothing_\varnothing$, the terms $t_1\,\{f_1/\mathtt{f}\}$ and $t_2\,\{f_2/\mathtt{f}\}$ are in the logical relation $\mathcal{T}[\![[\mathbb{N}]_\varnothing]\!]^\varnothing_\varnothing$. Notice that we can make reduction steps on $t_2\,\{f_2/\mathtt{f}\}$. So applying Lemma 4, our goal becomes

$$\mathcal{T}[\![[\mathbb{N}]_\varnothing]\!]^\varnothing_\varnothing \left(W',\ f_1\ [\varnothing]\ (\lambda \mathtt{x} : \mathbb{N}\,.\,\mathtt{x} + \mathtt{x}),\ \Downarrow^{\mathsf{L}} f_2\ [\mathsf{L}]\ \left(\lambda \mathtt{x} : \mathbb{N}\,.\,\Uparrow H^{\mathsf{L}}\ \mathtt{x}\right)\right), \tag{3.1}$$

where $W_1' = W_1$ and $W_2' = W_2, \mathsf{L}$. In fact, we can show a slightly different result:

$$\mathcal{T}[\![[\mathbb{N}]_\varnothing]\!]^\varnothing_\delta \left(W',\ f_1\ [\varnothing]\ (\lambda \mathtt{x} : \mathbb{N}\,.\,\mathtt{x} + \mathtt{x}),\ \Downarrow^{\mathsf{L}} f_2\ [\mathsf{L}]\ \left(\lambda \mathtt{x} : \mathbb{N}\,.\,\Uparrow H^{\mathsf{L}}\ \mathtt{x}\right)\right), \tag{3.2}$$

where δ contains the mapping $\alpha \mapsto \langle \varnothing, \mathsf{L}, \phi \rangle$ and ϕ is the interpretation specifically chosen for α in this example:

$$\phi \stackrel{def}{=} \left\{ \left\langle W'',\ (\lambda \mathtt{x} : \mathbb{N}\,.\,\mathtt{x} + \mathtt{x})\ n,\ \left(\lambda \mathtt{x} : \mathbb{N}\,.\,\Uparrow H^{\mathsf{L}}\ \mathtt{x}\right)\ n,\ \{\langle W'', 2n, 2n\rangle\},\ \varnothing,\ \mathsf{L} \right\rangle \;\middle|\; \begin{array}{l} n \in \mathbb{N} \wedge \\ W' \subseteq W'' \end{array} \right\}.$$

Having the result of (3.2), we can use a weakening lemma (omitted) to obtain goal (3.1). Proper effect polymorphism allows us to interpret α in arbitrary ways, but as we shall see, this particular choice of ϕ allows us to establish logical relatedness. To obtain (3.2), we apply Lemma 2 with evaluation contexts $[\cdot]$ and $\Downarrow^{\mathsf{L}} [\cdot]$:

- We want to show $\mathcal{K}[\![[\mathbb{N}]_\alpha \rightsquigarrow [\mathbb{N}]_\varnothing]\!]^\varnothing_\delta \left(W', [\cdot], \Downarrow^{\mathsf{L}} [\cdot]\right)$. We apply the [LÖB] rule from Section 3.4.1: to prove this goal, we are allowed to assume

  $$\triangleright\, \mathcal{K}[\![[\mathbb{N}]_\alpha \rightsquigarrow [\mathbb{N}]_\varnothing]\!]^\varnothing_\delta \left(W', [\cdot], \Downarrow^{\mathsf{L}} [\cdot]\right). \tag{3.3}$$

  Unfolding the definition of $\mathcal{K}$ generates the following two goals:

(a) We want to show for any v_1 and v_2 related by $\mathcal{V}[\![\mathbb{N}]\!]^{\varnothing}_{\delta}$ in a future world W'' ($W' \subseteq W''$), the terms v_1 and $\Downarrow^{\mathsf{L}} v_2$ are related by $\mathcal{T}[\![[\mathbb{N}]_{\varnothing}]\!]^{\varnothing}_{\delta}$ in world W''. This is immediate, because the right-hand side evaluates to v_2 and the value relation is included in the term relation (Lemma 5).

(b) We want to show for any $K_1[s_1]$ and $K_2[s_2]$ related by $\mathcal{S}[\![[\mathbb{N}]_{\alpha}]\!]^{\varnothing}_{\delta}$ in a future world W'' ($W' \subseteq W''$), the terms $K_1[s_1]$ and $\Downarrow^{\mathsf{L}} K_2[s_2]$ are related by $\mathcal{T}[\![[\mathbb{N}]_{\varnothing}]\!]^{\varnothing}_{\delta}$ in world W''. Unfolding the definition of $\mathcal{S}$, we know there exists an outcome relation ψ such that

(i) $\mathcal{U}[\![\alpha]\!]^{\varnothing}_{\delta}\left(W'', s_1, s_2, \psi, \overline{\mathsf{L}_1}, \overline{\mathsf{L}_2}\right)$,

(ii) $\forall i.\ \mathsf{L}_1^{(i)} \curvearrowright K_1$ and $\forall i.\ \mathsf{L}_2^{(i)} \curvearrowright K_2$, and

(iii) $\forall W''', s_1', s_2'.\ W'' \subseteq W''' \Rightarrow \psi\left(W''', s_1', s_2'\right) \Rightarrow$
$\triangleright\, \mathcal{T}[\![[\mathbb{N}]_{\alpha}]\!]^{\varnothing}_{\delta}\left(W''', K_1\left[s_1'\right], K_2\left[s_2'\right]\right)$.

Because the interpretation of effect variable α is prescribed by the semantic substitution δ, and because $\delta(\alpha)$ is chosen as ϕ, we know s_1, s_2, ψ, $\overline{\mathsf{L}_1}$, and $\overline{\mathsf{L}_2}$ are precisely the terms, relation, and labels in ϕ. Thus we need to show

$$\mathcal{T}[\![[\mathbb{N}]_{\varnothing}]\!]^{\varnothing}_{\delta}\left(W'', K_1[(\lambda \mathsf{x}:\mathbb{N}\,.\,\mathsf{x}+\mathsf{x})\ n], \Downarrow^{\mathsf{L}} K_2\left[\left(\lambda \mathsf{x}:\mathbb{N}\,.\,\Uparrow H^{\mathsf{L}}\ \mathsf{x}\right)\ \ n\right]\right).$$

Making evaluation steps on both sides, the goal becomes

$$\triangleright\, \mathcal{T}[\![[\mathbb{N}]_{\varnothing}]\!]^{\varnothing}_{\delta}\left(W'', K_1[2n], \Downarrow^{\mathsf{L}} K_2[2n]\right).$$

This new goal is guarded by the $\triangleright$ modality because evaluation occurred in the first computation. The new proof context is as follows, where the first assumption is the Löb induction hypothesis (3.3):

$$\frac{\triangleright\, \mathcal{K}[\![[\mathbb{N}]_{\alpha} \rightsquigarrow [\mathbb{N}]_{\varnothing}]\!]^{\varnothing}_{\delta}\left(W', [\cdot], \Downarrow^{\mathsf{L}} [\cdot]\right) \qquad \forall W''', s_1', s_2'.\ W'' \subseteq W''' \Rightarrow \psi\left(W''', s_1', s_2'\right) \Rightarrow \triangleright\, \mathcal{T}[\![[\mathbb{N}]_{\alpha}]\!]^{\varnothing}_{\delta}\left(W''', K_1\left[s_1'\right], K_2\left[s_2'\right]\right)}{\triangleright\, \mathcal{T}[\![[\mathbb{N}]_{\varnothing}]\!]^{\varnothing}_{\delta}\left(W'', K_1[2n], \Downarrow^{\mathsf{L}} K_2[2n]\right)}$$

We already have $\psi\,(W'',\,2n,\,2n)$, so by the second assumption in the proof context, we have $\triangleright\,\mathcal{T}[\![[\mathbb{N}]_\alpha]\!]^{\varnothing}_{\delta}\,(W'',\,K_1[2n],\,K_2[2n])$. Now we can apply rule [MONO] from Section 3.4.1: the presence of the $\triangleright$ modality in the goal cancels out the occurrences of $\triangleright$ in the assumptions. The new goal then follows from the definition of $\mathcal{K}[\![[\mathbb{N}]_\alpha \rightsquigarrow [\mathbb{N}]_\varnothing]\!]^{\varnothing}_{\delta}$.

- We are left to show $\mathcal{T}[\![[\mathbb{N}]_\alpha]\!]^{\varnothing}_{\delta}\,\left(W',\ f_1\ [\varnothing]\ (\lambda x{:}\mathbb{N}\,.\,x+x),\ f_2\ [\mathsf{L}]\ \left(\lambda x{:}\mathbb{N}\,.\,\Uparrow H^{\mathsf{L}}\ x\right)\right)$. Because $\mathcal{V}[\![\forall\alpha.\,(\mathbb{N}\to[\mathbb{N}]_\alpha)\to[\mathbb{N}]_\alpha]\!]^{\varnothing}_{\varnothing}\,(W',\ f_1,\ f_2)$, we have that the terms $f_1\ [\varnothing]$ and $f_2\ [\mathsf{L}]$ are related by $\mathcal{T}\left[\!\left[(\mathbb{N}\to[\mathbb{N}]_\alpha)\to[\mathbb{N}]_\alpha\right]_\varnothing\right]\!\right]^{\varnothing}_{\delta}$ in world W'. It then suffices to show that the values which $f_1\ [\varnothing]$ and $f_2\ [\mathsf{L}]$ are respectively applied to are related in any future world W'' ($W' \subseteq W''$):

$$\mathcal{V}[\![\mathbb{N}\to[\mathbb{N}]_\alpha]\!]^{\varnothing}_{\delta}\,\left(W'',\ \lambda x{:}\mathbb{N}\,.\,x+x,\ \lambda x{:}\mathbb{N}\,.\,\Uparrow H^{\mathsf{L}}\ x\right),$$

meaning that applications of these two lambda abstractions to the same natural number are related by $\mathcal{T}[\![[\mathbb{N}]_\alpha]\!]^{\varnothing}_{\delta}$ in world W''. By Lemma 5, we show the applications are actually in the smaller $\mathcal{S}[\![[\mathbb{N}]_\alpha]\!]^{\varnothing}_{\delta}$ relation:

$$\mathcal{S}[\![[\mathbb{N}]_\alpha]\!]^{\varnothing}_{\delta}\,\left(W'',\ (\lambda x{:}\mathbb{N}\,.\,x+x)\ n,\ \left(\lambda x{:}\mathbb{N}\,.\,\Uparrow H^{\mathsf{L}}\ x\right)\ n\right).$$

With the evaluation contexts being $[\cdot]$, the following conditions are straightforward to show for $\psi \stackrel{def}{=} \{\langle W'', 2n, 2n\rangle\}$:

(i) $\mathcal{U}[\![\alpha]\!]^{\varnothing}_{\delta}\,\left(W'',\ (\lambda x{:}\mathbb{N}\,.\,x+x)\ n,\ \left(\lambda x{:}\mathbb{N}\,.\,\Uparrow H^{\mathsf{L}}\ x\right)\ n,\ \psi,\ \varnothing,\ \mathsf{L}\right)$,

(ii) $\mathsf{L} \curvearrowright [\cdot]$, and

(iii) $\forall W''', s_1', s_2'.\ W'' \subseteq W''' \Rightarrow \psi\left(W''',\ s_1',\ s_2'\right) \Rightarrow \triangleright\,\mathcal{T}[\![[\mathbb{N}]_\alpha]\!]^{\varnothing}_{\delta}\left(W''',\ s_1',\ s_2'\right)$.

3.6 Related Work

Previous work proposes ways to make algebraic effects composable. Leijen [102] suggests using an `inject` function to prevent client code from meddling with the

effect-handling internals of library functions. Applying `inject` to a computation causes effects raised from that computation to bypass the innermost handler enclosing it. Biernacki et al. [18] propose a "lift" operator that works in a similar fashion: computations surrounded by a lift operator $[\cdot]_{\mathbb{F}}$ bypass the innermost effect handler for $\mathbb{F}$. The programmer can use `inject` or lift to prevent effects of a client-provided function from being intercepted by the effect-polymorphic, higher-order function that applies it. Both of these type systems use effect rows and row polymorphism, and distinguish different occurrences of the same effect name in a row.

The very use of effect rows in these approaches does not seem to be without limitations. In particular, it poses challenges to composing polymorphic effects. For example, because α, β is not a legal effect row, this effect-polymorphic higher-order function type does not seem to be expressible using effect rows: $\forall\alpha.\,\forall\beta.\,((T_1 \to [T_2]_\alpha) \to [T_3]_\beta) \to (T_1 \to [T_2]_\alpha) \to [T_3]_{\alpha,\beta}$.

Biernacki et al. [18] show that effect polymorphism in a core language equipped with the lift operator satisfies parametricity; we borrow useful techniques from their logical-relations definition. The type system of Biernacki et al. poses restrictions on "subeffecting" (cf. subtyping): it rejects—by fiat—an effect variable α as a subeffect of $\mathbb{F}, \alpha$. The absence of accidental handling hinges upon this restriction: the programmer must thread lift operators through effect-polymorphic code to please the type checker. For example, function `fiterate` from Section 3.1 would not type-check in their system because the effect of `f(x)` (i.e., effect variable `E`) is not a subeffect of `Yield[X], E`. The programmer would have to choose between (a) declaring variable `f` with type `X → bool /Yield[X], E`, and (b) surrounding `f(x)` with a lift operator. In contrast, because it rests on the intuitive principle

that code should only handle effects it is locally aware of, tunneling requires no essential changes to effect-polymorphic code.

Zhang et al. [193] propose an alternate semantics for exceptions in their Genus language, in which exceptions are tunneled through contexts that are not statically aware of them. While we build on this insight, this prior work is limited to exceptions rather than more general algebraic effects, and importantly, the mechanism is not shown formally to be abstraction-safe. The kind of exception polymorphism it supports is also more limited: functions are polymorphic in the latent exceptions of only those types that are annotated `weak`. It is argued that trading `weak` annotations for explicit effect variables reduces annotation burden. However, this approach makes it cumbersome, if not impossible, to define exception-polymorphic *data structures*, such as the `cachingFun` class in Section 3.2.4. The `weak` annotations are essentially a mechanism for region-capability effects: values of weak types have a stack discipline and thus can only be used in a second-class way, but data structures require a finer-grained notion of region capability.

Functional programming languages like ML and Haskell do not statically check that exceptions are handled, so we do not consider them fully type-safe. Interestingly, accidental handling can be avoided in SML, because SML exception types are generative [121] and because a handler can only handle lexically visible exception types. However, the type system does not ensure that accidental handling is avoided or that exceptions are handled at all. Bračevac et al. [26] observe the need to disambiguate handlers for invocations of the same algebraic effect operation. Compared with their proposed solution of generative effect signatures, tunneling addresses the issue straightforwardly: handlers can be specified explicitly for each invocation of the effect operation.

Brachthäuser and Schuster [24] encode algebraic effect handlers as a Scala library named Effekt. Like our use of handler polymorphism, the encoding passes handlers down to the place where effect operations are invoked, using Scala's *implicits* feature [137] and in particular, implicit function types [136], to resolve implicit arguments as handler objects in scope. Clients of Effekt do not have to worry about accidental handling, but this approach does not guarantee the absence of run-time errors. In addition to the handling code, a stack-marking *prompt* must be passed down too, so that when the effect operation is invoked, the continuation up to the prompt is captured and passed to the handling code. But there is no static checking that the prompt obeys the stack discipline—type-safety relies on client code using the library in a disciplined way.

It is hypothesized that this safety issue could be remedied by using the `@local` annotation provided in a Scala extension [141]. Parameters of functions and local variables can be annotated `@local`, making them second-class. In contrast to the Genus `weak` annotation [193], `@local` is applied to uses of types (instead of definitions of types), so it seems no lighter-weight than explicit effect variables. Like the `weak` annotations, `@local` cannot offer the fine-grained notion of region capability needed to express effect-polymorphic data structures.

Our use of capability effects to ensure soundness is adapted from work on region-based memory management [51, 86, 171]. A capability is a set of live memory regions. To prevent accesses to deallocated memory regions, computations are typed with capability effects that specify the set of regions they might access. We apply this idea to ensure continuations of handling code are accessible. Our type system is simpler than a full-fledged region type system because safety concerns only lexical regions delimited by effect handlers.

The problem of accidentally handled effects generalizes the problem of variable capture in early programming languages (e.g., Lisp) that supported dynamically scoped variables. Dynamically scoped variables do not have to be dynamically typed; Lewis et al. [105] provide a type system for them, treating them as implicit parameters. To avoid variable capture, Lewis et al. ban the use of implicitly parameterized functions as first-class values, losing the extensibility that makes dynamically scoped variables attractive. Tunneled algebraic effects offer abstraction-safe dynamically scoped variables without sacrificing their expressive power.

Kammar et al. [99] distinguish between deep and shallow semantics for handlers. A shallow handler is discarded after it is first invoked, while a deep handler can continue to handle the rest of the computation it envelops. Handlers for tunneled algebraic effects are deep. Shallow handlers pose challenges to modular reasoning, because it is difficult to reason statically about how effects raised from the rest of the computation are handled.

The effect constructs in our core language are essentially a pair of delimited control operators [53, 69]. With delimited control, one operator C (cf. $\Uparrow$ in $\lambda_{\downarrow\Uparrow}$) captures the continuation delimited by a corresponding operator of the other kind $\mathcal{D}$ (cf. $\Downarrow$ in $\lambda_{\downarrow\Uparrow}$). Among the variety of previous delimited control operators, ours are closest to those with named prompts [60, 88]. Rather than pairing a C operation with the *dynamically* closest enclosing $\mathcal{D}$, these mechanisms allow uses of $\mathcal{D}$ to be named and consequently referenced by invocations of C, enabling static reasoning. Although embedded in statically typed languages, the earlier mechanisms do not guarantee type safety—a C operation can go unhandled.

CHAPTER 4

GENUS: LIGHTWEIGHT, FLEXIBLE OBJECT-ORIENTED GENERICS

Generic programming provides the means to express algorithms and data structures in an abstract, adaptable, and interoperable form. Specifically, genericity mechanisms allow polymorphic code to apply to different types, improving modularity and reuse. Despite decades of work on genericity mechanisms, current OO languages still offer an unsatisfactory tradeoff between expressiveness and usability. These languages do not provide a design that coherently integrates desirable features—particularly, retroactive extension and dynamic dispatch. In practice, existing genericity mechanisms force developers to circumvent limitations in expressivity by using awkward, heavyweight design patterns and idioms.

The key question is how to expose the operations of type parameters in a type-safe, intuitive, and flexible manner within the OO paradigm. The following somewhat daunting Java signature for method `Collections::sort` illustrates the problem:

```
<T extends Comparable<? super T>> void sort(List<T> l)
```

The subtyping constraint constrains a type parameter `T` using the `Comparable` interface, ensuring that type `T` is comparable to itself or to one of its supertypes. However, `sort` can only be used on a type `T` if that type argument is explicitly declared to implement the `Comparable` interface. This restriction of nominal subtyping is alleviated by structural constraints as introduced by CLU [108, 110] and applied elsewhere (e.g., [43, 54]), but a more fundamental limitation remains: items of type `T` cannot be sorted unless `T` has a `compareTo` operation to define the sort order. That limitation is addressed by *type classes* in Haskell [176].

Inspired by Haskell, efforts have been made to incorporate type classes into OO languages with language-level support [154, 160, 169, 182] and the Concept design pattern [137]. However, as we argue, these designs do not fully exploit what type classes and OO languages have to offer when united.

This chapter introduces a new genericity mechanism, embodied in a new extension of Java called Genus. The genericity mechanism enhances expressive power, code reuse, and static type safety, while remaining lightweight and intuitive for the programmer in common use cases. Genus supports *models* as named constructs that can be defined and selected explicitly to witness constraints, even for primitive type arguments; however, in common uses of genericity, types implicitly witness constraints without additional programmer effort. The key novelty of models in Genus is their deep integration into the OO style, with features like model generics, model-dependent types, model enrichment, model multimethods, constraint entailment, model inheritance, and existential quantification further extending expressive power in an OO setting.

The chapter compares Genus to other language designs; describes its implementation; shows that Genus enables safer, more concise code through experiments that use it to reimplement existing generic libraries; and presents performance measurements that show that a naive translation from Genus to Java yields acceptable performance and that with simple optimizations, Genus can offer very good performance. A formal static semantics for a core version of

```
class AbstractVertex
  <EdgeType extends AbstractEdge<EdgeType, ActualVertexType>,
   ActualVertexType extends AbstractVertex<EdgeType, ActualVertexType>>
{ … }

class AbstractEdge
  <ActualEdgeType extends AbstractEdge<ActualEdgeType, VertexType>,
   VertexType extends AbstractVertex<ActualEdgeType, VertexType>>
{ … }
```

Figure 4.1. Parameter clutter in generic code

Genus is available in the technical report [191]; there we show that termination of default model resolution holds under reasonable syntactic restrictions.

4.1 The Need for Better Genericity

Prior work has explored various approaches to constrained genericity: subtyping constraints, structural matching, type classes, and design patterns. Each of these approaches has significant weaknesses.

The trouble with subtyping. Subtyping constraints are used in Java [22], C# [63, 100], and other OO languages. In the presence of nominal subtyping, subtyping constraints are too inflexible: they can only be satisfied by classes explicitly declared to implement the constraint. Structural subtyping and matching mechanisms (e.g., [43, 54, 110, 126]) do not require an explicit declaration that a constraint is satisfied, but still require that the relevant operations exist, with conformant signatures. Instead, we want retroactive modeling, in which a *model* (such as a type class instance [176]) can define how an existing type satisfies a constraint that it was not planned to satisfy ahead of time.

```
class TreeSet<T> implements Set<T> {
  TreeSet(Comparator<? super T> comparator) { … }
  …
}

interface Comparator<T> { int compare(T o1, T o2); }
```

Figure 4.2. Concept design pattern

Subtyping constraints, especially when F-bounded [38], also tend to lead to complex code when multiple type parameters are needed. For example, Figure 4.1 shows a simplification of the signatures of the classes `AbstractVertex` and `AbstractEdge` in the FindBugs project [71]. The vertex and the edge types of a graph have a mutual dependency that is reflected in the signatures in an unpleasantly complex way (see Figure 4.3 for our approach).

Concept design pattern. Presumably because of these limitations, the standard Java libraries mostly do *not* use constraints on the parameters of generic classes in the manner originally envisioned [22]. Instead, they use a version of the Concept design pattern [127] in which operations needed by parameter types are provided as arguments to constructors. For instance, a constructor of `TreeSet`, a class in the Java collections framework, accepts an object of the `Comparator` class (Figure 4.2). The `compare` operation is provided by this object rather than by `T` itself.

This design pattern provides missing flexibility, but adds new problems. First, a `comparator` object must be created even when the underlying type has a comparison operation. Second, because the model for `Comparator` is an ordinary (first-class) object, it is hard to specialize or optimize particular instantiations of generic code. Third, there is no static checking that two `TreeSets` use the same

ordering; if an algorithm relies on the element ordering in two `TreeSets` being the same, the programmer may be in for a shock.

In another variant of the design pattern, used in the C++ STL [125], an extra parameter for the class of the comparator distinguishes instantiations that use different models. However, this approach is more awkward than the `Comparator` object approach. Even the common case, in which the parameter type has exactly the needed operations, is just as heavyweight as when an arbitrary, different operation is substituted.

Type classes and concepts. The limitations of subtyping constraints have led to recent research on adapting type classes to OO languages to achieve retroactive modeling [160]. However, type classes have limitations: first, constraint satisfaction must be uniquely witnessed, and second, their models define how to adapt a single type, whereas in a language with subtyping, each adapted type in general represents all of its subtypes.

No existing approach addresses the first limitation, but an attempt is made by JavaGI [182] to fit subtyping polymorphism and dynamic dispatch into constrained genericity. As we will argue (Section 4.4.1), JavaGI's limited dynamic dispatch makes certain constraints hard to express, and interactions between subtyping and constraint handling make type checking subject to nontermination.

Beyond dynamic dispatch, it is important for OO programming that extensibility applies to models as well. The essence of OO programming is that new behavior can be added later in a modular way; we consider this post-factum *enrichment* of models to be a requirement.

Goals. What is wanted is a genericity mechanism with multiple features: retroactive modeling, a lightweight implicit approach for the common case,

multiparameter type constraints, non-unique constraint satisfaction with dynamic, extensible models, and model-dependent types. The mechanism should support modular compilation. It should be possible to implement the mechanism efficiently; in particular, an efficient implementation should limit the use of wrapper objects and should be able to specialize generic code to particular type arguments—especially, to primitive types. Genus meets all of these goals. We have tried not only to address the immediate problems with generics seen in current OO languages, but also to take further steps, adding features that support the style of programming that we expect will evolve when generics are easier to use than they are now.

4.2 Type Constraints in Genus

4.2.1 Type Constraints as Predicates

Instead of constraining types with subtyping, Genus uses explicit *type constraints* similar to type classes. For example, the constraint

```
constraint Eq[T] {
  boolean equals(T other);
}
```

requires that type `T` have an `equals` method.[1] Although this constraint looks like a Java interface, it is really a predicate on types, like a (multiparameter) type class in Haskell [98]. We do not call constraints "type classes" because there are differences and because the name "class" is already taken in the OO setting.

Generic code can require that actual type parameters satisfy constraints. For example, here is the `Set` interface in Genus (simplified):

[1] We denote Genus type parameters using square brackets, to distinguish Genus examples from those written in other languages (especially, Java).

```
constraint GraphLike[V,E] {
  Iterable[E] V.outgoingEdges();
  Iterable[E] V.incomingEdges();
  V E.source();
  V E.sink();
}
```

Figure 4.3. `GraphLike` is a multiparameter constraint

```
interface Set[T where Eq[T]]  { … }
```

The *where clause* "`where Eq[T]`" establishes the ability to test equality on type `T` within the scope of `Set`. Consequently, an instantiation of `Set` needs a witness that `Eq` is satisfied by the type argument. In Genus, such witnesses come in the form of *models*. Models are either implicitly chosen by the compiler or explicitly supplied by the programmer.

Multiparameter constraints. A constraint may be a predicate over multiple types. Figure 4.3 contains an example in which a constraint `GraphLike[V,E]` declares graph operations that should be satisfied by any pair of types `[V,E]` representing vertices and edges of a graph. In a multiparameter constraint, methods must explicitly declare receiver types (`V` or `E` in this case). Every operation in this constraint mentions both `V` and `E`; none of the operations really belongs to any single type. The ability to group related types and operations into a single constraint leads to code that is more modular and more readable than that in Figure 4.1.

Prerequisite constraints. A constraint can have other constraints as its *prerequisites*. For example, `Eq[T]` is a prerequisite constraint of `Comparable[T]`:

```
constraint OrdRing[T] extends Comparable[T] {
  static T T.zero();
  static T T.one();
  T T.plus(T that);
  T T.times(T that);
}
```

Figure 4.4. Constraint `OrdRing` contains static methods

```
constraint Comparable[T] extends Eq[T] {
  int compareTo(T other);
}
```

To satisfy a constraint, its prerequisite constraints must also be satisfied. Therefore, the satisfaction of a constraint *entails* the satisfaction of its prerequisites. For example, the Genus version of the `TreeSet` class from Figure 4.2 looks as follows:

```
class TreeSet[T where Comparable[T]] implements Set[T] { … }
```

The type `Set[T]` in the definition of `TreeSet` is well-formed because its constraint `Eq[T]` is entailed by the constraint `Comparable[T]`.

Static constraint members. Constraints can require that a type provide static methods, indicated by using the keyword `static` in the method declaration. In Figure 4.4, constraint `OrdRing` specifies a static method (`zero`) that returns the identity of the operation `plus`.

All types `T` are also automatically equipped with a static method `T.default()` that produces the default value for type `T`. This method is called, for instance, to initialize the elements of an array of type `T[]`, as in the following example:

```
Map[V,W] SSSP[V,E,W](V s)
where GraphLike[V,E], Weighted[E,W], OrdRing[W], Hashable[V] {
  TreeMap[W,V] frontier = new TreeMap[W,V]();
  Map[V,W] distances = new HashMap[V,W]();
  distances.put(s, W.one()); frontier.put(W.one(), s);
  while (frontier.size() > 0) {
    V v = frontier.pollFirstEntry().getValue();
    for (E vu : v.outgoingEdges()) {
      V u = vu.sink();
      W weight = distances.get(v).times(vu.weight());
      if (!distances.containsKey(u) ||
          weight.compareTo(distance.get(u)) < 0) {
        frontier.put(weight, u);
        distances.put(u, weight);
      }
    }
  }
  return distances;
}
```

Figure 4.5. A highly generic method for Dijkstra's single-source shortest-path algorithm. Definitions of `Weighted` and `Hashable` are omitted. Ordering and composition of distances are generalized to an ordered ring. (A more robust implementation might consider using a priority queue instead of `TreeMap`.)

```
class ArrayList[T] implements List[T] {
  T[] arr;
  ArrayList() { arr = new T[INITIAL_SIZE]; } // Calls T.default()
  ...
}
```

The ability to create an array of type `T[]` is often missed in Java.

4.2.2 Prescribing Constraints Using Where Clauses

Where-clause constraints enable generic algorithms, such as the version of Dijkstra's shortest-path algorithm in Figure 4.5, generalized to ordered rings. (The usual behavior is achieved if `plus` is min, `times` is +, and `one` is 0.) The where clause of `SSSP` requires only that the type arguments satisfy their respective constraints—no subtype relationship is needed.

Where-clause constraints endow typing contexts with assumptions that the constraints are satisfied. So the code of `SSSP` can make method calls like `vu.sink()` and `W.one()`. Note that the where clause may be placed after the formal parameters as in CLU; this notation is just syntactic sugar for placing it between the brackets.

Unlike Java `extends` clauses, a where clause is not attached to a particular parameter. It can include multiple constraints, separated by commas. Each constraint requires a corresponding model to be provided when the generic is instantiated. To allow models to be identified unambiguously in generic code, each such constraint in the where clause may be explicitly named as a *model variable*.

Another difference from Java `extends` clauses is that a where clause may be used without introducing a type parameter. For example, consider the `remove` method of `List`. Expressive power is gained if its caller can specify the notion of equality to be used, rather than requiring `List` itself to have an intrinsic notion of equality. Genus supports this genericity by allowing a constraint `Eq[E]` to be attached to `remove`:

```
interface List[E] {
  boolean remove(E e) where Eq[E];
  …
}
```

We call this feature *model genericity*.

4.2.3 Witnessing Constraints Using Models

As mentioned, generic instantiations require witnesses that their constraints are satisfied. In Genus, witnesses are provided by models. Models can be inferred—a process we call *default model resolution*—or specified explicitly, offering both convenience in common cases and expressivity when needed. We start with the *use* of models and leave the *definition* of models until Section 4.3.

Using default models. It is often clear from the context which models should be used to instantiate a generic. For instance, the `Set[T]` interface in the `TreeSet` example (Section 4.2.1) requires no further annotation to specify a model for `Eq[T]`, because the model can be uniquely resolved to the one promised by `Comparable[T]`.

Another common case is that the underlying type already has the required operations. This case is especially likely when classes are designed to support popular operations; having to supply models explicitly in this case would be a nuisance. Therefore, Genus allows types to *structurally conform* to constraints. When the methods of a type have the same names as the operations required by a constraint, and also have conformant signatures, the type automatically generates a *natural model* that witnesses the constraint. For example,[2] the type `Set[String]` means a `Set` that distinguishes strings using `String`'s built-in `equals` method. Thus, the common case in which types provide exactly the operations required by constraints is simple and intuitive. In turn, programmers have an incentive to standardize the names and signatures of popular operations.

[2]We assume throughout that the type `String` has methods "`boolean equals(String)`" and "`int compareTo(String)`."

Genus supports using primitive types as type arguments, and provides natural models for them that contain common methods. For example, a natural model for `Comparable[int]` exists, so types like `TreeSet[int]` that need that model can be used directly.

Default models can be used to instantiate any generic—not just generic classes. For example, consider this sort method:

```
void sort[T](List[T] l) where Comparable[T] { … }
```

The call `sort(x)`, where `x` is a `List[int]`, infers `int` both as the type argument and as the default model. Default model resolution, and more generally, type and model inference, are discussed further in Section 4.3.4 and Section 4.3.7.

Using named models. It is also possible to explicitly supply models to witness constraints. To do so, programmers use the `with` keyword followed by models for each of the where-clause constraints in the generic. These models can come from programmer-defined models (Section 4.3) or from model variables declared in where clauses (Section 4.2.2). For example, suppose model `CIEq` tests `String` equality in a case-insensitive manner. The type `Set[String with CIEq]` then describes a `Set` in which all strings are distinct without case-sensitivity. In fact, the type `Set[String]` is syntactic sugar for `Set[String with String]`, in which the `with` clause is used to explicitly specify the natural model that `String` automatically generates for `Eq[String]`.

A differentiating feature of our mechanism is that different models for `Eq[String]` can coexist in the same scope, allowing a generic class like `Set`, or a generic method, to be instantiated in more than one way in a scope:

```
Set[String] s0 = …;
Set[String with CIEq] s1 = …;
s1 = s0; // illegal assignment: different types.
```

The ordering that an instantiation of `Set` uses for its elements is *part of the type*, rather than a purely dynamic argument passed to a constructor as in the Concept pattern. Therefore, the final assignment statement is a static type error. The type checker catches the error because the different models used in the two `Set` instantiations allow `Sets` using different notions of equality to be distinguished. The use of models in types is discussed further in Section 4.3.5.

It is also possible to express types using *wildcard models*. For example, the type `Set[String with ?]` is a supertype of both `Set[String]` and `Set[String with CIEq]`. Wildcard models are actually syntactic sugar for existential quantification (Section 4.5).

4.3 Models

Models can be defined explicitly to allow a type to satisfy a constraint when the natural model is nonexistent or undesirable. For example, the case-insensitive string equality model `CIEq` can be defined concisely:

```
model CIEq for Eq[String] {
  bool equals(String str) { return equalsIgnoreCase(str); }
}
```

Furthermore, a model for case-insensitive `String` ordering might be defined by reusing `CIEq` via *model inheritance*, to witness the prerequisite constraint `Eq[String]`:

```
model CICmp for Comparable[String] extends CIEq {
  int compareTo(String str) { return compareToIgnoreCase(str); }
}
```

It is also possible for `CICmp` to satisfy `Eq` by defining its own `equals` method. Model inheritance is revisited in Section 4.4.3.

Models are immutable: they provide method implementations but do not have any instance variables. Models need not have global scope; modularity is achieved through the Java namespace mechanism. Similarly, models can be nested inside classes and are subject to the usual visibility rules.

4.3.1 Models as Expanders

Operations provided by models can be invoked directly, providing the functionality of *expanders* [181]. For example, the call `"x".(CIEq.equals)("X")` uses `CIEq` as the expander to test equality of two strings while ignoring case. Natural models can similarly be selected explicitly using the type name:

```
"x".(String.equals)("X")
```

Using models as expanders is an integral part of our genericity mechanism: the operations promised by where-clause constraints are invoked using expanders. In Figure 4.5, if we named the where-clause constraint `GraphLike[V,E]` with model variable `g`, the call `vu.sink()` would be sugar for `vu.(g.sink)()` with `g` being the expander. In this case, the expander can be elided because it can be inferred via default model resolution (Section 4.3.4).

4.3.2 Parameterized Models

Model definitions can be generic: they can be parameterized with type parameters and where-clause constraints. For example, model `ArrayListDeepCopy` (Figure 4.6) gives a naive implementation of deep-copying `ArrayLists`. It is generic with respect to the element type `E`, but requires `E` to be cloneable.

As another example, we can exploit model parameterization to implement the transpose of any graph. In Figure 4.7, the `DualGraph` model is itself a model

```
constraint Cloneable[T] { T clone(); }

model ArrayListDeepCopy[E] for Cloneable[ArrayList[E]] where Cloneable[E] {
  ArrayList[E] clone() {
    ArrayList[E] l = new ArrayList[E]();
    for (E e : this) { l.add(e.clone()); }
    return l;
  }
}
```

Figure 4.6. A parameterized model

for `GraphLike[V,E]`, and is parameterized by another model for `GraphLike[V,E]` (named g). It represents the transpose of graph g by reversing its edge orientations.

4.3.3 Non-Uniquely Witnessing Constraints

Previous languages with flexible type constraints, such as Haskell, JavaGI, and $\mathcal{G}$, require that witnesses be unique at generic instantiations, whether witnesses are scoped globally or lexically. By contrast, Genus allows multiple models witnessing a given constraint instantiation to coexist in the same context. This flexibility increases expressive power.

For example, consider Kosaraju's algorithm for finding strongly connected components in a directed graph [6]. It performs two depth-first searches, one following edges forward, and the other on the transposed graph, following edges backward. We would like to reuse the same generic depth-first-search algorithm on the same graph data structure for both traversals.

In Figure 4.7, the where clause of `SCC` introduces into the context a model for `GraphLike[V,E]`, denoted by model variable `g`. Using the `DualGraph` model,

```
model DualGraph[V,E] for GraphLike[V,E] where GraphLike[V,E] g {
  V E.source() { return this.(g.sink)(); }
  V E.sink() { return this.(g.source)(); }
  Iterable[E] V.incomingEdges() { return this.(g.outgoingEdges)(); }
  Iterable[E] V.outgoingEdges() { return this.(g.incomingEdges)(); }
}

void SCC[V,E](V[] vs) where GraphLike[V,E] g {
  … new DFIterator[V,E with g]() …
  … new DFIterator[V,E with DualGraph[V,E with g]]() …
}

class DFIterator[V,E] where GraphLike[V,E] { … }
```

Figure 4.7. Kosaraju's algorithm. Highlighted code is inferred if omitted.

the algorithm code can then perform both forward and backward traversals. It instantiates `DFIterator`, an iterator class for depth-first traversal, twice, with the original graph model `g` and with the transposed one. Being able to use two different models to witness the same constraint instantiation in `SCC` enables more code reuse. The highlighted `with` clauses can be safely elided, which brings us to default model resolution.

4.3.4 Resolving Default Models

In Genus, the omission of a `with` clause triggers default model resolution. Default model resolution is based on the following four ways in which models are *enabled* as potential default choices. First, types automatically generate natural models when they structurally conform to constraints. Natural models, when they exist, are always enabled as default candidates. Second, a where-clause constraint enables a model within the scope of the generic to which the where clause is

attached. For example, in method `SCC` in Figure 4.7 the where clause enables a model as a default candidate for `GraphLike[V,E]` within `SCC`. Third, a `use` declaration, e.g.,

```
use ArrayListDeepCopy;
```

enables the specified model as a potential default way to clone `ArrayLists` in the compilation unit in which the declaration resides. Fourth, a model itself is enabled as a potential default model within its definition.

Default model resolution works as follows:

1. If just one model for the constraint is enabled, it becomes the default model.
2. If more than one model is enabled, programmer intent is ambiguous. In this case, Genus requires that programmers make their intent explicit using a `with` clause. Omitting the `with` clause is a static error in this case.
3. If *no* model is explicitly enabled, but there is in scope a single model for the constraint, that model becomes the default model for the constraint.

Resolution for an elided expander in a method call works similarly. The only difference is that instead of searching for a model that witnesses a constraint, the compiler searches for a model that contains a method applicable to the given call. In typical use, this would be the natural model.

These rules for default models make generics and expanders easy to use in the common cases; in the less common cases where there is some ambiguity about which model to use, they force the programmer to be explicit and thereby help prevent hard-to-debug selection of the wrong model.

Letting each compilation unit choose its own default models is more flexible and concise than using Scala implicits, where a type-class instance can only be designated as implicit at the place where it is defined, and implicit definitions

```
class TreeSet[T] implements Set[T with c] where Comparable[T] c {
  TreeSet() { … }
  void addAll(Collection[? extends T] src) {
    if (src instanceof TreeSet[? extends T with c]) {
      addFromSorted((TreeSet[? extends T with c]) src);
    } else { … }
  }
  void addFromSorted(TreeSet[? extends T with c] src) {
    … // specialized code in virtue of the same ordering in src and this
  }
  …
}
```

Figure 4.8. `TreeSet` in Genus. Highlighted code is inferred if omitted.

are then imported into the scope, with a complex process used to find the most specific implicit among those imported [133]. We aim for simpler rules.

Genus also achieves the conciseness of Haskell type classes because uniquely satisfying models are allowed to witness constraints without being enabled, just as a unique type class instance in Haskell satisfies its type class without further declarations. But natural models make the mechanism lighter-weight than in Haskell, and the ability to have multiple models adds expressive power (as in the `SCC` example in Figure 4.7).

4.3.5 Models in Types

Section 4.2.3 introduced the ability to instantiate generic types with models, which become part of the type (i.e., model-dependent types). Type safety benefits from being able to distinguish instantiations that use different models.

The `addFromSorted` method in `TreeSet` (Figure 4.8) adds all elements in the source `TreeSet` to `this` one. Its signature requires that the source `TreeSet` and

`this` one use the same ordering. So a `TreeSet` with a different ordering cannot be accidentally passed to this method, avoiding a run-time exception.

Including the choice of model as part of the type is unusual, perhaps because it could increase annotation burden. Models are not part of types in the Concept design pattern (e.g., as realized in Scala [137]), because type class instances are not part of instantiated types. $\mathcal{G}$ [160] allows multiple models for the same constraint to be defined in one program (albeit only one in any lexical scope), yet neither at compile time nor at run time does it distinguish generic instantiations with distinct models. This raises potential safety issues when different modules interoperate.

In Genus, the concern about annotation burden is addressed by default models. For example, the type `TreeSet[? extends T]` in Figure 4.8 is implicitly instantiated with the model introduced by the where clause (via constraint entailment, Section 4.4.2). By contrast, Scala implicits work for method parameters, but *not* for type parameters of generic classes.

4.3.6 Models at Run Time

Unlike Java, whose type system is designed to support implementing generics via erasure, Genus makes models and type arguments available at run time. Genus allows testing the type of an object from a parameterized class at run time, like the `instanceof` test and the type cast in Figure 4.8.

Reifiability creates opportunities for optimization. For example, consider `TreeSet`'s implementation of the `addAll` method required by the `Collection` interface. In general, an implementation cannot rely on seeing the elements in the order expected by the destination collection, so for each element in the source collection, it must traverse the destination `TreeSet` to find the correct

position. However, if both collections use the same ordering, the merge can be done in a more asymptotically efficient way by calling the specialized method `addFromSorted`.

4.3.7 Default Model Resolution: Algorithmic Issues

Recursive resolution of default models. Default model resolution is especially powerful because it supports recursive reasoning. For example, the `use` declaration in Section 4.3.4 is syntactic sugar for the following parameterized declaration:

```
use [E where Cloneable[E] c] ArrayListDeepCopy[E with c]
for Cloneable[ArrayList[E]];
```

The default model candidacy of `ArrayListDeepCopy` is valid for cloning objects of any instantiated `ArrayList` type, provided that the element type satisfies `Cloneable` too. Indeed, when the compiler investigates the use of `ArrayListDeepCopy` to clone `ArrayList[Foo]`, it creates a subgoal to resolve the default model for `Cloneable[Foo]`. If this subgoal fails to be resolved, `ArrayListDeepCopy` is not considered as a candidate.

Recursive resolution may not terminate without additional restrictions. As an example, the declaration "`use DualGraph;`" is illegal because its recursive quest for a model of the same constraint causes resolution to cycle. The issue is addressed in Section 4.8 and the technical report [191] by imposing syntactic restrictions.

When a `use` declaration is rejected by the compiler for violating the restrictions, the programmer always has the workaround of explicitly selecting the model. By contrast, the inability to do so in Haskell or JavaGI makes it impossible to have a model like `DualGraph` in these languages.

Unification vs. default model resolution. Since Genus uses models in types, it is possible for models to be inferred via unification when they are elided. This inference potentially raises confusion with default model resolution.

Genus distinguishes between two kinds of where-clause constraints. Constraints for which the model is required by a parameterized type, such as `Eq[T]` in the declaration `void f[T where Eq[T]](Set[T] x)`, are called *intrinsic constraints*, because the `Set` must itself hold the corresponding model. By contrast, a constraint like `Printable[T]` in the declaration

```
void g[T where Printable[T]](List[T] x)
```

is *extrinsic* because `List[T]` has no such constraint on `T`.

Inference in Genus works by first solving for type parameters and intrinsic constraints via unification, and only then resolving default models for extrinsic constraints. To keep the semantics simple, Genus does not use default model availability to guide unification, and it requires extrinsic where-clause constraints to be written to the right of intrinsic ones. Nevertheless, it is always possible for programmers to explicitly specify intent.

4.3.8 Constraints/Models vs. Interfaces/Objects

The relationship between models and constraints is similar to that between objects and interfaces. Indeed, the Concept pattern can be viewed as using objects to implement models, and JavaGI extends interfaces to encode constraints. In contrast, Genus draws a distinction between the two, treating models as second-class values that cannot be stored in ordinary variables. This design choice has the following basis:

- Constraints are used in practice very differently from "ordinary" types, as evidenced by the nearly complete separation between *shapes* and *materials* seen in an analysis of a very large software base [83]. In their parlance, interfaces or classes that encode multiparameter constraints (e.g., `GraphLike`) or constraints requiring binary operations (e.g., `Comparable`) are shapes, while ordinary types (e.g., `Set`) are materials. Muddling the two may give rise to nontermination (Section 4.8).
- Because models are not full-fledged objects, generic code can easily be specialized to particular using contexts.
- Because model expressions can be used in types, Genus has dependent types; however, making models second-class and immutable simplifies the type system and avoids undecidability.

4.4 Making Models Object-Oriented

4.4.1 Dynamic Dispatching and Enrichment

In OO programs, subclasses are introduced to specialize the behavior offered by their superclasses. In Genus, models define part of the behavior of objects, so models too should support specialization. Therefore, a model in Genus may include not only method definitions for the base type, but also methods defining more specific behavior for subtypes. These methods can be dispatched *dynamically* by code both inside and outside model declarations. Dynamic dispatch takes place not only on the receiver, but also on method arguments of the manipulated types. The expressive power of dynamic dispatch is key to OO programming [8], and multiple dispatch is particularly important for binary operations, which are

typically encoded as constraints. Our approach differs in this way from $\mathcal{G}$ and Scala, which do not support dynamic dispatch on model operations.

For example, model `ShapeIntersect` in Figure 4.9 gives multiple definitions of `intersect`, varying in their expected argument types. In a context where the model is selected, a call to `intersect` on two objects statically typed as `Shape` will resolve at run time to the most specific method definition in the model. In JavaGI, multiple dispatch on `intersect` is impossible, because its dispatch is based on "self" types [32], while the argument types (including receiver) as well as the return type of an `intersect` implementation do not necessarily have to be the same.

Existing OO type hierarchies are often extended with new subclasses in ways not predicted by their designers. Genus provides *model enrichment* to allow models to be extended in a modular way, in sync with how class hierarchies are extended; here we apply the idea of open classes [45] to models. For example, if `Triangle` is later introduced to the `Shape` hierarchy, the model can then be separately enriched, as shown in the `enrich` declaration in Figure 4.9.

Model multimethods and model enrichment create the same challenge for modular type checking that is seen with other extensible OO mechanisms. For instance, if two modules separately enrich `ShapeIntersect`, these enrichments may conflict. Like Relaxed MultiJava [119], Genus can prevent such errors with a load-time check that there is a unique best method definition for every method invocation, obtaining mostly modular type checking and fully modular compilation. The check can be performed soundly, assuming load-time access to the entire program. If a program loads new code dynamically, the check must be performed at the time of dynamic loading.

```
constraint Intersectable[T] { T T.intersect(T that); }

model ShapeIntersect for Intersectable[Shape] {
  Shape Shape.intersect(Shape s) { … }
  // Rectangle and Circle are subclasses of Shape:
  Rectangle Rectangle.intersect(Rectangle r) { … }
  Shape Circle.intersect(Rectangle r) { … }
  …
}

enrich ShapeIntersect {
  Shape Triangle.intersect(Circle c) { … }
  …
}
```

Figure 4.9. An extensible model with multiple dispatch

4.4.2 Constraint Entailment

As seen earlier (Section 4.2.1), a constraint entails its prerequisite constraints. In general, a model may be used as a witness not just for the constraint it is declared for, but also for any constraints entailed by the declared constraint. For example, a model for `Comparable[Shape]` can be used to witness `Eq[Shape]`.

A second way that one constraint can entail another is through *variance* on constraint parameters. For example, since in constraint `Eq` the type parameter only occurs in contravariant positions, a model for `Eq[Shape]` may also be soundly used as a model for `Eq[Circle]`. It is also possible, though less common, to use a model to witness constraints for supertypes, via covariance. Variance is inferred automatically by the compiler, with bivariance downgraded to contravariance.

A model enabled for some constraint in one of the four ways discussed in Section 4.3.4 is also enabled for its prerequisite constraints and constraints that can be entailed via contravariance. Accommodating subtyping extends the

expressivity of default model resolution, but poses new challenges for termination. The technical report [191] shows that encoding "shape" types (in the sense of Greenman et al. [83]) as constraints helps ensure termination.

4.4.3 Model Inheritance

Code reuse among models can be achieved through model inheritance, signified by an `extends` clause (e.g., model `CICmp` in Section 4.3). Unlike an `extends` clause in a class or constraint definition, which creates an *is-a* relationship between a subclass and its superclass or a constraint and its prerequisite constraint, an `extends` clause in a model definition is *merely* for code reuse. The inheriting model inherits all method definitions *with compatible signatures* available in the inherited model. The inheriting model can also override these inherited definitions.

Model inheritance provides a means to derive models that are otherwise rejected by constraint entailment. For example, the model `ShapeIntersect` (Figure 4.9) soundly witnesses the same constraint for `Rectangle`, because the selected method definitions have compatible signatures, even though constraint `Intersectable` is invariant with respect to its type parameter. The specialization to `Rectangle` can be performed succinctly using model inheritance, with the benefit of a more precise result type when two rectangles are intersected:

```
model RectangleIntersect for Intersectable[Rectangle]
extends ShapeIntersect { }
```

4.5 Use-Site Genericity

Java's wildcard mechanism [173] is in essence a limited form of existential quantification. Existentials enable genericity at *use sites*. For example, a Java method

with return type `List<? extends Printable>` can be used by generic calling code that is able to print list elements even when the type of the elements is unknown to the calling code. The use-site genericity mechanism of Genus generalizes this idea while escaping some limitations of Java wildcards. Below we sketch the mechanism.

4.5.1 Existential Types

Using subtype-bounded existential quantification, the Java type `List<? extends Printable>` might be written more type-theoretically as `∃U≤Printable.List[U]`. Genus extends this idea to constraints. An existential type in Genus is signified by prefixing a quantified type with type parameters and/or where-clause constraints. For example, if `Printable` is a constraint, the Genus type corresponding to the Java type above is

```
[some U where Printable[U]]List[U]
```

The initial brackets introduce a use-site type parameter `U` and a model for the given constraint, which are in scope in the quantified type; the syntax emphasizes the connection between existential and universal quantification.

The presence of prefixed parameters in existential types gives the programmer control over the existential binding point, in contrast to Java wildcard types where binding is always at the generic type in which the wildcard is used as a type argument. For example, no Java type can express `∃U.List[List[U]]`, meaning a *homogeneous* collection of lists in which each list is parameterized by the *same* unknown type. This type is easily expressed in Genus as

```
[some U]List[List[U]]
```

```
[some T where Comparable[T]]List[T] f () {
  return new ArrayList[String]();
}

sort(f());
[U] (List[U] l) where Comparable[U] = f();   // bind U
l.first().compareTo(l.last());          // U is comparable
U[] a = new U[64];              // use run-time info about U
l = new ArrayList[U]();                  // new list, same U
```

Figure 4.10. Working with existential quantification

Genus also offers convenient syntactic sugar for common uses of existential types. A single-parameter constraint can be used as sugar for an existential type: e.g., `Printable`, used as a type, is sugar for

```
[some U where Printable[U]]U
```

allowing a value of any printable type. The wildcard syntax `List[?]` represents an existential type, with the binding point the same as in the Java equivalent. The type with a wildcard model `Set[String with ?]` is sugar for

```
[some Eq[String] m]Set[String with m]
```

Subtyping and coercion. Genus draws a distinction between *subtyping* and *coercion* involving existential types. Coercion may induce extra computation (i.e., existential packing) and can be context-dependent (i.e., default model resolution), while subtyping cannot. For example, the `return` expression in Figure 4.10 type-checks not because `ArrayList[String]` is a subtype of the existential return type, but because of coercion, which works by packing together a value of type `ArrayList[String]` with a model for `Comparable[String]` (in this case, the natural model) into a single value. The semantics of subtyping involving where-

clause-quantified existential types is designed in a way that makes it easy for programmers to reason about subtyping and joining types.

Capture conversion. In Java, wildcards in the type of an expression are instantiated as fresh identifiers when the expression is type-checked, a process called *capture conversion* [82]. Genus extends this idea to constraints: in addition to fresh type variables, capture conversion generates fresh models for where-clause constraints, and enables them in the current scope.

For example, at line 1 in Figure 4.10, when the call to `sort` (defined in Section 4.2.3) is type-checked, the type of the call `f()` is capture-converted to `List[#T]`, where `#T` is the fresh type variable that capture conversion generates for `T`, and a model for `Comparable[#T]` becomes enabled in the current context. Subsequently, the type argument to `sort` is inferred as `#T`, and the default model for `Comparable[#T]` resolves to the freshly generated model.

4.5.2 Explicit Local Binding

Capture conversion is convenient but not expressive enough. Consider a Java object typed as `List<? extends Comparable>`. The programmer might intend the elements of this homogeneous list to be comparable to one another, but comparisons to anything other than `null` do not type-check.

The awkwardness is addressed in Genus by *explicit local binding* of existentially quantified type variables and where-clause constraints, giving them names that can be used directly in the local context. An example of this mechanism is found at line 2 in Figure 4.10. The type variable `U` can be used as a full-fledged type in the remainder of the scope.

As its syntax suggests, explicit local binding can be viewed as introducing an inlined generic method encompassing subsequent code. Indeed, it operates under the same rules as universally quantified code. For example, the where clause at line 2 enables a new model so that values of type `U` can be compared at line 3. Also, locally bound type variables are likewise reifiable (line 4). Moreover, the binding at line 2 is type-checked using the usual inference algorithm to solve for `U` and for the model for `Comparable[U]`: per Section 4.3.7, the former is inferred via unification and the latter via default model resolution—it is an extrinsic constraint. Soundness is maintained by ensuring that `l` is initialized upon declaration and that assignments to the variable preserve the meaning of `U`.

4.6 Implementation

We have built a partial implementation of the Genus language in Java. The implementation consists of about 39,000 lines of code, extending the Polyglot compiler framework [130]. Code generation works by translating to Java 5 code, relying on a Java compiler as a back end. The current compiler implementation does not yet specialize instantiations to particular type arguments.

4.6.1 Implementing Constraints and Models

Constraints and models in Genus code are translated to parameterized interfaces and classes in Java. For example, the constraint `Comparable[T]` is translated to a parameterized Java interface `Comparable<T>` providing a method `compareTo` with the appropriate signature: `int compareTo(T,T)`. Models are translated to Java classes that implement these constraint interfaces.

```
// Source code in Genus
class ArrayList[T] implements List[T] {
  T[] arr;
  ArrayList() { arr = new T[INITIAL_SIZE]; }
  ...
}

// Target code in Java
class ArrayList<T> implements List<T> {
  Object arr; // an array of run-time type int[] when T is instantiated on int
  RTT<T> rtt$T; // run-time type information about T
  ArrayList(RTT<T> rtt$T) {
    this.rtt$T = rtt$T;
    arr = rtt$T.newArray(INITIAL_SIZE);
  }
  ...
}
```

Figure 4.11. Translating the Genus class `ArrayList` into Java

4.6.2 Implementing Generics

Parameterized Genus classes are translated to correspondingly parameterized Java classes. However, type arguments and models must be represented at run time. So extra arguments carrying this information are required by class constructors, and constructor bodies are extended to store these arguments as fields. For example, class `ArrayList` has a translated constructor with the signature shown in Figure 4.11. Parameterized methods and models are translated in a similar way by adding extra arguments representing type and model information.

4.6.3 Supporting Primitive Type Arguments

A challenge for efficient generics, especially with a JVM-based implementation, is how to avoid uniformly wrapping all primitives inside objects when primitive types are used as type arguments. Some wrapping is unavoidable, but from the standpoint of efficiency, the key is that when code parameterized on a type `T` is instantiated on a primitive type (e.g., `int`), the array type `T[]` should be represented exactly as an array of the primitive type (e.g., `int[]`), rather than a type like `Integer[]` in which every array element incurs the overhead of individualized memory management.

Our current implementation uses a *homogeneous* translation to support this efficiency. The run-time type information object (e.g., `rtt$T` in Figure 4.11) for a type parameter `T` provides all operations about `T[]`. It has type `RTT<T>`, which provides the operations for creating and accessing arrays of (unboxed) `T`.

A more efficient approach to supporting primitive type arguments is to generate specialized code for primitive instantiations, as is done in C#. The design of Genus makes it straightforward to implement particular instantiations with specialized code.

4.7 Evaluation

4.7.1 Porting Java Collections Framework to Genus

To evaluate how well the language design works in practice, we ported all 10 general-purpose implementations in the Java collections framework (JCF) as well as relevant interfaces and abstract implementations, to Genus. The result is a

safer, more precise encoding and more code reuse with little extra programmer effort.

The single most interesting constrained generic in JCF is probably `TreeSet` (and `TreeMap`, which backs it). In its Java implementation, elements are ordered using either the element type's implementation of `Comparable` or a comparator object passed as a constructor argument, depending on which constructor is used to create the set. This ad hoc choice results in error-prone client code. In Genus, by contrast, the ordering is part of the `TreeSet` type, eliminating 35 occurrences of `ClassCastException` in `TreeSet`'s and `TreeMap`'s specs.

Genus collection classes are also more faithful to the semantics of the abstractions. Unlike a `Set[E]`, a `List[E]` should not necessarily be able to test the equality of its elements. In Genus, collection methods like `contains` and `remove` are instead parameterized by the definition of equality (Section 4.2.2). These methods cannot be called unless a model for `Eq[E]` is provided.

More powerful genericity also enables increased code reuse. For example, the `NavigableMap` interface allows extracting a descending view of the original map. In JCF, `TreeMap` implements this view by defining separate classes for each of the ascending and descending views. In contrast, Genus expresses both views concisely in a single class parameterized by a model that defines how to navigate the tree, eliminating 160 lines of code. This change is made possible by retroactive, non-unique modeling of `compareTo()`.

Thanks to default models—in particular, implicit natural models, for popular operations including `toString`, `equals`, `hashCode` and `compareTo`—client and library code ordinarily type-check without using `with` clauses. When `with` clauses are used, extra expressive power is obtained. In fact, the descending views are the *only* place where `with` clauses are needed in the Genus collection classes.

4.7.2 Porting the Findbugs Graph Library to Genus

We ported to Genus the highly generic Findbugs [71] graph library (~1000 non-comment LoC), which provides graph algorithms used for the intermediate representation of static analyses. In Findbugs, the entities associated with the graph (e.g., `Graph`, `Vertex`, `Edge`) are represented as Java interfaces; F-bounded polymorphism is used to constrain parameters. As we saw earlier (Section 4.1), the resulting code is typically more cumbersome than the Genus version.

We quantified this effect by counting the number of parameter types, concrete types and keywords (`extends`, `where`) in each type declaration, ignoring modifiers and the name of the type. Across the library, Genus reduces annotation burden by 32% yet increases expressive power. The key is that constraints can be expressed directly without encoding them into subtyping and parametric polymorphism; further, prerequisite constraints avoid redundancy.

4.7.3 Performance

The current Genus implementation targets Java 5. To explore the overhead of this translation compared to similar Java code, we implemented a small Genus benchmark whose performance depends heavily on the efficiency of the underlying genericity mechanism, and hence probably exaggerates the performance impact of generics. The benchmark performs insertion sort over a large array or other ordered collection; the actual algorithm is the same in all cases, but different versions have different degrees of genericity with respect to the element type and even to the collection being sorted. Element type `T` is required to satisfy a constraint `Comparable[T]` and type `A` is required to satisfy a constraint `ArrayLike[A,T]`, which requires `A` to act like an array of `T`'s. Both primitive values (`double`) and ordinary object types (`Double`) are sorted.

Table 4.1: Comparing performance of Java and Genus

	data structure	**Java** (s)	**Genus** (s) [spec.]
Non-generic sort	`double[]`	1.3	
	`Double[]`	3.8	
	`ArrayList[double]`	—	5.4 [4.0]
	`ArrayList[Double]`	9.6	14.5 [8.3]
Generic sort: **`Comparable[T]`**	`double[]`	—	19.3 [1.3]
	`Double[]`	7.7	10.0 [3.8]
	`ArrayList[double]`	—	6.7 [4.0]
	`ArrayList[Double]`	9.8	17.9 [8.3]
Generic sort: **`ArrayLike[A,T]`,** **`Comparable[T]`**	`double[]`	—	17.0 [1.3]
	`Double[]`	12.8	12.4 [3.8]
	`ArrayList[double]`	—	24.6 [4.0]
	`ArrayList[Double]`	12.8	24.8 [8.3]

The results from sorting collections of 100k elements are summarized in Table 4.1. Results were collected using Java 7 on a MacBook Pro with a 2.6GHz Intel Core i7 processor. All measurements are the average of 10 runs, with an estimated relative error always within 2%. For comparison, the same (non-generic) algorithm takes 1.1s in C (with `gcc -O3`). The Java column leaves some entries blank because Java does not allow primitive type arguments.

To understand the performance improvement that is possible by specializing individual instantiations of generic code, we used hand translation; as mentioned above, the design of Genus makes such specialization easy to do. The expected performance improvement is shown in the bracketed table entries. Specialization to primitive types is particularly useful for avoiding the high cost of boxing and unboxing primitive values, but the measurements suggest use of primitive type arguments can improve performance even without specialization (e.g., Genus `ArrayList[double]` is usually faster than Java `ArrayList<Double>`).

4.8 Formalization and Decidability

We have formalized the key aspects of the Genus type system, in the style of Featherweight Java [94]. Importantly, inference rules for subtyping, constraint entailment, and well-formedness (including model–constraint conformance in the presence of multimethods) are given. The formalization is provided in the technical report [191]. We are not aware of any unsoundness in the type system, but leave proving soundness to future work.

Default model resolution is an integral part of the formalization, matching the description in Sections 4.3.4, 4.3.7, and 4.4.2. It is formalized as a translation from one calculus into another—the source calculus allows default models while the target is default-model-free.

Syntactic restrictions for decidable resolution of type class instances [166] and decidable subtyping with variance [83] have been separately proposed. We formulate our termination condition for default model resolution by synthesizing these restrictions, and to the best of our knowledge, give the first termination proof for such resolution when coupled with variance.

4.9 Related Work

Much prior work on parametric genericity mechanisms (e.g., [4, 22, 40, 43, 100, 110, 126, 157]) relies on constraint mechanisms that do not support retroactive modeling. We focus here on more recent work that follows Haskell's type classes in supporting retroactive modeling, complementing the discussion in previous sections.

The C++ community developed the Concept design pattern, based on templates, as a way to achieve retroactive modeling [13]. This pattern is used

extensively in the STL and Boost libraries. Templates are not checked until instantiation, so developers see confusing error messages, and the lack of separate compilation makes compilation time depend on the amount of generic library code. The OO language $\mathcal{G}$ [160], based on System F^G [159], supports separate compilation but limits the power of concept-based overloading. By contrast, C++ Concepts [84] abandon separate compilation to fully support concept-based overloading. It was not adopted by the C++11 standard [158], however. Concept-based overloading is orthogonal to the other Genus features; it is not currently implemented but could be fully supported by Genus along with separate compilation, because models are chosen modularly at compile time.

In Scala, genericity is achieved with the Concept design pattern and *implicits* [137]. This approach is expressive enough to encode advanced features including associated types [42] and generalized constraints [63]. Implicits make using generics less heavyweight, but add complexity. Importantly, Scala does not address the problems with the Concept pattern (Section 4.1). In particular, it lacks model-dependent types and also precludes the dynamic dispatch that contributes significantly to the success of object-oriented programming [8].

JavaGI [182] generalizes Java interfaces by reusing them as type classes. Like a type class instance, a JavaGI `implementation` is globally scoped, must uniquely witness its interface, and may only contain methods for the type(s) it is declared with. Unlike in Haskell, a call to an interface method is dynamically dispatched across all `implementations`. Although dispatch is not based entirely on the receiver type, within an `implementation` all occurrences of an implementing type for `T` must coincide, preventing multiply dispatching `intersect` across the `Shape` class hierarchy (cf. Section 4.4.1).

Approaches to generic programming in recent languages including Rust [154] and Swift [169] are also influenced by Haskell type classes, but do not escape their limitations.

Type classes call for a mechanism for implicitly and recursively resolving evidence of constraint satisfaction. The implicit calculus [138] formalizes this idea and extends it to work for all types. However, the calculus does not have subtyping. Factoring subtyping into resolution is not trivial, as evidenced by the reported stack overflow of the JavaGI compiler [83].

No prior work brings type constraints to use sites. The use of type constraints as types [169, 182] is realized as existentials in Genus. "Material–Shape Separation" [83] prohibits types such as `List<Comparable>`, which do find some usage in practice. Existentials in Genus help express such types in a type-safe way.

Associated types [42, 127] are type definitions required by type constraints. Encoding functionally dependent type parameters as associated types helps make certain type class headers less verbose [76]. Genus does not support associated types because they do not arise naturally as in other languages with traits [137, 154] or module systems [57] and because Genus code does not tend to need as many type parameters as in generic C++ code.

Finally, in Table 4.2, we compare how Genus and other languages perform with respect to the desiderata identified by prior work [77, 137, 160] and us. Not all prior desiderata are reflected in the table. Since we consider support for associated types to be an orthogonal issue, our desiderata do not include *constraints on associated types* and *equality constraints*. Also due to orthogonality, we omit *type aliases* and *first-class functions*.

Table 4.2: Comparing various generics approaches

	C++11	SML/OCaml	Haskell	Java	C#	Cecil	C++ Concepts	Rust	Swift	Scala	$\mathcal{G}$	JavaGI	Genus
Multiparameter constraints (Section 4.2.1)	○	●	●[a]	○	○	◐	●	○	○	●	●	●	●
Multiple constraints (Section 4.2.2)	○	◐	●	●	●	●	●	●	●	●	●	●	●
Associated types access (Section 4.9)	●	●	●	○	○	◐	●	●	●	●	●	○	○
Retroactive modeling (Section 4.3)	○	●	●	○	○	●	●	●	●	●	●	●	●
Modular compilation (Section 4.4.1)	○	●	●	●	●	◐	○	○	●	●	●	●	●
Implicit argument deduction (Section 4.5.2)	●	●	●	●	●	◐	●	●	●	●	●	●	●
Modular type checking (Section 4.4.1)	○	●	●	●	●	◐	◐	●	●	●	●	◐	◐
Lexically scoped models (Section 4.3)	○	●	○	○	○	○	●	○	○	●	●	○	●
Concept-based overloading (Section 4.9)	○	○	○	○	○	●	●	○	●	◐	◐	○	○
Model generics (Section 4.2.2)	○	○	○	○	○	○	○	○	○	●	○	○	●
Natural models (Section 4.2.3, Section 4.3.4)	●	○	○	○	○	●	●	○	◐	○	○	◐	●
Non-unique modeling (Section 4.3.3)	○	○	○	○	○	○	○	○	○	●	○	○	●
Model-dependent types (Section 4.2.3, Section 4.3.5)	○	○	○	○	○	○	○	○	○	○	○	○	●
Run-time type/model info (Section 4.3.6, Section 4.6)	●	○	○	○	●	●	●	●	●	◐	●	○	●
Model enrichment (Section 4.4.1)	○	○	○	○	○	○	○	○	○	○	○	●	●
Multiple dispatch (Section 4.4.1)	○	○	○	○	○	●	○	○	○	○	○	◐	●
Constraint variance (Section 4.4.2)	○	○	○	○	○	○	○	○	◐	●	○	◐	●
Model inheritance (Section 4.4.3)	○	○	○	○	○	○	○	○	○	○	○	○	●
Use-site generics (Section 4.5)	○	○	◐[b]	◐	○	○	○	○	○	◐	○	◐	●

[a] Using `MultiParamTypeClasses`

[b] Using `ExistentialQuantification`

CHAPTER 5

FAMILIA: UNIFYING INTERFACES, TYPE CLASSES, AND FAMILY POLYMORPHISM

It is futile to do with more things that which can be done with fewer.

—William of Ockham

Types help programmers write correct code, but they also introduce rigidity that can interfere with reuse. In statically typed languages, mechanisms for polymorphism recover needed flexibility about the types that code operates over.

Subtype polymorphism [39] and *inheritance* [48] are polymorphism mechanisms that have contributed to the wide adoption of modern object-oriented (OO) languages like Java. They make types and implementations open to future type-safe extensions, and thus increase code extensibility and reuse.

Parametric polymorphism offers a quite different approach: explicitly parameterizing code over types and modules it mentions [108, 112, 120]. It has dominated in functional languages but is also present in modern OO languages. Parametric polymorphism becomes even more powerful with the addition of type classes [176], which allow existing types to be *retroactively adapted* to the requirements of generic code.

Harmoniously integrating these two kinds of polymorphism has proved challenging. The success of type classes in Haskell, together with the awkwardness of using F-bounded constraints [38] for generic programming, has inspired recent efforts to integrate type classes into OO languages [160, 182, 190]. However, type classes and instances for those type classes burden already feature-rich languages with entirely new kinds of interfaces and implementations. The difficulty of

adopting *concepts* in C++ [165] suggests that the resulting languages may seem too complex.

Meanwhile, work on object-oriented inheritance has increased expressive power by allowing inheritance to operate at the level of *families* of related classes and types [12, 44, 64, 66, 114, 115, 131, 132, 150, 170]. Such *family polymorphism*, including virtual types and virtual classes, supports coordinated, type-safe extensions to related types and classes contained within a larger module. These features have also inspired [144] the addition of *associated types* [42] to type classes. Associated types are adopted by recent languages such as Rust [154] and Swift [169]. However, the lack of family-level inheritance limits the expressive power of associated types in these languages.

Combining all these desirable features in one programming language has not been done previously, perhaps because it threatens to confront programmers with a high degree of surface complexity. Our contribution is a lightweight unification of these different forms of polymorphism, offering increased expressive power with low apparent complexity. This unified polymorphism mechanism is embodied in a proposed Java-like language that we call Familia.

The key insight is that a lightweight presentation of the increased expressive power can be achieved by using a single interface mechanism to express both data abstraction and type constraints, by using classes as their implementations, and by using classes (and interfaces) as modules. Both interfaces and classes can be extended. The expressive power offered by previous polymorphism mechanisms, including flexible adaptation and family polymorphism, falls out naturally. Specifically, this chapter makes the following contributions:

- We show how to unite object-oriented polymorphism and parametric polymorphism by generalizing existing notions of interfaces, classes, and

method calls (Sections 5.2 and 5.3). The extensibility of objects and the adaptive power of type classes both follow from this reinterpretation.

- We further show how to naturally integrate an expressive form of family polymorphism. The design accommodates features found in previous family-polymorphism mechanisms (including associated types and nested inheritance) in the above setting of generalized classes and interfaces, and goes beyond them by offering new expressive power. We present a case study of using Familia to implement a highly reusable program analysis framework (Section 5.4).
- We capture the new language mechanisms formally by introducing a core language, Featherweight Familia, and we establish the soundness of its type system (Section 5.5).
- We show the power of the unified polymorphism mechanism by comparing Familia with various prior languages designed for software extensibility (Section 5.6).

5.1 Background

Our goal is a lightweight, expressive unification of the state of the art in genericity mechanisms. A variety of complementary genericity mechanisms have been developed, with seemingly quite different characteristics.

Genericity via inheritance. Object-oriented languages permit a given interface to be implemented in multiple ways, making clients of that interface generic with respect to future implementations. Hence, we call this a form of *implementation genericity*. Inheritance extends the power of implementation genericity by

allowing the code of a class to be generic with respect to implementation changes in future subclasses; method definitions are *late-bound*. While the type-theoretic essence of class inheritance is parameterization [46], encoding inheritance in this way is more verbose and less intuitive [20].

Family polymorphism [65] extends the expressive power of inheritance by allowing late binding of the meaning of types and classes declared within a containing class, supporting the design of highly extensible and composable software [132]. Virtual types [135, 170] and associated types [42, 95] allow the meaning of a type identifier to be provided by subclasses; with virtual classes as introduced by Beta [65, 115], the code of a nested class is also generic with respect to classes it is nested within. The outer class can then be subclassed to override the behavior and structure of the entire family of related classes and types in a coordinated and type-safe way. Classes and types become members of an object of the family class. The late binding of type names means that all type names implicitly become hooks for later extension, without cluttering the code with a possibly large number of explicit type parameters.

There are two approaches to family polymorphism; in the nomenclature of Clarke et al. [44], the original *object family* approach of Beta treats nested classes as attributes of objects of the family class [12, 65, 115], whereas in the *class family* approach of Concord [97], Jx and J& [131, 132], and ˆFJ [93] nested classes and types are attributes of the family classes directly. The approaches have even been combined by work on Tribe [44]. Familia follows Jx by providing *nested inheritance* [131], a class family mechanism that allows both further binding (specialization of nested classes) at arbitrary depth in the class nesting structure, and also inheritance across families.

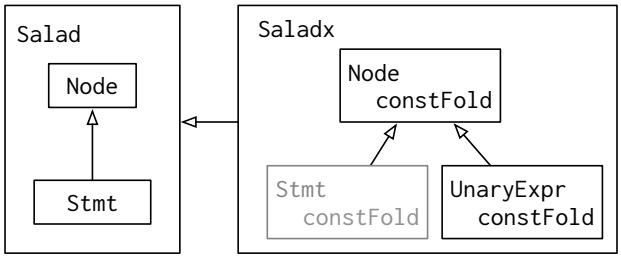


Figure 5.1. Applying family polymorphism to compiler construction

To see how support for coordinated changes can be useful, suppose we are building a compiler for a programming language called Saladx, which extends a previous language called Salad. The Salad compiler defines data structures (that is, types) and algorithms that operate on these types. We would like to reuse the Salad compiler code in a modular way, without modification. Figure 5.1 sketches how this can be done in a modular, type-safe way using nested inheritance.

The original compiler defines abstract syntax tree (AST) nodes such as `Node` and `Stmt`. The extended compiler defines a new module `Saladx` that inherits as a family from the original `Salad` module. The new module adds support for a new type of AST node, `UnaryExpr`, by adding a new class definition. Saladx also *further binds* the class `Node` to add a new method `constFold` that performs constant folding. Importantly, the rest of `Salad.Node` does not need to be restated. Nor does any code need to be written for `Saladx.Stmt`; this class *implicitly* exists in the `Saladx` module and inherits `constFold` from the new version of `Node`. References in the original Salad code to names like `Node` and `Stmt` now refer in the Saladx compiler to the Saladx versions of these classes. The Salad code is highly extensible without being explicitly parameterized.

By contrast, the conventional OO approach could extend individual classes and types from Salad with new behavior. However, each individually extended class could not access the others' extended behavior (such as `constFold`) in a

type-safe way. Alternatively, extensibility could be implemented for Salad by cluttering the code with many explicit type parameters.

Genericity via parametric polymorphism. Parametric polymorphism, often simply called *generics*, provides a more widely known and complementary form of genericity in which code is *explicitly* parameterized with respect to the types or modules of data it manipulates. Whereas implementation genericity makes client code and superclass code generic with respect to future implementations, parametric polymorphism makes *implementations* generic with respect to future *clients*. *Constrained* parametric polymorphism [108] ensures that generic code can be instantiated only on types meeting a constraint. These constraints act effectively as a second kind of interface.

Haskell's type classes [176] manifest these interfaces as named constraints to which programmers can explicitly adapt existing types. By contrast, most OO languages (e.g., Java and C#) use subtyping to express constraints on type parameters. Subtyping constraints are rigid: they express binary methods in an awkward manner, and more crucially, it is typically impossible to retroactively adapt types to satisfy the subtyping requirement. The rigidity of subtyping constraints has led to new OO languages that support type classes [160, 182, 190].

Combining genericity mechanisms. Genericity mechanisms are motivated by a real need for expressive power. Both family polymorphism and type classes can be viewed as reactions to the classic *expression problem* [179] on the well-known difficulty of extending both data types and the operations on them in a modular, type-safe way [151]. However, the approaches are complementary: type classes do not also provide the *scalable extensibility* [131] offered by family polymorphism, whereas family polymorphism lacks the *flexible adaptation* offered by type

classes. Despite becoming popular among recent languages that incorporate type classes [154, 169], associated types do not provide the degree of extensibility offered by an expressive family-polymorphism mechanism.

On the other hand, *data abstraction* is concerned with separating public interfaces from how they are implemented so that the implementation can be changed freely without affecting the using code. Implementations are defined in terms of a *representation* that is hidden from clients of the interface. Abstract data types, object interfaces, and type classes can all provide data abstraction [47]. Genericity mechanisms such as inheritance and parametric polymorphism are not essential to data abstraction. However, they add significant expressive power to data abstraction.

Languages that combine multiple forms of polymorphism tend to duplicate data abstraction mechanisms. For example, recent OO languages incorporate the expressive power of type classes by adding new language structures above and beyond the standard OO concepts like interfaces and classes [160, 182, 190]. Unfortunately, a programming language that provides data abstraction in more than one way is likely to introduce feature redundancy and threatens to confront the programmer with added surface complexity. Even for Haskell, it has been argued that type classes introduced duplication of functionality [55], and that the possibility of approaching the same task in multiple ways created confusion [142].

Our contribution is a clean way to combine data abstraction and these disparate and powerful polymorphism mechanisms in a compact package. As a result, programmers obtain the expressive power they need for a wide range of software design challenges, without confronting the linguistic complexity that would result from a naive combination of all the mechanisms.

5.2 Unifying object-oriented interfaces and type classes

Both object-oriented interfaces and type classes are important, but having both in a language can lead to confusion and duplication. Fortunately, both can be supported by a single, unified interface mechanism, offering an economy of concepts.

We unify interfaces with type classes by decoupling the *representation type* of an object-oriented interface from its *object type*. A representation type is an underlying type used to implement the interface; the implementations of interface methods operate on these representation types. An object type, on the other hand, specifies the externally visible operations on an object of the interface.

For example, an interface `Eq` describing the ability of a type `T` to be compared for equality can be written as shown in Figure 5.2.[1] This interface declares a single representation type `T` (in parentheses after the interface name `Eq`); the receiver of method `equals` hence has this representation type. Each implementation of this interface chooses some concrete type as the representation type.

As convenient syntactic sugar, an interface with a single representation type may omit its declaration, implicitly declaring a single representation type named `This`. In this usage, all non-static methods declared by the interface have implicit receiver type `This`. In Figure 5.2, the other three interfaces all exploit this sugar.

An interface may also declare ordinary type parameters for generic programming, grouped in square brackets to distinguish them from representation type parameters. For example, a generic set interface might be declared as shown in Figure 5.3a, where the interface `Set` has an explicit type parameter `E` representing its elements. In the figure, the omitted representation type of `Set` (i.e., `This`)

[1] Except as noted, Familia follows the syntactic and semantic conventions of Java [81].

```
interface Eq(T) {
  boolean T.equals(T);
}

interface PartialOrd extends Eq {
  boolean leq(This);
}
```

```
interface Hashable extends Eq {
  int hashCode();
}

interface Ord extends PartialOrd {
  int compare(This);
}
```

Figure 5.2. Four interfaces with single representation types. `Eq` explicitly names its representation type `T`; the others leave it implicit as `This`. The receiver types of the interface methods are the representation types.

```
interface Set[E where Eq(E)]
extends Collection[E] {
    int size();
    boolean contains(E);
    Self add(E);
    Self remove(E);
    Self addAll(Set[E]);
    ...
}
```

(a) Interface `Set` is parameterized by a type parameter and a `where`-clause constraint.

```
1 interface SortedSet[E]
2 extends Set[E] where Ord(E) {
3   E max() throws SetEmpty;
4   E min() throws SetEmpty;
5   Self subset(E, E);
6   ...
7 }
```

(b) Interface `SortedSet` extends `Set`. Its `where`-clause constraint `Ord(E)` entails `Eq(E)`.

Figure 5.3. Interfaces `Set` and `SortedSet`

is also the implicit representation type of `Collection[E]`, the interface being extended.

Using interfaces to constrain types. Interfaces can be used as type classes: that is, as constraints on types. In Figure 5.3a, `Set` has a `where`-clause `where Eq(E)`, which constrains the choice of types for `E` to those which satisfy the interface `Eq` and that therefore support equality. A `where`-clause may have several such

constraints, each constraining a type parameter by instantiating an interface using that type (`E` in this example) as the representation type. Hence, we also refer to representation types as *constraint parameters*.

As syntactic sugar, a `where`-clause may be placed outside the brackets containing type parameters (e.g., line 2 of Figure 5.3b). If kept inside the brackets, the parameters to the constraint may be omitted, defaulting to the preceding type parameter(s), as in line 1 of Figures 5.6a and 5.6b. A `where`-clause constraint can optionally be named (e.g., line 18 of Figure 5.5).

Using the object type. Each interface also defines an *object type* that has the same name as the interface. Using an interface as an object type corresponds to the typical use of interfaces in OO languages. In this case, the interface hides its representation type from the client. For example, in the variable declaration "`Hashable x;`", the representation type is an *unknown* type `T` on which the constraint `Hashable(T)` holds. The programmer can make the method call `x.hashCode()` in the standard, object-oriented way. Thus, the interface `Hashable` serves both as a type class that is a constraint on types and as an ordinary object type.

From a type-theoretic viewpoint, object types are existential types, as in some prior object encodings [28]. A method call on an object (e.g., `x.hashCode()`) implicitly unpacks the existentially typed receiver. This unpacking is made explicit in the core language of Section 5.5.

Subtype polymorphism and constraint entailment. Subtype polymorphism is an essential feature of statically typed OO languages. As Figures 5.2 and 5.3 show, interfaces can be extended in Familia. The declaration `extends Collection[E]` in the definition of interface `Set` introduces a subtype relationship between

`Set[E]` and `Collection[E]`. An interface definition can extend multiple interfaces.

Such subtype relationships are higher-order [145], in the sense that interfaces in Familia can be viewed as type operators that accept a representation type. When the interfaces are used as object types, this higher-order subtyping relation becomes the familiar, first-order subtyping relation between object types. When the interfaces are used to constrain types, this higher-order subtyping relation manifests in *constraint entailment*, a relation between constraints on types [182, 190]. For example, consider the instantiation of `Set` on line 2 of `SortedSet` in Figure 5.3b. The type `Set` can only be instantiated on a type T that satisfies the constraint `Eq(`T`)`; here, because `Ord` (transitively) extends `Eq`, constraint `Ord(E)` being satisfied entails `Eq(E)` being satisfied.

A second form of constraint entailment concerns varying the representation type, rather than the interface, when the interface is contravariant in its representation type. For example, all interfaces in Figures 5.2 and 5.3 are contravariant, because they do not use their representation types in covariant or invariant positions. Because `Eq` is contravariant, it is safe to use a class that implements `Eq(Set[E])` to satisfy the constraint `Eq(SortedSet[E])`. Figure 5.4 shows an example of an invariant interface, `LowerLattice`, which uses `This` on line 2 both covariantly (as a return type) and contravariantly (as receiver and argument types). Familia infers interfaces to be either contravariant or invariant in their constraint parameters, with most interfaces being contravariant. A constraint parameter that is inferred contravariant may also be explicitly annotated as invariant. Covariance and bivariance in constraint parameters are not supported because these forms of variance do not seem useful.

```
interface LowerLattice extends PartialOrd {
    This This.meet(This); // covariant and contravariant uses of This
    static This top(); // a static method
    static This meetAll(Iterable[This] c) {
        This glb = top();
        for (This elem : c) { glb = glb.meet(elem); }
        return glb;
    } // a static method with a default implementation
}
```

Figure 5.4. An invariant interface for a lower semilattice

Static and default methods in interfaces. Interfaces may also declare static methods that do not expect a receiver. For example, consider the interface `LowerLattice` in Figure 5.4. It describes a bounded lower semilattice, with its representation type `This` as the carrier set of the semilattice. Therefore, `LowerLattice` extends `PartialOrd` and additionally declares a binary method `meet()` and a static method `top()` that returns the greatest element.

Interfaces can also provide default implementations for methods. Method `meetAll` in `LowerLattice` computes the greatest lower bound of a collection of elements. However, an implementation of `LowerLattice` can override this default with its own code for `meetAll`.

The current interface `Self`. In Familia, all interfaces have access to a `Self` interface that precisely characterizes the current interface in the presence of inheritance. For example, in interface `Set` (Figure 5.3a), `Self` is used as the return type of the `addAll` method, meaning that the return type varies with the interface the method is enclosed within: in interface `Set`, the return type is understood as `Set[E]`; when method `addAll` is inherited into interface `SortedSet`, the return

type is understood as `SortedSet[E]`. Hence, adding all elements of a (possibly unsorted) set into a sorted set is known statically to produce a sorted set:

```
SortedSet[E] ss1 = …;
Set[E] s = …;
SortedSet[E] ss2 = ss1.addAll(s);
```

`This` and `Self` are implicit parameters of an interface, playing different roles. The parameter `This` stands for the representation type, which is instantiated by the implementation of an interface with the type of its representation. On the other hand, the parameter `Self` stands for the current interface, and its meaning is refined by interfaces that inherit from the current interface. Section 5.3.4 further explores the roles of `This` and `Self`.

Interfaces with multiple representation types. An interface can act as a multi-parameter type class if it declares multiple representation types—that is, if it has multiple constraint parameters. As an example, the `Graph` interface (lines 6–11 in Figure 5.5) constrains both `Vertex` and `Edge` types. Note that the implicit constraint parameter of the superinterface `Hashable` is used explicitly here. As seen on lines 7–10, when there are multiple constrained types, the receiver type of each defined operation must be given explicitly. Unlike interfaces with a single representation type, interfaces with multiple representation types do not define an object type.

5.3 Unifying OO classes and type-class implementations

All previous languages that integrate type classes into the OO setting draw a distinction between classes and implementations of type classes. Confusingly, different languages use different terminology to describe type-class implementations.

graphs

```
1 module graphs;
2 static List[List[V]] findSCCs[V,E](List[V] vertices) where Graph(V,E) {
3     … new postOrdIter[V,E](v) …
4     … new postOrdIter[V,E with transpose[V,E]](v) …
5 } // Implements Kosaraju's algorithm for finding strongly connected components
```

graphs.Graph

```
6 interface Graph(Vertex,Edge) extends Hashable(Vertex) {
7     Vertex Edge.source();
8     Vertex Edge.sink();
9     Iterable[Edge] Vertex.outgoingEdges();
10    Iterable[Edge] Vertex.outgoingEdges();
11 }
```

graphs.postOrdIter

```
12 class postOrdIter[V,E] for Iterator[V] where Graph(V,E) {
13    postOrdIter(V root) { … }
14    V next() throws NoSuchElement { … }
15    …
16 }
```

graphs.transpose

```
17 class transpose[Vertex,Edge] for Graph(Vertex,Edge)
18 where Graph(Vertex,Edge) g {
19    Vertex Edge.source() { return this.(g.sink)(); }
20    Vertex Edge.sink() { return this.(g.source)(); }
21    Iterable[Edge] Vertex.outgoingEdges() { … }
22    Iterable[Edge] Vertex.incomingEdges() { … }
23    int Vertex.hashCode() { return this.(g.hashCode)(); }
24 }
```

Figure 5.5. A generic graph module. Interface `Graph` has two constraint parameters, so the method receiver types in interface `Graph` (and also its implementing class `transpose`) cannot be omitted. The code in this graph is discussed in Section 5.3.

```
1 class hashset[E where Hashable] for Set[E] {
2     E[] table;
3     int size;
4     hashset() { table = new E[10]; size = 0; }
5     int size() { return this.size; }
6     boolean contains(E e) { … e.hashCode() … }
7     …
8 }
```

(a) The representation of `hashset` is its fields.

```
1 class mapset[E where Eq] for Set[E](Map[E,?]) {
2   boolean contains(E e) { return this.containsKey(e); }
3   int size() { return this.(Map[E,?].size)(); }
4   …
5 }
```

(b) The representation of `mapset` is a `Map` object.

Figure 5.6. Two implementations of the `Set` interface using different representations

Haskell has "instances", the various C++ proposals have "concept maps" [165], JavaGI [182] has "implementations", and Genus [190] has "models".

Familia avoids unnecessary duplication and terminology by unifying classes with type-class implementations. Classes establish the ability of an underlying representation to satisfy the requirements of an interface. The representation may be a collection of fields, in the usual OO style. Alternatively, unlike in OO style, the representation can be any other type that is to be retroactively adapted to the desired interface.

For example, to implement the interface `Set` (Section 5.2), we can define the class `hashset` shown in Figure 5.6a. Class `hashset` implicitly instantiates the representation type `This` of its interface `Set[E]` as a record type comprising its field types (i.e., `{E[] table; int size}`). Since `this` denotes the receiver and the receiver has this representation type, the field access on line 5 type-checks.

The receiver type of a class method is usually omitted when the class has a single representation type.

Classes are not types. If classes were types, code using classes would be less extensible because any extension would be forced to choose the representation type in a compatible way. We give classes lowercase names to emphasize that they are more like terms. A class can be used via its constructors; for example, `new hashset[E]()` produces a new object of type `Set[E]`.

A distinguishing feature of Familia is that a class can also instantiate its representation type explicitly, effectively *adapting* an existing type or types to an interface. Suppose we already had an interface `Map` and wanted to implement `Set` in terms of `Map`. As shown in Figure 5.6b, this adaptation can be achieved by defining a class that instantiates the representation type of `Set[E]` as `Map[E,?]`. Class `mapset` implements the `Set` operations by redirecting them to corresponding methods of `Map`. Note that the value type of the map does not matter, hence the wildcard `?`. Because expression `this` has type `Map[E,?]` in class `mapset`, the method call `this.containsKey(e)` on line 2 type-checks, assuming `Map` defines such a method.

A class like `mapset` has by default a single-argument constructor that expects an argument of the representation type. So an object `x` of type `Map[K,V]` can be used to construct a set through the expression `new mapset[K](x)`, an expression with type `Set[K]`. It is also possible to define other constructors to initialize the class' representation.

Classes can be extended via inheritance. A subclass can choose to implement an interface that is a subtype of the superclass interface, but cannot alter the representation type to be a subtype, which would be unsound. The fact that a

subclass can add extra fields is justified by treating the representation type of a class with fields as a nested type (Section 5.4.2).

5.3.1 Classes as Witnesses to Constraint Satisfaction

The ability of classes to adapt types to interfaces makes generic programming more expressive and improves checking that it is being used correctly. In particular, the Familia type system keeps track of which class is used to satisfy each type constraint. For example, suppose we want sets of strings that are unique up to case-insensitivity; we would like a `Set[String]` where string equality is defined in a case-insensitive way. Because interface `String` has an `equals` method

```
interface String { boolean equals(String); … }
```

it automatically structurally satisfies the constraint `Eq(String)` that is required to instantiate `Set`. However, it satisfies that interface in the wrong, case-sensitive way. We solve this problem in Familia by defining a class that rebinds the necessary methods:

```
class cihash for Hashable(String) {
  boolean equals(String s) { return equalsIgnoreCase(s); }
  int hashCode() { return toLowercase().hashCode(); }
}

Set[String with cihash] s1 = new hashset[String with cihash]();
Set[String] s2 = s1; // illegal
```

Notice that the types in this example keep track of the class being used to satisfy the constraint, a feature adopted from Genus [190]: a `Set[String]`, which uses `String` to define equality, cannot be confused with a `Set[String with cihash]`. Such a confusion might be dangerous because the implementation of `Set` might rely on the property that two sets use the same notion of equality.

The type `Set[String]` does not explicitly specify a class, so to witness the constraint `Eq(String)` for it, Familia infers a default class, which in this case is a *natural class* that Familia automatically generates because `String` structurally conforms to the constraint `Eq`. The natural class has an `equals` method that conforms to that required by `Eq(String)`, and its implementation simply calls through to the underlying method `String.equals`. We call these "natural classes" by analogy with natural models [190].

The natural class can also be named explicitly using the name `String`, so `Set[String]` is actually a shorthand for `Set[String with String]`. However, the natural class denoted by `String` is different from the `String` type.

We've seen that a class can be used both to construct objects and to witness satisfaction of `where`-clause constraints. In fact, even object construction is really another case of using a class to witness satisfaction of a constraint. For example, creating a `Set[E]` object needs both a value of some representation type T and a class that satisfies the constraint `Set[E]`(T). The job of a class constructor is to use its arguments to create a value of the representation type.

5.3.2 Classes as Dispatchers

Like other OO languages, Familia uses classes to obtain dispatch information. Unlike previous OO languages, Familia allows methods to be dispatched using classes that are *not* the receiver object's class.

The general form of a method call is

$$e_0.(d.\mathtt{m})(e_1, ..., e_n)$$

where e_0 is the receiver, class d is the *dispatcher*, and $e_1, ..., e_n$ are the ordinary method arguments. Dispatcher d provides the method `m` being invoked; the receiver e_0 must have the same type as the representation type of d.

This generalized notion of method invocation adds valuable expressive power. For an example, return to class `transpose` in Figure 5.5. On line 18, its `where`-clause constraint, named `g`, denotes some class for `Graph(Vertex,Edge)`. Class `transpose` implements the transpose of the graph defined by `g` by reversing all edge orientations in `g`; for example, method call `this.(g.sink)()` on line 19 uses class `g` to find the sink of a vertex and returns it as the source. Note that the transposed graph is implemented *without* creating a new data structure. On lines 2–5, the method `findSCCs()` demonstrates one use for the transposed graph. It finds strongly connected components via Kosaraju's algorithm [6], which performs two postorder traversals, one on the original graph and one on the transposed graph.

Dispatcher classes can usually be elided in method calls—Familia infers the dispatcher for an ordinary method call of form e_0`.m(`e_1, ..., e_n`)`—offering convenience in the common cases where there is no ambiguity about which dispatcher to use. This inference process handles method invocation for both object-oriented polymorphism and constrained parametric polymorphism.

In the common case corresponding to object-oriented polymorphism, this ordinary method call represents finding a method in the dispatch information from e_0's *own* class. For example, assuming `s1` and `s2` have type `String`, the method call `s1.equals(s2)` is syntactic sugar for using the natural class implicitly generated for `String` as the dispatcher: it means `s1.(String.equals)(s2)`. The natural class generated for an interface `I` actually implements the constraint `I(I)`. So the `equals` method in the natural class for `String` must have receiver type `String` and argument type `String`. Hence the expanded method call above type-checks.

In another common case corresponding to constrained parametric polymorphism, ordinary method call syntax may be employed by generic code to invoke operations promised by constraints on type parameters. For example, consider the method call `e.hashCode()` on line 6 in class `hashset` (Figure 5.6a). Familia infers the dispatcher to be the class passed in to satisfy the `where`-clause constraint `Hashable(E)`. So if the programmer named the constraint as in "`where Hashable(E) h`", the desugared method call would be `e.(h.hashCode)()`.

The generalized form of method calls provides both static and dynamic dispatch. While the dispatcher class is chosen statically, the actual method code to run is chosen dynamically from the dispatcher class. It is easy to see this when the dispatcher is a natural class; the natural class uses the receiver's own class to dispatch the method call. An explicit class can also provide specialized behavior for subtypes of the declared representation type; all such methods are dispatched dynamically based on the run-time type of the receiver.

In fact, class methods are actually multimethods that dispatch on all arguments whose corresponding types in the interface signature are `This`. Since Familia unifies classes with type-class implementations, the semantics of multimethods in Familia is the same as model multimethods in Genus [190]. For example, class `setPO` in Figure 5.7 implements a partial ordering for `Set` based on set containment. It has two `leq` methods, one for the base case, and the other for two sorted sets. Notice that in the second `leq` method, the receiver and the parameter are guaranteed to have the same ordering, because both occurrences of `SortedSet` indicate a subtype of the same `Set[E with eq]` type, assuming the `where`-clause constraint is named `eq`. Hence, the second `leq` method can be implemented in an asymptotically more efficient way. When class `setPO` is used

```
class setPO[E where Eq] for PartialOrd(Set[E]) {
    boolean Set[E].leq(Set[E] that) {        // base implementation
        return this.containsAll(that);
    }
    boolean SortedSet.leq(SortedSet that) { ... }   // specialization
    ...
}
```

Figure 5.7. The `leq` methods in class `setPO` are multimethods. The second `leq` method offers an asymptotically more efficient implementation for two sets sorted using the *same* order.

to dispatch a method call to `leq`, the most specific version is chosen based on the run-time types of the receiver and the argument.

5.3.3 Inferring Default Classes

As mentioned in Sections 5.3.1 and 5.3.2, Familia can infer default classes both for elided `with` clauses and for elided dispatchers. It does so based on how classes are *enabled* as potential default choices, similarly to how models are enabled in Genus [190]. If only one enabled class works in the `with` clause or as the dispatcher, that class is chosen as the default class. Otherwise, Familia requires the class to be specified explicitly.

Classes can be enabled in four ways: 1. Types automatically generate and enable natural classes. 2. A `where`-clause constraint enables a class within its scope. 3. A `use`-statement enables a class in the scope where the `use`-statement resides; for example, the statement "`use mapset;`" enables this class as a way to adapt `Map[E,?]` to `Set[E]`. 4. A class is enabled within its own definition; this enclosing class can be accessed via the keyword `self`.

For example, consider the method call on line 3 in class `mapset` (Figure 5.6b). If the dispatcher were elided, two classes would be enabled as potential dispatcher choices—one, the natural class generated for the receiver's type (i.e., `Map[E,?]`, which has a `size()` method), and the other, the enclosing class `mapset`, which defines a `size()` method with a compatible receiver type. Because of this ambiguity, Familia requires the dispatcher to be specified explicitly.

As another example, consider the method call `glb.meet(elem)` on line 6 in Figure 5.4. An enclosing class in this case is the class that implements the `LowerLattice` interface (this class inherits the method definition that contains this method call), and only this class is enabled as a potential dispatcher for the call. So Familia desugars the method call as `glb.(self.meet)(elem)`.

5.3.4 `Self`, `This`, `self`, and `this`

As discussed in Section 5.2, an interface definition has access to both the `Self` interface and the `This` type: `Self` is the current interface, while `This` is the type of the underlying representation of the current interface. Analogously, a class definition (as well as a non-static default method in an interface) has access to both the `self` class and the `this` term: `self` denotes the current class, while `this` denotes the underlying representation of the current class (or equivalently, the receiver). A class definition also has access to a `Self` interface that denotes the interface of the current class `self`. Hence, class `self` witnesses the constraint `Self(`T`)` where T is the type of `this`.

Although they sound similar, `Self` (or `self`) and `This` (or `this`) serve different purposes. Both `Self` and `self` are late-bound: in the presence of inheritance, their interpretation varies with the current interface or class. In this sense, `Self` and `self` provide a typed account of interface and class extensibility. Section 5.4

shows how the `self` class is further generalized to support type-safe extensibility at larger scale. On the other hand, the representation type `This` and the representation `this` provide *encapsulation*—objects hide their representations and object types hide their representation types—and *adaptation*—classes adapt their representations to interfaces.

Ignoring nesting (Section 5.4), objects in Familia are closest from a type-theoretic viewpoint to the denotational interpretation of objects by Bruce [29], who gives a denotation of an object type using two special type variables: the first represents the type of an object as viewed from the inside, and the second, the type of the object once it has been packed into an existential. These type variables roughly correspond to `This` and `Self` in Familia. This denotational semantics is intended as a formal model for OO languages, but no later language distinguishes between these two types. Familia shows that by embracing this distinction in the surface language, the same underlying model can express both object types and type classes.

5.3.5 Adaptive Use-Site Genericity

Familia further extends the adaptive power of interfaces to express use-site genericity ala Genus [190]. The adaptive power arises from the duality between use-site genericity and definition-site genericity (i.e., parametric polymorphism), which respectively correspond to existential and universal type quantification. Because of this adaptive power, we call this "use-site genericity" rather than "use-site variance", which only concerns subtyping [173].

Java uses wildcards and subtyping constraints to express use-site variance. For example, the type `List<? extends Set<E>>` describes all lists whose element type is a subtype of `Set<E>`. The corresponding type in Familia is

```
List[out Set[E]]
```

which is sugar for the explicit existential type

```
[some T where Set[E](T)]List[T]
```

where `Set[E]` is used to constrain the unknown type `T`, and the leading brackets denote constrained existential quantification. Therefore, one can assign a `List[SortedSet[E]]` to a `List[out Set[E]]` because the natural class generated for `SortedSet[E]` satisfies the constraint `Set[E](SortedSet[E])`. More interestingly, one can assign a `List[Map[E,?]]` to a `List[out Set[E]]` in a context in which class `mapset` (Figure 5.6b) is enabled. Note that this adaptation is asymptotically more efficient than with the Adapter pattern: when assigning a `List[Map[E,?]]` into a `List[out Set[E]]`, the resulting list is represented by a list of maps and a class that adapts `Map` to `Set`, rather than by wrappers that inefficiently wrap each individual map into a set.

5.4 Evolving families of classes and interfaces

Thus far we have seen how to unify OO classes and interfaces with type classes and their implementations. However, the real payoff comes from further unifying these mechanisms with family polymorphism, to support *coordinated* changes to related classes and types contained within a larger module.

5.4.1 Overview

As in many OO languages, Familia's module mechanism is based on nesting: classes and interfaces are modules that can contain classes, interfaces, and types. Familia has nesting both via classes and via a pure module construct that is

analogous to packages in Java or namespaces in C++. Apart from being able to span multiple compilation units, such a module is essentially a degenerate class that has no instances and does not implement an interface. Hence, both classes and modules define families containing their components. Familia interfaces can also contain nested components; a class inherits the nested components from its interfaces. Since nesting is allowed to arbitrary depth, a nested component may be part of multiple families at various levels.

Unlike most OO languages, Familia allows not only classes but all modules to be extended via inheritance. Following C++ convention [164], we call the module being extended the *base module* and the extending module, the *derived module*. Hence, superclass and base class are synonyms, as are subclass and derived class. We also slightly abuse the terminology "module" to mean not only the module construct but all families that contain nested components.

When a module is inherited, all components of the base module—including nested modules, classes, interfaces, types,[2] and methods—are inherited into the derived module; the inherited code is polymorphic with respect to a family it is nested within. Further, the derived module may *override* the nested components. In this sense, names of components nested inside a module are *implicit parameters* declared by their families.

Example: dataflow analysis. As an example where coordinated changes to a module are useful, consider the problem of developing an extensible framework for dataflow analysis. A dataflow analysis can be characterized as a four-tuple (G, I, L, F_n) [7]: the direction G that items flow on the control-flow graph (CFG), the set I of items being propagated, the operations $\sqcap$ and $\top$ of the semilattice L

[2]Nested interfaces are similar to nested types except that nested interfaces can be used to constrain types and that nested types need not necessarily be bound to interfaces.

```
dataflow.base
abstract module dataflow.base {
  class cfg for Graph(Peer,Edge);
  type Item;
  class itemlat for LowerLattice(Item);
}

dataflow.base.transfer
class transfer for Function[Item,Item](Node) {
  Item Node.apply(Item item) { return item; }
}

dataflow.base.worker
use cfg, itemlat;
class worker for Worker {
  Map[Peer,Item] items; // analysis result
  …
  void worklist(List[Peer] src) {
    List[List[Peer]] sccs = graphs.findSCCs[Peer,Edge](src);
    for (List[Peer] scc : sccs) {
      boolean change = false;
      do { // Iteratively computes result
        for (Peer p : scc) {
          Item newf = outflow(p);
          Item oldf = items.get(p);
          change |= !oldf.equals(newf);
          items.put(p,newf);
        }
      } while (change);
    }
  }
  Item outflow(Peer p) {
    Item conf = itemlat.meetAll(inFlows(p));
    return p.node().(transfer.apply)(conf);
  }
  List[Item] inFlows(Peer p) { … }
}
```

Figure 5.8. Excerpt from an extensible dataflow analysis framework. (A `Peer` is a vertex in the CFG, and has access to an AST node, `Node`.)

```
dataflow.liveness
module dataflow.liveness extends dataflow.base {
    // Backward analysis
    class cfg extends transpose[Peer,Edge with flowGraph];
    // Liveness analysis propagates sets of variables
    type Item = Set[Var];
}
```

```
dataflow.liveness.transfer
// Def-Use
class transfer for Function[Item,Item](Node) {
    Item LocalVar.apply(Item item) {
        return item.add(this.var());
    }
    Item LocalAssign.apply(Item item) {
        LocalVar n = this.left();
        return item.remove(n.var());
    }
}
```

```
dataflow.liveness.itemlat
class itemlat for LowerLattice(Item) {
    static Item top() fixes Item { return new hashset[Var](); }
    Item meet(Item that) { return this.addAll(that); }
    boolean leq(Item that) { return this.(setPO[Var].leq)(that); }
    boolean equals(Item that) { return this.(setPO[Var].equals)(that); }
}
```

Figure 5.9. An extension of the base dataflow framework: live variable analysis

formed by the items, and the transfer functions F_n associated with each type of AST node.

We would like to be able to define a generic dataflow analysis framework that leaves the four parameters either unspecified or partially specified, so that a specific analysis task (such as live variable analysis) can be obtained by instantiating or specializing the parameters. This can be achieved using family polymorphism,

but does not work with conventional OO languages since they do not support coordinated changes, as discussed in Section 5.1.

Figure 5.8 shows the code of the module `base`, which is nested inside the module `dataflow`. It provides a base implementation of the extensible dataflow analysis framework discussed earlier. The four parameters (G, I, L, F_n) of a dataflow analysis framework correspond to class `cfg`, type `Item`, class `itemlat`, and class `transfer`, respectively. The rest of the module—especially class `worker`, which implements the worklist algorithm (lines 5–23)—is generic with respect to the choice of these parameters. Therefore, writing a specific dataflow analysis amounts to instantiating these four parameters in a derived module. Crucially, this instantiation can be done in a lightweight, type-safe way by either binding or further-binding the four nested components.

To illustrate such an extension, Figure 5.9 shows a module that inherits from the base dataflow module and implements live variable analysis. Recall that live variable analysis is a backward analysis where the items are sets of local variables, the meet operator $\sqcap$ takes the union of two sets, the greatest element $\top$ is the empty set, and the transfer functions are defined in terms of the variables each CFG node defines and uses. The module definition (`liveness`) declares that it extends the `base` module. It provides the definitions of exactly these four implicit parameters.

The combined power of family inheritance and retroactive adaptation is apparent: with roughly 20 lines of straightforward code, we are able to implement a new program analysis. And this analysis is itself extensible; for example, it can be further extended to report unused variables. Implementing this example with the same extensibility in any previous language would require more boilerplate code or design patterns.

5.4.2 Further Binding

In Familia, all nested names are points of extensibility that can be further-bound in derived modules. In addition, base modules, superclasses, superinterfaces, supertypes, and interfaces of classes can be further-bound in derived modules.

Binding nested, unbound names. A derived module can *bind* the nested components left unbound in its base module. In the example, derived module `liveness` binds three components unbound in module `base`: it binds nested class `cfg` by using the transpose of `flowGraph` (we assume `flowGraph` is a class for constraint `Graph(Peer,Edge)`) as its superclass, it binds nested type `Item` to type `Set[Var]`, and it binds nested class `itemlat` to a nested class definition. A module can nest unbound methods, classes, or types if it is abstract or one of its families is abstract;[3] module `base` is declared abstract.

Unbound classes are not to be confused with abstract classes. An unbound class is one that a non-abstract, derived module of one of its enclosing families must provide a binding for. So unlike abstract classes, unbound classes *can* be used to satisfy constraints (including creating objects) and dispatch method calls. For example, consider the `worklist()` method in the `base` module, which computes the strongly connected components of the CFG (line 6) to achieve faster convergence [129]. Class `cfg`, though unbound, is used (as the default class) to satisfy the constraint required by the generic `findSCCs()` method from Figure 5.5. As another example, unbound class `itemlat` is used as the (inferred) dispatcher in the method call on line 13.

It is perfectly okay to give partial definitions to nested classes `cfg` and `itemlat` without declaring them abstract, because a non-abstract, derived module of their

[3] "Abstract methods" in previous OO languages actually mean unbound methods in Familia, while "abstract classes" retain their meaning.

family is required to complete the definitions. (Class `itemlat` is indeed partially defined because it inherits a default method implementation from its interface defined in Figure 5.4.) It follows that the above discussion about unbound classes applies to partially bound classes as well. Previous languages with family polymorphism do not support non-abstract classes that are unbound or partially bound.

Nested type `Item` is essentially an *associated type* of its family. Associated types are unbound type members of interfaces and superclasses [42, 95].[4] Associated types were introduced to reduce the number of type parameters, a rationale that applies here as well. However, unlike languages like Scala and Haskell that support associated types, it is possible in Familia to further-bind a previously bound nested name in a derived module.

Further-binding nested names. In Familia, a derived module can *further-bind* nested interface and class definitions, *specializing* and *enriching* the behavior of the corresponding definition in the base module. Further binding was introduced by Beta [113]. Languages that support virtual types but not full family polymorphism, such as Scala, can only simulate further binding through design patterns [134, 183].

In the dataflow example, class `transfer` in the derived module further-binds its counterpart in the base module. Recall from Section 5.3.2 that a class can provide specialized behavior for subtypes of the representation type and that all such methods are dynamically dispatched. The implementation of the transfer functions demonstrates how family inheritance interacts with this feature. The `transfer` class in the base module provides a default implementation of a transfer

[4] Associated types take different forms in different languages: `typedef` in C++, abstract type members in Scala, `associatedtype` in Swift, to name a few.

function through the method `apply`. The `transfer` class in the derived module `liveness` inherits this default implementation, and refines it by specializing the implementation of `apply` to handle the particular types of AST nodes that play an interesting role in live variable analysis. Dispatching on the receiver object is used to allow choosing the most specific method among all of the three `apply` methods at run time.

In addition to specialization, a further-bound interface or class definition can also enrich the corresponding definition in the base module. This enrichment was seen in the example from Section 5.1, in which the class `Node` and implicitly, all of its derived classes, were extended with a new method `constFold`. This example works in Familia as well. If `Node` were an interface rather than a class, Familia would check that every class implementing it in the derived module provides a definition for method `constFold`.

A nested type can also be further-bound to a subtype. In fact, further-binding is what allows subclasses to add new fields. Recall from Section 5.3 that the representation type of a class with fields is a record type containing the field types. We take this unification a step further with nested types: field types are essentially a nested record type that can be further-bound to a subtype. For example, consider classes `c1` and `c2` in Figure 5.10, whose representation types are a nested type. Class `c2` adds a new field `g` by further-binding the nested type `Fields` to a subtype. Since `this` has type `Fields` in class `c2`, the nested fields can be accessed via `this.f` and `this.g`.

Further-binding base modules. In Familia, not only can nested names be further-bound, but base modules, superclasses, superinterfaces, and interfaces that classes implement can also be further-bound. The utility of further-binding

```
interface I { int This.m(); }

class c1 for I(Fields) {
    type Fields = {int f}
    int Fields.m() {return this.(self.n)();}
    int Fields.n() {return this.f;}
}

class c2 for I(Fields) extends c1 {
    type Fields = {int f; int g}
    int Fields.n() {return this.f + this.g;}
}

// Testing code
{int f} t1 = {f = 1};
{int f; int g} t2 = {f = 0; g = 2}
I i = new c2(t2);
i.(I.m)();      // dispatcher is the natural class
t2.(c2.m)();    // dispatcher is c2
t1.(c2.m)();    // illegal
```

Figure 5.10. Classes `c1` and `c2` have fields as their representations (Section 5.4.2). The testing code illustrates how late binding ensures type safety (Section 5.4.3). Receiver types in method signatures and dispatcher classes in method calls are written out in this example.

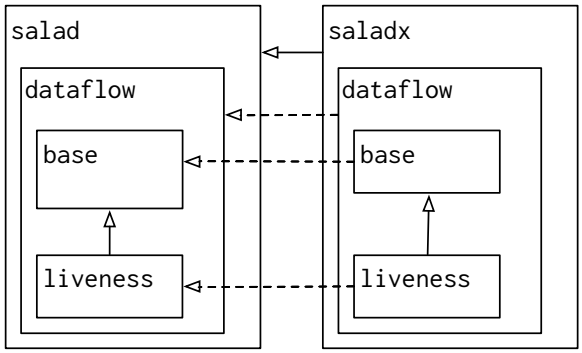
base modules can be demonstrated by the new opportunity in Familia to co-evolve related, non-nested families of classes and interfaces.

For example, suppose the dataflow framework in Figures 5.8 and 5.9 was developed for the Salad programming language from Section 5.1. Because the Saladx extension in Figure 5.1 adds unary expressions to Salad, we cannot expect the dataflow framework to automatically work correctly for Saladx. If the `dataflow` module happened to be nested within the `salad` module, family polymorphism with further binding would allow us to add a transfer function for unary expressions in class `saladx.dataflow.liveness.transfer` (which would

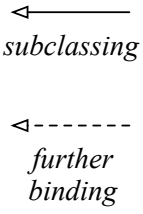
further-bind class `salad.dataflow.liveness.transfer`). This approach is illustrated in Figure 5.11a. Suppose, however, that the dataflow framework were implemented separately by a third party, and thus had to import the module `salad` rather than residing within it. The extensibility strategy just outlined would not work.

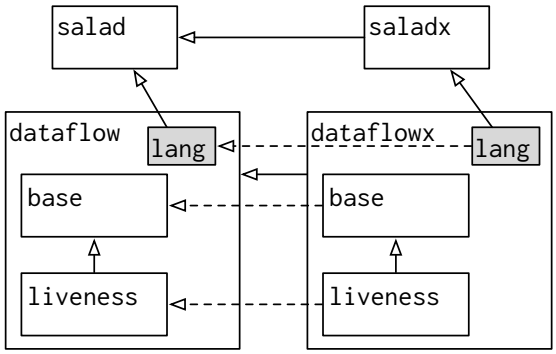
This need to co-evolve related families is addressed by further-binding a base module. Figure 5.11b illustrates how to co-evolve the dataflow framework and the Salad implementation, and Figure 5.12 shows the code. In Figure 5.12a, module `dataflow`, the dataflow framework for the base Salad language, declares a nested name `lang` and binds it by using `salad` as its base module. In Figure 5.12b, derived module `dataflowx`, the dataflow framework for Saladx, further-binds the base module of `lang` to `saladx`, and updates the transfer function to account for unary expressions. (For brevity, we also say that module `dataflow` binds nested name `lang` to `salad` and that module `dataflowx` further-binds it to `saladx`.) As we soon see in Section 5.4.3, the `dataflow` and `dataflowx` modules interpret names imported from the `salad` and `saladx` modules (e.g., `Node`) relative to their own nested component `lang`. This relativity ensures that the relationships between the related components in modules `salad` and `dataflow` are preserved when inherited into derived modules `saladx` and `dataflowx`; components of module `salad` cannot be mixed with components of module `dataflowx`, which work with `saladx`.

Familia supports this kind of further binding for other kinds of nested components than modules. For example, when a derived module further-binds the superclass of a nested class, Familia checks that the new superclass extends—up to transitivity and reflexivity—the original superclass. This check ensures that inherited code is type-safe with respect to the new binding.



(a) Module `dataflow` is nested within module `salad`, and is further-bound by the `dataflow` module in `saladx`. This approach requires module `dataflow` to be nested within module `salad`.





(b) Module `dataflow` imports `salad` by binding the base module of nested module `lang` to it. Derived module `dataflowx` further-binds the base module to `saladx`.

Figure 5.11. Two approaches to co-evolving modules `dataflow` and `salad`

5.4.3 Late Binding of Nested Names via `self`

A key to making family polymorphism sound is that relationships between nested components within a larger module are preserved when inherited into a derived module. Familia statically ensures that relationships between nested components are preserved through *late binding* of all nested names. To see what would go wrong without late binding, consider the using code in Figure 5.10. Without late binding of the nested name `Fields`, the last method call would type-check because the `m()` in `c2` is inherited from `c1` and its receiver type would

```
dataflow
module dataflow {
    module lang extends salad;
}
```

(a) Dataflow analysis for Salad

```
dataflowx
module dataflowx extends dataflow {
    module lang
    extends saladx;
}
dataflowx.liveness.transfer
class transfer for Function[Item,Item](Node) {
    Item UnaryExpr.apply(Item item) {
        return item.remove(this.var());
    }
}
```

(b) Dataflow analysis for extended Salad.

Figure 5.12. Evolving the `dataflow` module in accordance with the extension to Salad, using the approach illustrated by Figure 5.11b

be `{ int f }` instead of the late-bound `Fields`. This would result in a run-time error as `t1` does not have the `g` field.

Recall from Section 5.3.3 that Familia uses the keyword `self` to represent the current class or module. By adding a qualifier, `self` can also be used to refer to any enclosing class or module. Names that are not fully qualified are interpreted in terms of an enclosing class or module denoted by a `self`-prefix. For example, consider the mentions of the unqualified name `Fields` in class `c1` of Figure 5.10. Here `Fields` is syntactic sugar for the qualified name `self[c1].Fields`, where the prefix `self[c1]` refers to the enclosing class that is, or inherits from `c1`. When the code is inherited into class `c2`, the inferred `self`-prefix becomes `self[c2]`, a class that is, or inherits from `c2`. The unqualified name `Fields` then has a

new meaning in class `c2`: `self[c2].Fields`. In the using code in Figure 5.10, when class `c2` is used to invoke method `m`, the method signature is obtained by substituting the dispatcher for the `self` parameter in `int self[c2].Fields.m()`. This substitution suggests that the receiver should have type `{ int f; int g }`, so the last method call `t1.(c2.m)()` is rejected statically.

The further binding of the base module of nested name `lang` offers another example. The `transfer` class of the base dataflow module (Figure 5.8) mentions `Node`, defined by module `salad.ast`. Because the enclosing module `dataflow` binds `lang` to `salad` (Figure 5.12a), `Node` is expanded to `lang.ast.Node`. Further desugaring the mention of `lang` yields `self[dataflow].lang.ast.Node`. Hence, in the derived module `dataflowx` (Figure 5.12b), the unqualified name `Node` is reinterpreted as `self[dataflowx].lang.ast.Node`. Similarly, module `dataflowx` interprets the unqualified name `Stmt` as `self[dataflowx].lang.ast.Stmt`, so the subtyping relationship between `Stmt` and `Node` is preserved.

Importantly, late binding in Familia supports separate compilation with modular type checking—existing code need not be rechecked or recompiled when inherited into derived modules. For example, derived module `liveness` inherits method `outflow()` (lines 19–22 of Figure 5.8), which takes the meet of all incoming flows of a node and computes the outgoing flow by applying the transfer function. The call to `apply()` on line 21 in the base module need not be rechecked in a derived module, say `dataflowx.liveness`, which interprets the receiver type as `self[dataflowx].lang.ast.Node`, the formal parameter type `self[dataflowx.liveness].Item`, and the dispatcher class `self[dataflowx.liveness].transfer`. This guarantees that the method can only be invoked using arguments and dispatcher that are compatible.

It is occasionally useful to locally turn off late binding. Consider implementing the static `top()` method in class `liveness.itemlat` (Figure 5.9). We would like to create a new `hashset[Var]` and return it. However, it would not type-check because the return value would have type `Set[Var]`, which is different from the expected, late-bound return type `Item`. This issue is addressed in a type-safe and modular way by the `fixes`-clause in the method signature (line 2). A `fixes`-clause is followed by a late-bound name. In this case, the clause `fixes Item` allows the method to equate types `Item` and `Set[Var]` so that the `return` statement type-checks. To ensure type safety, the `fixes`-clause also forces a derived module of `liveness` (or a family thereof) to override `top()` if it further-binds its nested type `Item` to a different type than `Set[Var]`.

Because `self`-qualifiers can be omitted and inferred, family polymorphism becomes transparent to the novice programmer. And code written without family polymorphism in mind can be extended later without modification.

5.5 A core language

To make the semantics of the unified polymorphism mechanism more precise, we define a core language called Featherweight Familia(abbreviated F^2), capturing the key aspects of Familia.

5.5.1 Syntax and Notation

F^2 follows the style of Featherweight Java [94], and makes similar assumptions and simplifications. F^2 omits a few convenient features of Familia: uses of nested names are fully expanded, the class used by a method call or a `with`-clause is always explicit, and natural classes are encoded explicitly rather than generated

programs	$\mathcal{P}$	::= $\diamond\{\overline{\mathcal{I}}\ \overline{C}\ e\}$
class definitions	C	::= `class` $\mathtt{c}[\varphi]$ `for` $Q(T)$ `extends` $P\ \{\overline{\mathcal{I}}\ \overline{C}\ \overline{\mathcal{M}}\}$
interface definitions	$\mathcal{I}$	::= `interface` $\mathtt{I}^{\nu}[\varphi]$ `extends` $Q\ \{\overline{\mathtt{m}:S}\}$
method definitions	$\mathcal{M}$	::= $\mathtt{m} : S\ (\overline{\mathtt{x}})\ \{e\}$
parameterization	$[\varphi]$	::= $[\,\overline{\mathtt{X}}$ `where` $\overline{\mathtt{p}\ \mathtt{for}\ H(T)}\,]$
instantiation	$[\omega]$	::= $[\,\overline{T}$ `with` $\overline{d}\,]$
interface variance	ν	::= $-$ | 0
method signatures	S	::= $[\varphi]\,\overline{T_1} \to T_2$
types	T	::= `int` | $\mathtt{X}$ | H
interface paths	H	::= Q | $Q^!$
inexact class paths	P	::= $P^!.\mathtt{c}[\omega]$
inexact interface paths	Q	::= $P^!.\mathtt{I}[\omega]$
exact class paths	$P^!$	::= $\diamond$ | $P^!.\mathtt{c}^![\omega]$ | `self`$[P]$
exact interface paths	$Q^!$	::= $P^!.\mathtt{I}^![\omega]$ | `Self`$[Q]$ | $d.\mathtt{itf}$
dispatchers	d	::= $P^!$ | $\mathtt{p}$
expressions	e	::= n | $\mathtt{x}$ | `pack` (e, d) | `unpack` e_1 `as` $(\mathtt{x}, \mathtt{p})$ `in` e_2 | $e_0.(d.\mathtt{m}[\omega])(\overline{e_1})$
class names	$\mathtt{c}$	
interface names	$\mathtt{I}$	
method names	$\mathtt{m}$	
class variables	$\mathtt{p}$	
type variables	$\mathtt{X}$	
term variables	$\mathtt{x}$	

Figure 5.13. Featherweight Familia: Syntax

implicitly. For simplicity, F^2 does not model certain features of Familia whose formalization would be similar to that in Featherweight Genus [191]: interfaces with multiple constraint parameters, multimethods, and use-site genericity.

Figure 5.13 presents the syntax of F^2. An overline denotes a (possibly empty) sequence or mapping. For example, $\overline{\mathtt{X}}$ denotes a sequence of type variables,

$\overline{\texttt{m}:S}$ denotes an (unordered) mapping that maps method names $\overline{\texttt{m}}$ to method signatures $\overline{S}$, and $\varnothing$ denotes an empty sequence or mapping. The i-th element in $\overline{\bullet}$ is denoted by $\bullet^{(i)}$. To avoid clutter, we write $[\varphi]$ to denote a bracketed list of type variables and `where`-clause constraints, and $[\omega]$ to denote the arguments to these parameters. A `where`-clause constraint in $[\varphi]$ is explicitly named by a class variable `p`. Substitution takes the form $\bullet\{\bullet/\bullet\}$, and is defined in the usual way. We introduce an abbreviated notation for instantiating parameterized abstractions: $\bullet\{\omega/\varphi\}$ substitutes the types and classes in $[\omega]$ for their respective parameters in $[\varphi]$. Type variables, term variables, and class variables are all assumed distinct in any environment. Type variables `X` include `This`, the implicit constraint parameter of all interfaces. Term variables `x` include `this`. We use R^* to mean the reflexive, transitive closure of binary relation R.

A program $\mathcal{P}$ comprises interface and class definitions ($\mathcal{I}$ and C) and a "main" expression. A class definition can contain its own nested classes, interfaces, and methods. An interface definition has the implicit representation type `This`, and its variance with respect to `This` is signified by v. All classes implement some interface in F^2, although they do not have to in Familia. Familia supports multiple interface inheritance and nested `type` declarations, permits interfaces to contain components other than nested methods, and infers interface variance. For simplicity, F^2 does not model these features. In F^2, classes do not have explicit fields because field types are essentially a nested record type (Section 5.4.2) and because record types can be simulated by interface types.

A *class path* represents the use of a class. Class paths have exactness, a notion that is also seen in previous approaches to type-safe extensibility (e.g., Bruce and Foster [30], Bruce et al. [31], Nystrom et al. [131, 132], Ryu [156]). An *exact class path* $P!$ denotes a particular class, while an *inexact class path* P abstracts over all

of its subclasses (including itself). Inexact class paths are of the form $P!.\texttt{c}[\omega]$ and can be used in `extends`-clauses in class headers as superclasses. Similar to Featherweight Java, F^2 assumes the well-foundedness condition that there are no cycles in inheritance chains, as well as the existence of a distinguished, universal superclass `empty`.

An exact class path $P!$ may take one of the following forms:

(1) $\diamond$, denoting the program $\mathcal{P}$ that nests everything;

(2) $P!.\texttt{c}![\omega]$, denoting class $\texttt{c}[\omega]$—not including a subclass thereof—nested within $P!$; or

(3) $\texttt{self}[P]$, denoting an enclosing class that must either be P, extend P, or further-bind P.

For example, in a class definition named $\texttt{c}_2$ nested within class definition $\texttt{c}_1$ which is nested within $\mathcal{P}$, the current class $\texttt{c}_2$ is referred to as $\texttt{self}[\texttt{self}[\diamond.\texttt{c}_1].\texttt{c}_2]$. Familia uses the lighter syntax $\texttt{self}[\texttt{c}_1.\texttt{c}_2]$ to denote this path, or even just `self` if $\texttt{c}_2$ is the immediately enclosing class, but the heavier syntax in F^2 makes it straightforward to perform substitution for outer `self`-parameters like $\texttt{self}[\diamond.\texttt{c}_1]$. Some paths with valid syntax cannot appear in F^2 programs: for example, the path $\texttt{self}[\diamond.\texttt{c}_1!.\texttt{c}_2]$. Nevertheless, the static semantics may create such paths to facilitate type checking. Given an inexact class path $P_1 = P_2!.\texttt{c}[\omega]$, we use $P_1!$ to mean the exact class path $P_2!.\texttt{c}![\omega]$.

Although F^2 requires explicit exactness annotations (i.e., !), they are usually not needed in Familia. The exactness of certain uses of classes is obvious and thus inferred: class paths used in `extends`-clauses are inexact, but class paths used in `with`-clauses, `pack`-expressions, and as dispatchers in method calls are all exact.

An *interface path H* represents the use of an interface. Like class paths, interface paths can be exact (Q!) or inexact (P!.I[ω]). Inexact paths can be used in `extends`-clauses in interface headers and `for`-clauses in class headers. The distinguished interface `Any` is the universal superinterface.[5]

An exact interface path Q! may take one of the following forms:

(1) P!.I![ω], denoting interface I[ω]—not including a subinterface thereof—nested within P!;

(2) Self[Q], an enclosing interface that must either be Q, extend Q, or further-bind Q; or

(3) d.itf, the interface implemented by dispatcher d.

For example, in the class definition of $\mathtt{c}_2$ nested within $\mathtt{c}_1$ which is nested within $\mathcal{P}$, the interface implemented by the current class is denoted by the path self[self[$\diamond$.$\mathtt{c}_1$].$\mathtt{c}_2$].itf. In Familia, this interface is denoted by the lighter syntax Self[$\mathtt{c}_1$.$\mathtt{c}_2$]. Familia also supports interface paths with inexact prefixes (i.e., P!.$\overline{\mathtt{c}[\omega_1]}$.I[$\omega_2$]); they are not modeled in F^2 for simplicity.

A type T is either the integer type, a type variable X, or an object type denoted by an interface path, which can be either exact or inexact. Inexactly typed values may have a run-time type that is a subtype, while exactly typed values cannot. A dispatcher d is either an exact class path or a class variable. Dispatchers are used in method calls e_0.(d.m[ω])($\overline{e_1}$), object-creation expressions pack (e, d), and `with`-clauses. Function *FTV*(•) returns the type variables occurring free in •. Function *FCV*(•) returns free class variables.

[5]Class empty and interface Any are nested directly within the program $\mathcal{P}$. Class empty is parameterized by a type variable X and implements the constraint $\diamond$.Any(X). A class extending $\diamond$.empty instantiates X as its representation type.

$$
\begin{array}{lrcl}
\text{values} & v & ::= & n \mid \texttt{pack}\ (v, P^!) \\
\text{evaluation contexts} & \mathcal{E} & ::= & [\cdot] \mid \mathcal{E}.(P^!.\texttt{m}[\omega])(\overline{e}) \mid v_0.(P^!.\texttt{m}[\omega])(\overline{v_1}, \mathcal{E}, \overline{e}) \mid \\
 & & & \texttt{pack}\ (\mathcal{E}, P^!) \mid \texttt{unpack}\ \mathcal{E}\ \texttt{as}\ (\texttt{x}, \texttt{p})\ \texttt{in}\ e_2
\end{array}
$$

$$\boxed{\diamond\!\rightsquigarrow \mathbb{P}^! \vdash e_1 \longrightarrow e_2}$$

[UNPACK] $$\diamond\!\rightsquigarrow \mathbb{P}^! \vdash \texttt{unpack}\ (\texttt{pack}\ (v, P^!))\ \texttt{as}\ (\texttt{x}, \texttt{p})\ \texttt{in}\ e \longrightarrow e\{v/\texttt{x}\}\{P^!/\texttt{p}\}$$

[CALL] $$\frac{\diamond\!\rightsquigarrow \mathbb{P}^!_{\text{prog}} \vdash P^! \rightsquigarrow \mathbb{P}^!_{\text{disp}} \qquad \mathbb{P}^!_{\text{disp}}(\texttt{m}) = [\varphi]\,\overline{T_1} \to T_2\ (\overline{\texttt{x}})\ \{\,e\,\}}{\diamond\!\rightsquigarrow \mathbb{P}^!_{\text{prog}} \vdash v_0.(P^!.\texttt{m}[\omega])(\overline{v_1}) \longrightarrow e\{v_0/\texttt{this}\}\{\omega/\varphi\}\{\overline{v_1}/\overline{\texttt{x}}\}}$$

Figure 5.14. Featherweight Familia: Operational semantics

$$
\begin{array}{lrcl}
\text{type environments} & \Delta & ::= & \varnothing \mid \Delta, \texttt{Self}[Q] \mid \Delta, \texttt{X} \\
\text{class environments} & K & ::= & \diamond\!\rightsquigarrow \mathbb{P}^! \mid K, \texttt{self}[P] \mid K, \texttt{p}\ \texttt{for}\ H(T) \\
\text{term environments} & \Gamma & ::= & \varnothing \mid \Gamma, \texttt{x}:T
\end{array}
$$

Figure 5.15. Featherweight Familia: Environment syntax

5.5.2 Dynamic Semantics

Figure 5.14 presents the operational semantics of Featherweight Familia, including its values, evaluation contexts, and reduction rules. Object values take the form $\texttt{pack}\ (v, P^!)$. Reduction rule [UNPACK] unpacks an object. Rule [CALL] reduces a method call. The method body to evaluate is retrieved from $\mathbb{P}^!_{\text{disp}}$, the *linkage* of the dispatcher $P^!$. A linkage provides a dispatch table indexed by method names, as discussed in more detail in Section 5.5.3.

5.5.3 Static Semantics

The complete static semantics of Featherweight Familiacan be found in the accompanying technical report [187]. Below we explain the judgment forms and discuss selected judgment rules shown in Figures 5.16 and 5.18.

Environments. The syntax of environments is shown in Figure 5.15. A type environment Δ contains `Self[`Q`]` parameters as well as type variables. A class environment K always contains the linkage of the entire program ($\diamond \rightsquigarrow \mathbb{P}!$). It may also contain `self`-parameters as well as class variables. For example, checking program $\mathcal{P}$ adds the program linkage into K, checking class `c` (nested within $\mathcal{P}$) adds `self[`$\diamond$`.c]` into K, and checking interface `I` (nested within `c`) adds `Self[self[`$\diamond$`.c].I]` into Δ.

Constrained parametric polymorphism. As shown in Figure 5.13, all nested components ($\mathcal{C}$, $\mathcal{I}$, and $\mathcal{M}$) can be parameterized, so their well-formedness rules require the well-formedness of the parameters `[`φ`]`, which is expressed using $\Delta; K \vdash$ `[`φ`]` OK. Subsequent checks in these well-formedness rules are performed under the environments Δ, Δ_φ and K, K_φ, where Δ_φ and K_φ consist of the type parameters and class parameters of `[`φ`]`. The well-formedness rules of class paths, interface paths, and method-call expressions correspondingly check the validity of the substitution of arguments `[`ω`]` for parameters `[`φ`]`. These checks use the judgment form $\Delta; K \vdash \{\omega/\varphi\}$ OK, and its rule is given by [INST] in Figure 5.16. In addition to the well-formedness of the arguments, it requires the constraints implemented by the dispatchers to entail the corresponding `where`-clause constraints. Constraint entailment is expressed using the judgment form $K \vdash H_1(T_1) \leq H_2(T_2)$.

$$\boxed{\Delta;\, K \vdash \{\omega/\varphi\}\ \mathsf{OK}}$$

[INST]
$$\frac{(\forall i)\ \Delta;\, K \vdash T_1^{(i)}\ \mathsf{OK} \qquad (\forall i)\ \Delta;\, K \vdash d^{(i)}\ \mathsf{OK} \qquad (\forall i)\ K \vdash d^{(i)}\ \mathsf{dispatches}\ T_3^{(i)} \qquad (\forall i)\ K \vdash d^{(i)}.\mathtt{itf}(T_3^{(i)}) \ \leq\ H^{(i)}(T_2^{(i)})\left\{\overline{T_1}\,/\,\overline{\mathsf{X}}\right\}}{\Delta;\, K \vdash \left\{\overline{T_1}\ \mathtt{with}\ \overline{d}\ /\ \overline{\mathsf{X}}\ \mathtt{where}\ \overline{\mathsf{p}\ \mathtt{for}\ H(T_2)}\right\}\ \mathsf{OK}}$$

$$\boxed{\Delta;\, K;\, \Gamma \vdash e : T}$$

[E-PACK]
$$\frac{\Delta;\, K \vdash d\ \mathsf{OK} \qquad K \vdash d\ \mathsf{dispatches}\ T \qquad \Delta;\, K;\, \Gamma \vdash e : T}{\Delta;\, K;\, \Gamma \vdash \mathtt{pack}\ (e, d) : d.\mathtt{itf}}$$

[E-UNPACK]
$$\frac{\Delta;\, K;\, \Gamma \vdash e_1 : H \qquad \Delta,\, \mathsf{X};\, K,\, \mathsf{p}\ \mathtt{for}\ H(\mathsf{X});\, \Gamma, \mathsf{x}{:}\mathsf{X} \vdash e_2 : T \qquad \mathsf{X} \notin FTV(T) \qquad \mathsf{p} \notin FCV(T)}{\Delta;\, K;\, \Gamma \vdash \mathtt{unpack}\ e_1\ \mathtt{as}\ (\mathsf{x}, \mathsf{p})\ \mathtt{in}\ e_2 : T}$$

[E-CALL]
$$\frac{\begin{array}{c}\Delta;\, K \vdash d\ \mathsf{OK} \qquad K \vdash d\ \mathsf{dispatches}\ T_0 \qquad \Delta;\, K;\, \Gamma \vdash e_0 : T_0 \\ K \vdash d.\mathtt{itf} \rightsquigarrow \mathbb{Q}! \qquad \mathbb{Q}!\{T_0/\mathtt{This}\}(\mathsf{m}) = [\varphi]\,\overline{T_1} \to T_2 \\ \Delta;\, K \vdash \{\omega/\varphi\}\ \mathsf{OK} \qquad (\forall i)\ \Delta;\, K;\, \Gamma \vdash e_1^{(i)} : T_1^{(i)}\{\omega/\varphi\}\end{array}}{\Delta;\, K;\, \Gamma \vdash e_0.(d.\mathsf{m}[\omega])(\overline{e_1}) : T_2\{\omega/\varphi\}}$$

$$\boxed{\vdash \mathcal{P}\ \mathsf{OK}}$$

[PROG]
$$\frac{\begin{array}{c}flatten\left(\diamond\{\overline{\mathcal{I}}\ \ \overline{C}\ \ e\}\right) = \mathbb{P}! \qquad K \stackrel{def}{=} \diamond\rightsquigarrow\mathbb{P}! \\ K \vdash \mathbb{P}!\ \ \mathsf{I\text{-}Conform} \qquad K \vdash \mathbb{P}!\ \ \mathsf{FB\text{-}Conform} \\ (\forall i)\ \varnothing;\, K;\, \diamond \vdash \mathcal{I}^{(i)}\ \mathsf{OK} \qquad (\forall i)\ \varnothing;\, K;\, \diamond \vdash C^{(i)}\ \mathsf{OK} \qquad \varnothing;\, K;\, \varnothing \vdash e : T\end{array}}{\vdash \diamond\{\overline{\mathcal{I}}\ \ \overline{C}\ \ e\}\ \mathsf{OK}}$$

Figure 5.16. Featherweight Familia: Selected well-formedness rules

Object-oriented polymorphism. The typing of pack- and unpack-expressions is given by rules [E-PACK] and [E-UNPACK] in Figure 5.16. The expression pack (e, d) packs e and the dispatcher d into an object, where e is the underlying representation and d is the class implementing the object type $d.\mathtt{itf}$. The expression unpack e_1 as (x, p) in e_2 unpacks object e_1 into its representation and class (bound to x and p, respectively) and evaluates e_2, in which x and p may occur free. The standard existential-unpacking rule requires the freshly generated

type variable not to occur free in the resulting type; likewise, rule [E-UNPACK] requires the same of the freshly generated class variable.

While Familia automatically generates natural classes for interfaces, F^2 gives a concrete encoding of natural classes via `unpack`-expressions. For example, suppose variable x_0 has object type `◇.I`, an interface that requires a single method `m:int→int`. Then invoking `m` on x_0 can be written as x_0`.(◇.natural_I!.m)(8)`, where the natural class `natural_I` is defined as follows:

```
class natural_I for ◇.I(◇.I) extends ◇.empty[◇.I]{
    m : int→int (x1) { unpack this as (x2,p) in x2.(p.m)(x1) }
}
```

The natural class implements the method by unpacking the receiver and subsequently calling the method with the unpacked class as the dispatcher and the unpacked representation as the receiver. Some prior object encodings formalize objects as explicit existentials [2, 29]. The unpacking of receiver objects in natural classes is akin to the way these encodings unpack existentially typed objects before sending messages to them.

The way that natural classes are encoded suggests that not all interfaces have natural classes. In fact, an interface such as `Eq` that uses its constraint parameter in contravariant or invariant positions other than the receiver type does not have a natural class. The reason is that the encoding of such natural classes would involve unpacking objects of the representation type every time the representation type appears in the method signature, including one for the receiver, and that the unpacked receiver does not necessarily have the same representation type as the other occurrences. The lack of natural classes for interfaces like `Eq` means these interfaces cannot be used as object types as other interfaces (e.g., `Set[E]`) can. This restriction is not a limitation, though; a survey

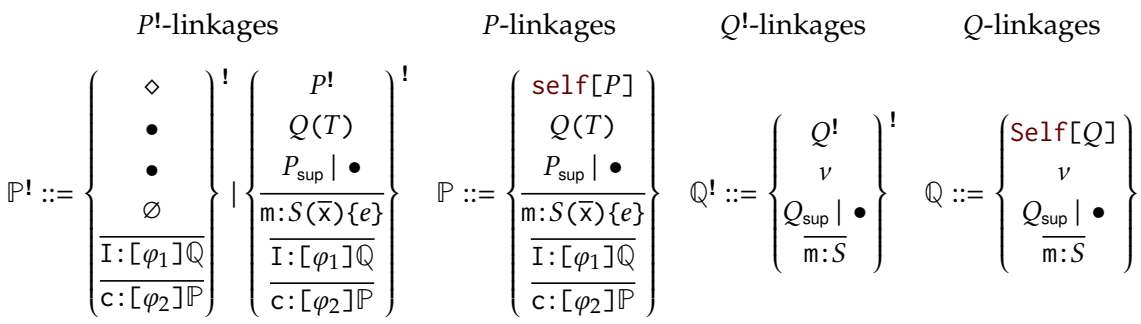


Figure 5.17. Featherweight Familia: Linkage syntax

of a large corpus of open-source Java code finds that in practice, programmers never use interfaces like `Eq` as types of objects [83].

Method calls. Rule [E-CALL] in Figure 5.16 type-checks a method call. The method signature that the call is checked against is retrieved from the linkage of the dispatcher's interface; this linkage contains signatures for the methods the interface requires. When the dispatcher d is a natural class, we get a typical object-oriented method call; when d is a class variable, we have a method call enabled by a type-class constraint; and when d is any other class, we have an "expander call" [181] that endows the receiver with new behavior. Rule [E-CALL] unifies these cases.

Family polymorphism. Central to the semantics is the notion of *linkages*. A class linkage $\mathbb{P}$ (or $\mathbb{P}^!$) collects information about a class path of the form P (or $P^!$). As shown in Figure 5.17, a class linkage is a tuple comprising (1) the path, (2) the constraint being implemented, (3) the superclass, (4) nested method definitions, (5) linkages of nested interfaces, and (6) linkages of nested classes. The linkage of an inexact class path P is parameterized by a `self[`P`]` parameter; substitution for `self[`P`]` in that linkage is thus capture-avoiding. We emphasize this fact by

$$\boxed{K \vdash P \rightsquigarrow \mathbb{P} \qquad K \vdash P^! \rightsquigarrow \mathbb{P}^!}$$

$$[\text{P}]\ \frac{\begin{array}{c} K \vdash P^! \rightsquigarrow \mathbb{P}^!_{\text{fam}} \qquad \mathbb{P}^!_{\text{fam}}(\mathsf{c}) = [\varphi]\mathbb{P}_{\text{nest}} \\ \mathit{parent}(\mathbb{P}_{\text{nest}}\{\omega/\varphi\}) = P_{\text{sup}} \\ K \vdash P_{\text{sup}} \rightsquigarrow \mathbb{P}_{\text{sup}} \qquad \mathbb{P}_{\text{sup}} \oplus \mathbb{P}_{\text{nest}}\{\omega/\varphi\} = \mathbb{P} \end{array}}{K \vdash P^!.\mathsf{c}[\omega] \rightsquigarrow \mathbb{P}} \qquad [\text{P!-PROG}]\ \frac{\Diamond \rightsquigarrow \mathbb{P}^! \in K}{K \vdash \Diamond \rightsquigarrow \mathbb{P}^!}$$

$$[\text{P!-SELF}]\frac{K \vdash P \rightsquigarrow \mathbb{P}}{K \vdash \texttt{self}[P] \rightsquigarrow \mathbb{P}^!} \qquad [\text{P!-NEST}]\frac{K \vdash P^!.\mathsf{c}[\omega] \rightsquigarrow \mathbb{P}}{K \vdash P^!.\mathsf{c}^![\omega] \rightsquigarrow \mathbb{P}^!\left\{P^!.\mathsf{c}^![\omega] \big/ \texttt{self}[P^!.\mathsf{c}[\omega]]\right\}}$$

$$\boxed{\mathbb{P}_1 \oplus \mathbb{P}_2 = \mathbb{P}_3}$$

$$[\text{CONCAT-P}]\ \frac{\begin{array}{c} \overline{\mathsf{m}_1 : S_1(\overline{\mathsf{x}_1})\{e_1\}}\{\texttt{self}[P_2]/\texttt{self}[P_1]\} \oplus \overline{\mathsf{m}_2 : S_2(\overline{\mathsf{x}_2})\{e_2\}} = \overline{\mathsf{m}_3 : S_3(\overline{\mathsf{x}_3})\{e_3\}} \\ \overline{\mathsf{I}_1 : [\varphi_{11}]\mathbb{Q}_1}\{\texttt{self}[P_2]/\texttt{self}[P_1]\} \oplus \overline{\mathsf{I}_2 : [\varphi_{21}]\mathbb{Q}_2} = \overline{\mathsf{I}_3 : [\varphi_{31}]\mathbb{Q}_3} \\ \overline{\mathsf{c}_1 : [\varphi_{12}]\mathbb{P}_1}\{\texttt{self}[P_2]/\texttt{self}[P_1]\} \oplus \overline{\mathsf{c}_2 : [\varphi_{22}]\mathbb{P}_2} = \overline{\mathsf{c}_3 : [\varphi_{32}]\mathbb{P}_3} \end{array}}{\left\{\begin{array}{c} \texttt{self}[P_1] \\ \dots \\ \dots \\ \hline \overline{\mathsf{m}_1 : S_1(\overline{\mathsf{x}_1})\{e_1\}} \\ \hline \overline{\mathsf{I}_1 : [\varphi_{11}]\mathbb{Q}_1} \\ \hline \overline{\mathsf{c}_1 : [\varphi_{12}]\mathbb{P}_1} \end{array}\right\} \oplus \left\{\begin{array}{c} \texttt{self}[P_2] \\ Q_2(T_2) \\ P_{\text{sup2}} \\ \hline \overline{\mathsf{m}_2 : S_2(\overline{\mathsf{x}_2})\{e_2\}} \\ \hline \overline{\mathsf{I}_2 : [\varphi_{21}]\mathbb{Q}_2} \\ \hline \overline{\mathsf{c}_2 : [\varphi_{22}]\mathbb{P}_2} \end{array}\right\} = \left\{\begin{array}{c} \texttt{self}[P_2] \\ Q_2(T_2) \\ P_{\text{sup2}} \\ \hline \overline{\mathsf{m}_3 : S_3(\overline{\mathsf{x}_3})\{e_3\}} \\ \hline \overline{\mathsf{I}_3 : [\varphi_{31}]\mathbb{Q}_3} \\ \hline \overline{\mathsf{c}_3 : [\varphi_{32}]\mathbb{P}_3} \end{array}\right\}}$$

$$\boxed{K \vdash Q \rightsquigarrow \mathbb{Q} \qquad K \vdash Q^! \rightsquigarrow \mathbb{Q}^!}$$

$$[\text{Q}]\ \frac{\begin{array}{c} K \vdash P^! \rightsquigarrow \mathbb{P}^! \qquad \mathbb{P}^!(\mathsf{I}) = [\varphi]\mathbb{Q}_{\text{nest}} \qquad \mathit{parent}(\mathbb{Q}_{\text{nest}}\{\omega/\varphi\}) = Q_{\text{sup}} \\ K \vdash Q_{\text{sup}} \rightsquigarrow \mathbb{Q}_{\text{sup}} \qquad \mathbb{Q}_{\text{sup}} \oplus \mathbb{Q}_{\text{nest}}\{\omega/\varphi\} = \mathbb{Q} \end{array}}{K \vdash P^!.\mathsf{I}[\omega] \rightsquigarrow \mathbb{Q}} \qquad [\text{Q!-NORM}]\ \frac{\begin{array}{c} K \vdash Q^!_1 \Rrightarrow Q^!_2 \\ K \vdash Q^!_2 \rightsquigarrow \mathbb{Q}^! \end{array}}{K \vdash Q^!_1 \rightsquigarrow \mathbb{Q}^!}$$

$$[\text{Q!-SELF}]\ \frac{K \vdash Q \rightsquigarrow \mathbb{Q}}{K \vdash \texttt{Self}[Q] \rightsquigarrow \mathbb{Q}^!} \qquad [\text{Q!-NEST}]\ \frac{K \vdash P^!.\mathsf{I}[\omega] \rightsquigarrow \mathbb{Q}}{K \vdash P^!.\mathsf{I}^![\omega] \rightsquigarrow \mathbb{Q}^!\left\{P^!.\mathsf{I}^![\omega] \big/ \texttt{Self}[P^!.\mathsf{I}[\omega]]\right\}}$$

$$[\text{Q!-I-P}]\ \frac{\begin{array}{c} K \vdash P^! \rightsquigarrow \mathbb{P}^! \\ \mathit{interface}(\mathbb{P}^!) = Q \qquad K \vdash Q \rightsquigarrow \mathbb{Q} \end{array}}{K \vdash P^!.\texttt{itf} \rightsquigarrow \mathbb{Q}^!\left\{P^!.\texttt{itf} \big/ \texttt{Self}[Q]\right\}} \qquad [\text{Q!-I-CV}]\ \frac{\mathsf{p}\ \texttt{for}\ Q(T) \in K \qquad K \vdash Q \rightsquigarrow \mathbb{Q}}{K \vdash \mathsf{p}.\texttt{itf} \rightsquigarrow \mathbb{Q}^!\{\mathsf{p}.\texttt{itf}/\texttt{Self}[Q]\}}$$

$$\boxed{K \vdash Q^!_1 \Rrightarrow Q^!_2}$$

$$[\text{NORM-ABS}]\ \frac{K \vdash \Diamond.\overline{\mathsf{c}^![\omega]} \rightsquigarrow \mathbb{P}^! \qquad \mathit{interface}(\mathbb{P}^!) = Q}{K \vdash \Diamond.\overline{\mathsf{c}^![\omega]}.\texttt{itf} \Rrightarrow Q^!} \qquad [\text{NORM-CV}]\ \frac{\mathsf{p}\ \texttt{for}\ Q^!(T) \in K}{K \vdash \mathsf{p}.\texttt{itf} \Rrightarrow Q^!}$$

Figure 5.18. Featherweight Familia: Selected rules for computing linkages

putting this `self`-path, instead of the inexact path, as the first element of the tuple. Given the linkage $\mathbb{P}$ of an inexact class path P, we use $\mathbb{P}^!$ to mean the linkage of the exact class path `self[`P`]`. Interface linkages ($\mathbb{Q}^!$ and $\mathbb{Q}$) contain fewer components. In the linkage of an inexact interface path Q, `Self[`Q`]` may occur free. Looking up an (exact) linkage for a nested component named `id` is denoted by $\mathbb{P}^!$(`id`) or $\mathbb{Q}^!$(`id`).

The well-formedness rules for paths can be found in the technical report, but here Figure 5.18 presents the rules that compute linkages for paths. The corresponding judgment forms are $K \vdash P^! \rightsquigarrow \mathbb{P}^!$, $K \vdash P \rightsquigarrow \mathbb{P}$, $K \vdash Q^! \rightsquigarrow \mathbb{Q}^!$, and $K \vdash Q \rightsquigarrow \mathbb{Q}$.

Linkages are computed in an outside-in manner. As shown in rule [PROG] in Figure 5.16, the linkage of a program is obtained via the helper function *flatten*(•) and added to the environment. Rule [P!-PROG] in Figure 5.18 retrieves this linkage from the environment. The helper function *flatten*(•) is defined in the technical report; it does not do anything interesting except converting the program text into a tree of linkages. Importantly, linkages nested within an outer linkage do not contain components that are inherited. Thus all nested linkages are *incomplete*.

Rule [P] in Figure 5.18 computes the linkage of an inexact class path $P^!$`.c[`ω`]`. It first computes the linkage $\mathbb{P}^!_{\mathsf{fam}}$ of the family $P^!$, from which the nested linkage $\mathbb{P}_{\mathsf{nest}}$ corresponding to nested class `c` is obtained. The helper function *parent*(•) finds the superclass, whose linkage is $\mathbb{P}_{\mathsf{sup}}$. While $\mathbb{P}_{\mathsf{nest}}$ is incomplete, the superclass linkage $\mathbb{P}_{\mathsf{sup}}$ is complete. The complete linkage of $P^!$`.c[`ω`]` is then obtained by *concatenating* $\mathbb{P}_{\mathsf{sup}}$ with $\mathbb{P}_{\mathsf{nest}}\{\omega/\varphi\}$, using the linkage concatenation operator $\oplus$ defined by rule [CONCAT-P] in Figure 5.18. Operator $\oplus$ is defined recursively; a nested linkage is concatenated with the corresponding linkage it further-binds

to produce a new nested linkage (see other $\oplus$-rules in the technical report). Importantly, $\oplus$ replaces the `self`-parameter of the first linkage with that of the second linkage; this substitution is key to late binding of nested names. Linkage concatenation is also what enables dynamic dispatch for object-oriented method calls (i.e., calls using a natural class as the dispatcher), because a method in a linkage $\mathbb{P}$ overrides less specific methods of the same name in linkages to which $\mathbb{P}$ is concatenated.

Rule [P!-NEST] shows that the linkage of an exact path $P!.c![\omega]$ is obtained by substituting $P!.c![\omega]$ for $\texttt{self}[P!.c[\omega]]$ in the linkage of $P!.c[\omega]$.

Some interface paths are equivalent. For example, interface path `p.itf`, where `p` is declared to witness constraint $Q!(T)$, is equivalent to $Q!$. This equivalence relation is captured by path normalization ($\equiv\!\!\Rightarrow$). Rules [NORM-ABS] and [NORM-CV] in Figure 5.18 simplify interface paths of form $d.\texttt{itf}$. A path is simplified to its normal form after finite steps of simplification ($\equiv\!\!\Rightarrow^*$). Other normalization rules are purely structural; they and the normal forms can be found in the technical report. The linkage computation rules in Figure 5.18—except for [Q!-NORM]—are defined for paths of normal forms. Rule [Q!-NORM] suggests that the linkage of a path is the same as that of its normal form. The equivalence relation $\equiv$ (shown in the technical report) is then the symmetric closure of $\equiv\!\!\Rightarrow^*$. Because of this path equivalence, substitution for $\texttt{self}[P]$ (resp. $\texttt{Self}[Q]$) also replaces other `self`-paths (resp. `Self`-paths) that are equivalent with $\texttt{self}[P]$ (resp. $\texttt{Self}[Q]$).

Soundness of family polymorphism hinges on a few conformance checks. For example, if a nested interface definition adds new methods in a derived module, classes implementing the interface in the base module should also be augmented

in the derived module to define the new methods. This conformance of classes to interfaces is expressed by the judgment form $K \vdash \mathbb{P}!$ I-Conform.

Another conformance condition, $K \vdash \mathbb{P}!$ FB-Conform, requires that nested classes and interfaces conform to classes and interfaces they further-bind. In particular, the superclass (or interface) of a nested class in a derived module should be a subclass (or subinterface) of that of the further-bound class (or interface) in the base module. Also, a nested, further-binding interface should not change its variance with respect to the representation type. These checks ensure that inherited code still type-checks in derived modules.

The rules performing the conformance checks above are given in the technical report. They work by recursively invoking the checks on nested classes. At the top level, they are invoked from rule [PROG] in Figure 5.16.

Decidability. Because F^2 does not infer default classes, decidability of its static semantics is trivial: the well-formedness rules and the linkage-computation rules are syntax-directed (the subsumption rule can be easily factored into the other expression-typing rules, and the path normalization rules are defined algorithmically), and a subtyping algorithm works by climbing the subtyping hierarchy. Inference of default classes in Familia could potentially introduce nontermination, similar to how default model inference in Genus could lead to nontermination [191]. We expect that termination can be guaranteed by enforcing syntactic restrictions in the same fashion as in Genus. In particular, the restrictions include Material–Shape separation [83], which, in the context of Familia, prevents the supertype of an interface definition from mentioning the interface being defined; for example, an interface like `Double` cannot be declared to implement `Comparable[Double]`. Java allows such supertypes so that types like `Double` can satisfy F-bounded constraints [38] like `<T extends Comparable<? super T>>`.

Familia would use a `where`-clause instead (i.e., `[T where Ord(T)]`), eliminating possible nontermination when checking subtyping [85] while adding the flexibility of retroactively adapting types to constraints.

5.5.4 Soundness

We establish the soundness of Featherweight Familiathrough the standard approach of progress and preservation [185]. The key lemmas and their proofs can be found in the technical report.

Lemma 9 (PROGRESS)**.**
If (i) $\vdash \mathcal{P}\ \mathsf{OK}$, (ii) $\mathit{flatten}(\mathcal{P}) = \mathbb{P}^{!}_{\mathsf{prog}}$, and (iii) $\varnothing;\ \diamond \rightsquigarrow \mathbb{P}^{!}_{\mathsf{prog}};\ \varnothing \vdash e : T$, then either e is a value or there exists e' such that $\diamond \rightsquigarrow \mathbb{P}^{!}_{\mathsf{prog}} \vdash e \longrightarrow e'$.

Lemma 10 (PRESERVATION)**.**
If (i) $\vdash \mathcal{P}\ \mathsf{OK}$, (ii) $\mathit{flatten}(\mathcal{P}) = \mathbb{P}^{!}_{\mathsf{prog}}$, (iii) $\varnothing;\ \diamond \rightsquigarrow \mathbb{P}^{!}_{\mathsf{prog}};\ \varnothing \vdash e : T$, and also (iv) $\diamond \rightsquigarrow \mathbb{P}^{!}_{\mathsf{prog}} \vdash e \longrightarrow e'$, then $\varnothing;\ \diamond \rightsquigarrow \mathbb{P}^{!}_{\mathsf{prog}};\ \varnothing \vdash e' : T$.

Theorem 5 (SOUNDNESS)**.**
If (i) $\vdash \diamond\{\,\overline{I}\ \ \overline{C}\ \ e\,\}\ \mathsf{OK}$, (ii) $\mathit{flatten}\Big(\diamond\{\,\overline{I}\ \ \overline{C}\ \ e\,\}\Big) = \mathbb{P}^{!}_{\mathsf{prog}}$, and (iii) $\diamond \rightsquigarrow \mathbb{P}^{!}_{\mathsf{prog}} \vdash e \longrightarrow^{*} e'$, then either e' is a value or there exists e'' such that $\diamond \rightsquigarrow \mathbb{P}^{!}_{\mathsf{prog}} \vdash e' \longrightarrow e''$.

5.6 Related work

One way to evaluate programming language designs is by comparison with prior work, and indeed decades of prior work on language support for extensible and composable software has developed many mechanisms for extensibility and

genericity. However, we argue that no prior work integrates the different forms of polymorphism as successfully.

Constrained parametric polymorphism. Central to a parametric-polymorphism mechanism is a way to specify and satisfy constraints on type parameters. Many prior languages have experimented with either nominal subtyping constraints (e.g., Java and C#) or structural matching constraints (e.g., CLU [110] and Cecil [43]). Both approaches are too inflexible: types must be preplanned to either explicitly declare they implement the constraint or include the required methods with conformant signatures. At the same time, typing is made difficult by the interaction between inheritance and constraints that require binary methods. F-bounded polymorphism [38] and match-bounded polymorphism [1, 31, 33] are proposed to address this typing problem. However, they do not address the more urgent need to allow types to retroactively satisfy constraints they are not prepared to satisfy; this inflexibility is an inherent limitation of subtyping-based approaches.

To allow retroactive adaptation, recent work follows Haskell type classes [176]. JavaGI [182] supports generalized interfaces that can act as type classes. A special *implementation* construct is used as a type-class instance. Genus [190] introduces *constraints* and *models* on top of interfaces and classes. It avoids the *global uniqueness* limitation of Haskell and JavaGI—that type-class instances are globally scoped and that a given constraint can only be satisfied in one way. To avoid complicating the easy case, Genus allows constraints to be satisfied structurally via *natural models*. Genus also supports model-dependent types that strengthen static checking and model multimethods that offer convenience and extensibility. Familia incorporates all these Genus features without requiring

extra constructs for constraints or models. Unlike Familia, neither JavaGI nor Genus supports associated types.

Generic programming in Rust [154] and Swift [169] is inspired by type classes as well. In Rust, objects and generics are expressed using the same constructs (`trait` and `impl`), but Rust lacks support for implementation inheritance. These languages also have the limitation of global uniqueness. Dreyer et al. [57] and Devriese and Piessens [55] integrate type classes into ML and Agda, respectively, with a goal of not complicating the host language with duplicate functionality. Although not intended for generic programming, expanders [181] and CaesarJ wrappers [12] support statically scoped adaptation of classes to interfaces.

In Scala, generics are supported by using the Concept design pattern, made more convenient by implicits [137]: traits act as constraints, and trait objects are implicitly resolved arguments to generic abstractions. This approach does not distinguish types instantiated with different trait objects (cf. Familia types that keep track of the classes used to satisfy constraints), and does not allow specializing behavior for subtypes of the constrained type (cf. class multimethods in Familia). Scala also supports *higher-order polymorphism* by allowing higher-kinded type parameters and virtual types [123]. Familia supports higher-order polymorphism via nested, parameterized types and interfaces. Because nested components can be further-bound, higher-order polymorphism in Familia goes beyond Scala's higher-kinded virtual types.

Family polymorphism. Prior work on family polymorphism has been largely disjoint from work on parametric polymorphism. Virtual types [92, 170, 172] are unbound type members of an interface or class. They support family polymorphism [65], with families identified by an instance of the enclosing class. Virtual types inspired Haskell to add associated types to type classes [41, 42]. Virtual

types are not really "virtual": once they are bound in a class, their bindings cannot be refined as can those of virtual methods and virtual classes. In this sense, they act more like type parameters; in fact, virtual types are considered an alternative approach to parametric polymorphism [170]. It is understood that virtual types are good at expressing mutually recursive bounds [32]; this use of virtual types in generic programming is largely subsumed by the more flexible approach of multi-parameter type classes [98] available in Haskell, JavaGI, Genus, and Familia.

Virtual classes, both based on object families [12, 23, 64, 113, 115], and class families [44, 131, 132, 150], offer a more powerful form of family polymorphism than virtual types do: a subclass can specialize and enrich nested classes via further binding. Path-dependent types are used to ensure type safety for virtual types and virtual classes (e.g., Ernst et al. [66], Nystrom et al. [131]). A variety of other mechanisms support further binding, including virtual classes, mixin layers [162], delegation layers [140], and variant path types [93]. The family-polymorphism mechanism in Familia is closest to that in Jx [131]. Our use of prefix types is adapted from Jx; the fact that `self`-prefixes can be inferred makes family polymorphism lightweight in Familia.

Unlike the class-family approach taken in Familia, the object-family approach (virtual classes) does not readily support cross-family inheritance. For example, with virtual classes, class `a.b.c` cannot extend class `a.d.e` because class `a.b.c` has no enclosing instance of `a.d`. Tribe [44] and Scala support cross-family inheritance for virtual classes and virtual types, respectively, but by adding extra complexity to virtual classes or by resorting to verbose design patterns. Few prior languages support coordinated evolution of related, non-nested families. Cross-family inheritance and cross-family coevolution are crucial to deploying

family polymorphism at large scale, where we expect components from different modules to be frequently reused and composed.

Scala supports virtual types but not virtual classes, simulating the latter with a design pattern [134]. While this pattern seems effective at a small scale for tasks like the Observer pattern, it does not scale to a larger setting where cross-family inheritance is needed, where entire frameworks are extended, and where further binding is therefore needed at arbitrary depth. The effort required to encode virtual classes in Scala appears to be significant [183]. Scala also supports mixin composition. A mixin has an unbound superclass that is bound in classes that incorporate the mixin. Familia is expressive enough to encode mixin composition via late binding of superclasses, rather than requiring a separate language mechanism for mixins.

Familia extends the expressivity and practicality of family polymorphism. It allows classes to be unbound yet non-abstract. It also allows externally imported names to coevolve with the current module by further-binding base modules. Prior languages that support family polymorphism beyond virtual types have omitted support for parametric polymorphism. We believe support for parametric polymorphism is still important, because applicative instantiation of generic abstractions is often more convenient and interoperable [32].

CHAPTER 6

CONCLUSION

This dissertation looks at key abstraction mechanisms for organizing software. In particular, it explores new linguistic abstractions for dealing with non-local control behavior (such as exception handling), and for code reuse and extensibility. We have designed these linguistic abstractions with the guidelines (a) that they make it easier for the programmer to express what we thought they needed to express, (b) that they provide the guarantees which we considered are important to provide, and (c) that the resulting language as a whole is as simple and coherent as we could make it.

Accepting blame for tunneled exceptions. The new exception mechanism presented in Chapter 2 combines the benefits of static checking with the flexibility of unchecked exceptions. We were guided in the design of this mechanism by thinking carefully about the goals of an exception mechanism, by much previous work, and by many discussions found online. Our formal results and experience suggest that our approach improves assurance that exceptions are handled. The evaluation shows that the mechanism works well on real code. It adds negligible cost when exceptions are not being used; exception tunneling comes with a small performance penalty that appears to be more than offset in practice by avoiding the run-time overhead of wrapping exceptions. We hope this work helps programmers use exceptions in a principled way and gives language implementers an incentive to make exceptions more efficient.

Abstraction-safe effect handlers via tunneling. Chapter 3 argues that tunneling is also the right semantics for algebraic effects. As we have shown formally,

it makes them abstraction-safe, preserving modular reasoning. Because algebraic effects generalize other mechanisms such as exceptions, dynamically scoped variables, and coroutines, the tunneling semantics fixes not only algebraic effects generically, but also the design of several specific language features. We have provided a strong foundation for the design of algebraic-effect mechanisms that are not only type-safe, but also abstraction-safe. Our new semantics should be a useful guide for future language designs and also motivate support for algebraic effects in mainstream languages.

Lightweight, flexible object-oriented generics. The Genus design presented in Chapter 4 is a novel and harmonious combination of language ideas that achieves a high degree of expressive power for generic programming while handling common usage patterns simply. Our experiments with using Genus to reimplement real software suggests that it offers an effective way to integrate generics into object-oriented languages, decreasing annotation burden while increasing type safety. Our benchmarks suggest the mechanism can be implemented with good performance.

Unifying interfaces, type classes, and family polymorphism. The Familia design presented in Chapter 5 achieves a high degree of expressive power by unifying multiple powerful mechanisms for type-safe polymorphism. The resulting language has low surface complexity—it can be used as an ordinary Java-like object-oriented language that supports inheritance, encapsulation, and subtyping. With little added syntax, several powerful features become available: parametric polymorphism with flexible type classes, wrapper-free adaptation, and deep family polymorphism with cross-family inheritance and cross-family coevolution. We have described the language intuitively with examples that

illustrate its expressive power. Its operational and static semantics are captured by a core language that we have proved type-safe. Comparisons with previous mechanisms for generic programming show that Familia improves expressive power in a lightweight way.

BIBLIOGRAPHY

[1] Martín Abadi and Luca Cardelli. On subtyping and matching. *ACM Trans.on Programming Languages and Systems*, 18(4), July 1996.

[2] Martín Abadi, Luca Cardelli, and Ramesh Viswanathan. An interpretation of objects and object types. In *23rd ACM Symp. on Principles of Programming Languages (POPL)*, 1996.

[3] Ada 95. Ada 95 reference manual: language and standard libraries, 1997.

[4] Ole Agesen, Stephen N Freund, and John C Mitchell. Adding type parameterization to the Java language. In *12th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1997.

[5] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *15th European Symposium on Programming*, 2006. Extended/corrected version available as Harvard University TR-01-06.

[6] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1983.

[7] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[8] Jonathan Aldrich. The power of interoperability: Why objects are inevitable. In *ACM Int'l Symp. on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!)*, 2013.

[9] Apache Commons. The Apache Commons project. https://commons.apache.org/.

[10] Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans.on Programming Languages and Systems*, 23(5), September 2001.

[11] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *34th ACM Symp. on Principles of Programming Languages (POPL)*, 2007.

[12] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Lecture Notes in Computer Science: Transactions on Aspect-Oriented Software Development I*. 2006.

[13] Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[14] Brian Aydemir, Arthur Charguéraud, Benjamin C Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *35th ACM Symp. on Principles of Programming Languages (POPL)*, 2008.

[15] Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. *Logical Methods in Computer Science*, Volume 10, Issue 4, December 2014.

[16] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1), 2015.

[17] Nick Benton and Uri Zarfaty. Formalizing and verifying semantic type soundness of a simple compiler. In *Proceedings of the 9th ACM SIGPLAN*

International Conference on Principles and Practice of Declarative Programming, 2007.

[18] Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. Handle with care: Relational interpretation of algebraic effects and handlers. *Proc. ACM on Programming Languages*, 2(POPL), December 2017.

[19] Andrew P Black. *Exception handling: The case against*. PhD thesis, University of Oxford, 1982.

[20] Andrew P. Black, Kim B. Bruce, and R. James Noble. The essence of inheritance. In *A List of Successes That Can Change the World*, volume 9600 of *LNCS*. Springer, March 2016.

[21] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *21st ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2006.

[22] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *13th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, October 1998.

[23] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in Newspeak. In *24th European Conf. on Object-Oriented Programming*, 2010.

[24] Jonathan Immanuel Brachthäuser and Philipp Schuster. Effekt: Extensible algebraic effects in Scala (short paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, 2017.

[25] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Algebraic effects for the masses. *Proc. ACM on Programming Languages*, 2 (OOPSLA), October 2018.

[26] Oliver Bračevac, Nada Amin, Guido Salvaneschi, Sebastian Erdweg, Patrick Eugster, and Mira Mezini. Versatile event correlation with algebraic effects. *Proc. ACM on Programming Languages*, 2(ICFP), August 2018.

[27] Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[28] Kim Bruce, Luca Cardelli, and Benjamin Pierce. Comparing object encodings. *Information and Computation*, 155(1):108–133, 1999.

[29] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4 (2):127–206, 1994.

[30] Kim B. Bruce and J. Nathan Foster. LOOJ: Weaving LOOM into Java. In *18th European Conf. on Object-Oriented Programming*, 2004.

[31] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *11th European Conf. on Object-Oriented Programming*, June 1997.

[32] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *12th European Conf. on Object-Oriented Programming*, July 1998.

[33] Kim B. Bruce, Angela Schuett, Robert van Gent, and Adrian Fiech. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM Trans.on Programming Languages and Systems*, 25(2):225–290, March 2003.

[34] Peter A. Buhr and W. Y. Russell Mok. Advanced exception handling mechanisms. *IEEE Trans. Softw. Eng.*, 26(9), September 2000.

[35] Bruno Cabral and Paulo Marques. Exception handling: A field study in Java and .NET. In *21st European Conf. on Object-Oriented Programming*, 2007.

[36] Bruno Cabral and Paulo Marques. Hidden truth behind .NET's exception handling today. *IET Software*, 1(6), 2007.

[37] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2001.

[38] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Conf. on Functional Programming Languages and Computer Architecture*, 1989.

[39] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2–3):138–164, 1988. Also in *Readings in Object-Oriented Database Systems*, S. Zdonik and D. Maier, eds., Morgan Kaufmann, 1990.

[40] Robert Cartwright and Guy L. Steele, Jr. Compatible genericity with run-time types for the Java programming language. In *13th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, October 1998.

[41] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones. Associated type synonyms. In *10th ACM SIGPLAN Int'l Conf. on Functional Programming*, 2005.

[42] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *32nd ACM Symp. on Principles of Programming Languages (POPL)*, 2005.

[43] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *20th European Conf. on Object-Oriented Programming*, volume 615, 1992.

[44] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: A simple virtual class calculus. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, 2007.

[45] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *15th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2000.

[46] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *4th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 1989.

[47] William R. Cook. On understanding data abstraction, revisited. In *24th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2009.

[48] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, January 1990. Also STL-89-17, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, CA, July 1989.

[49] Coq 8.7. The Coq proof assistant. http://coq.inria.fr. Version 8.7.

[50] Russ Cox. Generics: Problem overview. https://go.googlesource.com/proposal/+/master/design/go2draft-generics-overview.md, 2018. Accessed on July 17, 2019.

[51] Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *26th ACM Symp. on Principles of Programming Languages (POPL)*, 1999.

[52] CWE. Common weakness enumeration list. http://cwe.mitre.org/data/.

[53] Olivier Danvy and Andrzej Filinski. Abstracting control. In *ACM Conf. on LISP and Functional Programming*, 1990.

[54] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *10th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, October 1995. ACM SIGPLAN Notices 30(10).

[55] Dominique Devriese and Frank Piessens. On the bright side of type classes: Instance arguments in Agda. In *16th ACM SIGPLAN Int'l Conf. on Functional Programming*, 2011.

[56] Derek Dreyer. Milner award lecture: The type soundness theorem that you really want to prove (and now you can). In *45th ACM Symp. on Principles of Programming Languages (POPL)*, 2018.

[57] Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. In *34th ACM Symp. on Principles of Programming Languages (POPL)*, 2007.

[58] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *24th Annual IEEE Symposium on Logic In Computer Science (LICS)*, 2009.

[59] Derek Dreyer, Georg Neis, and Lars Birkedal. The impact of higher-order state and control effects on local relational reasoning. *Journal of Functional Programming*, 22(4-5):477–528, 2012.

[60] R. Kent Dyvbig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6): 687–730, November 2007.

[61] Bruce Eckel. Does Java need checked exceptions? `http://www.mindview.net/Etc/Discussions/CheckedExceptions`, 2003.

[62] ECMA International. ECMAScript 2018 language specification. Standard-ECMA 262, June 2018.

[63] Burak Emir, Andrew Kennedy, Claudio Russo, and Dachuan Yu. Variance and generalized constraints for C# generics. In *20th European Conf. on Object-Oriented Programming*, 2006.

[64] Erik Ernst. *gbeta—a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, 1999.

[65] Erik Ernst. Family polymorphism. In *15th European Conf. on Object-Oriented Programming*, 2001.

[66] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *33rd ACM Symp. on Principles of Programming Languages (POPL)*, January 2006.

[67] Simon Marlow et. al. Haskell 2010 language report. haskell.org, 2010.

[68] Manuel Fähndrich, Jeffrey S. Foster, Alexander Aiken, and Jason Cu. Tracking down exceptions in standard ML programs. Technical report, EECS Department, UC Berkeley, 1998.

[69] Mattias Felleisen. The theory and practice of first-class prompts. In *15th ACM Symp. on Principles of Programming Languages (POPL)*, 1988.

[70] FindBugs bug descriptions. Findbugs bug descriptions. http://findbugs.sourceforge.net/bugDescriptions.html/.

[71] findbugs-release. Findbugs. http://findbugs.sourceforge.net/.

[72] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *7th ACM SIGPLAN Int'l Conf. on Functional Programming*, 2002.

[73] Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. Behavioral contracts and behavioral subtyping. In *ESEC/FSE-9*, 2001.

[74] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[75] Alessandro F Garcia, Cecilia M.F Rubira, Alexander Romanovsky, and Jie Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 2001.

[76] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy G. Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, 2003.

[77] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17(2):145–205, March 2007.

[78] Brian Goetz. Java theory and practice: The exceptions debate. `http://www.ibm.com/developerworks/library/j-jtp05254`, 2004.

[79] Brian Goetz. Exception transparency in Java. `http://blogs.oracle.com/briangoetz/entry/exception_transparency_in_java`, 2010.

[80] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Comm. of the ACM*, 18:683–696, December 1975.

[81] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 3nd edition, 2005.

[82] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 3rd edition, 2005. ISBN 0321246780.

[83] Ben Greenman, Fabian Muehlboeck, and Ross Tate. Getting F-bounded polymorphism into shape. In *35th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2014.

[84] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *21st ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2006.

[85] Radu Grigore. Java generics are Turing complete. In *44th ACM Symp. on Principles of Programming Languages (POPL)*, 2017.

[86] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2002.

[87] Carl Gunter and John C. Mitchell, editors. *Theoretical aspects of object-oriented programming*. MIT Press, 1994.

[88] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ml-like languages. In *7th Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, 1995.

[89] Anders Hejlsberg, Bill Venners, and Bruce Eckel. Remaining neutral on checked exceptions. `http://www.artima.com/intv/handcuffs.html`, 2003.

[90] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley, 1st edition, October 2003.

[91] Daniel Hillerström and Sam Lindley. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*, 2016.

[92] Atsushi Igarashi and Benjamin Pierce. Foundations for virtual types. In *Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, June 1999.

[93] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. In *22nd ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007.

[94] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans.on Programming Languages and Systems*, 23(3):396–450, 2001.

[95] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. Associated types and constraint propagation for mainstream object-oriented generics. In *20th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2005.

[96] Patricia Johann, Alex Simpson, and Janis Voigtländer. A generic operational metatheory for algebraic effects. In *25th Annual IEEE Symposium on Logic In Computer Science (LICS)*, 2010.

[97] Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP)*, June 2004.

[98] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell Workshop*, 1997.

[99] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *18th ACM SIGPLAN Int'l Conf. on Functional Programming*, 2013.

[100] Andrew Kennedy and Don Syme. Design and implementation of generics for the .NET Common Language Runtime. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2001.

[101] Donald Ervin Knuth. *The TEXbook*. Addison-Wesley Reading, 1984.

[102] Daan Leijen. Koka: Programming with row polymorphic effect types. In *5th Workshop on Mathematically Structured Functional Programming*, 2014.

[103] Daan Leijen. Type directed compilation of row-typed algebraic effects. In *44th ACM Symp. on Principles of Programming Languages (POPL)*, 2017.

[104] Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans.on Programming Languages and Systems*, 22(2), March 2000.

[105] Jeffrey R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit parameters: Dynamic scoping with static types. In *27th ACM Symp. on Principles of Programming Languages (POPL)*, 2000.

[106] Sam Lindley, Conor McBride, and Craig McLaughlin. Do be do be do. In *44th ACM Symp. on Principles of Programming Languages (POPL)*, 2017.

[107] B. Liskov and A. Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, SE-5(6):546–558, November 1979.

[108] B. Liskov, A. Snyder, R. Atkinson, and J. C. Schaffert. Abstraction mechanisms in CLU. *Comm. of the ACM*, 20(8):564–576, August 1977. Also in S. Zdonik and D. Maier, eds., *Readings in Object-Oriented Database Systems*.

[109] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Trans.on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[110] Barbara Liskov, Russell Atkinson, Toby Bloom, J. Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*. Springer-Verlag, 1984. Also published as Lecture Notes in Computer Science 114, G. Goos and J. Hartmanis, Eds., Springer-Verlag, 1981.

[111] M. Donald MacLaren. Exception handling in PL/I. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, 1977.

[112] David MacQueen. Modules for Standard ML. In *1984 ACM Symposium on Lisp and Functional Programming*, August 1984.

[113] O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[114] Ole Lehrmann Madsen. Semantic analysis of virtual classes and nested classes. In *14^{th} ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, November 1999.

[115] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism for object-oriented programming. In *4^{th} ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, October 1989.

[116] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: The

Java Unsafe API in the wild. In *2015 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2015.

[117] Bertrand Meyer. *Eiffel: The Language*. 1992.

[118] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, 2001. ISBN 0-7356-1448-2.

[119] Todd Millstein, Mark Reay, and Craig Chambers. Relaxed MultiJava: Balancing extensibility and modular typechecking. In *18th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2003.

[120] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[121] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[122] J. G. Mitchell, W. Maybury, and R. Sweet. Mesa language manual. Technical Report CSL-78-1, Xerox Research Center, Palo Alto, CA, February 1978.

[123] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In *23rd ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2008.

[124] J. H. Morris, Jr. *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology, 1968.

[125] David R. Musser, Gillmer J. Derge, and Atul Saini. *The STL Tutorial and Reference Guide*. Addison-Wesley, 2nd edition, 2001.

[126] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *24th ACM Symp. on Principles of Programming Languages (POPL)*, January 1997.

[127] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, 7(5), June 1995.

[128] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[129] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[130] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *12th Int'l Conf. on Compiler Construction (CC'03)*, April 2003.

[131] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *19th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, October 2004.

[132] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: Nested intersection for scalable software composition. In *21st ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, October 2006.

[133] Martin Odersky. *The Scala Language Specification*. EPFL, 2014. Version 2.9.

[134] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *20th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, October 2005.

[135] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings of 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, July 2003.

[136] Martin Odersky, Olivier Blanvillain, Fengyun Liu, Aggelos Biboudis, Heather Miller, and Sandro Stucki. Simplicitly: Foundations and applications of implicit function types. *Proc. ACM on Programming Languages*, 2 (POPL), December 2017.

[137] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *25th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2010.

[138] Bruno C.d.S. Oliveira, Tom Schrijvers, Wontae Choi, Wonchan Lee, and Kwangkeun Yi. The implicit calculus: A new foundation for generic programming. In *33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2012.

[139] OpenJDK javac. The `javac` compiler. `http://hg.openjdk.java.net/`.

[140] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *16th European Conf. on Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, 2002.

[141] Leo Osvald, Grégory Essertel, Xilun Wu, Lilliam I. González Alayón, and Tiark Rompf. Gentrification gone too far? Affordable 2nd-class values for fun and (co-)effect. In *2016 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2016.

[142] Luke Palmer. Haskell antipattern: Existential typeclass. https://lukepalmer.wordpress.com/2010/01/24/haskell-antipattern-existential-typeclass, 2010.

[143] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[144] Simon Peyton Jones. Classes, Jim, but not as we know them—type classes in Haskell: What, why, and whither. In *23rd European Conf. on Object-Oriented Programming*, 2009.

[145] Benjamin Pierce and Martin Steffen. Higher-order subtyping. *Theoretical Computer Science*, 176(1), 1997.

[146] Andrew M Pitts and Ian Stark. Operational reasoning for functions with local state. *Higher order operational techniques in semantics*, 1998.

[147] Gordon Plotkin and John Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, Feb 2003.

[148] Gordon Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, Volume 9, Issue 4, December 2013.

[149] Piotr Polesiuk. IxFree: Step-indexed logical relations in Coq. In *3rd International Workshop on Coq for Programming Languages (CoqPL)*, 2017.

[150] Xin Qi and Andrew C. Myers. Homogeneous family sharing. In *25th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, October 2010.

[151] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schu-

man, editor, *New Directions in Algorithmic Languages*. Institut de Recherche d'Informatique et d'Automatique, Le Chesnay, France, 1975. Reprinted in [87], pages 13–23.

[152] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, 1983.

[153] Martin P. Robillard and Gail C. Murphy. Designing robust Java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-8)*, 2000.

[154] Rust 2014. Rust programming language. `http://doc.rust-lang.org/0.11.0/rust.html`, 2014.

[155] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *26th European Conf. on Object-Oriented Programming*, 2012.

[156] Sukyoung Ryu. `ThisType` for object-oriented languages: From theory to practice. *ACM Trans.on Programming Languages and Systems*, 38(3), April 2016.

[157] C. Schaffert et al. An introduction to Trellis/Owl. In *1st ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, September 1986.

[158] Jeremy G. Siek. The C++0x concepts effort. Arxiv preprint arXiv:1201.0027, December 2011.

[159] Jeremy G. Siek and Andrew Lumsdaine. Essential language support for generic programming. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

[160] Jeremy G. Siek and Andrew Lumsdaine. A language for generic programming in the large. *Science of Computer Programming*, 76(5):423–465, 2011.

[161] Saurabh Sinha and Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9), September 2000.

[162] Yannis Smaragdakis and Don Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, April 2002.

[163] Guy L. Steele, Jr. *Common LISP: the Language*. Digital Press, second edition, 1990. ISBN 1-55558-041-6.

[164] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[165] Bjarne Stroustrup. The C++0x "remove concepts" decision. *Dr. Dobbs*, July 2009.

[166] Martin Sulzmann, Gregory J. Duck, Simon Peyton-Jones, and Peter J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1):83–129, January 2007.

[167] *Java Language Specification*. Sun Microsystems, version 1.0 beta edition, October 1995. Available at `ftp://ftp.javasoft.com/docs/\penalty\z@javaspec.ps.zip`.

[168] SunFlow. SunFlow: the open-source rendering engine. Open-source software, 2007.

[169] swift.org. Swift programming language. https://docs.swift.org/swift-book, 2014.

[170] Kresten Krab Thorup. Genericity in Java with virtual types. In *European Conf. on Object-Oriented Programming*, 1997.

[171] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[172] Mads Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1998.

[173] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahé, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *2004 ACM Symposium on Applied Computing*, 2004.

[174] Marko van Dooren and Eric Steegmans. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In *20th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2005.

[175] Bill Venners. Failure and exceptions: A conversation with James Gosling, Part II. http://www.artima.com/intv/solid.html, 2003.

[176] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *16th ACM Symp. on Principles of Programming Languages (POPL)*, 1989.

[177] Philip Wadler. Theorems for free! In *4th Conf. on Functional Programming Languages and Computer Architecture (FPCA)*, September 1989.

[178] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, 2009.

[179] Philip Wadler et al. The expression problem, December 1998. Discussion on Java-Genericity mailing list.

[180] Rod Waldhoff. Java's checked exceptions were a mistake. http://radio-weblogs.com/0122027/stories/2003/04/01/JavasCheckedExceptionsWereAMistake.html, 2003.

[181] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *21st ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, October 2006.

[182] Stefan Wehr and Peter Thiemann. JavaGI: The interaction of type classes with interfaces and inheritance. *ACM Trans.on Programming Languages and Systems*, 33(4):12:1–12:83, July 2011.

[183] Manuel Weiel, Ingo Maier, Sebastian Erdweg, Michael Eichberg, and Mira Mezini. Towards virtual traits in Scala. In *Scala '14*, July 2014.

[184] Westley Weimer and George C. Necula. Exceptional situations and program reliability. *ACM Trans.on Programming Languages and Systems*, 30(2), March 2008.

[185] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[186] Yizhou Zhang and Andrew C. Myers. Familia: Unifying interfaces, type classes, and family polymorphism. In *2017 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, October 2017.

[187] Yizhou Zhang and Andrew C. Myers. Unifying interfaces, type classes, and family polymorphism. Technical report, September 2017.

[188] Yizhou Zhang and Andrew C. Myers. Abstraction-safe effect handlers via tunneling : Technical report. Technical Report 1813–60202, Cornell University Computing and Information Science, November 2018.

[189] Yizhou Zhang and Andrew C. Myers. Abstraction-safe effect handlers via tunneling. In *45th ACM Symp. on Principles of Programming Languages (POPL)*, January 2019.

[190] Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. Lightweight, flexible object-oriented generics. In *36th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2015.

[191] Yizhou Zhang, Matthew C. Loring, Guido Salvaneschi, Barbara Liskov, and Andrew C. Myers. Genus: Making generics object-oriented, expressive, and lightweight. Technical Report 1813–39910, Cornell University Computing and Information Science, June 2015.

[192] Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. Accepting blame: Expressive checked exceptions. Technical Report 1813–43784, Cornell University Computing and Information Science, April 2016.

[193] Yizhou Zhang, Guido Salvaneschi, Quinn Beightol, Barbara Liskov, and Andrew C. Myers. Accepting blame for safe tunneled exceptions. In *37th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, June 2016.