The Thesis committee for Scott Lasater Kilpatrick

Certifies that this is the approved version of the following thesis:

Ad Hoc: Overloading and Language Design

APPROVED BY

SUPERVISING COMMITTEE:

William R. Cook, Supervisor

Eric E. Allen, Supervisor

Ad Hoc: Overloading and Language Design

by

Scott Lasater Kilpatrick, B.S.

THESIS

Presented to the Faculty of the Graduate School

of The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCES

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2010

Informal, quasi-empirical mathematics does not grow through a monotonous increase in the number of indubitably established theorems but through the incessant improvement of guesses by speculation and criticism, by the logic of proofs and refutations.

Proofs and Refutations

Imre Lakatos

Acknowledgments

First and foremost, I'd like to thank Greg Lavender, whose early support was instrumental for my academic career. Many thanks also go to the infinitely patient and supportive Eric Allen, fellow intern Justin Hilburn, and the past and present researchers of the Sun Labs Programming Languages Research Group: David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, and Guy Steele. Additionally, William Cook provided useful insights and made my work on Fortress possible.

Finally, thanks to the many accommodating Austin cafés and the background music of Dave Brubeck, Duke Ellington, and *Air*.

Ad Hoc: Overloading and Language Design

by

Scott Lasater Kilpatrick, M.S.C.S.
The University of Texas at Austin, 2010

Supervisors: William R. Cook
Eric E. Allen

The intricate concepts of ad-hoc polymorphism and overloading permeate the field of programming languages despite their somewhat nebulous definitions. With the perspective afforded by the state of the art, object-oriented Fortress programming language, this thesis presents a contemporary account of ad-hoc polymorphism and overloading in theory and in practice. Common language constructs are reinterpreted with a new emphasis on overloading as a key facility.

Furthermore, concrete problems with overloading in Fortress, encountered during the author's experience in the development of the language, are presented with an emphasis on the *ad hoc* nature of their solutions.

Table of Contents

Chapter 1

Introduction

Strachey [62] coined the term *ad-hoc polymorphism*, and Cardelli and Wegner [15] refined it as a kind of polymorphism in which "a function works, or appears to work, on several different types [...] and may behave in unrelated ways for each type." They describe *overloading* as a kind of ad-hoc polymorphism in which the context surrounding a particular use of a name determines which function it denotes. This idea of overloading materializes in many programming languages today — from practical to theoretical, from functional to object-oriented — albeit with tremendous differences in scope, presentation, and implementation.

Along with overloading comes the paired notion of *dispatch*: if the former specifies *what* can be phrased in the language, it is the latter that specifies *how* those phrases are interpreted. One cannot easily explain overloading and dispatch in a particular language independently of one another, given that the practical limitations of dispatch sometimes govern the specification of overloading in unexpected ways. For example, many object-oriented languages exhibit this *ad hoc* ad-hoc polymorphism due to a fixed, restrictive dispatch semantics, known as *single dynamic dispatch* (Section 3.2). In the Fortress programming language [3], however, overloading is emphasized as a fundamental language concept; the cost of dispatch is then subsumed as a requisite, resulting in a more consistent interpretation of over-

loading. With this unique interpretation Fortress poses some interesting research problems, as well as design challenges that must be addressed.

This thesis is divided into three parts: In Chapter 2 overloading and its related concepts are introduced, motivated, and explained with a new perspective, including the identification of three primary language features induced by overloading. Chapter 3 presents overloading as it exists in formal calculi and more practical programming languages, with emphasis on type classes, object classes, and multiple dispatch systems; each system is viewed with the lens of the generalized description of overloading from Chapter 2. In Chapter 4, concrete problems with Fortress overloading are presented and solved, based on the author's experience in the development of the language. Also presented are further observations about language design, inspired by Fortress. Finally, Chapter 5 concludes this thesis with some final thoughts on overloading and language design.

Chapter 2

Overloading in general

2.1 Intuition and motivation

Overloading is a linguistic phenomenon wherein a single name can have multiple interpretations, with a unique one implicitly determined from the context. A name that exhibits this behavior is said to be **overloaded**, and each possible interpretation is called an **overloading** of that name.[1] Such an informal definition unfortunately necessitates further definitions of its constituents: The concept of a *name* seems straightforward enough, but what is an *interpretation*? Is the interpretation of a name its annotated type? What about a *context*? The enclosing lexical scope of the name, perhaps?

This generality actually speaks to the broad relevance of overloading to language. Informally introduced to the world of programming languages as *ad-hoc polymorphism*, the phenomenon really existed long before computer science. In the subsequent sections we discuss overloading in these earlier languages[2] and then its arrival in programming languages.

[1] Note that the word "overloading" is itself overloaded in the English language.

[2] For consistency, the somewhat anachronistic term *overloading* will be used throughout.

2.1.1 Natural languages

Natural human languages, which lack any deterministic, formal semantics whatsoever, exhibit a phenomenon similar to overloading. A sentence in natural language might involve a word with context-dependent meaning, much like a program does. Indeed, the English word "program" has twelve distinct *senses* according to the *New Oxford American Dictionary, 2nd Edition*, five of which are reproduced in Figure 2.1. That a word might ambiguously denote one of many senses is known in the linguistics community as *polysemy*, and the act of determining the correct sense is called *word sense disambiguation*, an ubiquitous computational task in natural language processing [38, Ch. 19].

The definition in Figure 2.1 presents five distinct senses of the word "program" organized into an ontology: At the top level, senses are categorized by their *parts of speech*, here *noun* and *transitive verb*. Each boldface number represents a unique sense for the word when used with that part of speech; for example, the two unique noun entries given for "program". Each sense might involve *subsenses* thereof; for example, the verb entry given for "program" defines a base sense and two subsenses. A subsense, also called a *hyponym* [38, §19.2.2], denotes a more specific meaning than its parent. Later, in Section 3.3.8, one might realize a connection between polysemy and the expressive overloading of Fortress.[3]

[3] In particular, aside from the relation between hyponymy and subtyping, the partitioning of senses by parts of speech corresponds to the notion of exclusion in Fortress.

pro·gram
noun

1 a planned series of future events, items, or performances *: a weekly program of films*
2 a series of coded software instructions to control the operation of a computer or other machine.

verb [trans.]

1 provide (a computer or other machine) with coded instructions for the automatic performance of a particular task *: it is a simple matter to program the computer to recognize such symbols.*
• input (instructions for the automatic performance of a task) into a computer or other machine *: simply **program in** your desired volume level.*
• (often **be programmed**) figurative cause (a person or animal) to behave in a predetermined way *: all members of a particular species are **programmed to** build nests in the same way.*

Figure 2.1: Excerpt of the definition of "program".

2.1.2 Informal mathematics

Informal mathematical notation, as we know it today, has evolved over thousands of years of human discovery and pedagogy. Statements in this widely understood language include

$$\text{(a)}\quad 2+2=4 \qquad \text{(b)}\quad \ln e = 1 \qquad \text{(c)}\quad (e^{-1})\, e = 1,$$

i.e., trivial statements about (a) the sum of two integers, (b) an identity of the natural logarithmic function, and (c) a multiplicative identity of the real number denoted by e. Though a syntactic trick hides among the latter two: the space between ln and e means *apply the* ln *function to* e, but the space

between (e^{-1}) and e means *the product of e^{-1} and e*. That is, space — technically called *juxtaposition* — is usually overloaded in informal mathematical notation.

Despite having two possible meanings, juxtaposition actually admits a straightforward interpretation through analysis of the shapes of its operands. In (b) juxtaposition means *function application* because we know that the left operand, ln, is a function over the real numbers and that the right operand, e, is such a real number. In (c) it means *multiplication* because the left and right operands are elements of the structure of real numbers (which defines such a multiplication operator). And in the well-known "double angle" identity

$$\sin 2\theta = 2 \sin \theta \cos \theta$$

the two distinct meanings coexist peacefully and unambiguously.

As another example of overloading in mathematics, consider the definition of a group homomorphism in abstract algebra below.

> Let G and H be groups and let $f : G \to H$. The function f is a *group homomorphism* if, for every x and y in G,
>
> $$f(x \cdot y) = f(x) \cdot f(y) \, .$$

This definition exhibits two subtle forms of overloading: First, the names G and H refer both to the group structures themselves and to those structures' underlying sets. Second, a single symbol $\cdot$ represents each group's operator — with operands x and y (elements of G) it means G's operator and with operands $f(x)$ and $f(y)$ (elements of H) it means H's.

This interpretation via shape analysis bears a resemblance to *type checking* in a statically typed language, such as the *simply typed lambda calculus*. A static analyzer for a formal language based on mathematical notation might, in the presence of juxtaposition, simultaneously type check a term and choose its meaning (for later evaluation)[4]. Such a mechanism will be explored in more detail in Section 2.3.

2.2 From polymorphism

In 1967 Strachey [62] informally characterized the ideas behind overloading as *ad-hoc polymorphism*, in contrast with *parametric polymorphism*, which he considered the "more regular" kind of polymorphism. He described *polymorphic* functions as those with "several forms depending on their arguments." Strachey's definitions should be understood in context, however, as they predate the seminal works on polymorphism, Girard's *System F* [29] and Reynolds' *polymorphic lambda calculus* [53], by several years.

Cardelli and Wegner [15] expanded on Strachey's classification of polymorphism almost two decades later. They formulated a hierarchy of polymorphism which, at the top level, comprises the categories *universal* and *ad-hoc*. Universal polymorphism describes those functions which operate on all (possibly infinitely many) types that share a common structure; the two subcategories are the familiar *parametric* polymorphism and a new class, *inclusion* polymorphism, which they devised in order to model the

[4]Indeed, a stated goal of the Fortress programming language is to properly type check and interpret mathematical expressions like $n(n+1)\sin 3nx \log\log x$.

subtyping and inheritance of the object-oriented paradigm.

In the following sections we discuss parametric, inclusion, and ad-hoc polymorphism in more detail.

2.2.1 Parametric polymorphism

In [62] Strachey identifies a need to statically analyze the types of polymorphic function applications. He simultaneously describes and motivates parametric polymorphism with an example: the type of the now-standard polymorphic `map` function, which he writes as

$$(\alpha \Rightarrow \beta,\ \alpha\ \texttt{list}) \Rightarrow \beta\ \texttt{list}$$

("where α and β stand for any type"). Essentially, this is identical to the contemporary presentation with universally quantified types [29, 53]

$$\forall\alpha\forall\beta.(\alpha \to \beta) \to \texttt{List}\ \alpha \to \texttt{List}\ \beta$$

with the key difference being the explicitly quantified type variables. Cardelli and Wegner make clear this notion of type parameters for parametric polymorphism in [15], further defining such functions as **generic**[5].

According to Harper's contemporary account [34, Ch. 24], parametrically polymorphic expressions "codify generic (type-independent) behaviors that are shared by all instances of the pattern." That is, a generic function has only a single implementation which can be *instantiated* for multiple (if not infinitely many) types at will; for example, using the polymorphic

[5]We take **generic** functions to mean parametrically polymorphic functions, as Cardelli and Wegner do, throughout. Other authors have used *generic* to describe other notions, *e.g.* Section 3.3.4.

`map` function to transform a list of integers into a list of strings, or vice versa, in a way that can be checked statically. Parametric polymorphism is not limited to functions, however; the type of lists, `List t`, and the value of the empty list, `nil`, are both typically defined once and used parametrically. In this way parametric polymorphism permits *code reuse*, a major practical contribution to software engineering.

A key theoretical result of parametric polymorphism is the notion of *parametricity* (introduced as the *abstraction theorem* by Reynolds [54]): the genericity of a polymorphic type correlates with the deducibility of an inhabiting expression's behavior [34, §24.3]. For example, from the polymorphic type $\forall\alpha\,.\,\alpha \to \alpha$, one can determine the *exact* behavior of any expression inhabiting this type; in fact, there is only a single expression, the polymorphic identity function $\Lambda\alpha\,.\,\lambda x\!:\!\alpha\,.\,x$. Wadler exuberantly declared this type-based deductive power as "theorems for free!" [65] Parametric polymorphism therefore affords program analysis and reasoning beyond its obvious application to software engineering.

2.2.2 Inclusion polymorphism

Aside from parametric, the other kind of universal polymorphism, as defined by Cardelli and Wegner, is inclusion polymorphism. In this kind of polymorphism, expressions are classified by a preordered family $(\mathcal{S}, \sqsubseteq)$, and any expression in set $S' \in \mathcal{S}$ can be used where one in set $S \in \mathcal{S}$ is expected, for any S such that $S' \sqsubseteq S$. Such an expression is polymorphic over all the supersets of its set, S, while only having the single representation S.

Subtyping and class-based inheritance both fall into this category. With subtyping, $\mathcal{S}$ is the set of all types and $\sqsubseteq$ is the subtyping relation

```
# the classes of all living things
class LivingThing():
  # living things are created with an age field
  def __init__(self, age):
    self.age = age

# the class of human beings - inherits from LivingThing
class Human(LivingThing): ...

# define a function that compares ages of living things
def age_le(thing1, thing2):
  return thing1.age <= thing2.age

# some_plant is a LivingThing; steve_albini is a Human
... age_le(some_plant, steve_albini) ...
```

Figure 2.2: Examples of inclusion polymorphism with class-based inheritance in Python. The `compare_ages` function works transparently on the `Human` object, `steve_albini`.

on types. Any expression of type S' can be used wherever S is expected, such as in a variable binding or function application. With inheritance, $\mathcal{S}$ is the set of all object classes and $\sqsubseteq$ is the inheritance or extension relation on classes. Any member defined for objects of class S can be accessed on an object of class S' since S' inherits the members from S. Figure 2.2 contains sample code illustrating inclusion polymorphism in the form of class-based inheritance in the dynamically typed language Python.

2.2.3 Ad-hoc polymorphism

Historically, ad-hoc polymorphism has characterized arithmetic operators which have definitions on disparate types (see Figure 2.3). The term bears the pejorative phrase *ad hoc* because, according to Strachey [62], with this polymorphism "there is no single systematic way of determining the type of the result from the type of the arguments." Notably, Strachey describes ad-hoc polymorphism as an inherent quality of programming to be studied; in [62] he never explains *why* a language should support it, except implicitly as a convenience for programmers accustomed to informal mathematical notation (Section 2.1.2). Strachey also recognizes the need for "rules of limited extent which reduce the number of cases" of these multiply defined operators and foretells that such rules would be "ad hoc both in scope and content." (Rules that do precisely this will be presented with the Fortress programming language in Section 3.3.8.)

Whereas Strachey implies that ad-hoc polymorphism is irregular, Cardelli and Wegner [15] assert that it is merely "apparent" polymorphism. They explain that "ad-hoc polymorphism is obtained when a function works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type." Their definition of *overloading* largely matches that presented at the beginning of Section 2.1, with the important exception that they describe it as a "convenient syntactic abbreviation." They describe another kind of ad-hoc polymorphism, *coercion*, as on the contrary a *semantic* operation. Phrasing overloading as a syntactic convenience certainly agrees with its use in mathematics, but this unduly limits its functionality in programming. In Section 2.5 we shall see how overloading also achieves important abstraction and special-

```
// integer addition
opr + (m: Integer, n: Integer): Integer = ...

// floating point addition
opr + (a: Real, b: Real): Real = ...

// string concatenation
opr + (s1: String, s2: String): String = ...

2 + 2          // evaluates to 4
3.14 + 2.72    // evaluates to 5.86
"foo" + "bar"  // evaluates to "foobar"
```

Figure 2.3: Common examples of ad-hoc polymorphism via overloading in pseudocode: + is overloaded for addition and concatenation.

ization properties.

In this thesis, unless otherwise specified, *overloading* is taken to be synonymous with *ad-hoc polymorphism*, while *polymorphism* and *genericity* are taken to be synonymous with *parametric polymorphism*.

2.3 Mapping names to interpretations

So far we have only peripherally described the implicit mapping of an overloaded name to a unique interpretation. However, this name resolution, called **dispatch**, plays an important role in both the specification and the implementation of overloading: it is the mechanism by which the implicit control flow of overloading is realized.

2.3.1 Overloading as binding

All programming languages, at some level, support the binding of a name to a syntactic object. With *lexical scoping* a binding of name x only exists within some lexically apparent extent, and references to x are determined statically; with *dynamic scoping* each bound name x has its own stack, to which new bindings of x are pushed at run time, and from which the top is accessed as the current value of x. In a language with dynamic scoping, or with lexical scoping and *shadowing*, binding can be seen as a form of overloading in which the dispatch mechanism trivially accesses the most recently defined binding.

The means of name lookup (*i.e.* dispatch) varies between lexical and dynamic scoping: the former happens statically and the latter at run time. More than an implementation concern, this distinction affects program behavior. For example, the following lambda calculus expression (with `let` and integers) evaluates to `1` with lexical scoping and shadowing and to `2` with dynamic scoping:

```
let x = 1 in
  let f = λ y. x in
    let x = 2 in f 0
```

As with name lookup in conventional binding, the dispatch semantics of a system with overloading affects program behavior; hence the statement in Chapter 1 that a specification of overloading is incomplete without describing dispatch.

2.3.2 Dispatch in context

The notion of *context* introduced in the definition of overloading from the beginning of this chapter has remained largely unspecified. Up to this point it has generally been interpreted as the argument type to which an overloaded function is applied. (In Chapter 3 we shall see that this interpretation is common in practice.) Context might also involve other type information like the expected type of the application.

More generally, though, dispatch is not confined to statically determined type information. Dispatch is not even confined to the *static* or compile-time phase of execution; it can also be performed in the *dynamic* or run-time phase of execution. During run time, instead of reasoning about the context using the *types* of subexpressions, the dispatch mechanism can reason about the precise *values* of those subexpressions. Thus the context established dynamically is in a sense "more specific" than that established statically.

When a system only performs dispatch in the static phase, we say it exhibits **static dispatch** semantics; when it performs dispatch also in the dynamic phase, we say it exhibits **dynamic dispatch** semantics. Static and dynamic dispatch are also known, respectively, as *type-based* and *value-based* dispatch, for obvious reasons.

2.3.3 Abstract and concrete overloadings

An abstract overloading specifies only the signature of an interpretation[6] rather than an interpretation itself; it is a fictional overloading that only exists for static analysis. A concrete overloading specifies the signature and the definition of an interpretation; *i.e.* it is an overloading that may actually be executed at run time.

For example, an abstract overloading for + might have signature $\forall \alpha \in \texttt{Numeric} \,.\, \alpha \to \alpha \to \alpha$, where α ranges over all types in the set of numeric types, `Numeric`. Concrete overloadings that satisfy this abstract one include those with signatures `Int` $\to$ `Int` $\to$ `Int` and `Complex` $\to$ `Complex` $\to$ `Complex`, whereas the concrete overloading with signature `String` $\to$ `String` $\to$ `String` does not satisfy it. Because an abstract overloading contains no definition (and thus no code), a crucial invariant must be maintained: dispatch must have selected a concrete overloading for any overloaded name reference before the execution of that reference.

For any abstract overloading A, there exists some (possibly empty) set of overloadings $\mathcal{D}_A$ that satisfy it. If every overloading in $\mathcal{D}_A$ is concrete, then A is called the **principal overloading** of $\mathcal{D}_A$; moreover, if the set of overloadings for the name f is exactly $\{A\} \cup \mathcal{D}_A$, then A is further called the **principal overloading** of f. As a helpful analogy, one can think of abstract overloadings as types and concrete overloadings as values. Systems in which only concrete overloadings satisfy abstract ones correspond to systems in which values inhabit only a single type, and systems in which any overloading may satisfy an abstract one correspond to systems with

[6] Alternatively, an abstract overloading declares a *shape*, *type*, *specification*, *scheme*, or *template*.

subtyping. If the name f has a principal overloading A, a reference to f corresponds to an expression whose *principal type*, the most general type it inhabits, is the type of A [23]. Continuing the analogy, when an overloaded name f does have a principal overloading A, then the type of f can be *inferred* as the signature of A.

2.4 Parametric overloading and extensibility

Strachey states in [62] that "polymorphism of both classes [ad-hoc and parametric] presents a considerable challenge to the language designer, but it is not one which we shall take up here," and Cardelli and Wegner say nothing about mixing ad-hoc and parametric polymorphism in [15]. Overloading of polymorphic functions has indeed presented some difficulty in programming language design. For example, consider the overloaded functions in Figure 2.4 with varying genericity. In Chapter 3 we shall explore systems that support overloaded polymorphic functions, including the author's work on Fortress.

$foo(xs\colon \mathrm{List}[\![\mathrm{Any}]\!], ys\colon \mathrm{List}[\![\mathrm{Any}]\!])$	$\circledast$ 1
$foo[\![T]\!](xs\colon \mathrm{List}[\![T]\!], ys\colon \mathrm{List}[\![T]\!])$	$\circledast$ 2
$foo[\![T, U]\!](xs\colon \mathrm{List}[\![T]\!], ys\colon \mathrm{List}[\![U]\!])$	$\circledast$ 3

Figure 2.4: An example in Fortress of three overloadings each parameterized by a different number of type variables.

Alternatively, mixing the two classes of polymorphism might involve parameterizing a function definition over all types for which an overloaded name is defined, like the Haskell function

```
double :: Num a => a -> a
double x = x + x
```

This function is parametric over all types `a` that satisfy the predicate `Num a` and thus have the `+` operator defined. This property is essential for extensible overloading: without ranging over all possible types that support the overloaded name, that name could only refer to the fixed set of interpretations known during compilation of that name. In general we consider systems with an "open world" approach in which overloadings defined modularly can be augmented with those defined in client programs. For example, we hope that the `double` function will work on overloadings of `+` defined by some client program, such as `Quaternion` $\rightarrow$ `Quaternion` $\rightarrow$ `Quaternion`, which are unknown to the program that defines `double`.

2.5 A means to multiple ends

There appear to be three fundamental programming language features induced by overloading (among other means), although a literature search yields no such exposition. These overloading *raisons d'être* are outlined below and briefly related to their realization in practice. (Chapter 3 will explore these ideas in practice in more detail.)

2.5.1 Convenience

In natural languages and informal mathematics it is simply more convenient to overload names and operations than to uniquely refer to each interpretation. Whether that means writing explicit dot symbols for multiplication or subscripting each use of an ambiguous word with a unique

identifier from a standard dictionary, precision comes with the cost of more syntax and larger texts.

Sometimes a function is called with the same argument pattern often enough to justify an additional overloading that is implemented in terms of the old one. For example, `println()` and `println(Object)` overloadings of `println(String)` in place of repeated calls like `println("")` and `println(obj.toString)`, or constructor overloadings that provide default values for certain fields. These additional overloadings provide convenient access to semantically related behavior, but they are by no means necessary. One could think of them as syntactic sugar (*cf.* Cardelli and Wegner's statement) for functions called `println_String`, `println_void`, and `println_Object`.[7] In that case, the desugaring would necessarily be type-directed, since `println(x)` could desugar to either `println_String(x)` or `println_Object(x)` depending on the type of `x`. Further, what if the type of `x` is `Object` at compile time and `String` at run time? Desugaring to `println_Object` would mean that the "wrong" function is called during execution.[8]

Other times a function name is used so commonly for two completely different argument types that the overloadings are provided as an abuse of notation — such as integral and floating point arithmetical operations. Standard ML offers overloading for arithmetic operators and record types as a convenient compromise, but leaves the rest of the work of naming and abstraction to the module system [33, §8.3]. The standard library of

[7] Even if overloading could be desugared out of a program, it still provides a convenient facility for programmers; `let`-binding is sugar for a lambda abstraction and application, but this facility is still ubiquitous among programming languages.

[8] In the case of `println` this distinction is probably moot.

OCaml, on the other hand, provides distinct operators like `+.` and `*.` for floating point operations.[9]

2.5.2 Abstraction

Most programming languages offer facilities for defining and using abstract data types. Each representation type for an abstract type is defined along with some core, axiomatic functions that designate the behavior of the abstract type, like `insert` and `empty` for the abstract type `Set`. The abstract type and its abstract functions form a contract for users: any representing type and functions that implement the abstraction may be used in its place. Client programs then refer to the abstraction but a specific representation is actually used during execution.

This general notion of data abstraction is modeled with existentially quantified types in the polymorphic lambda calculus and in System F. An existential type comprises an abstract type variable representing each concrete type and a constituent type specifying the structure of the whole abstract type. The existential type is inhabited by a representation pair: a concrete type to instantiate the variable and a type for the whole structure (defined by the abstract type structure with the concrete type substituted for the variable). References to an abstract type (*e.g.* `Set`) must be directed to some representation during execution (*e.g.* `TreeSet`), and in order to use that representation, it must first be "opened" over some lexical scope; only in that scope are its concrete definitions available (*e.g.* `insert`). Notably, any reference to, say, `insert` in a client program uniquely maps to a single

[9] See the `Pervasives` module documentation: http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html#6_Floatingpointarithmetic

representation that was explicitly specified in that program. Module-based programming languages like ML take this approach.

An alternate interpretation of data abstraction sees each concrete function existing in the same namespace — all representations are merely overloadings of the abstract definitions. In the Java programming language, an `interface` construct declares abstract methods that must be defined by any implementing class. Every implementation of that interface defines a concrete overloading of each abstract method. In a client program, an invocation of `isEmpty` on a receiver of abstract interface type `Set` refers to the abstract overloading at compile time (`Set.isEmpty()`) and to some concrete overloading at run time (`TreeSet.isEmpty()`). The overloading resolution and inheritance mechanisms of Java guarantee that a unique, concrete overloading will be executed at run time, thus maintaining the aforementioned invariant. Haskell's system of type classes also achieves data abstraction through overloading, albeit with some fundamental differences from Java, as we shall see in Sections 3.1 and 3.2.

2.5.3 Specialization

When one overloading applies to a strict subset of the arguments to which another overloading applies, we say that the former specializes the latter, or that the former is *more specific* than the latter. Generally in this case the more specific overloading's domain (*i.e.* its parameter type) is a subtype of the other's, and knowing about the subtype's structure affords that overloading more information about its parameter.

For example, consider an overloaded function `process` with overloadings $\texttt{process}_1\texttt{(List)}$ and $\texttt{process}_2\texttt{(ArrayList)}$, where `ArrayList` is a

random-access subtype of `List`. Because $\texttt{process}_2$ applies to those expressions of type `List` which also have type `ArrayList`, while $\texttt{process}_1$ applies to *all* expressions of type `List`, we say that $\texttt{process}_2$ is more specific than $\texttt{process}_1$. An `ArrayList` expression has additional structure beyond that of `List` — namely, a random-access array for the list. Thus the definition of $\texttt{process}_2$ can make use of that additional structure and possibly yield a more efficient implementation: perhaps the random-access algorithm of $\texttt{process}_2$ requires constant time in the length of the list while the sequential algorithm of $\texttt{process}_1$ requires linear time.

At first glance, specialization via overloading seems like an unnecessarily indirect route — why not simply refer to the specialized overloading instead of the overloaded name? In the previous example, if a programmer could write the subscript on the overloadings, she could then write $\texttt{process}_2\texttt{(exp)}$ in her programs whenever the expression `exp` is indeed an `ArrayList`, thus circumventing overloading entirely. However, the type of `exp` at run time might be different (*e.g.* a subtype) from that at compile time. In this case, $\texttt{process}_1$ is the only applicable overloading when the programmer writes the code, but $\texttt{process}_2$ might be applicable during execution. The more specific — or "more appropriate" or "more efficient" or simply "better" — overloading is selected automatically as an implementation detail.

In effect, with overloading, the compiler and/or run time environment of the language usurps control from the programmer and exploits its knowledge of the precise values being computed. This inherent lack of programmer control summarizes the opposition to overloading among

programmers. Others, including the author of this thesis, view the above benefits of implicit control flow as outweighing that cost.

Chapter 3

Overloading in the field

3.1 Type classes

Originally introduced by Wadler and Blott [66] for Haskell [50], **type classes** have become a popular facility for ad-hoc polymorphism. They devised type classes as a means to specify type-safe definitions of equality (thus deprecating ML's then notion of `eqtypes`) and, as usual, arithmetic operations. Since then type classes have been adapted for use in other languages, such as modular type classes for ML by Dreyer *et al.* [25], `concepts` for C++ by Siek [56, 57], generalized interfaces for Java by Wehr *et al.* [67], and, more literally, type classes for Coq by Sozeau and Oury [59].

3.1.1 Overview

Type classes encapsulate overloaded function declarations, with separate *instances* that define the behavior of those functions (called *class methods*) for any particular *predicated type*. Parametric polymorphism is then augmented to express type class constraints, providing a way to quantify a type variable — and thus a function definition — over all types that instantiate the type class. Kaes, in a similar, independently discovered system [39], dubbed the parameterization of a function over all interpretations of its overloaded references as *parametric overloading*.

In the type system, a type class `C` acts as a predicate over types: the

predicate `C` τ is satisfied iff an implementing instance of `C` is provided for type τ. Each predicated type consists of a constituent type and a set of class constraints on type variables, $\texttt{C}_i\ \texttt{a}_i$, that must be satisfied. The methods declared in a class `C` are top-level and available in any scope, but for each reference a particular instance's definition of that reference must be executed at run time. The implementation chosen for a reference is located within the current lexical scope — it is either defined in a top-level instance definition or its existence is hypothesized. For example, the `double` function from Section 2.4 (reproduced in Figure 3.1) posits the existence of an instance satisfying `Num a` which is required for the reference to `+`. Such an instance must be supplied so that some interpretation of the overloaded reference will be executed.

3.1.2 Analysis of overloading features

Abstraction Type classes exhibit abstraction through overloading exactly as described in Section 2.5.2: a constraint `C a` directly corresponds to an abstract data type whose type structure is given by the declared method types in the class `C`. As an implementation, Wadler and Blott described a translation of a type class-based language into the polymorphic lambda calculus (without type classes).[1] This *dictionary passing transformation* converts type classes into polymorphic dictionary types, instances into corresponding dictionary expressions that hold the method implementations, and overloaded references into dictionary lookups. Whereas the original code implicitly ex-

[1] Wadler, along with Hall, Hammond, and Peyton Jones, formalized this elaboration semantics for Haskell [32] as part of its typing relation, which Faxén more completely (but less concisely) defined much later in [27].

```
--------- originally compiled library code ---------
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  neg :: a -> a

-- The ‘prim‘ functions are primitive numeric
-- operations implemented specially.
instance Num Int where
  (+) = primIntAdd
  (*) = primIntMul
  neg = primIntNeg

instance Num Float where
  (+) = primFloatAdd
  (*) = primFloatMul
  neg = primFloatNeg

-- The (Num a) and (Num b) assumptions are used
-- in the implementations of the methods.
instance (Num a, Num b) => Num (a, b) where
  (x1, x2) + (y1, y2) = (x1 + y1, x2 + y2)
  (x1, x2) * (y1, y2) = (x1 * y1, x2 * y2)
  neg (x1, x2) = (neg x1, neg x2)

--------- separately compiled client code ---------
instance Num Quaternion where ...

double :: Num a => a -> a
double x = x + x
```

Figure 3.1: An example usage of overloading in Haskell.

presses data abstraction with an overloaded function reference, the translated code explicitly opens the abstract data type for the required class and

accesses its concrete representation of that function. Dreyer, Harper, and Chakravarty provide a complete account of this correspondence between the implicit data abstraction of type classes and the explicit data abstraction of ML modules in [25].

Convenience In systems with type classes, overloaded functions must be contained in some type class, and their signatures must be equivalent, modulo substitution of the *class parameter*. (Kaes's system also requires each overloaded name to have a single *overloading scheme*.) This uniformity is necessary for an overloaded function call to admit a principal type; with a principal type for some function call's context, the type checker can determine the constraints under which a correct overloaded definition will be found. Because of this requirement, type classes are ill-suited for fixed, *ad hoc* sets of overloaded functions like

$$\mathit{println}() : () = \mathit{println}(\text{“”})$$

$$\mathit{println}(s : \text{String}) : () = \ldots$$

or functions lacking uniform variance in the domain and range[2] like

$$\mathit{bar}(x : \mathbb{Z}) : \text{Boolean} = (x = 0)$$

$$\mathit{bar}(x : \text{Boolean}) : \mathbb{Z} = \texttt{if } x \texttt{ then } 1 \texttt{ else } 2 \texttt{ end}$$

$$\mathit{bar}(x : \text{String}) : \text{String} = x$$

[2]With *multi-parameter type classes* [26], one could define functions as these. A reference to the method `bar`, however, would require an explicit type annotation like `:: Int -> Bool` to apply to an `Int`.

The rich type inference afforded by type classes also contributes to convenience, as programmers may omit most (if not all) specifications of type class constraints for parametric overloading.

Specialization In many type class systems, no two instances can "overlap" in which types they cover. This means that there can be no ambiguity when choosing an interpretation and effectively rules out specialization. To define a specialized function like `process` from Section 2.5.3, a single definition must be parameterized over some type class that specifies overloaded list operations for traversal and indexed access. However, this `process` function must have the exact same code for all list instances — the only algorithmic variation between the instances occurs in the implementations of the class methods. It is possible (and likely) that the class methods and the implementations thereof do not offer the algorithms needed for this particular function, `process`. Then the only way a function can be specialized is if the specialization lies entirely within the domain of the predefined methods of the type class it concerns. Since individual instances cannot be modified after their declaration and since no two instances can be defined on the same types, client programmers cannot define overloaded functions on existing types that are specialized to their needs.

Dispatch Each type class method declares a top-level function available in any scope and uniquely determines the class that declared it, so we can "lift" it to an abstract overloading (as in Section 2.3.3): a method `f` $::\tau$ declared in class `C a` lifts out to an abstract overloading named `f` with signature `C a` $\Rightarrow \tau$. Furthermore, each definition of `f` in an instance $\theta \Rightarrow$ `C`

υ lifts out to a concrete overloading of `f` with signature $[\upsilon/\mathtt{a}]\theta \Rightarrow [\upsilon/\mathtt{a}]\tau$. To resolve an overloaded reference into an interpretation, the type checker finds a concrete overloading whose type unifies with the expected type of the reference and whose predicate is satisfied in the calling scope.

During type checking (*i.e.* statically) dispatch must select some overloading for each reference to `f`. Since no two instances can overlap there can be at most one matching overloading; this overloading may be abstract (declared in the class itself) or concrete (defined in some instance). Dispatch must guarantee that a concrete overloading is executed at run time. With dynamic dispatch, a concrete overloading replaces the abstract one at run time, but systems with type classes do not typically implement dynamic dispatch semantics. Instead, these systems exhibit what could be described as **deferred static dispatch**; for an overloaded reference in some function definition, the dispatch choice is *deferred* to the calling function.[3] With this semantics, dispatch is decided statically and those static decisions are propagated dynamically.

Deferred static dispatch is implemented easily with the dictionary passing transformation: Each type class constraint on a function's signature becomes an explicit dictionary parameter which must be supplied when calling this function. That dictionary contains the deferred implementation for the overloaded references in the called function.

[3]Other authors describe type class dispatch semantics as "type-based." This suggests that the types in the context of an overloaded reference determine the chosen implementation, but since the implementation is passed in as an argument, this is not the case.

Extensibility When a function such as `double` is parameterized over instances of a type class, any instance thereof can be used in the function, even those declared in a separately compiled program. However, as mentioned above, only a single instance can be defined on any particular type. The modular type classes of Dreyer *et al.* [25] and the *named instances* of Kahl and Scheffczyk [40] provide ways to circumvent this restriction.

3.1.3 System O

Odersky, Wadler, and Wehr define a variation on type classes in [49] called System O which actually omits type classes altogether. Instead of specifying overloaded functions in semantically related groups, programs in System O declare overloaded functions directly. Thus each type predicate has the form `m` $::\tau$, meaning that the overloaded name `m` has an implementation at type τ. Because overloaded names are divorced from their interpretations, in System O one can overload + for numbers and for strings; with type classes, + would typically belong to a numeric type class which prevents the inclusion of strings (compare Figures 3.1 and 3.2). System O resembles the original parametric overloading system of Kaes [39] and it extends the Hindley-Milner style of ML more conservatively than do type classes.

Like with type classes, System O admits a rich type inference algorithm. In System O, however, that algorithm is complete: no type annotations are ever required. Every program also has a dynamic semantics irrespective of its type. (With type classes a program might admit two different types, and its dynamic behavior could vary depending on which type was chosen statically.) Furthermore System O can be implemented with a very

```
--------- originally compiled library code ---------
over (+)

-- The ‘prim‘ functions are primitive numeric
-- operations implemented specially.
inst (+) :: Int -> Int -> Int
     (+) = primIntAdd

inst (+) :: Float -> Float -> Float
     (+) = primFloatAdd

-- Able to define string concatenation overloading now.
inst (+) :: String -> String -> String
     (+) = stringConcat

-- Able to define unary plus too.
inst (+) :: Int -> Int
     (+) x = x

--------- separately compiled client code ---------
inst (+) :: Quaternion -> Quaternion -> Quaternion
     (+) = ...

double :: ((+) :: a -> a -> a) => a -> a
double x = x + x
```

Figure 3.2: An example usage of an overloaded addition operation in System O.

similar dictionary passing transformation, taking a well-typed System O program into a well-typed System F program.

Unlike type classes and Kaes's system [39], each concrete overloading can have a unique type structure; there is no declared overloading scheme or template that all implementations must obey. System O does enforce

its own restrictions on overloading types though: each overloading constraint's type must start with a type variable in order to guarantee complete type inference.

The analysis of overloading in System O varies little from that of type classes. However, System O makes explicit the connection to abstract and concrete overloadings in the generalized account from Chapter 2: instance declarations directly define concrete overloadings, while local overloading type assumptions declare abstract overloadings. These abstract overloadings may be used statically but obviously cannot be used dynamically. The dictionary passing transformation guarantees that at run time a concrete overloading is provided in place of each abstract one.

3.1.4 λ^{O}

Shields and Peyton Jones recognized the utility of ad-hoc overloading and specialization typically found in object-oriented languages. To model such languages within the context of type class overloading, they devised a system called λ^{O} that incorporates these features and subtyping in Haskell [55].

The type classes of λ^{O} resemble the overloading of System O in that each class specifies a single overloaded name and each instance a single interpretation. A key innovation of λ^{O} is the introduction of *closed classes*: classes of which no further instances may be defined. Closed classes can be used for simplifying type constraints. For example, the following λ^{O} -style overloadings of an addition operation

```
class closed (+) a where a
```

```
instance (+) (Int -> Int -> Int) where primIntAdd
instance (+) (Float -> Float -> Float) where primFloatAdd
```

permit the simplification of the type of the client function `succ` from the seemingly most general one, which involves a constraint on `(+)`, to a much simpler one

```
-- succ :: ((+) (a -> Int -> b)) => a -> b
succ :: Int -> Int
succ x = x + 1
```

because the only possible instance whose second parameter is an `Int` is that defined above. Additionally, those closed definitions mean that the following client function is no longer well-typed because there is no instance on `Bools`:

```
-- f :: ((+) (a -> Bool -> b)) => a -> b
-- f x = x + True
```

Closed classes therefore give finer-grained control of abstraction and extensibility than ordinary type classes.

Another change to type classes in λ^{O} is the inclusion of *overlapping instances*, which are instances that cover non-disjoint sets of implementing types. The main utility of overlapping instances lies in the ability to specialize type class method implementations. Virtually every object-oriented language offers this kind of specialization (see Section 3.2), which inspired its inclusion in type classes for λ^{O} .

3.1.5 Constraint handling rules

Much of the work required by the type system in order to reason about type class-based overloading involves constraints on overloaded names. In [64] Stuckey and Sulzmann generalize the parametric overloading of type classes and System O with *constraint handling rules* (CHRs). The design space of overloading does not significantly change in their system. As the CHRs serve mainly to model the constraint logic in the type system, the deferred static dispatch semantics and type inference are enriched to better distinguish ambiguity and unambiguity.

The CHR system of Stuckey and Sulzmann generalizes type class extensions like *constructor classes* [37], *multiparameter classes* [26], and *functional dependencies* [36]. With CHRs, however, programmers are burdened with an additional metalanguage of constraints. In the opinion of this author, a well-designed programming language instead provides facilities that, on one hand, the programmer utilizes to specify an ontology of program abstractions, and on the other hand, the type system utilizes to verify properties about that ontology.

3.2 Object classes

First introduced in Simula in 1967 [6], and popularized in Smalltalk [30], C++ [63], and Java [31], object-oriented (OO) languages have dominated programming practice. Whereas type classes originate from formal presentations in the research literature, much of the body of knowledge sur-

rounding objects involves informal presentations and implementations.[4] In this section we describe a general OO language, taking care to address the most salient concepts among such languages.

3.2.1 Overview

An *object* encapsulates data and exhibits certain behaviors or *methods* that can be *invoked* on it in order to access its data in a functional manner; when a method is invoked on an object, we call that object the *receiver*. A method is invoked on an object exactly like an ordinary function call, with the exception that the receiver is a syntactically distinct argument to the call. For example, in the expression `x.m(y, z)`, `x` is the receiver, `m` the method, and `x` and `y` the additional arguments, of which there could be zero or more.

A *class* acts as a generating template for objects that share the same behaviors albeit with possibly distinct internal data. Thus every object is an *instance* of some class and respects every method defined in that class. In statically typed OO languages each class corresponds to a type, and any instance of that class has that type. Classes can be organized into an ontology through an *extension* relation. If a class `C` is declared to *extend* another class `D`, then `C` *inherits* each method in `D` and the data encapsulated by `C` is a superset of that of `D`. However, subclasses can *override* inherited methods by redefining them, possibly making use of the new internal data in the subclass. In this way overriding allows one to specialize a method's implementation for the subclass.

With overriding comes multiple interpretations for a single method

[4] Igarashi, Pierce, and Wadler devised Featherweight Java [35] as a formal calculus for the Java type system, partly to rectify this problem.

name (*i.e.* which inherited method definition to use), so we need a kind of dispatch mechanism to determine a unique one. Thus, although it is not typically presented as such, *overriding is a form of overloading*.[5] To decide which interpretation to use, the dispatch mechanism determines which ones are *applicable* for the invocation and then chooses the least applicable interpretation, according to some order. An overloading is applicable for an invocation if the argument types are all subtypes of their corresponding parameter types (including the receiver argument/parameter).

Normally, the interpretation of a method invocation is decided statically based on the types of all the arguments (including the receiver). If the invoked method is a *virtual* method, however, then the interpretation is decided statically for type checking and again at run time: the class of the evaluated receiver — along with only the types of the remaining arguments — decides the interpretation. Figure 3.3 below illustrates the difference between virtual and non-virtual methods. Some languages, like C++, allow users to specify which methods are virtual, while others, like Java, force all methods to be virtual. Note that in dynamically typed OO languages, like Smalltalk, one can think of all methods as virtual since there is no static phase.

3.2.2 Analysis of overloading features

Abstraction In some languages methods can be declared `abstract`, which effectively transforms the method definition into a method obligation (with no implementation). These abstract method obligations are inherited just

[5]This statement is not quite correct for a system that lacks multiple dynamic dispatch semantics, as explained in Section 3.3.1.

```
class A {
  def foo(): Int = 1
  virtual def bar(): Int = 2
}
class B extends A {
  def foo(): Int = -1
  def bar(): Int = -2
}

val b: A = new B
b.foo()   // evaluates to 1
b.bar()   // evaluates to -2
```

Figure 3.3: Example of virtual methods and overriding in OO pseudocode. Note that the method `foo` is not virtual but the method `bar` is, and that `b` has type `A` but is actually an instance of `B` at run time.

like normal methods.[6] Classes which have declared or inherited abstract methods are themselves abstract, and these classes can never exist at run time. Thus each abstract method in a class corresponds to an abstract overloading, and the other methods to concrete overloadings. As usual, at run time only a concrete overloading may be executed; this property is guaranteed since only concrete classes exist at run time.

As discussed in Section 2.5.2, abstract classes and methods provide a means of data abstraction in OO languages. An expression whose type is an abstract class is like an abstract data type, which at run time will be

[6]The propagation of both concrete method definitions and abstract method obligations is a primary function of *mixin modules*. In the original work on the subject, Bracha and Lindstrom called these *definitions* and *declarations* [10], respectively, while more recently Dreyer and Rossberg called them *exports* and *imports* [24].

replaced with an evaluated object, an instance of a concrete class. To further enforce the correspondence to abstract data types, Java provides `interface` constructs [31, Ch. 9] which are entirely abstract entities.

Convenience Many OO languages allow direct, ad-hoc overloading of methods, such as `println(String)` and `println()`, in addition to method overriding. Due to phrasing overriding as a kind of overloading, dispatch largely remains unaffected by this ad-hoc overloading: it must simply consider more interpretations for a method invocation than those inherited or overridden.

Specialization Overriding and virtual methods allow programmers to specialize methods for receiver values. For example, consider the classes `List` and `ArrayList`, with the latter extending the former. The virtual method `get(Int)` defined in `List` accesses the `i`-th element of the list; its implementation might walk the list starting from the 0-th element until the `i`-th is reached — a linear-time operation. However, the random-access `ArrayList` class can override this method to perform a constant-time access for the `i`-th element. Any invocation of `get` on a receiver whose type is `List` but is actually an `ArrayList` at run time will execute the specialized, constant-time implementation, despite the appearance of a linear-time operation during type checking.

Unfortunately, in virtually every OO language, specialization only occurs on the receiver argument of a (virtual) method invocation. Since each overriding method definition only knows about its own representation, it cannot specialize according to the other arguments — Cook called

this principle *autognosis* in his discussion of abstract data types and objects [20]. In Section 3.3 we will explore what is required of OO systems that allow specialization on all arguments.

Dispatch The correspondence of method overriding to overloading results in a correspondence in dispatch semantics. Using the definitions from Section 2.3, choosing an interpretation for the invocation of non-virtual methods corresponds to *static dispatch*. For virtual methods, however, the correspondence is not so direct. We defined *dynamic dispatch* as the case when dispatch determines an interpretation based on the values of all arguments to a function, but in virtual method invocation, only the value of the *receiver* argument is considered. We therefore use the term **single dynamic dispatch**[7] to describe the choice of interpretation for virtual method invocation.

Extensibility As discussed in Section 2.2.2, OO languages exhibit inclusion polymorphism. Any class can be extended by another class[8], possibly in a separately compiled program, thus allowing any instance of the subclass to act as an instance of the superclass. Any function parameter of some class `C` can be inhabited at run time by an object whose class is a subclass of `C`.

Furthermore, object classes, like type classes, exhibit a form of parametric overloading. With object classes, a function definition in which an

[7] Other authors have called this semantics *single dispatch*, *dynamic dispatch*, and *virtual dispatch*.

[8] In many OO languages a class can be declared to have no further subclasses.

abstract method `m` is invoked on an argument `arg` (with an abstract class type) is, obviously, parameterized by that argument. At run time some concrete overloading of `m` will be executed, and the class of the evaluated object which is passed in for `arg` provides that concrete overloading. Therefore the argument `arg` itself acts as the parameter which determines the interpretation of the "overloaded" name `m` — like a type class constraint.

3.2.3 Multiple inheritance

We have so far only considered extension of a single class, *i.e.*, single inheritance. However, it is certainly useful to model classes that inherit from multiple superclasses, particularly when each superclass represents some fine-grained, abstract attribute like `Sized` or `Positional`. Multiple inheritance poses tricky problems for OO language design [58]. For this thesis, the relevant problem concerns the inheritance of concrete method definitions of the same name from two different classes: which definition should be executed when such a method is invoked on an instance of the multiply inheriting subclass? (Figure 3.4 below provides a concrete example.) With single inheritance, no such ambiguity exists.

One solution to this problem is to impose a *linearization* of multiple inheritance [58] so that each class always inherits from one other, such as in Scala [48, §5.1.2]. Another solution is to require any such subclass to override the method in question. In this case, when the method is invoked at run time on an object of the subclass, its own overriding definition will be executed unambiguously.[9] We shall see more of this style of preventing

[9]Because Java `interfaces` only allow abstract method *declarations*, any concrete class

```
class Cowboy {
  def draw() = // destroy
}
class Artist {
  def draw() = // create
}
class CowboyArtist extends { Cowboy, Artist } { }

marlboro_manet.draw()
```

Figure 3.4: Example of ambiguity caused by multiple inheritance in OO pseudocode. `marlboro_manet` is an instance of class `CowboyArtist`, which inherits `draw` from both superclasses.

overloading ambiguity in Section 3.3.

3.2.4 Polymorphism

Canning *et al.* introduced *F-bounded polymorphism* as a basis for (parametric) polymorphism in statically typed OO languages [13]. Like in the second-order lambda calculus with subtyping, System $F_{\leq}$[10] [22], the type variables of universally quantified types can have supertype bounds; this reinforces the notion of parametric overloading discussed previously. For example, if a function is parameterized by a type variable $\alpha <: \texttt{Num}$, where `Num` is an abstract numeric class, then the methods of `Num` can be invoked on an argument of type α.

that extends that interface must provide a definition for those methods. Consequently, Java allows multiple inheritance of `interfaces` only.

[10]The now-standard calculus $F_{\leq}$ has its roots in the language Fun devised by Cardelli and Wegner in the same essay discussed in Section 2.2 [15].

Polymorphism existed in C++ in the form of template classes and template functions, which compilers expand into intermediate source code. Following the work of Bracha *et al.* on GJ [11], "generics" were added to Java. Igarashi *et al.* modeled GJ as part of their Featherweight Java core calculus, calling it Featherweight GJ [35, §3]. These polymorphic languages — and consequently any language based on the Java Virtual Machine [42], like Scala — all make use of *type erasure* for execution. With type erasure, all parametric polymorphism is compiled out of the program so that the run time environment need not consider it. Type erasure sometimes conflicts with overloading in unexpected ways; for example, the following overloaded declarations in Scala are invalid: [11]

```
def foo[T](xs: List[T]): List[T]  // general
def foo(xs: List[Int]): List[Int] // specialized #1
def foo(xs: List[String]): List[String]       // #2
```

The Scala compiler erases the polymorphism from these declarations, resulting in three overloadings with identical signatures: [12]

```
def foo(xs: List): List
def foo(xs: List): List
def foo(xs: List): List
```

(Even if these overloadings were valid, the specialized ones would only be executed when the argument is a `List[Int]` or `List[String]` statically,

[11] The analogous definitions in Java would similarly be invalid.

[12] OO languages generally prohibit overloadings with equal parameter types (including the receiver parameter) because dispatch must distinguish between two applicable overloadings by their parameter types. In the more type-based approach to dispatch found in type class languages like Haskell, this restriction is not always necessary.

due to the way virtual methods work. With multimethods, presented in Section 3.3, this is not the case.)

3.2.5 Local type inference

With polymorphic methods comes type application on invocations. Many functional languages adopt the Hindley-Milner type system which admits complete or near-complete type inference: users never worry about applying the types that instantiate polymorphic functions. Unfortunately, incorporating full type inference into systems that combine subtyping and polymorphism has mostly eluded researchers. To alleviate the burden of explicit type application, languages like Scala (and to some extent, Java) employ local type inference [52], a technique which infers type application for most invocations.

Returning to the overloaded polymorphic functions in Fortress from Figure 2.4, consider the function call $foo(listOfInts, listOfStrings)$. This call could refer to the monomorphic overloading 1, or it might refer to the polymorphic overloadings 2 and 3 with an implicit type application of $[\![\text{Any}]\!]$ (2) or $[\![\mathbb{Z}, \text{String}]\!]$ (3) or even $[\![\text{Any}, \text{Any}]\!]$ (3).[13] The interaction of local type inference should thus be taken into account for overloading.

3.3 Multiple dispatch

As demonstrated in the last section, object methods cannot specialize their implementations according to their argument types. That limitation

[13] Assuming covariant lists, $\text{List}[\![\mathbb{Z}]\!]$ and $\text{List}[\![\text{String}]\!]$ are both subtypes of $\text{List}[\![\text{Any}]\!]$. However, in the current version of Fortress this is not the case.

precludes the kind of specialization that was deemed an important effect of overloading in Section 2.5.3. Some OO languages, on the other hand, provide a facility for *multiple dynamic dispatch*, in which any method argument can be specialized. This semantics originated in OO variants of Common Lisp [60] [7] [8] but has remained absent from mainstream OO languages. In this section we describe multiple dispatch and its challenges, and briefly summarize systems with such semantics.

3.3.1 Overview

An object method whose invocations depend on the representations of *all* argument values is called a **multimethod**. Multimethods are a natural extension of virtual methods: the interpretation for a multimethod invocation is decided dynamically according to the classes of all argument values, whereas the interpretation of a virtual method depends on the class of only the receiver value.

The uniformity with which arguments determine the executed overloading characterizes a more intuitive semantics for overloading and dispatch in OO languages. In single dispatch systems, like those presented in Section 3.2, overriding cannot be *directly* considered a form of overloading precisely because an overriding method definition only specializes the receiver. Figure 3.5 presents a common bug in Java programs due to the improper conflation of dynamic overriding and static overloading: The invocation `w1.equals(w2)` dispatches to the `Object.equals` definition because the static type of `w2` is `Object`, forcing the dynamic dispatch mechanism to

only find interpretations with an `Object` parameter type.[14] If Java exhibited multiple dynamic dispatch semantics, then overriding would truly be a form of overloading, and this example would behave as expected. As we shall see in Section 3.3.3, though, multiple dynamic dispatch brings with it some hefty obstacles.

3.3.2 Analysis of overloading features

The relation of multiple dispatch to the generalized overloading largely follows that of OO languages with single dispatch (Section 3.2.2). The main difference lies in the specializing nature of multimethods and in the expressivity they offer. In single dispatch systems the overloading consists of overriding (for which each overloading has a very uniform signature) and static overloading (which can lead to unexpected run-time behavior). In multiple dispatch systems, programmers are free to overload methods in virtually any manner they wish, specializing any parameter.

3.3.3 Challenges

The design of a multiple dispatch system is complicated primarily by three concerns: ambiguity, modularity, and static checking. We describe these concerns generally and then address them in several prominent multiple dispatch systems.

[14] This particular example is also an instance of the *binary method problem* [12]. However, the unintuitive distinction between overloading and overriding is a more general concern.

```
// implicit superclass to all Java classes
class Object {
  Boolean equals(Object other) {
    // check for referential equality
    return this == other;
  }
}

// a class defined by the programmer
class Widget {
  int state;
  Widget(int s) {
    this.state = s;
  }
  Boolean equals(Widget other) {
    // check for equivalent Widget state
    if (this.state == other.state) return true;
    else return false;
  }
}

Widget w1 = new Widget(5);
Object w2 = new Widget(5);
w1.equals(w2) // false!
```

Figure 3.5: In this Java program, the `Widget.equals` definition is presumed to *override* the `Object.equals` definition but actually *overloads* it.

3.3.3.1 Ambiguity

In a well-typed program, dispatch should never encounter any nondeterminism or ambiguity when choosing an interpretation for a multimethod invocation. Moreover, at run time *some* applicable interpretation must be present for the invocation.

We saw in Section 3.2.3 how ambiguity creeps into OO languages with single dispatch and multiple inheritance. The solutions presented in that context — linearization of the class hierarchy, and requiring an overriding definition in the class that inherits two conflicting definitions — do not entirely apply in the multiple dispatch context. As we shall soon see, ambiguity occurs even when the class hierarchy is linearized, and furthermore the ambiguity occurs at parameters other than the receiver, precluding the straightforward overriding approach.

```
// B is a subtype of A
class A {}
class B extends A {}

def m(x: A, y: B): Int = 1
def m(x: B, y: A): Int = 2

m(new B, new B) // static error
```

Figure 3.6: Scala program with a statically ambiguous method invocation.

Even without the complications of multiple inheritance, what was only *statically* ambiguous in single dispatch systems is *dynamically* ambiguous in multiple dispatch systems: The Scala program in Figure 3.6 signals a static error during type checking since both overloadings apply for the first invocation, with neither being more specific than the other. The programmer can disambiguate the invocation by adding a static type assumption to an argument to prevent one of the overloadings from applying, for example, `m(b, b: A)`, which dispatches to the second overloading. But if this new invocation were interpreted with multiple dispatch semantics, at run

time the argument values would both have type `B`: now the same ambiguity occurs at run time instead of compile time! Since there is no more specific overloading, it would not be unreasonable for this hypothetical system to preserve the static interpretation when dynamic ambiguity is encountered; in this case, the invocation would still dispatch to the second overloading. (We shall return to this point shortly.) The programmer could more accurately disambiguate the invocation by instead providing a new overloading that is more specific than the other two:

```
def m(x: B, y: B): Int = 3
```

Note that instead of defining an entirely new implementation, which is simply the expression `3` here, the programmer could recursively invoke the method as above, thus choosing the second overloading.

In the early multimethod systems like CLOS [7, 60] (Section 3.3.4) ambiguous multimethods are mere inconveniences to be automatically resolved *ad hoc*, or they produce run-time errors. Lécluse and Richard [41] describe a means of statically checking for ambiguity in an OO language with single dispatch.[15] The Kea language of Mugridge *et al.* [47] provides a specification of dynamic ambiguity but not an algorithm. Agrawal *et al.* survey static type checking of multimethods but do not consider ambiguity in [1]; instead their algorithms focus on ensuring that *some* interpretation for an invocation exists at run time. Castagna *et al.* devised a multimethod calculus that prevents ambiguity [17] (Section 3.3.5). Chambers and Leavens

[15] Although it is single dispatch, their system is based on "structured values" rather than classes, so the facilities for eliminating ambiguity in the object receiver parameter are not present.

more carefully consider ambiguity as part of a type system with multiple dispatch in [18], followed later with Millstein in [45, 46, 19] (Section 3.3.6).

3.3.3.2 Modularity

A program component should be type checked given only its own method definitions and the interfaces of those other components on which it depends. Furthermore, the compilation tools for the system should have the capability to compile that component separately from other components.

The main complication of modularity lies in the expansion of available method definitions and types at run time: When a component *C* is type checked, it knows about some set of methods and classes. If another component *D* uses *C* and extends those methods and classes, then execution of *D* leads to execution of *C*, and *C*'s knowledge of the methods and types now lags behind that of *D*. A multimethod invocation in *C* will then be executed without knowing of additional definitions thereof (provided by *D*). For example, consider again the program in Figure 3.6, with the aforementioned semantics of executing the statically chosen interpretation in the presence of dynamic ambiguity. If this program were executed due to a call from another module, a module which provides the disambiguating, more specific definition on two `B` parameters, then that executed interpretation would not actually be the most specific available.

The modular extension of classes affects not only the run time dispatch, but also the ambiguity prevention in the presence of multiple inheritance. Whereas with single dispatch two method definitions (each from a different class) clash in the receiver parameter when inherited by a common subclass, with multiple dispatch the clash could occur in any number of pa-

rameters. Consider two overloaded definitions $\texttt{m}_1$ and $\texttt{m}_2$, between which only the i^{th} parameter type differs: in $\texttt{m}_1$ that type is `Foo` and in $\texttt{m}_2$ it is `Bar`. In the class hierarchy visible in the current component, `Foo` and `Bar` are entirely unrelated — there is no object which is an instance of both classes and thus no ambiguity between the methods. However, another program that uses this (statically checked) component might declare a new class `Baz` that extends both `Foo` and `Bar`. An invocation of `m` on an expression of type `Foo` now becomes troublesome: an instance of class `Baz` that is passed in for the i^{th} argument in that invocation causes dynamic ambiguity. Therefore, to preserve the modularity of the "open world" of classes and methods, any mechanism that prevents ambiguity must reason about *what could be defined* instead of merely *what is currently defined*.

Modularity has been prominently investigated in the context of multiple dispatch systems by Chambers, Leavens, and Millstein [18, 45, 46, 19] (Section 3.3.6). Earlier, the Kea language [47] provided support for separate compilation of classes with multimethods. Other systems that have explored the challenges of modular multiple dispatch include Half & Half [5], JPred [28], Fortress [2, 3] (Section 3.3.8), and CZ [43].

3.3.3.3 Static checking

A multiple dispatch system should certainly admit a sound and decidable algorithm[16] for statically verifying the well-formedness of multimethod definitions and invocations in programs. These checks should be performed modularly and they should catch all ambiguous invocations —

[16]The verification algorithm can be incomplete iff it can detect when it cannot verify a program.

in effect the multimethods themselves are type checked.

The Common Lisp-based multiple dispatch systems, being dynamically typed, do not admit any static checking or compilation. All other multiple dispatch systems mentioned thus far do specify static checking. In the following sections, though, we shall see that not all of them verify the same properties with the same strategies.

3.3.4 Common Lisp Object System

In the Common Lisp Object System (CLOS) [8], based heavily on CommonLoops [7] (which originated the term *multimethod*), methods are defined outside the scope of classes — a method is not owned by any particular class. Consequently methods are not distinguished syntactically from normal functions with the "dotted" form. By convention the first argument of a method is the object upon which the method acts; in other words, it is semantically the receiver but not syntactically. Each method definition with name *m* defines a multimethod overloading of a *generic function* named *m* and can specify the types of any number of its parameters. Those parameters with specified types will only apply in method invocations whose corresponding argument values have subtypes thereof; *i.e.* any number of parameters may have unspecified types, so any argument would apply to those.

CLOS multimethods are totally ordered by specificity, meaning that for any non-empty list of applicable overloadings for an invocation, there is always a most specific one. This is achieved by linearizing multiple inheritance and by employing **asymmetric multiple dispatch** semantics — multimethod parameter lists are ordered not by usual covariant subtyping on

entire tuple types (*i.e.* **symmetric multiple dispatch**), but lexicographically by the order of the parameters.[17] Consider the methods of Figure 3.6 (without their implicit receiver parameters), which have domain tuple types `(A, B)` and `(B, A)` respectively. The second tuple is not a subtype of the first tuple, but its first element, `B`, is indeed a subtype of the other's first element, `A`, which makes the second method more specific than the first.

3.3.5 λ&

The λ& calculus of Castagna, Ghelli, and Longo [17] is the first core calculus devised to model statically typed languages with subtyping and multiple dispatch. Their calculus extends the simply typed lambda calculus with multimethods, *i.e.*, functions with multiple "branches," and application of these multimethods to operands. More importantly, their calculus has the distinction of the first multiple dispatch system to *embed unambiguity into the type system*. In terms of overloading, the possibility of ambiguous interpretations at run time is ruled out statically, not unlike the unambiguous overloading found in type class systems.

An overloaded function f has type $T_f = \{U_i \to V_i\}_{i \in I}$, where $U_i \to V_i$ is the arrow type of the i^{th} overloading of f. Crucially, however, the type T_f is only well-formed if for all $i, j \in I$,

1. if $U_i \leq U_j$ then $V_i \leq V_j$, and
2. if U_i and U_j have a common (non-$\bot$) subtype then there exists a unique $h \in I$ such that $U_h = \inf\{U_i, U_j\}$.

[17] In CLOS one can actually change the order in which parameters are tested for specificity, per method.

Condition 1 states that if f_i is more specific than f_j, its return type must be a subtype of f_j's — this guarantees type safety of dynamically choosing a more specific overloading on an invocation. Condition 2 states that if there is a common subtype of f_i's and f_j's domains, then there must be a unique overloading f_h, the domain of which is equal to the greatest common subtype of those domains — this guarantees the existence of an overloading that disambiguates two others, for any possible invocation.

We return again to the example in Figure 3.6. If those Scala methods are translated to λ& as λ abstractions m_1 and m_2 with types $T_1 = (A, B) \to$ Int and $T_2 = (B, A) \to$ Int respectively, then the overloaded function that combines them, $m = m_1 \,\&\, m_2$, has type $T = \{T_1, T_2\}$. However, the type (B, B) is a subtype of both domains but no overloading with that domain exists in T — then T is not a well-formed type! Therefore we must add another overloading whose domain is (B, B) and whose return type is a subtype of Int; now $T = \{T_1, T_2, (B, B) \to \text{Int}\}$. (Note that this overloading has a signature identical to that of the disambiguating definition discussed for Figure 3.6 earlier.) In this way, λ& precludes ambiguity on *any possible invocation* as part of the type checking process on overloaded function definitions, without needing to check each invocation site.

As simply a core calculus, λ& lacks any facility for modularity or abstract declarations of multimethods, but Castagna *et al.* described its application to OO languages in [17]. The Fortress language [3] is such an application, with extensions for modularity and robustness. In particular, determining if two types have a common subtype is tricky with the extensible open world of types in OO languages; Fortress provides a solution to this problem, which will be discussed in Section 3.3.8.

3.3.6 System M

Millstein and Chambers [45] devised Dubious, a classless OO language with multimethods, and later Clifton, Millstein, Leavens, and Chambers [19] developed MultiJava, an extension of Java with multimethods, based on the modularity and dispatch semantics of Dubious. Both languages rely on modularity restrictions called System M [45, §4.2] to statically curb dynamic ambiguity. In particular, multiple inheritance from concrete classes in different modules is forbidden, and abstract (`interface`) types cannot be used as multimethod specializers, *i.e.*, as parameter types of concrete, specialized overloadings. Additionally, whereas only multimethod signature types must be checked for ambiguity in $\lambda\&$, in MultiJava all possible argument (tuple) types must be checked against all multimethod definitions, per module.

3.3.7 ML$_{\leq}$

Instead of extensible records, the ML$_{\leq}$ language of Bourdoncle and Merz [9] uses ML-style data types as the basis for classes: a data type declaration like

```
datatype list[α] = nil | cons of (α * list[α]);
```

corresponds in ML$_{\leq}$ to a type constructor `list[`α`]` and two data type constructors `nil[]` and `cons[`α`]` (both subtypes of `list[`α`]`), which would all be grouped into a semantically related, extensible class `List`. Multimethods are defined within classes similarly to CLOS, with the exception that each "generic function" must have a principal type of which all overloadings' domains are subtypes (as with the languages based on System M). In ML$_{\leq}$

multimethods generalize the pattern matching cases of function definitions in ML, using subtyping instead of unification on types to check applicability.

The key innovation in $ML_{\leq}$ lies in its (parametric) polymorphism: of all the multiple dispatch systems discussed thus far, none have supported polymorphism. Bourdoncle and Merz observe that the polymorphic domain types of functions correspond to existential types, and that the System $F_{\leq}$ style of subtyping generalizes the monomorphic subtyping needed for an overloading to satisfy its principal type. Along with ML's polymorphism, $ML_{\leq}$ maintains ML's rich type inference, except in the case of generic function signatures.

The $ML_{\leq}$ system does not support modularity, however, and like MultiJava, every possible argument type must be checked for ambiguity between all multimethods.

3.3.8 Fortress

The Fortress programming language [3] offers truly modular multiple dispatch [2]. Methods may be defined in classes (known as *traits* when abstract and *objects* when concrete) as *dotted methods* (dotted syntax for invocation) or as *functional methods* (normal top-level function call syntax for invocation); in the latter case, methods may be overloaded with similarly named top-level functions.

Fortress extends the approach to unambiguity of $\lambda\&$ with a notion of **exclusion**: two types exclude when they have no common subtype. Similarly to how the subtyping relation in OO languages is induced by nominal

extension of classes, the exclusion relation is induced by nominal exclusion of classes.

Considering the hierarchy of types as a *lattice* under the subtyping order, for each pair of overloadings f_1, f_2 of some function f, there must be a unique overloading of f whose domain is the *meet* of the domains of f_1 and f_2. (This interpretation is equivalent to the second condition for well-typedness of overloaded function types in $\lambda\&$ from Section 3.3.5.) However, in Fortress there is another possibility: The domains of f_1 and f_2 may exclude instead. In that case, no ambiguous argument value for which both overloadings apply could ever exist, so a disambiguating overloading is unnecessary.

In Section 3.3.3.2 we saw how a modular multiple dispatch system with multiple inheritance can produce ambiguity when two classes `Foo` and `Bar` are both extended by a class `Baz` in a later module. In Fortress if `Foo` and `Bar` exclude, then there can never exist a type like `Baz` that is a subtype of both; the type checker can thus make use of this fact when type checking the method definitions which previously caused ambiguity.

Moreover, the semantic partitioning of methods according to the number of declared parameters (as done in each of the other systems presented) is generalized with exclusion, as any two tuple types with different numbers of constituent types exclude. This eliminates the requirement for all overloadings of some function to have the same number of parameters and, with reasonable exclusion defined between library classes like numbers and strings, it permits convenient, overloaded definitions on disparate types like *println* and *bar* from Section 3.1.2.

Static checking in Fortress is performed modularly, with each com-

ponent being checked only with knowledge of the interfaces of those it imports. Unlike System M, Fortress allows multiple inheritance across module boundaries, and unlike $ML_{\leq}$, Fortress supports polymorphic, overloaded functions and methods in a very restricted manner, similar to their treatment in the FGJ calculus [35].[18]

3.3.9 Away from objects

Although it is presented here in the context of objects and classes, multiple dynamic dispatch semantics can be employed in other kinds of languages. The unique specialization capability afforded by this semantics requires a preorder on types, a framework for overloaded function definitions, and run-time type information; these concepts are realized naturally in OO languages, but they are not confined to those languages. Castagna discusses in [16] how specialization (with multiple dynamic dispatch) is a covariant relation on function types entirely separate from subtyping, a contravariant relation. Additionally, Allen *et al.* explain how even Haskell can adopt unambiguous multiple dynamic dispatch semantics similar to Fortress in [4, App. A].

[18] Recent work by Allen *et al.*, however, has generalized overloading in Fortress to cover polymorphism [4].

Chapter 4

Lessons learned from Fortress

4.1 Self-type idiom

With type classes the method declarations always have a reference to the instantiating type. For example, in the `Num` class defined in Figure 3.1, the type variable `a` is instantiated with `Int`, `Float`, and so forth, within each implementing instance. We've also seen the function `double` as an example of a function that is parameterized by instances of `Num`.

In an OO language, if one were to model `Num` as an abstract (object) class, one might naïvely declare it as

```
class Num {
  def add(other: Num): Num
  def mul(other: Num): Num
  def neg(): Num
}
```

where the methods would be invoked on numeric objects like `x.add(y)` and `intFive.neg()` — the object receiver is actually the left operand. This abstract declaration says that for any concrete class `T` that extends `Num`, `T` must implement these methods. But then `T` would need to be implemented where the right operand, the `other` parameter, still has the abstract type `Num`, *not* the same type `T`. Unlike the abstract declarations in the type class, which

were declared on the class parameter type `a`, the OO declarations do not (and should not) require that substitution of the instantiating type for recursive references to `Num`.

Phrased differently, the problem lies in specifying *binary methods* — methods whose parameter types should vary covariantly by the extension relation along with the receiver type — which has plagued OO languages for decades [12]. Much work has gone into ways of specifying such methods accurately, but here we only concern ourselves with the approach of *F-bounded polymorphism*, as this has been adopted not only by Fortress but by major OO languages like Java and Scala.

4.1.1 Using F-bounds

The insight of Canning *et al.* [14] involved modeling such classes using polymorphic types with recursive bounds as an extension of bounded polymorphism in Fun [15]; they called this approach *F-bounded polymorphism*. The recursive type variable would allow one to make use of the instantiating class type by passing it as a parameter to the abstract class. The above code would be rewritten as

```
class Num[T] {
  def add(other: T): T
  def mul(other: T): T
  def neg(): T
}
```

and all implementations thereof would be defined like

```
class Int extends Num[Int]
```

The `double` function would then be defined as

```
def double[T <: Num[T]](x: T): T = x.add(x)
```

However, consider a default implementation[1] of the `mul` method in terms of the `add` method: there would be invocations of the form `x.add(y)` where `x` and `y` both have type `T`. In order for this invocation to be well-typed (for `x` to exhibit the `add` behavior), `T` needs to be a subtype of `Num[T]`. As is, `T` is unbounded, so we must modify the definition of `Num` to read

```
class Num[T <: Num[T]] { ... }
```

as in the signature of `double`. This approach is taken in the standard library of Java, *e.g.*, the abstract superclass of all `enum` types,[2]

```
class Enum<E extends Enum<E>> { ... }
```

This convoluted manner of identifying the name of the implementing type — the type of the concrete class that inherits from the abstract class — with the type parameter of the abstract class is known as the *self-type idiom*.

This approach is still not ideal. In particular, nothing prevents programmers from writing a class like

```
class Foo extends Num[Int] { ... }
```

[1] *i.e.*, defined inside the abstract class `Num`.

[2] In the Java Platform Standard Edition 6: http://download-llnw.oracle.com/javase/6/docs/api/java/lang/Enum.html

wherein the type parameter `T` no longer stands in for the implementing class type. Therefore the *desired* class relationships are permitted, as witnessed above, but so too are *undesired* relationships like `Foo` and `Num[Int]`. This is not disastrous but also not ideal.

4.1.2 Ill-typed Fortress code

Early in the development of Fortress, before a static type checker had been implemented, a large portion of the library code was written under the assumption of well-typedness. Some abstract classes (or *traits*, as they are known in Fortress) had the form

```
trait Comparable⟦T extends Comparable⟦T⟧⟧
  lessThan(other: T): Boolean
  greaterThan(other: T): Boolean = other.lessThan(self)
end
```

which inconspicuously introduced another typing problem. Generally in OO languages the type of the receiver reference within a class definition, `self`, is simply the class type applied to its type variables —Comparable$[\![T]\!]$ in this case. However, the implementation of *lessThan* uses `self` where type T is expected, but Comparable$[\![T]\!]$ is not a subtype of T. Intuitively we've used the self-type idiom to identify T with the type of `self`, but now we must *force* them to be equivalent in the type system.[3]

[3]The reverse direction of the subtyping relation is already true by the declared bound of T.

4.1.3 An *ad hoc* solution

Previous efforts to address this problem involved an additional type Self which would be instantiated with the implementing type during execution and which would eliminate the need for the type parameter T. However, in a chain of subtypes like MyInt <: Int <: TotallyOrdered[4] <: Comparable, the run time environment cannot determine that the type of `self` within the definition of the *leq* method, Self, should be String and not MyString or TotallyOrdered instead. The subtyping between MyString and String and between TotallyOrdered and Comparable are fundamentally different from that between String and Comparable, though; the latter relation represents an implementation, whereas the others represent normal class extensions of data and methods. An OO system might instead provide an alternative typing relation for implementation; for example, the JavaGI system of Wehr *et al.* [67] provides *generalized interfaces* which act much like type classes (and thus circumvent the problem entirely).[5]

In Fortress, however, a different strategy has been employed to solve the typing problem with the self type idiom [61]. The solution does not introduce new concepts to the language but instead utilizes an existing concept in a new way. Like the nominal exclusion relation explained in Section 3.3.8, Fortress trait types can specify what traits they are extended by in an optional `comprises` clause. For example, consider the definition of ConsLists which precludes any other classes from extending it:

[4]The trait TotallyOrdered simply extends Comparable with further abstract declarations to be implemented.

[5]Java provides the `implements` relation between classes and interfaces, but it does not make use of an auxiliary self type.

`trait` ConsList⟦T⟧ `comprises` { Cons⟦T⟧, Nil⟦T⟧ } ... `end`

`object` Cons⟦T⟧ `extends` ConsList⟦T⟧ ... `end`

`object` Nil⟦T⟧ `extends` ConsList⟦T⟧ ... `end`

The key insight of the author involved two changes to the comprises clause, inspired by the (more restricted) *explicit self types* of classes in Scala [48, §5.1]. First, the receiver `self` within a trait Foo⟦$T_1, \ldots, T_m$⟧ that comprises types $U_1, \ldots, U_n$ may be given the more precise type Foo⟦$T_1, \ldots, T_m$⟧ $\cap$ $(U_1 \cup \ldots \cup U_n)$, where $\cap$ and $\cup$ denote intersection and union types respectively. This type models the statically enforced semantics that any object at run time that will instantiate the `self` parameter must also belong to some type U_i that extends Foo⟦$T_1, \ldots, T_m$⟧. Second, a comprises clause may contain a naked type variable T, as in

`trait` Comparable⟦T⟧ `comprises` {T} ... `end`

With this definition, the only type allowed to extend Comparable⟦Foo⟧ is Foo itself, which addresses the problem of prohibiting undesired type relationships. Moreover, since any type appearing in a comprises clause by definition extends the comprising type in some declaration, we know that T is a subtype of Comparable⟦T⟧. Now, with the more precise self type, the type of `self` is Comparable⟦T⟧ $\cap$ T, which is a trivial subtype of T, and since T is a subtype of Comparable⟦T⟧ (and since T is a subtype of itself) we have that $T <:$ Comparable⟦T⟧ $\cap$ $T <: T$. Therefore the type of `self` and T are now equivalent! The self-type idiom can then be employed in a well-typed, more syntactically compact way. More examples, and the precise semantics currently implemented in Fortress, are explained in [61].

The fix outlined above has transformed ill-typed code to well-typed code with few textual changes. Although the new semantics encode the intuition about relationships between classes, the methodology behind the fix (using the comprises clause) is entirely *ad hoc*. Even worse, it is an *ad hoc* workaround to an *ad hoc* workaround — the unintuitive use of F-bounded polymorphism simply to reference the type of an implementation. With type classes, and even with existential types (Section 2.5.2), the reference to an implementing type is a fundamental part.

4.2 Expressive, excessive overloading

The permissive overloading of Fortress, especially with recent developments for polymorphism [4], embodies a new interpretation of a familiar programming environment. All functions and methods are open to any amount of specialization or convenient overloading by satisfying a single set of rules,[6] and there need be no explicit, textual connective between those overloaded definitions. In systems with type classes, every overloading is explicitly bound to an abstract overloading that describes it, its *principal overloading*; *i.e.* each implementation of a type class method is defined within an explicit instantiation of that type class. In systems with multiple dispatch, every multimethod overloading must specialize a single principal overloading, so the common multimethod name explicitly connects them. In Fortress, however, the overloadings exhibit *implicit* connections to be discovered and checked as part of the type system. One might compare the overloaded functions of Fortress to the expressions of a system with

[6] In Fortress the rules that must be satisfied for an overloaded set that mixes top-level functions and functional methods are slightly more complicated.

inferred, principal types, or liken Fortress overloading to a system of structural rather than nominal subtyping.

In exchange for implicit, expressive overloading, the Fortress type system imposes a burden on the programmer: the programmer[7] must reason about the well-formedness of overloaded functions through the satisfaction of "overloading rules" [3, §15.6]. These rules, which boil down to the well-formedness conditions from λ&, constitute a proof strategy to verify the essential properties of type safety and unambiguity. This proof strategy might be considered *ad hoc* since the programmer employs it independently for each overloaded set. Other systems, however, only express overloading through particular language constructs, which essentially prove type safety and unambiguity "by construction." Those constructs, like Haskell's type classes or even Java's (object) classes, serve the dual purpose of structuring data and expressing overloading in such a constrained way that guarantees validity. Whether the implicit expressivity of λ&/Fortress overloading is more beneficial to programmers than the explicit uniformity of type class (and object class) overloading remains to be seen.

4.2.1 A concrete concern

We have already seen how the nominal exclusion relation assists in verifying the validity of an overloaded set. Many convenient overloadings that users should be able to write, like $\mathit{println}(\mathrm{String})$ and $\mathit{println}(\mathbb{Z})$, presume the exclusion of many trait types defined in libraries whenever such an exclusion seems intuitive (here, no value should be simultaneously a

[7] Since only the overloaded function *definitions* are checked, this burden perhaps falls more on library writers than on clients.

string and an integer). However, since exclusion between trait types can *only* be specified on the declaration of one of the excluding traits, the libraries *must* define all such exclusions; the exclusion relation is not modularly extensible.

In Section 3.3.8 we considered the case where two existing traits Foo and Bar could exclude each other to allow an overloaded function to be defined on both because, after all, it might not make sense for any trait to extend both. But in the case where Foo and Bar are declared in completely unrelated components, the importing component in which the overloaded function is to be defined cannot make those two traits exclude. The existence of overloadings on these two types would necessitate a third overloading on their meet — but since these traits are entirely unrelated (conceptually) there is no such type on which to define the overloading. In other words, the programmer cannot modularly express the semantics of exclusion within the Fortress type system. One solution to this problem would be to allow first-class intersection types in function parameters, as such types are precisely the "greatest common subtypes" of their constituents. Even in this case, the programmer would need to provide a moot definition of the meet overloading, perhaps by raising an impossible exception:[8]

$$f(\text{Foo} \cap \text{Bar}) = \texttt{throw}\ \text{ImpossibleCase}$$

[8]The observant reader might notice the resemblance of this problem (and this *ad hoc* solution) to that of guaranteeing exhaustiveness of pattern matching [44].

4.3 Nominal relations

A nominally declared extension relation between class types is common to virtually every class-based OO language. Extension organizes classes into a hierarchy of inheritance while also providing essential semantics for the type system — (nominal) subtyping. Some languages distinguish between interface inheritance and implementation inheritance, and others, like CZ [43], provide entirely different relations for inheritance and subtyping.[9]

In the same way as extension, the exclusion and comprises relations of Fortress serve multiple roles in the language: they act as documentation about the ontology of traits, but they also provide additional means of describing relationships in the type system, which affords more precision and expressivity for overloading and abstract specification (as demonstrated in the previous two sections). In recent work on polymorphism for Fortress overloading [4], the exclusion relation is augmented beyond the declared, nominal exclusion between trait types by utilizing a subtle fact of OO class tables.

Encoding ever-increasing knowledge about types by augmenting nominal type relations may yield greater expressivity, but this approach lacks a coherent, generalized core, or even a definite point of completion.[10] This approach to language design is analogous to augmenting a logical system

[9] Cook *et al.* [21] showed that in the context of OO languages, inheritance (extension) is not the same relation as subtyping.

[10] The same phenomena is occuring in the constant development of Haskell. Various *ad hoc* yet intuitive extensions to its type class system regularly appear in the research literature.

with cursory, *ad hoc* exceptions instead of capturing the essence in a core set of definitions.

Chapter 5

Conclusion

The underlying idea behind overloading, using a single name with different meanings depending on context, stems from informal languages — from the malleable, hierarchical polysemy in natural language, to the more precise use of symbols for distinct, concrete interpretations in informal mathematical language. This precedent justifies and explains the use of overloading in programming languages, which is known as ad-hoc polymorphism.

Since its early, informal definition, overloading has been understood as an unnecessary, *ad hoc* convenience that models arithmetic operations. However, through the lens of a generalized system of overloading, it can be interpreted as providing three key features: abstraction, convenience, and specialization. Many common programming language constructs — type classes, object classes, and multimethods — manifest overloading as a fundamental aspect, albeit not explicitly. Conventionally thought to provide those features themselves, these constructs instead serve as the means of expressing and organizing overloading in a way that satisfies well-formedness and safety properties; *i.e.*, overloading is the byproduct of other, more fundamental abstractions.

In the Fortress programming language, however, overloading — or more accurately, multiple dynamic dispatch — has been designed as such

a fundamental language facility. A notion of object classes still serves to organize overloading, but as a convenient common case of a more general framework of overloading. This framework offers even greater abstraction, convenience, and specialization than the overloading of other systems, as it exports greater control to programmers to define and specialize functions as they wish. With that control comes additional responsibility to only write overloaded functions that maintain crucial type safety properties. While the type checker can verify those properties using new nominal relations on types, programmers are burdened more than with other systems in which overloading well-formedness is ensured by construction.

The different methodologies in expressing overloading suggest a new spectrum of language design. On one end, well-formed overloading is induced from organizational type structures and is more restrictive; on the other end, well-formed overloading is expressed and reasoned about *ad hoc* by the programmer and is more permissive. It is left as an exercise to the reader to determine which is more fitting.

Bibliography

[1] Rakesh Agrawal, Linda G. Demichiel, and Bruce G. Lindsay. Static type checking of multi-methods. *SIGPLAN Not.*, 26(11):113–128, 1991.

[2] Eric Allen, J. J. Hallett, Victor Luchangco, Sukyoung Ryu, and Guy L. Steele, Jr. Modular multiple dispatch with multiple inheritance. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1117–1121, New York, NY, USA, 2007. ACM.

[3] Eric Allen, David Chase, Joe Hallet, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification, version 1.0. Technical report, Sun Labs, 2008. URL http://labs.oracle.com/projects/plrg/Publications/fortress.1.0.pdf.

[4] Eric Allen, Justin Hilburn, Scott Kilpatrick, Sukyoung Ryu, David Chase, Victor Luchangco, and Guy L. Steele Jr. Type-checking modular multiple dispatch with parametric polymorphism and multiple inheritance. Submitted for publication, July 2010. URL http://userweb.cs.utexas.edu/~scottk/papers/multipoly.pdf.

[5] Gerald Baumgartner, Martin Jansche, and Konstantin Läufer. Half & Half: Multiple dispatch and retroactive abstraction for java. Technical report, 2002.

[6] G.M. Birtwhistle, O.J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Chartwell-Bratt Ltd, 1979. ISBN 086238009X.

[7] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: merging Lisp and object-oriented programming. *SIGPLAN Not.*, 21(11):17–29, 1986.

[8] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System specification. *SIGPLAN Not.*, 23(SI):1–142, 1988.

[9] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 302–315, New York, NY, USA, 1997. ACM.

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

[11] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the java programming language. *SIGPLAN Not.*, 33(10):183–200, 1998.

[12] Kim Bruce, Luca Cardelli, Giuseppe Castagna, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theor. Pract. Object Syst.*, 1(3): 221–242, 1995.

[13] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280, New York, NY, USA, 1989. ACM.

[14] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280, New York, NY, USA, 1989. ACM.

[15] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985. ISSN 0360-0300.

[16] Giuseppe Castagna. Covariance and contravariance: conflict without a cause. *ACM Trans. Program. Lang. Syst.*, 17(3):431–447, 1995.

[17] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *SIGPLAN Lisp Pointers*, V(1): 182–192, 1992. ISSN 1045-3563.

[18] Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. *ACM Trans. Program. Lang. Syst.*, 17(6):805–843, 1995.

[19] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. Multijava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28(3):517–575, 2006.

[20] William R. Cook. On understanding data abstraction, revisited. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 557–572, New York, NY, USA, 2009. ACM.

[21] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, New York, NY, USA, 1990. ACM.

[22] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption. In *CAAP '90: Proceedings of the fifteenth colloquium on CAAP'90*, pages 132–146, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[23] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM.

[24] Derek Dreyer and Andreas Rossberg. Mixin' up the ml module system. *SIGPLAN Not.*, 43(9):307–320, 2008.

[25] Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. In *POPL '07: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 63–70, New York, NY, USA, 2007. ACM.

[26] Dominic Duggan and John Ophel. Type-checking multi-parameter type classes. *J. Funct. Program.*, 12(2):133–158, 2002.

[27] Karl-Filip Faxén. A static semantics for Haskell. *Journal of Functional Programming*, 12(4-5):295–357, 2002.

[28] Christopher Frost and Todd Millstein. Modularly typesafe interface dispatch in JPred. In *In FOOL/WOOD '06: International Workshop on Foundations and Developments of Object-Oriented Languages*. ACM Press, 2006.

[29] Jean-Yves Girard. Une extension de l'interprétation de gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings 2nd Scandinavian Logic Symposium*, pages 63–92, Amsterdam, 1971. North-Holland.

[30] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6.

[31] James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java^{TM} Language Specification*. Addison-Wesley Professional, third edition, 2005. ISBN 0321246780.

[32] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18:241–256, 1996.

[33] Robert Harper. *Programming in Standard ML*. 2005. URL `http://www.cs.cmu.edu/~rwh/smlbook/online.pdf`. Draft of August 20, 2009.

[34] Robert Harper. *Practical Foundations for Programming Languages*. 2008. URL `http://www.cs.cmu.edu/~rwh/plbook/book.pdf`. Draft of June 13, 2010 at 23:45.

[35] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[36] Mark P. Jones. Type classes with functional dependencies. In *ESOP/ETA (LNCS)*, pages 230–244. Springer-Verlag, 2000.

[37] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Journal of functional programming*, pages 52–61. ACM Press, 1995.

[38] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009. ISBN 0131873210.

[39] Stefan Kaes. Parametric overloading in polymorphic programming languages. In *ESOP '88: Proceedings of the 2nd European Symposium on Programming*, pages 131–144, London, UK, 1988. Springer-Verlag.

[40] Wolfram Kahl and Jan Scheffczyk. Named instances for Haskell type classes. In Ralf Hinze, editor, *Proc. Haskell Workshop 2001*, 2001.

[41] Christophe Lécluse and Philippe Richard. Manipulation of structured values in object-oriented databases. In Richard Hull, Ronald Morrison, and David W. Stemple, editors, *Proceedings of the Second International Workshop on Database Programming Languages, 4-8 June, 1989, Salishan Lodge, Gleneden Beach, Oregon*, pages 113–121. Morgan Kaufmann, 1989. ISBN 1-55860-072-8.

[42] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0201432943.

[43] Donna Malayeri and Jonathan Aldrich. Cz: multiple inheritance without diamonds. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 21–40, New York, NY, USA, 2009. ACM.

[44] Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17(03):387–421, 2007.

[45] Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 279–303, London, UK, 1999. Springer-Verlag.

[46] Todd David Millstein. *Reconciling software extensibility with modular program reasoning*. PhD thesis, 2003. Chair-Chambers, Craig.

[47] Warwick B. Mugridge, John Hamer, and John G. Hosking. Multi-methods in a statically-typed programming language. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 307–324, London, UK, 1991. Springer-Verlag. ISBN 3-540-54262-0.

[48] Martin Odersky. *The Scala Language Specification, Version 2.7*. EPFL Lausanne, Switzerland, 2009. URL http://www.scala-lang.org/docu/files/ScalaReference.pdf.

[49] Martin Odersky, Philip Wadler, and Martin Wehr. A second look at overloading. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 135–146, New York, NY, USA, 1995. ACM.

[50] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. URL http://haskell.org/definition/haskell98-report.pdf.

[51] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-16209-1.

[52] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[53] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423, London, UK, 1974. Springer-Verlag. ISBN 3-540-06859-7.

[54] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[55] Mark Shields and Simon Peyton Jones. Object-oriented style overloading for Haskell. In *First Workshop on Multi-language Infrastructure and Interoperability (BABEL'01), Firenze, Italy*, September 2001.

[56] Jeremy G. Siek. *A language for generic programming*. PhD thesis, Indiana University, Indianapolis, IN, USA, 2005.

[57] Jeremy G. Siek and Andrew Lumsdaine. A language for generic programming in the large. *Science of Computer Programming*, 2008.

[58] Ghan Bir Singh. Single versus multiple inheritance in object oriented programming. *SIGPLAN OOPS Mess.*, 5(1):34–43, 1994.

[59] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 278–293, Berlin, Heidelberg, 2008. Springer-Verlag.

[60] Guy L. Steele Jr. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990. ISBN 1-55558-041-6.

[61] Guy L. Steele Jr. New self type idiom, November 2009. URL http://projectfortress.sun.com/Projects/Community/blog/NewSelfTypeIdiom. Project Fortress blog.

[62] Christopher Strachey. Fundamental concepts in programming languages. Lecture Notes, International Summer School in Computer Programming, Copenhagen, 1967. Reprinted in *Higher-Order and Symbolic Computation*, 13(1/2), pp. 1–49, 2000.

[63] Bjarne Stroustrup. *The C++ Programming Language: Special Edition*. Addison-Wesley Professional, 3 edition, February 2000. ISBN 0201700735.

[64] Peter J. Stuckey and Martin Sulzmann. A theory of overloading. *ACM Trans. Program. Lang. Syst.*, 27(6):1216–1269, 2005.

[65] Philip Wadler. Theorems for free! In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 347–359, New York, NY, USA, 1989. ACM.

[66] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76, New York, NY, USA, 1989. ACM.

[67] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. JavaGI: Generalized interfaces for Java. In *ECOOP 2007, Proceedings. LNCS, Springer-Verlag (2007) 25*, pages 347–372. Springer-Verlag, 2007.

Vita

Scott Lasater Kilpatrick, a native Texan, was graduated from James Martin High School in Arlington, Texas with *magna cum laude* distinction. In 2008 he earned a B.S. in Computer Sciences and a B.S. in Mathematics with high honors from the University of Texas at Austin, one of eighteen Dean's Honored Graduates for the College of Natural Sciences. Immediately after receiving his baccalaureate degrees, he entered the Graduate School at the University of Texas at Austin in the Department of Computer Science. After receiving his master's degree from the University of Texas at Austin, having completed twenty years of public education in Texas, he will enroll in the doctoral program at the Max Planck Institute for Software Systems in Saarbrücken, Germany under the supervision of Derek Dreyer. He will miss eating tacos in the Austin swelter.

Email Address: `scottk@cs.utexas.edu`
Permanent Address: 2015 Misty Creek Dr.
Arlington, Texas 76017

This thesis was typed by the author.