Type Classes and Instance Chains:

A Relational Approach

by

John Garrett Morris

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Mark P. Jones, Chair
Sergio Antoy
James G. Hook
Andrew P. Tolmach
M. Paul Latiolais

Portland State University
2013

ABSTRACT

Type classes, first proposed during the design of the Haskell programming language, extend standard type systems to support overloaded functions. Since their introduction, type classes have been used to address a range of problems, from typing ordering and arithmetic operators to describing heterogeneous lists and limited subtyping. However, while type class programming is useful for a variety of practical problems, its wider use is limited by the inexpressiveness and hidden complexity of current mechanisms. We propose two improvements to existing class systems. First, we introduce several novel language features, instance chains and explicit failure, that increase the expressiveness of type classes while providing more direct expression of current idioms. To validate these features, we have built an implementation of these features, demonstrating their use in a practical setting and their integration with type reconstruction for a Hindley-Milner type system. Second, we define a set-based semantics for type classes that provides a sound basis for reasoning about type class systems, their implementations, and the meanings of programs that use them.

ACKNOWLEDGMENTS

This dissertation, and the research it describes, would not have been possible without the help and support of numerous others over my time as a graduate student. First and foremost, I thank Mark Jones, my advisor, for his advice, encouragement, and for many hours of stimulating and enlightening discussions. Next, I should thank the other members of the HASP research group, and in particular James Hook and Andrew Tolmach, for their feedback and suggestions over the course of this research and development of this dissertation. Finally, I must thank my family for their love and unwavering support, particularly over the past several years.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Type systems play a central role in assuring program correctness. By providing sound, decidable approximations of semantic properties, types allow compilers to detect entire classes of program errors automatically, and they establish properties of programs and program components, providing a basis for further understanding or verification efforts. However, these benefits come at a cost: limitations in a typing discipline in turn limit the programs that can be well-typed under that discipline. Thus, improving the expressiveness of type systems—that is, increasing the fidelity of their approximation of underlying semantic properties—is a central concern for the development of typed programming languages, and for program validation as a whole.

One early challenge in the development of type systems was to support *polymorphism*, in which the same symbol or expression can have multiple meanings, corresponding to multiple, distinct types. Following Strachey [62], we identify two classes of polymorphism, *parametric* and *ad hoc*.

- Parametric polymorphism arises when an expression can take on any of a regularly formed family of types. A typical example of parametric polymorphism is the function that reverses a list. This function can be applied only to list values, and always returns list values. However, assuming a uniform representation of list types, it does not depend upon the type of the elements of the list—it could be applied equally well to lists of integers, of Boolean values, or of more complex types. Thus, we can imagine that the reverse function should be able to take on any type in the set

$$\{[\tau] \to [\tau] \mid \tau \in \mathit{Type}\},$$

where *Type* is the set of all types, and we write $[\tau]$ for the type of lists with elements of type τ, and $\tau \to \tau'$ for the type of functions from values of type τ to values of type τ'. This is called "parametric" because such sets can be captured by writing parameters in types. Thus, instead of the set comprehension above, it is common to write the type of the reverse function as `[t]` $\to$ `[t]`, where the type parameter `t` is implicitly universally quantified.

- Ad-hoc polymorphism, also called overloading, arises when an expression's types cannot be described just with types and type variables. For example, we might expect the `elem` function, which determines whether a candidate value is in a particular list, to be applicable to list of integers or character strings, but not to lists of functions, as we do not expect to have a computable equality test for functions. Similarly, we might expect arithmetic operations, like addition or multiplication, to be applicable to integers and fractional values, but not to character strings. Finally, the implementation of an ad-hoc polymorphic value is likely to be different at each of its types. For example, while we might expect that the reverse function will have uniform implementation for different list element types, we would not expect a common implementation of addition for integers and floating-point values.

There have been numerous approaches to strongly-typed parametric polymorphism, including the Hindley-Milner system [5, 21, 41], the Girard-Reynolds polymorphic lambda calculus [15, 52], and, more recently, mechanisms for generics in Java and C#. We will focus on the Hindley-Milner system and its extensions, as it has several advantages for our purposes. First, it has seen relatively wide adoption, underlying the ML and Haskell programming languages and their various dialects. Second, its semantics has been well-studied, providing a basis to study the semantics of overloading. Third, expressions in a Hindley-Milner setting have principal, or most general, types. Fourth, the principal type of an expression in a Hindley-Milner setting can be computed from the expression itself, without requiring programmer-supplied type annotation; the process of doing so is called *type inference*. It is possible to define type inference algorithms in settings that lack principal types; however, because there is no single, most general type for an expression in such a setting, the inferred type of an expression may not correspond to its intended use. The programmer is therefore required either to understand the details of the type inference algorithm, or to defensively add type annotation even when they may not be necessary. By contrast, the ML or Haskell programmer can be confident that the inferred type of an expression is general enough for any well-typed use of the expression. Thus, the existence of principal types is not just an interesting theoretical property, but an important contribution to the usability of languages and tools built on the Hindley-Milner type system.

There have been fewer widely-adopted mechanisms for strongly-typed, ad-hoc polymorphism in functional programming languages. Standard ML, for example, provides a syntactic distinction between variables that range over any type and variables that range over any type supporting equality; thus, the `elem` function could be given a type using the latter kind of variable. However, this mechanism is not generic: it is not extensible to support user-supplied notions of equality, or other sources of ad-hoc polymorphism. Ocaml provides an equality operation that ranges over all types, generating run-time exceptions if it is applied to types such as functions. While general purpose, this lacks the degree of static assurance provided by type-based approaches. One successful approach to strongly-typed ad-hoc polymorphism is provided by type classes, which were originally proposed by Wadler and Blott [70], as an extension of the Hindley-Milner type system. Type classes play three roles in programs:

- First, type classes provide a generic mechanism to constrain the instantiation of type variables, and thus to provide principal types for expressions with ad-hoc polymorphism. For example, Wadler and Blott describe a type class, named `Eq`, that contains those types that support equality. This class can be used to constrain the instantiation of variables in the types of polymorphic values. For example, we would infer the type `Eq t` $\Rightarrow$ `t` $\rightarrow$ `[t]` $\rightarrow$ `Bool` for the `elem` function, where the constraint `Eq t` indicates that variable `t` can only be instantiated with types from the `Eq` class. As we did for parametric polymorphism above, we can interpret this type as a set comprehension, but one in which the domains of variables is limited to the set $\mathit{Eq} \subseteq \mathit{Type}$:

$$\{\tau \rightarrow [\tau] \rightarrow \mathit{Bool} \mid \tau \in \mathit{Eq}\}.$$

- Second, type classes provide a way to describe the type-specific implementations of ad-hoc polymorphic expressions. For example, we expect the equality function for comparing integers to have a different implementation from the equality operator for character strings. Wadler and Blott observed that these implementation correspond to proofs (that is, the implementations witness the proofs) that Booleans or integers are in the class `Eq`. This observation has two applications. First, it can help to derive the implementation of overloaded values; for example, we can conclude from the `Eq t` predicate in its type that, for a given type τ the implementation of `elem` at type $\tau \rightarrow [\tau] \rightarrow \mathit{Bool}$ may depend on the proof that $\tau \in \mathit{Eq}$, that is, on the τ-specific implementation of the equality function. Second, it allows proofs that families of types are in classes. For example, given a witness of $\tau \in \mathit{Eq}$, we can generically construct a witness that $[\tau] \in \mathit{Eq}$. These mechanisms admit a good deal of automation. While the programmer must provide the basic proof rules, such as that $\mathit{Int} \in \mathit{Eq}$ and that $\tau \in \mathit{Eq} \implies [\tau] \in \mathit{Eq}$, the compiler can then use them to automatically construct larger proofs, such as to demonstrate that the types [Int] and [[Int]] are also in the *Eq* class.

- Third, type classes define properties on types. We have already relied on this interpretation when we wrote set comprehensions such as $\{\tau \to [\tau] \to \mathit{Bool} \mid \tau \in \mathit{Eq}\}$, which interpreted classes as subsets of *Type*, and thus, as properties on *Type*. When they introduced type classes, Wadler and Blott proposed that some class predicates might apply to multiple types; for example, the predicate `Coerce a b` would indicate that values of type `a` could be coerced into values of type `b`. We can interpret these multi-parameter classes as relations on types; this allows us to begin using classes to capture static information beyond simply the witness of class membership. For example, Jones [31, 32] built on this notion of classes as relations on types to describe mechanisms by which the satisfiability of predicates could be used to improve type inference, both by reducing ambiguity in inferred types and by detecting erroneous programs closer to the original source of the error.

Since their introduction, there has been significant interest, both scholarly and practical, in type classes. Unfortunately, this work has frequently uncovered complexities in the class system, and much of it has failed to gain wide acceptance. Wadler and Blott's formal treatment of classes and their implementations [70] provided lexically scoped instances; however, this approach has not been reconciled with principal typing. Haskell compilers have long supported overlapping instances [50], permitting the definition of more generic instances. However, this leads to both complications in reasoning about the type system, and undermines equational approaches to reasoning about the meanings of Haskell expressions. Functional dependencies [32] saw widespread adoption as a tool to combine overloading and type-level reasoning. However, perceived implementation difficulties led to the proposal of new mechanisms, such as indexed type families [56], which separate type classes from other approaches to type-level reasoning. Finally, Haskell implementations differ in their interpretations of these extensions; for example, the Hugs compiler [24] provides a more flexible system of functional dependencies than does GHC [14], while GHC's implementation of overlapping instances is more permissive than that of Hugs. Thus, we believe that the design and implementation of type class systems remains an open research problem, of import both to the development of Haskell, and to the development of strongly-typed programming in general.

1.1 INSTANCE CHAINS AND THE HABIT CLASS SYSTEM

We have recently been involved in the design of the programming language Habit, a dialect of Haskell intended for high-assurance low-level programming, and particularly in the specification of its type system and implementation of its type inference mechanisms. In doing so, we have developed a new collection of type class features (alternative clauses, exclusion, and backtracking, which we collectively term *instance chains*), intended to support, and expand upon, the expressive capabilities of Haskell type classes a mechanism both for typing overloaded values and for type-level computation, while providing a sound basis for reasoning about Habit programs, and avoiding the difficulties encountered with Haskell type classes and their extensions. This dissertation describes the results of that effort, in terms of the design of the Habit language, the semantics of classes and of overloaded expressions, and the implementation of a Habit compiler.

Language design. The design of the Habit class system is motivated by existing uses of the Haskell class system and its extensions, and by the difficulties and inexpressiveness those uses have encountered. Based on a survey of the literature and of Hackage, an online repository of Haskell libraries and applications, we have identified three significant patterns, beyond simple overloading, in the use of the Haskell class system and its extensions. Each of these patterns has a clearer, or less problematic, expression in the Habit class system than it does in Haskell. We provide a collection of examples to validate these claims and, as a further demonstration of the features of the Habit class system, show a new solution to the expression problem. The expression problem is a classic benchmark in programming language expressiveness [2, 37, 53, 69]: it requires the extension of both the constructors of, and the operations over, an abstract data type. Our solution builds on existing Haskell approaches [38, 64] to the expression problem; we improve on those approaches in two respects. First, we support greater flexibility in the injection of values into extensible data types. Second, we provide a generic mechanism to describe operations over extensible data types, whereas existing solutions require the definition of a new type class for each such operation.

Semantics of classes. Our semantics of classes provides a sound basis for reasoning about the meaning of classes and class predicates. By providing this semantics, we hope: first, to avoid confusion about the meaning of class system features, as observed in Haskell; second, to provide a foundation for future extensions to the Habit class system; and, third, to provide a standard for implementations of the Habit class system. We begin with an intuitive notion of the meanings of classes, interpreting each class as a mappings from types (or tuples of types) to the corresponding implementations of the class methods We then give a semantics for class predicates by building Kripke frame models [36] from our interpretations of classes. Finally, we define computable notions of acceptability (which describes whether the compiler accepts a given collection of class and instance declarations) and entailment (which describes the proofs the compiler can compute), and we show that each of these notions is sound with respect to our semantics.

Semantics of overloading. We build on our semantics of classes to give a semantics of overloaded expressions, completing our semantics of overloading in Habit. We adopt an approach originally developed by Ohori [48] in his semantics of Core ML; unlike other semantics for ML-like languages, Ohori's approach is suited to both ad-hoc and parametric polymorphism. We show that our notion of the meanings of classes naturally gives rise to the semantics of their methods, and that the soundness of entailment then gives the soundness of the (simplified) Habit type system with respect to our semantics. Unlike previous semantics of type classes, our approach gives meaning to class methods directly, rather than by translation; we believe this permits more direct reasoning about overloaded values and their meanings.

Implementation. It is not enough to be able to reason about overloaded values; we would like to be able to compile them as well. Thus, we conclude by discussing our Habit predicate solver, the implementation of Habit class system in the HASP group's complete Habit compiler. We describe three aspects of the interface between the solver and the type inference component of the Habit compiler: first, the interpretation of entailment proofs as evidence, suitable for compiling overloading; second, the simplification of inferred predicate sets; and third, the computation of type equalities implied by the satisfiability of predicate sets. Finally, we give a broad overview of the data structures and abstractions that make up our solver's implementation and its interface to the remainder of the compiler.

1.2 OUTLINE OF DISSERTATION

The primary contributions of this dissertation are:

1. The design of the instance chain mechanisms (alternative clauses, exclusion, and backtracking search) in the Habit class system;

2. A collection of examples, drawn partially from a survey of a large, public repository of Haskell libraries and applications, motivating the design of instance chains, and demonstrating their expressiveness;

3. A novel semantics of type classes, based on Kripke frame semantics, relating class and instance declarations to intuitive models of classes and their meanings;
4. Validity and entailment relations, describing the interaction between classes and typing, and proofs of their soundness;
5. The first translation-free semantics for programs with overloading, building on Ohori's semantics of Core ML and our semantics of type classes; and,
6. A discussion of our practical implementation of instance chains, and its interactions with type inference, in the context of a prototype compiler for Habit.

The remainder of the dissertation proceeds as follows. Related work is summarized at the end of each chapter.

Chapter 2 gives a brief introduction to: the Hindley-Milner type system; Wadler and Blott's proposal of type classes; Jones's generalization of type systems with predicates; and, several extensions of type classes implemented by the various Haskell compilers.

Chapter 3 describes the language design aspects of our work, including: our survey of type classes in Haskell programs; an intuitive description of the new features of the Habit class system; and, our case study of the expression problem. This chapter addresses contributions (1) and (2).

Chapter 4 develops the semantics of classes, and the logic of class predicates, including our development of the validity and entailment relations, and our proofs of the soundness of each. This chapter addresses contributions (3) and (4).

Chapter 5 develops the semantics of overloaded expressions; we begin with an introduction to Ohori's approach to semantics and Harrison's [18] extension of that approach to support polymorphic recursion, the continue to describe our semantics of overloading and to prove its soundness. This chapter addresses contribution (5).

Chapter 6 describes the Habit predicate solver, its interface to the remainder of the Habit compiler, and gives a high-level overview of its implementation. This chapter addresses contribution (6).

Finally, chapter 7 sketches several directions for future development of this work.

2. A BRIEF INTRODUCTION TO POLYMORPHISM AND OVERLOADING

In this chapter, we provide a brief introduction to the treatment of polymorphism in typed functional languages, particularly Haskell. Following Strachey [62], we identify two classes of polymorphism:

- *Parametric polymorphism* arises when an expression has regular meanings over its range of types; the name arises from the use of type parameters in describing the types of such value; and,
- *Ad-hoc polymorphism*, or overloading, arises when the meaning of an expression is distinct at different types.

We begin by discussing the Hindley-Milner type system (§2.1), an approach to parametric polymorphism adopted in the programming languages ML, Haskell, and their dialects. We go on to discuss type classes (§2.2), a feature of Haskell that extends the Hindley-Milner system to accommodate ad-hoc polymorphism. Next, we discuss Jones's theory of qualified types (§2.3), which generalizes the treatment of type classes to apply to arbitrary predicates on types, and re-establishes many of the advantageous attributes of Hindley-Milner typing. We will rely on Jones's type system in the remainder of the dissertation, allowing us to focus on the class system itself. Finally, we give an overview of several commonly used extensions of Haskell type classes: multi-parameter type classes (§2.4.1), functional dependences (§2.4.2), and overlapping instances (§2.4.3). In particular, we identify a serious flaw in latter (§2.4.4), setting the stage for the features of the Habit class system that are developed in the remainder of this dissertation.

2.1 THE HINDLEY-MILNER TYPE SYSTEM

The Hindley-Milner type system [5, 21, 41] provides one approach to parametric polymorphism. It assigns *type schemes* to expressions, capturing generic usage through (implicitly quantified) type variables. For example, the `reverse` function, which reverses the elements of a list, could be typed as

```
reverse :: [t] → [t]
```

meaning that, for any type `t`, `reverse` transforms a list with elements from `t` into another list with elements from `t`. A polymorphic value can be used at any type that is a *generic instance*—a substitution of types for quantified type variables—of its type scheme; in the case of `reverse`, these include the types `[Int] → [Int]` and `[(Int, Bool)] → [(Int, Bool)]`, but not `[Int] → [Bool]`. Intuitively, if *Type* is the set of all types, then we expect `reverse` to take on any type in the set

$$\{[\tau] \to [\tau] \mid \tau \in \mathit{Type}\}.$$

This is clearly not the only type scheme that we could assign to `reverse`; we could also assign it types such as `[(a, b)] → [(a, b)]` or even `[Int] → [Int]`. In this case, it should be apparent that any type scheme we could assign to `reverse` is (the quantification of) one of the generic instances of the type scheme `[a] → [a]`, and thus that `[a] → [a]` is, in some sense, the most general type scheme of the `reverse` function. Such a type scheme is called a *principal* type, and is unique (up to renaming of bound variables). A significant feature of Hindley-Milner typing, in contrast to some other systems of parametric polymorphism, is that every expression has a principal type. Further, there is a process, called *type inference*, that automatically computes the principal type of any well-typed expression. This provides for strong typing—with the associated semantic guarantees—without requiring programmers to provide explicit type annotations in programs.

The Hindley-Milner type system, however, provides no support for ad-hoc polymorphism, limiting the expressive power of Hindley-Milner types. For example while the `reverse` function is fully generic—we can expect to apply it to lists of any type—the list membership function `elem` (which determines if a particular value is an element of a list) must be able to compare its arguments for equality, and thus could only be applied to lists whose elements support computable equality comparisons. Thus, while the `elem` function could have numerous Hindley-Milner types, such as `Int` $\to$ `[Int]` $\to$ `Bool` or `Char` $\to$ `[Char]` $\to$ `Bool`, it does not have a principal type.

A variety of approaches have been used to support ad-hoc polymorphism in Hindley-Milner systems: Standard ML [42], for example, provides distinguished type variables that can only range over types supporting equality, and overloads arithmetic functions to support both the built-in integer and floating point types. However, these mechanisms are not extensible, either to arithmetic operations on user-defined types, such as complex numbers or matrices, or to allow the introduction new overloaded operators, such as ordering comparators.

2.2 TYPE CLASSES

Type classes, proposed by Wadler and Blott [70] during the design of the programming language Haskell, provide an extensible mechanism for typing overloaded functions, allowing the definition of both new overloaded operators and new instances of existing overloaded functions. A type class captures an open (i.e., extensible) set of types, associating with each member of the class the implementation of a specific collection of functions, called the *class methods*. This set is populated by a collection of *instance declarations*, each describing a particular way of implementing the class methods for a given type or (parametric) range of types. For example, the `Eq` class, used in Haskell to capture the types that support equality. This class is declared:

```
class Eq t
    where (==) :: t → t → Bool
          (≠) :: t → t → Bool
```

specifying that for a type to be in `Eq`, it must provide an implementation of equality `(==)` and inequality `(≠)` functions. The Haskell standard libraries provide instances of `Eq` for base types, such as the type of 32-bit integers, like the following:

```
instance Eq Int
    where x == y  =  isZero (x - y)
          x ≠ y  =  not (isZero (x - y))
```

as well as instances for parameterized types, such as the following instance for pairs:

```
instance (Eq a, Eq b) ⇒ Eq (a, b)
    where (x, y) == (x', y')  =  x == x' && y == y'
          (x, y) ≠ (x', y')  =  x ≠ x' || y ≠ y'
```

We can characterize the `Eq` class by identifying those types it includes. Given the instances above, we would expect the `Eq` class to contain at least the following subset of *Type*, the set of all types:

$$\{\mathit{Int}, (\mathit{Int}, \mathit{Int}), ((\mathit{Int}, \mathit{Int}), \mathit{Int}), (\mathit{Int}, (\mathit{Int}, \mathit{Int})), \ldots\}.$$

We use the `Eq` class to provide principal types for expressions that depend on equality, such as the `elem` function in the prior section, introducing type-class *predicates* that restrict, or *qualify*, the variables that appear in their types. For example, the type of the equality operator reflects the restriction that it can only be instantiated at types in *Eq*:

```
(==) :: Eq t ⇒ t → t → Bool
```

In contrast to the `reverse` function, which can be used at any generic instance of its type scheme, the `(==)` function can only be used at generic instances that satisfy its constraints. For example, it can take on types

$$\text{Int} \to \text{Int} \to \text{Bool}$$

or

$$(\text{Int}, \text{Int}) \to (\text{Int}, \text{Int}) \to \text{Bool}$$

but not types such as

$$(\text{a} \to \text{b}) \to (\text{a} \to \text{b}) \to \text{Bool},$$

as we do not expect a decidable equality predicate for functions. We can use the overloaded equality operator to define the list membership function:

```
elem x []        =  False
elem x (y : ys)  =  x == y || elem x ys
```

which would have the principal type scheme

```
elem :: Eq t ⇒ t → [t] → Bool
```

including the qualifier, `Eq t`, arising from the use of `(==)`. Both of these uses of the `Eq` class are open: the programmer can either add new types to the `Eq` class, or write new functions that make use of its methods, without having to make any changes to existing code or usage.

Wadler and Blott give meaning to overloaded expressions, such as the definition of `elem`, using a *dictionary-passing translation*. In this approach, an expression with a qualified type is translated into a function with one (additional) parameter for each predicate in its qualifier. This parameter receives a tuple of type-specific implementations of class methods, called a *dictionary*, derived from the instance declarations in the program. References to class methods within the expression are translated into references to components of the dictionary arguments, while other overloaded expressions are translated to add suitable dictionary arguments. For example, the `Eq` class has two methods—the equality and inequality functions—so a dictionary for this class to might be described by tuple of functions of the following type

```
type EqD a = (a → a → Bool, a → a → Bool)
```

and each instance declaration would be translated into a corresponding value of type `EqD`; for example, the instance for integers above would be translated to the following value of type `EqD Int`:

```
eqIntD = (λx y → isZero (x - y), λx y → not (isZero (x - y)))
```

We can now demonstrate the dictionary-passing translation of the `elem` function. As shown above, the type scheme of the `elem` function has one qualifier; therefore, the type scheme of the translated function (for clarity, called `elemD`) will have one dictionary parameter:

```
elemD :: EqD t → t → [t] → Bool
```

The translation of the body of the `elem` function is straight-forward:

```
elemD (eq, neq) x []        =  False
elemD (eq, neq) x (y : ys)  =  eq x y || elemD (eq, neq) x ys
```

Note that the call to `(==)`, a class method, has been transformed into a call to the parameter `eq`, and that the recursive call to `elem` has been translated to a call to `elemD`, with the dictionary argument `(eq, neq)`. We could apply a similar approach to translate instances with qualifiers into dictionary constructors; for example, the instance of `Eq` for pairs might be translated:

```
eqPairD :: EqD a → EqD b → EqD (a, b)
eqPairD (eqA, neqA) (eqB, neqB) = (eqPair, neqPair)
    where eqPair (x, y) (x', y')  = eqA x x' && eqB y y'
          neqPair (x, y) (x', y') = neqA x x' || neqA y y'
```

Finally, where qualifiers can be proved in the original program, dictionary values are inserted in the translated program. For example, we would not expect a use of `elem` on list of integers to have a qualified type, as the constraint `Eq Int` can be discharged:

```
fiveIsOdd :: Bool
fiveIsOdd = elem 5 [1,3..]
```

In the translation of this code, `elemD` still needs dictionary arguments; however, we can provide those arguments using the dictionaries built from the instance declarations in the source program:

```
fiveIsOddD :: Bool
fiveIsOddD = elemD eqIntD 5 [1,3..]
```

The version we have presented here follows Wadler and Blott's description of type classes. To formalize type classes, they present a typing and translation relation in which classes and instances are lexically scoped, instead of being top-level, global declarations. This introduces some formal difficulties not present in their more intuitive description; in particular, as they observe, the introduction of local instances undermines principal typing.

2.3 THE THEORY OF QUALIFIED TYPES

In his *theory of qualified types* [28], Jones develops a general system of typing with qualification. Rather than focus on a particular form of predicate, as Wadler and Blott did in their work on type classes, Jones adopts an abstract notion of predicate. He demonstrates that his system can describe not only typing with type classes, but also forms of record typing and subsumption. He presents a generic overloaded lambda calculus, called OML, and its type system; because he treats predicates abstractly, Jones's type system distinguishes the generic manipulation of predicates necessary for typing from the domain-specific entailment relation among predicates.

Jones gives a notion of principal typing for OML; this is made more complex by the presence of predicates in types. We have previously described a principal type scheme for a given expression as being one such that any other type of the expression is a generic instance of its principal type, and we have claimed that the principal type is unique, up to renaming of bound variables. For example, we could give the `reverse` function either of the type schemes `[a]` $\rightarrow$ `[a]` or `[b]` $\rightarrow$ `[b]`; however, while these types are syntactically distinct, we can transform either to match the other, by suitable substitution for the bound variables `a` and `b`. This is no longer true for qualified types. For example, with the instances in the Haskell standard libraries, the predicate `Eq [t]` holds if, and only if, the predicate `Eq t` also holds. Thus, we can observe that, in an intuitive sense, the two type schemes `Eq t` $\Rightarrow$ `t` $\rightarrow$ `t` and `Eq [u]` $\Rightarrow$ `u` $\rightarrow$ `u` describe the same sets of types; however, there is no transformation of the variables `t` and `u` such that one will equal the other. To account for this difficulty, Jones develops the notion of a *principal satisfiable type* [28]. Intuitively, given some initial predicates P_0, a principal satisfiable type $P \Rightarrow \tau$ of some expression is one such that, for any other type $Q \Rightarrow \tau'$ of the same expression, τ' is a generic instance of τ, and, if Q follows from P_0, the instantiation of P also follows from P_0. Jones shows that each OML expression has a principal satisfiable type, and gives a type inference algorithm that computes such a type scheme for any well-typed OML expression.

Jones also develops a translation-based semantics for overloaded expressions, generalizing the approach of Wadler and Blott. As he treats predicates abstractly, he also introduces a notion of *evidence*, an abstraction of the implementation of predicates. Different forms of predicates give rise to different forms of evidence; for example, type classes might use dictionaries as evidence, as in the Wadler and Blott system, while evidence for record predicates might correspond to offsets into underlying data structures. As his type system is independent of the form of predicates, so his translation is independent of the form of evidence. His semantics is based on a translation to a typed lambda calculus with explicit introduction and elimination of evidence values, and a version of the entailment relation annotated to compute evidence.

One consequence of any instantiation of Jones's system of qualified types, including type classes, is that part of the semantics of programs is determined automatically by the compiler, based solely on the typing derivations of expressions. We would hope that, just as each of an expression's principal qualified types describes the same collection of types, each distinct translation of the expression has the same meaning. Jones refers to this property as the *coherence* of the translation. In some cases, we cannot hope to have a coherent translation. For example, the Haskell report defines classes `Show` and `Read` for converting values to and from textual representations, including class methods with the types

```
show :: Show t ⇒ t → String
read :: Read t ⇒ String → t
```

Given these functions, the expression `show ∘ read` has type

```
(Show t, Read t) ⇒ String → String
```

where the type parameter `t` in the `Show` and `Read` predicates does not appear in the type of the expression. Such types are called *ambiguous*, as, in the translation of an expression with an ambiguous type, the choice of dictionaries is necessarily arbitrary. Jones shows that, assuming a constraint on the entailment relation he calls uniqueness of evidence, the translations of expressions with unambiguous types in his semantics are coherent.

Jones's system is a natural foundation for our work on type classes: it allows us to focus on the class predicates and their manipulation, without having to simultaneously address the concerns of typing and type inference. His approach makes some assumptions of the entailment relation, such as uniqueness of evidence; we shall thus have to demonstrate that our entailment relation meets those assumptions. We will give a more formal recounting of the types and terms of OML when we discuss the semantics of overloading in Chapter 5; while we assume his type system, our semantics is not based on translation.

2.4 EXTENSIONS OF TYPE CLASSES

We conclude this background material by discussing several commonly used extensions to the Haskell class system: multi-parameter type classes, functional dependencies, and overlapping instances. We have two purposes in doing so. First, multi-parameter type classes and functional dependencies are both central to the Habit class system, and will appear regularly in the remainder of the dissertation. Second, the discussion of functional dependencies and overlapping instances will demonstrate applications of the notions of principal satisfiable types and coherence, described in the previous section.

2.4.1 Multi-Parameter Type Classes

Although Wadler and Blott [70] focussed on type classes with a single parameter (corresponding to sets of types with associated operators), they proposed that type classes could also apply to more than one parameter. They gave the example of a class to capture valid coercions between types; for example:

```
class Coerce a b
  where coerce :: a → b
instance Coerce Int Float
  where coerce = fromIntegral
instance Coerce a b ⇒ Coerce [a] [b]
  where coerce = map coerce
```

Just as single-parameter type classes can be interpreted as sets of types, multi-parameter type classes can be interpreted as relations on types (i.e., sets of tuples of types). From the instances above, we would expect `Coerce` to include the following subset of *Type* × *Type*

$$\{\langle Int, Float\rangle, \langle [Int], [Float]\rangle, \dots\},$$

where we write type tuples with angle brackets to distinguish them from the tuple type constructor.

2.4.2 Functional Dependencies

Despite their conceptual simplicity, many anticipated uses of multi-parameter type classes were problematic. For example [32], we might hope to use a type class to capture the relationship between collection types `c` and their element types `e`:

```
class Elems c e
    where empty  :: c
          insert :: e → c → c
          elem   :: e → c → Bool
```

(A fully-featured collections class might have many more methods; however, these are sufficient for our purposes.) This class might be populated for lists:

```
instance Eq t ⇒ Elems [t] t where ...
```

and could similarly be populated for binary search trees:

```
instance Ord t ⇒ Elems (BTree t) t where ...
```

Binary search trees need an ordering on their key type; in this case, that requirement is captured by the `Ord` qualification, which provides an (overloaded) `(<)` operator.

Unfortunately, any attempt to use this class will be problematic. One problem can be observed in the type of the `empty` method:

```
empty :: Elems c e ⇒ c
```

The type of `empty` is ambiguous: any attempt to discharge this predicate would require an arbitrary choice by the compiler as to the instantiation of `e`, and thus the evidence for the predicate `Elems c e`. As expressions containing `empty` cannot have a coherent translation, the compiler will reject its definition.

A further problem can be demonstrated with the `insert` function. Using it, we could write the following function

```
insert2 c = insert True (insert 'x' c)
```

to insert both the Boolean constant `True` and the character constant `'x'` into the collection `c`. This function has the type:

```
insert2 :: (Elems c Bool, Elems c Char) ⇒ c → c
```

Neither our examples of lists nor binary trees support this kind of usage. We will describe a well-typed heterogenous collection later (§3.2.2); however, its types would not fit the pattern required by the `Elems` class. Thus, while not erroneous given the current definitions, this constraint is likely to be unsatisfiable.

Both of these problems might be resolved if we restricted the `Elems` class to homogeneous collections. We could then observe that the type of `empty` is not truly ambiguous, as the type of the collection is specified and thus the type of elements is determined, and that `insert2` is an error, as it would require a collection type with distinct element types, an unsatisfiable constraint.

To restrict the `Elems` class to homogeneous collections, we must require that, for any type τ, there be at most one type τ' such that the predicate Elems τ τ' holds. This is an example of a *functional dependency* [40]. Functional dependencies are properties of a relation; the `Elem` class as populated by the prior instance has such a dependency. However, as Haskell type classes are open, we cannot be sure that future instances will preserve the dependency. Jones proposed adding *functional dependency constraints* to classes, such that all instances of that class were required to maintain certain dependencies [32]. For example, we could add a functional dependency constraint to the `Elems` class:

```
class Elems c e | c → e
```

This constraint plays two roles. First, it requires that all instances of the `Elems` class maintain the functional dependency. For example, given the instances earlier, it would be an error to add an instance such as the following

```
instance Elems [Int] Char
    where ...
```

as there would be two element types (`Int` and `Char`) associated with the collection type `[Int]`. Second, it provides additional information about the satisfiability of `Elems` constraints, addressing the concerns with the earlier uses of the `Elems` class. In the case of `empty`, it allows the compiler to conclude that, for any type `c`, there is at most one type such that `Elems c e` is satisfiable, with one evidence value for `Elems c e`, and thus that the type of `empty`, `Elems c e ⇒ c` is not truly ambiguous. In the case of `insert2`, it allows the compiler to determine that the constraints `Elems c Bool` and `Elems c Char` could only be satisfied if `Bool` and `Char` where the same type; instead, because they are distinct, the compiler rejects this definition.

2.4.3 Overlapping Instances

Two instances *overlap* if they could apply to the same predicate. For example, consider a type class `C` with the following instances:

```
instance C (a, [b]) where ...
instance C ([a], b) where ...
```

These instances overlap: either could be used to solve predicates like C ([Int], [Int]). However, the compiler has no guarantee that the class methods are implemented equivalently for both instances—that is, there is no guarantee that the evidence for C ([Int], [Int]) is unique—and so a program with both instances may have multiple, distinct interpretations. To avoid this kind of (potential) incoherence, Haskell 98 prohibits any overlap between instances.

This restriction is sometimes inconvenient. For one example, the `Show` class in the Haskell standard libraries includes types whose values have a textual representation:

```
class Show t
    where show :: t → String
          ...
```

Conventionally, `show` generates the Haskell syntax for its argument. For example, Haskell's syntax for lists surrounds the elements with brackets and separates them with commas. We could write a `Show` instance for lists that used this syntax:

```
instance Show t ⇒ Show [t]
    where show xs = "[" ++ intercalate "," (map show xs) ++ "]"
```

In addition to its standard syntax for lists, Haskell has special syntax to allow lists of characters to be written as character strings, delimited by double quotes. We might like to add a special instance of `Show` to handle this case, as in this simplified example:

```
instance Show [Char]
    where show xs = "\"" ++ xs ++ "\""
```

These two instances overlap: a `[Char]` value could be rendered either as a list or as a string. As a result, a program containing both instances would be rejected by a Haskell compiler.

Popular Haskell compilers have long supported language extensions that allows overlapping instances, as long as the instances can be ordered by specificity [26, 50].

This extension attempts to allow programmers to provide both general and type-specific instances, such as in the `Show` example above. We can specify this extension as follows. Given two instances for some class `D`:

```
instance P1 ⇒ D τᵢ where ...
instance P2 ⇒ D υᵢ where ...
```

these instances overlap if there is some instantiation of the variables in the τ_i such that the equal an instantiation of the variables in the υ_i. The first instance is more specific than the second if the τ_i can be instantiated to match the υ_i, but not vice versa. In the example instances of `C` given at the start of the section, both instances overlap, but neither is more specific than the other. Both are more specific than the instance

```
instance C (a, b) where ...
```

and less specific than the instance

```
instance C ([a], [b]) where ...
```

In resolving any individual predicate, the compiler chooses the most specific instance such that the instance conclusion unifies with the goal predicate. Note that the contexts (`P1` and `P2` above) do not factor into this determination; if the most specific instance does not solve the predicate, the compiler does not attempt to use less specific instances.

While the overlapping instances extension has a long history of use in the Haskell community, it is (despite the prior paragraphs) still mostly unspecified. This has two effects. First, different compilers implement the extension differently. For example, Hugs requires that any pair of overlapping instances be orderable by specificity; in contrast, GHC only requires that overlapping instances be orderable at predicates used in the program. Thus, GHC would accept a set of instances like

```
instance C (a, b)   where ...
instance C ([a], b) where ...
instance C (a, [b]) where ...
```

and only indicate an error if the programmer used a predicate of the form `C ([t], [u])`. Hugs, on the other hand, would reject the program because of the overlap between the second two instances, regardless of the predicates that appeared in the remainder of the program. Second, the interaction of the overlapping instances extension with other class system features is unspecified, or unsupported. For instance, the Haskell report specifies a simplification process, called context reduction, that attempts to reduce the complexity of inferred predicates [49, §4.5.3]. Given the instance of `Show` for lists above, for example, context reduction would simplify the predicate `Show [t]` to `Show t`. However, in the presence of overlapping instances, this simplification may not be sound, as there can be instances of `Show [t]` that do not correspond to instances of `Show t`. For another example, Hugs does not take the overlapping instances extension into account when validating instances against functional dependency constraints. While GHC continues to support overlapping instances, it does not support overlap in indexed type families (a feature for type-level programming), as such overlaps could introduce soundness issues.

2.4.4 The Coherence Problem

Despite their practical utility, we argue that, to be useful, overlapping instances must rely on an incoherent translation; this reliance, in turn, can be used to generate apparently nonsensical behavior. Thus, we believe that even if the problems of specification mentioned in the previous section could be resolved, overlapping instances would remain problematic in Haskell, or Haskell-like languages.

Suppose that we had a class with a generic instance, such as the following.

```
module A where
```

```
class C t where f :: t → Int
instance C a where f _ = 0
```

Our intention is that this instance provides a default implementation of method `f`, while allowing its behavior to be refined at specific types. Consider the following declaration, located in the same module as the class definition:

```
module A where
  ...
  g :: Char → Int
  g = f
```

Ought this definition type check? It seems like it must: context reduction requires that the constraint `C Int`, introduced by the use of class method `f`, be discharged, and there is a most specific instance that solves it. However, now consider a second module, as follows:

```
module B where
  import A
  instance C Char where f _ = 1

  b = f 'c' == g 'c'
```

The programmer may be surprised to discover that `b` is `False`. Simple equational reasoning would suggest that, as `g` is defined to be `f`, `b` amounts to the expression `f 'c' == f 'c'`, which is surely `True`. However, this interpretation fails to take account of the overlapping instances. In module `A`, where `g` is defined, the only instance of `C`, and thus only implementation of `f`, is the generic one. However, in module `B`, a second, more specific instance of `C` is available, and thus a different implementation of `f` is chosen.

We argue that this problem is unavoidable. First, this pattern is essential to overlapping instances: if the generic instance was not chosen to resolve some predicates, it would serve no use in the program. For translations using the generic instance to be coherent, it is not sufficient that there be no more specific instance at the expression itself; such a criterion is met in both definitions above. Instead, there must be no more specific instance at any use of the expression. However, this condition cannot be met: Haskell provides no mechanism for a module to constrain the instances of the modules that include it. Thus, any successful use of overlapping instances must depend on the incoherent selection of generic instances.

3. PROGRAMMING WITH INSTANCE CHAINS

In the previous chapter, we argued that some popular extensions of the Haskell class system are problematic: in particular, we identified coherence issues with overlapping instances and implementation divergence with both functional dependencies and overlapping instances. Despite these challenges, however, these features are central to many interesting examples of type-class programming in Haskell. One goal of Habit is to preserve and expand the scope and capabilities of Haskell-like type-class programming, while simplifying the underlying model of classes, and avoiding the problems encountered with Haskell type classes. To this end, we have developed a set of language features, collectively termed *instance chains*, that provide for many of the uses of overlapping instances, and extend the possibilities of type-class programming. This chapter describes the process that led to the design of instance chains, and demonstrates their use in type-level programming. We will discuss their semantics and the practical issues of implementing instance chains more fully in subsequent chapters.

This chapter begins with a survey of type-class programming in Haskell, drawing from the type class literature and from Hackage, a public repository of Haskell libraries and applications. We present data on the use of overlapping instances (§3.1), and describe several common type-class programming techniques (§3.2). We describe the language features of instance chains, and relate them to the results of our survey (§3.3). Finally, we work through an in-depth case study, demonstrating the use of instance chains to implement extensible data types (§3.4).

3.1 OVERLAPPING INSTANCES IN PRACTICE

Many of the problems with overlapping instances result from the open-ended nature of overlap: as any non-ground instance could, potentially, be overlapped by another instance elsewhere in the program, it becomes impossible to reason about the meaning of program components in isolation. Thus, we were curious about how important this open-endedness is to programs that depend on overlapping instances.

To help answer this question, in 2009, we surveyed the frequency and uses of overlapping instances in Hackage[1], a large online repository of Haskell libraries and applications. Our survey is distinguished from the folklore and informal input that often guide language design efforts both by being based on a large code library and by having an infrastructure to automate data collection. As much as possible, we reused the Hackage infrastructure to simplify the mechanics of the survey. In particular, we used and extended GHC [14] and `cabal-install` [20], a tool to download and install packages (and their dependencies) automatically from Hackage. We hoped to answer the following questions:

- How significant are overlapping instances in practice? In particular, how many of the projects on Hackage use overlapping instances?
- What are common syntactic patterns in the use of overlapping instances? How many instances overlap each other? Are these instances contained in the same module? The same package?
- What are common semantic patterns in the use of overlapping instances? What problems do programmers rely on overlapping instances to solve?

In turn, we expected that the answers to these questions would inform the design of

[1]`http://hackage.haskell.org`

instance chains, with which we hoped to capture the uses of overlapping instances, but to avoid their semantic difficulties.

The remainder of this section describes Hackage (§3.1.1), reviews the methodology of our survey (§3.1.2), and summarizes our results (§3.1.3). Finally, we address whether an alternative methodology, based on package metadata instead of compiler instrumentation, could have produced comparable results with significantly less effort (§3.1.4).

3.1.1 Hackage

Hackage is a large, online repository of Haskell libraries and applications. It organizes Haskell code into packages, each of which consists of a collection of source files along with a metadata file called a `.cabal` file. Each `.cabal` file contains the name and version of the package and the names and version ranges of the package's dependencies, and may optionally contain preferred optimization and profiling settings, language extensions used within the package, and compiler flags specified directly. The build and dependency information can, in turn, vary depending on the local configuration and available libraries. The `.cabal` file options also include ways to activate a number of standard Haskell preprocessors; however, unlike Makefiles, they cannot specify arbitrary additional tools or further modify the build process.

In addition to the online repository of packages, there are several other tools in the Hackage infrastructure. Among those most relevant to this work are Cabal (the "Common Architecture for Building Applications and Libraries") [1], which defines a library for building packages based on their `.cabal` files; and `cabal-install`, a tool for automatically downloading and installing packages and their dependencies.

While Cabal provides limited support for other Haskell compilers, such as Hugs, NHC and JHC, the majority of the language extensions that Cabal recognizes are only supported by GHC. Therefore, we used GHC for our survey and will restrict our attention to it for the remainder of this section.

3.1.2 Methodology

Our goal was to collect usage information on overlapping instances for as many of the packages on Hackage as possible. We hoped this would give us both an idea of how frequently Haskell programmers used overlapping instances, and a catalog of how they are used. In turn, these results would drive the design of the Habit class system.

We divided the survey into two stages: first, to find which packages use overlapping instances; and second, to identify the overlapping instances within each of those packages. While it would be possible to examine source code for overlapping instances by hand, this process would be vulnerable to human error and would become impractical for larger numbers of packages. Instead, we instrumented GHC to detect overlapping instances and to output information about the location of each such instance as it was encountered. We then attempted to build as many packages from Hackage as possible and collected the output of our instrumentation. This section describes our approach and evaluates its effectiveness.

Determining package sets

The Hackage infrastructure requires that any set of packages that it installs includes at most one version of each package [3]; unfortunately, because different packages on Hackage have conflicting requirements, this means that installing all of Hackage at once is not possible. Therefore, our first task was to determine the largest set of packages to check for overlapping instances.

To find such a set, we were inspired by Duncan Coutts' description of using Hackage for regression testing [4]. First, we used `cabal-install` to generate a list of all available packages. We then attempted a dry run of installing those packages. Predictably, `cabal-install` detected conflicting version requirements.

At this point, our approach differed slightly from that described by Coutts. Rather than attempting to restrict the selection of packages to get a close to optimal choice, we moved conflicting packages to a separate package list. As a consequence, we had a number of package sets, each internally consistent but inconsistent with all of the other sets.

This approach was moderately effective. Our initial package list included 1195 packages. From this, we constructed five package lists: the first contained 992 packages, and the remaining four included 139 more. This left 64 packages (5% of the total) that we made no attempt to install, because:

- They required C libraries or a version of GHC not available on our survey machine;
- They had internally inconsistent dependency requirements; or,
- They depended on packages that we were not attempting to install.

While our approach is simple to describe, filtering incompatible packages out of packages lists can be time consuming. In particular, if a given package is incompatible with a list, not only that package but all packages dependent on it must be removed from the list. To assist with this operation, we developed rudimentary support for tracing reverse dependencies through the Hackage database. Similar functionality is now independently available [66].

Instrumenting GHC

Our next task was to instrument the compiler to announce overlapping instances. By doing so, we avoided time-consuming and error-prone manual inspection of Haskell source code.

As described in Section 2.4.3, GHC orders overlapping instances by specificity when attempting to resolve a predicate and emits an error if the applicable instances cannot be so ordered. Thus, predicate resolution might seem like an ideal location for our instrumentation; however, the same set of overlapping instances might be detected numerous times, while other sets of overlapping instances might never be detected because no predicate required their use. Instead, we instrumented GHC's instance validation process. When validating instances, GHC checks that each new instance does not duplicate an instance it has already encountered. To do so, GHC computes all the instances that unify with the new instance. This corresponds precisely to the list of overlapping instances, so we added code to the duplicate instance check to output that list.

This check detects overlaps that are otherwise irrelevant to the compilation process. For example, consider the following overlapping instances:

```
instance C (a, [b])
instance C ([a], b)
```

Our overlap detection would output this set of instances. On the other hand, GHC will not check that it can order these instances until it attempts to resolve a predicate of the form `C ([a], [b])`. In fact, as long as a program does not require GHC to resolve a predicate of that form, it would not even need to enable overlapping instance support. On the other hand, because of the open-world nature of Haskell models, and as one of the options we were considering for Habit was a strict limitation on overlap akin to that implemented by Hugs, we were still interested in detecting this sort of unused overlap.

Collecting Results

Having identified consistent sets of packages and constructed an instrumented compiler, we were ready to generate our survey data. Following the technique described by Coutts, we compiled each set of packages independently. While we cannot avoid installing packages—a package can only be built if all of its dependencies are installed—we were able to use `cabal-install`'s existing functionality to ensure that each set of packages was installed to a distinct location and used a distinct local package database. As a result, the packages installed in one package set were not visible when building any other package set, and each of the sets could be built without conflicting with any other set.

Unlike Coutts' regression tests, we were interested in more information than whether each package compiled successfully; we also needed the overlapping instance information emitted during compilation. This meant that we had to extract the survey results manually from the build logs of each package, instead of being able to use the build reports that `cabal-install` generates automatically. Luckily, our output strings were easily identified by regular expression matching, so collecting the overlapping instances from the different package sets was relatively easy.

Alternatively, in the process of instrumenting GHC, it would have been possible to output the information that we collected to particular files, possibly specified by a command line option; this would have eliminated the need for the regular expression pass over the build output. We did not take this step in performing our survey, as the output of our instrumentation was easy to detect and our changes to GHC were otherwise quite local.

Evaluation

In this section, we consider the effectiveness of our methodology.

One advantage of our approach is that it required relatively little new code. While we had to modify the GHC type checker to emit details about overlapping instances, we were able to make use of the existing structure of the duplicate instance check. In total, we added 10 lines to GHC, not including comments. The changes to `cabal-install` to generate reverse dependences were larger—around 140 lines—but were localized to the implementation of a single additional command.

We were also able to achieve decent coverage of Hackage. We attempted to compile 1131 (95%) of 1195 packages, without making any attempt to repair broken dependencies manually or to install packages that either depended on absent C libraries or required non-Cabal build processes. Unfortunately, of these 1131 packages, only 826 packages (73%) built and installed successfully. The primary cause of build failures was our choice of which compiler to instrument. At the time that we performed the survey, the latest released version of GHC was 6.10.2, while the version in development was 6.11.20090330. One significant change from GHC 6.10 to 6.11 was that GHC's build system had been retooled and simplified. After several unexpected build failures using the 6.10 build tools, we decided to use 6.11 for the survey. While this resolved our build issues, it also had negative consequences. In addition to the compiler itself, GHC provides several packages, including the `base` package that includes the Haskell prelude as well as numerous primitive operations and basic combinators. GHC 6.11 included both Versions 3 and 4 of the `base` library, whereas GHC 6.10 had included only Version 3. As `base` Version 4 had not yet been released, some packages did not support the changes that it made, but still had dependencies on `base` without upper bounds. Cabal attempted to build these packages using `base` Version 4, which failed during compilation.

We believe that these deficiencies would be significantly reduced if the survey were redone now. The improvements to the build system have been (long since) incorporated into released versions of GHC, and incompatibilities with versions of the `base` library are also reduced by new requirements of Cabal and `cabal-install` [65].

A final note is that our methodology seems to be most suited to asking positive questions, such as "how often are overlapping instances used?" or "how many packages use GADTs?" because it is possible to identify code implementing these extensions within the compiler and to introduce local instrumentation at those points. It seems harder to adapt our approach to questions such as "how many packages only use language features in Haskell 98", as answering that question would require establishing that none of a (large) set of extensions are used. Instead of instrumenting a single point in the compiler, it would be necessary to check each extension of Haskell 98 and report whether any of them are used, most likely requiring non-local code changes and data collection.

3.1.3 Results

Of the 826 packages built during our survey, 57 (7%) used at least one overlapping instance. While this may seem like a relatively small proportion of the total code base, we think this level of usage is not insignificant, given that overlapping instances are an experimental and somewhat arcane feature of the Haskell type system.

In the packages that used overlapping instances, we found a total of 445 instances either overlapping or overlapped by other instances. We partitioned these instances into sets, where each instance in a set overlaps at least one other instance in the set, and no instances outside the set. The 445 overlapping instances partition into 123 sets. (Intuitively, imagine a graph with a vertex for each instance, and an edge between two vertices if their corresponding instances overlap. Our overlapping sets correspond to connected components in the graph.) We can draw further conclusions about the use of overlapping instances by examining the sets.

Our first question was how frequently the open-endedness of overlapping instances was necessary in practice. To answer this question, we determined whether the instances in each set were located in the same module, in different modules within the same package, or in different packages (Figure 3.1). Out of the 123 sets, 19 included overlapping instances from different modules, and 6 (of those 19) included overlapping instances from different packages. THe majority (104, or 85%) of the sets only included instances from a single module. This suggests that,

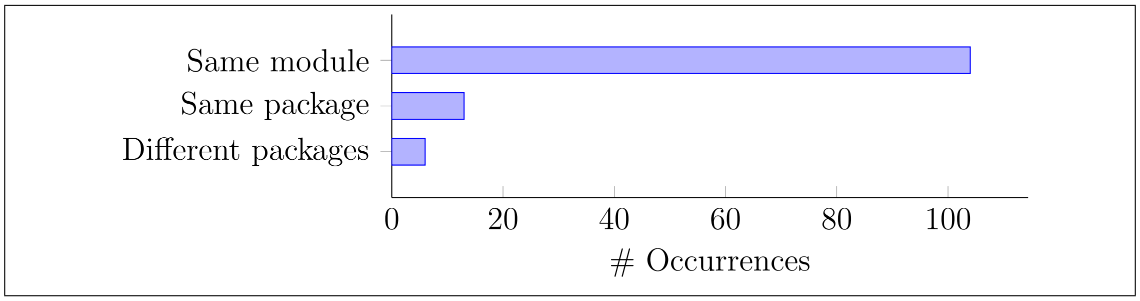


Figure 3.1: The vast majority of overlapping instance sets in our survey were in related source files.

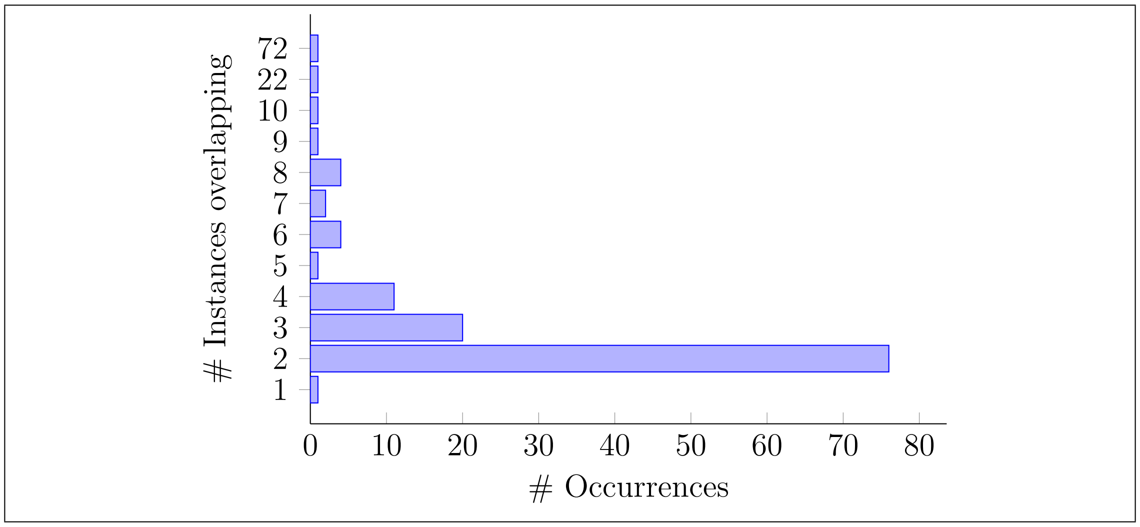


Figure 3.2: The majority of overlapping instance sets were relatively small.

while applications exist for instances overlapping across modules, most overlapping instances are defined locally.

We also analyzed the size (number of instances) of each set (Figure 3.2). On average, each set had 3.6 instances, although 76 (62%) of the sets had only two. The average is pulled up by several outliers: for example, one set of overlapping instances contains 72 instances. This resulted from the definition of a new `Show` instance:

```
instance JSON a ⇒ Show a where ...
```

that overlapped all other instances of the `Show` class. (One could argue that this instance is an abuse of the `Show` class, as its output is in JSON format instead of the Haskell syntax that most `Show` instances use.) As a final note, our data includes one set of overlapping instances containing only one instance; this resulted from a program containing two different modules, each of which defined exactly the same instance. As a result, the program in question was rejected by the compiler; however, because our data was generated simultaneously with compilation, we still detected the identical instances.

We found these results broadly encouraging: while there are some examples that make use of the full generality of overlapping instances, many of those we found involve relatively small numbers of instances in related modules. This suggested that approaches without the open-ended nature of overlapping instances could capture many of their usage patterns, while avoiding much of their complexity.

3.1.4 Using CABAL Metadata

The mechanism described in the previous sections may seem overly elaborate, especially given that support for overlapping instances must be enabled by specific compiler flags. As compiler flags are listed in `.cabal` files, it would seem that most packages that used overlapping instances could be detected by searching the `.cabal` files for the relevant compiler options or language extensions [61], and much of the previous work—particularly that which involved compiling large portions of Hackage—could have been avoided. There were several technical reasons that convinced us to take our more labor-intensive approach:

- While `.cabal` files are one place that language extensions may be specified, they are not the only place. Individual source files may also specify language extensions and compiler flags in compiler pragmas. Additionally, there are multiple ways that users can enable GHC's support for overlapping instances, including the `OverlappingInstances` language option, the

`-XOverlappingInstances` compiler flag, or the older `-fallow-overlapping-instances` compiler flag.

- The presence of overlapping instance support only enables the definition of overlapping instances; it does not require them. This means that packages that declare overlapping instance support may not actually contain any overlapping instances.

- Most significantly, GHC only requires that overlapping instance support be enabled in the module that defines the less specific (overlapped) instances. For example, consider the example instances for `Show` from Section 2.4.3:

  ```
  instance Show t ⇒ Show [t] where ...
  instance Show [Char] where ...
  ```

  If these instances were in separate modules (perhaps even in separate packages), then only the module that contained the `Show [t]` instance would need overlapping instance support enabled. As a consequence, while examining those modules that had overlapping instance support would allow us to detect all instances that could potentially be overlapped, it would not indicate whether, or how often, any of those instances were actually overlapped.

Having completed the survey, we returned to the question about whether using the Cabal metadata would be a suitable substitute for building all of Hackage. Surprisingly, we found that only 13 of the 57 packages that contained overlapping instances declared the corresponding language extension or GHC flag in their Cabal metadata. However, 59 packages that did not actually contain any overlapping instances included the overlapping instances flag in their metadata. We can imagine several reasons for this:

- Packages may use overlapping instances to provide default implementations for new classes without providing any more specific implementations. In this case, the package author would need to enable overlapping instance support, but our method would only find overlapping instances if there were more specific implementations elsewhere on Hackage.

- Package authors may use standard `.cabal` file templates, or may not remove options from `.cabal` files when they are no longer applicable.
- Package authors may prefer to use source file language pragmas when particular features or options are only needed in a portion of an entire package.

3.2 PROGRAMMING WITH TYPE CLASSES

This section provides several examples of the use of overlapping instances and functional dependencies that we found in our survey and in the literature. We discuss three usage patterns: *type functions*, illustrated by an implementation of type-level Peano arithmetic; *alternative implementations*, illustrated by an implementation of heterogeneous lists; and, *default implementations*, as illustrated in the prior discussion of overlapping instances (§2.4.3). We will return to these examples in the next section to motivate the corresponding features of instance chains.

3.2.1 Type Functions

This section illustrates the use of type classes to implement type functions, or type-level computations. To do so, we describe the implementation of several arithmetic operations at the type level, based on work by Thomas Hallgren [17].

Hallgren begins by defining Peano numbers at the type level (Figure 3.3), including types for zero and successor (Line 1) and for Boolean values (Line 2). As these types will be used only for type-level computation, they have no value-level constructors. Next, he defines the `Lte` class to implement the $\leq$ relation at the type level (Lines 4-7). Note that, as indicated by the functional dependency, Hallgren actually defines the characteristic function of the $\leq$ relation. This allows him

```haskell
data Z; data S n
data T; data F

class Lte m n b | m n → b
instance Lte Z (S n) T
instance Lte (S n) Z F
instance Lte m n b ⇒ Lte (S m) (S n) b
```

Figure 3.3: Type-level Peano numerals

```haskell
data Nil; data Cons x xs

class Sort xs ys | xs → ys
instance Sort Nil Nil
instance (Sort xs ys, Insert x ys zs) ⇒ Sort (Cons x xs) zs

class Insert x xs ys | x xs → ys
instance Insert x Nil (Cons x Nil)
instance (Lte x y b, InsertCons b x y ys r) ⇒ Insert x (Cons y ys) r

class InsertCons b x y xs ys | b x y xs → ys
instance InsertCons T x y xs (Cons x (Cons y ys))
instance Insert y xs ys ⇒ InsertCons F x y xs (Cons y ys)
```

Figure 3.4: Type-level insertion sort

more flexibility in using the `Lte` class, as he can use it to determine either that one number is less than or equal to another, or that it is greater.

To demonstrate the expressivity of this framework, Hallgren uses the `Lte` class to define a type-level insertion sort (Figure 3.4). To do so, he begins with a type-level representation of lists (Line 1). The implementation of sorting is straightforward: he provides a base case for empty lists (Line 4), and an inductive case (Line 5) that, given a non-empty list, sorts the remainder of the list and inserts the head element into the sorted list. The implementation of insertion (Lines 7-13) is not as simple. In particular, Hallgren defines insertion using two classes: the conditional behavior (depending on whether the element being inserted is less than the head of the list) is factored into its own class, `InsertCons`. It might seem more obvious to combine `Insert` and `InsertCons` into one definition, such as:

```
1  class Insert x xs ys | x xs → ys
2  instance Insert x Nil (Cons x Nil)
3  instance Lte x y T ⇒
4      Insert x (Cons y ys) (Cons x (Cons y ys))
5  instance (Lte x y F, Insert x ys zs) ⇒
6      Insert x (Cons y ys) (Cons y zs)
```

However, these instances would not be accepted by the compiler. Although we can see intuitively that the instances at Lines 3 and 4-5 cannot both apply to the same `x`, `y` and `ys`—as that would require both `Lte x y T` and `Lte x y F` to hold, violating the functional dependency on `Lte`—the overlapping instances extension only makes use of syntactic relationships between instances. In this case, the latter two instances for `Insert` fail to present such a syntactic relationship, and so would be rejected.

Hallgren reports that the Haskell implementation that he was using (Hugs 98) did not discharge the type constraints in uses of his insertion sort example. However, his instances work without modification in our compiler.

Another useful operation on Peano numbers is the greatest common divisor; for example, work on typing low-level data structures in Haskell has relied on a

```
1  class Subt m n p | m n → p
2  instance Subt Z n Z
3  instance Subt m Z m
4  instance Subt m n (S p) ⇒ Subt m (S n) p
5
6  class Gcd m n p | m n → p
7  instance Gcd m m m
8  instance (Lte n m b, Gcd1 b m n p) ⇒ Gcd m n p
9
10 class Gcd1 b m n p | b m n → p
11 instance (Subt m n m', Gcd m' n p) ⇒ Gcd1 T m n p
12 instance (Subt n m n', Gcd m n' p) ⇒ Gcd1 F m n p
```

Figure 3.5: Type-level greatest common divisor

type-level GCD operator [7]. Our implementation (Figure 3.5) begins by defining a (bounded) subtraction operation (Lines 1-4). We can then implement Euclid's algorithm (Lines 6-12), again using two classes to handle the conditional. As in the insertion sort example, the instances in a direct encoding of Euclid's algorithm could not introduce incoherence, but would still be rejected by the compiler because they are not syntactically distinguished by the instance conclusion.

3.2.2 Alternative Implementations

In the previous section, we demonstrated a collection of instances, in which the choice among instances was based on a type-level computation. In this section, by contrast, we will demonstrate alternative instances based on the structure of types. To do so, we will make use of overlapping instances. Our examples are drawn from a definition of type-safe heterogeneous lists, or *h-lists* [35]. Unlike standard Haskell lists, the elements of h-lists do not all have to have the same type. By reflecting

```
1 data Nil = Nil
2 data Cons t ts = t :*: ts
3
4 x :: Cons Char (Cons Bool Nil)
5 x = 'c' :*: (True :*: Nil)
6
7 y :: Cons Bool (Cons Char Nil)
8 y = True :*: ('c' :*: Nil)
```

Figure 3.6: Heterogeneous list types and examples

the types of the elements in the type of the list, type-safe operations over h-lists can be defined. As the implementation of these operations depends on the types of their arguments, we will rely on type classes and overlapping instances in their definition.

Heterogeneous lists are built from the type constructors `(:*:)` and `Nil`. Unlike the examples in the prior section, these constructors have both type- and value-level components. Note that the types of the head of the list (`t`, in Line 2) and the tail (`ts`) are reflected in the type of the list. This is demonstrated by the values `x` and `y`, in which the type of the list captures the type of each element of the list. Finally, we have introduced a distinguished value, `Nil`, to terminate lists at both the type and value levels.

Our first goal is to define a function `project` that projects a value from an h-list, based on the type of the value. In defining this operation, we want to ensure that it is only applied to lists that have a value of that type (so that its value is always defined) and that do not have multiple values of that type (so that there is no arbitrary choice of which value to project). This operation, for example, might be the basis of an extensible record system built using h-lists [34].

We can define the `project` function with two classes: the `HasOne` class, which

```
1  class HasOne t l where project :: l → t
2  class HasNone t l
3
4  instance HasNone t ts ⇒ HasOne t (Cons t ts)
5     where project (t :∗: _) = t
6
7  instance HasOne t ts ⇒ HasOne t (Cons u ts)
8     where project (_ :∗: ts) = project ts
9
10 class Fail t
11 data TypeExists t
12
13 instance HasNone t Nil
14 instance Fail (TypeExists t) ⇒ HasNone t (Cons t ts)
15 instance HasNone t ts ⇒ HasNone t (Cons u ts)
```

Figure 3.7: `project` class and implementation

implements the `project` operation, and the `HasNone` operation, which ensures that the projection is unique (Figure 3.7). The implementation of `HasOne` is straightforward. The first instance (Lines 4-5) covers the case where the goal type is at the head of the h-list, and does not appear anywhere in the remainder of the list. The second instance (Lines 7-8) covers the case where the goal type is not at the head, and does appear in the remainder of the list.

This example illustrates the use of overlapping instances to implement conditionals, and demonstrates several recurring patterns in the use of overlapping instances.

- First, it may not be obvious how the compiler chooses which instance to apply. It is not based on position in source code (as in expression-level conditional constructs), nor is it based on some kind of backtracking search (as in Prolog clauses). Rather, it is based on concluding that the first instance (Lines 4-5) describes a strictly smaller set of types than the second (Lines 6-7). A compiler can detect this by attempting to find substitutions such that the first instance matches the second (that is to say, the first instance can be instantiated to be syntically identical to the second), and vice versa. The substitution of `t` for `u` is sufficient for the second to match the first. However, there is no substitution that makes the first match the second. Therefore, the compiler can conclude that the second is more general, and check it only if the first does not apply.

- Second, consider the predicate

  ```
  HasOne Char (Cons Char (Cons Char ()))
  ```

  It might seem that the second instance would prove this predicate: although the first instance does not apply (because `Char` appears in the tail of the list), there is exactly one `Char` in the tail of the list. However, a Haskell compiler will only check one matching instance. Because the predicate matches the first instance, the compiler will never check the second, and will indicate an error when the preconditions of the first cannot be met.

- Finally, note that failure conditions (the type either appearing no, or multiple, times) are implicit. It would be possible for a programmer to add new instances of `HasOne`, inadvertently changing its behavior.

Next, we discuss the `HasNone` predicate. As this class is only used as a precondition for instances, it does not need any member functions. Its definition is structurally similar to that of `HasOne`: there is an instance for the base case (Line 13), one for the case where the goal matches the head of the list (Line 14), and one for the case where they differ (Line 15). However, the implementation of the

```
class Remove t ts us | t ts → us where remove :: ts → us

instance Remove t Nil Nil
    where remove Nil = Nil
instance Remove t ts us ⇒ Remove t (Cons t ts) us
    where remove (_ :∗: ts) = remove ts
instance Remove t ts us ⇒ Remove t (Cons u ts) us
    where remove (u :∗: ts) = u :∗: remove ts
```

Figure 3.8: Removing elements of a particular type from an h-list

matching instance (Line 14) is complicated. Intuitively, `HasNone` should not hold in this case; however, we cannot omit the case, as the final instance (Line 15) is general enough to include it. Instead, the original authors relied on creating a context that cannot be satisfied—in this case, the predicate `Fail (TypeExists t)`, named to give some indication as to the reason for the unsatisfiable predicate. Of course, were the `Fail` class and `TypeExists` types accessible outside the definition of this instance, a new instance satisfying `Fail (TypeExists t)` could be added. Thus, either the class or the type must be hidden using an additional mechanism, such as the Haskell module system.

This technique for encoding choice can be combined with the type-level computation mechanism from the previous section. For example, in defining an operation that removes all elements of a type from an h-list, we must compute not only the resulting list, but also its type (Figure 3.8). This example illustrates the difficulties introduced by the interaction of class system extensions: both GHC and Hugs claim that these instances violate the functional dependency constraint on class `Remove` (showing that their implementation of functional dependencies does not take overlapping instance resolution into account).

This section has demonstrated that overlapping instances provide a powerful technique for encoding alternative instances. However, we believe that this technique is also fragile: the intention of the programmer is (somewhat) obscured, and the method does not easily scale to more than two or three alternatives. This technique does not require open-ended overlapping instances; indeed, it would be surprising, and likely incorrect, were a user to add additional instances to one of the h-list classes. As a consequence, the instances are usually also syntactically grouped.

3.2.3 Default Implementations

Overlapping instances can also be used to provide default implementations for complex behavior, based on other pre-existing classes or other assumptions; this is arguably the purpose for which they were originally intended. We have already shown one such example (§2.4.3) in which we provided generic behavior for showing lists and specific behavior for showing lists of characters. Further examples of this pattern are quite common in generics and serialization packages—for example, the `EMGM` package [22] uses overlapping instances to provide default implementations that reduce the burden of writing new generic functions. However, this is precisely the usage that relies upon incoherent instance selection (§2.4.4).

It is sometimes unclear whether a given set of instances are intended to implement alternatives, as in the prior section, or to provide a default implementation along with several specific implementations. For example, the following instances appear in the monad transformer library [30], as implemented in the `mmtl` package:

```
instance MonadState s (State s) where ...

instance (MonadTrans t, Monad (t (State s)))
        ⇒ MonadState s (t (State s)) where ...
```

There are two ways we could interpret these instances:

- Any state monad should include the `State` type. This pair of instances provides a complete implementation of the `MonadState` class.

- The `State` type provides one way, but not the only way, to implement state monads. This pair of instances is not the complete implementation of the `MonadState` class.

It is not clear from the code which of these interpretations is intended.

3.3 DESIGN OF THE HABIT CLASS SYSTEM

In designing the Habit class system, our goal was to support, and expand upon, the overloading and type-level programming possible with the Haskell class system, while avoiding the semantic complexities introduced by some of its extensions. We began with a Haskell-style class system, extended with functional dependencies. However, we were reluctant to support overlapping instances: while our survey suggested that while many interesting examples of type-level programming used overlapping instances, the extension itself is poorly defined, theoretically problematic, and not well suited to many of its uses. Instead, we attempted to develop and define new language features that: would better support the majority of the uses of overlapping instances; would interact well with other class system features; and, would avoid the problems introduced by the inferred ordering and open-ended nature of the overlapping instances extension.

To that end, we developed three new class system features, which we refer to collectively as instance chains. These features are:

- Alternative clauses, allowing programmer-specified ordering of instances;

- Exclusion, allowing programmers to both assert that and test whether particular types are excluded from classes; and,

```
Type       ::= ... | ClassName {Type}
Pred       ::= ClassName Type {Type} [= Type] [fails]
Class      ::= class ClassName TyVar {TyVar} [= TyVar]
                     [ | Constraint {, Constraint } ]
                     [where Decls]
Constraint ::= Pred
            |  {TyVar} → {TyVar}
Clause     ::= Pred [if Pred {, Pred} ] [where Decls]
Chain      ::= instance Clause {else Clause}
```

Figure 3.9: Concrete syntax of Habit predicates, classes, and instances

- Backtracking search, providing a more flexible mechanism for instance selection consistent with the logical interpretation of instances.

We believe that these features are both more expressive and easier to understand that overlapping instances. In the remainder of this section, we will discuss the syntax of Habit predicates and classes and describe each of the features in detail. In particular, we will show that these features naturally express the type function and alternatives patterns that we identified in the previous section. Finally, we will conclude this section by discussing one approach to providing default implementations in Habit.

3.3.1 Habit Class System Syntax

The Habit class system incorporates a number of new syntactic features, including those necessary to support instance chains, as well as features designed to simplify and improve the use of functional dependencies, and to clarify the meaning of class and instance declarations. The surface syntax for Habit predicates, classes, and instances is given in Figure 3.9, excerpted, with some simplification, from the Habit language report [51]. Non-terminals are indicated by initial capitals, optional elements are surrounded by brackets, and optional repeatable elements are surrounded by braces. The Habit syntax differs from that of Haskell as follows:

- Predicates can be negated by appending the keyword `fails`; that is, if a predicate $C\,\vec{\tau}$ is satisfied by proving that the tuple of types $\vec{\tau}$ is an instance of class C, then a predicate $C\,\vec{\tau}\,\text{fails}$ is satisfied by proving that the tuple of types $\vec{\tau}$ is not an instance of C.

- Habit provides *functional notation* [7, 33], improving the notation for functionally determined types. In particular, if the final type τ_n in a predicate $C\ \tau_1 \ldots \tau_n$ is functionally determined, then (a) its predicates may be written $C\ \tau_1 \ldots \tau_{n-1} = \tau_n$, and (b) $C\ \tau_1 \ldots \tau_{n-1}$ may be written as a type, representing the unique type τ_n such that the predicate $C\ \tau_1 \ldots \tau_n$ holds.

- Habit syntax treats superclass constraints and functional dependency constraints uniformly. The Haskell syntax for superclass constraints is misleading: it suggests that the superclass constraints imply class membership, when really they are implied by class membership.

- Habit reverses the order of the instance hypotheses and conclusion; this draws emphasis to the instance being defined, as it appears first in the declaration.

- Habit allows multiple instance clauses to be chained together using the `else` keyword, providing a method for programmer specification of instance ordering.

Combined, these features can give Habit class and instance declarations a more functional feel than the equivalent Haskell declarations. For example, the Haskell instance declaration:

```
instance (Lte n m T, Subt m n m', Gcd m' n p) ⇒ Gcd m n p
```

such as might be used in a direct encoding of Euclid's algorithm (§3.2.1), can be written in Habit as:

```
instance Gcd m n = Gcd (Subt m n) n if Lte n m = T
```

There are some syntactic restrictions on Habit predicates and classes not captured in the syntax, such as:

- All clauses in a chain must be for the same class;
- Clauses asserting `fails` predicates must not provide method implementations; and,
- The use of = in predicates is not allowed if the last parameter is not determined.

We will use Habit syntax for the remainder of this document. However, with the exception of features specifically necessary for instance chains (the `fails` keyword in predicates and `else` keyword in instance declarations), none of our work depends on these changes to language syntax.

3.3.2 Alternative clauses

Many of the examples that we collected (§3.1) use overlapping instances to implement choice among instances. We have argued that this encoding is fragile: it relies on an inferred ordering between instances, and requires obscure reformulations to express even simple non-syntactic choices, such as in the `InsertCons` and `Gcd` examples (§3.2.1). In contrast, Habit provides a direct means of expressing alternative instances. Habit classes can be populated by multiple, non-overlapping instance chains, each of which may contain a number of potentially overlapping clauses, separated by the keyword `else`. During instance selection, clauses within a chain are checked in order; the order in which chains are checked is irrelevant, as they are not allowed to overlap. Using instance chains allows clearer expression of programmer intentions and simplifies the encoding of algorithms that would be complex or impossible to express with overlapping instances.

For example, the h-list classes (§3.2.2) relied on the ordering inferred by the overlapping instances extension. In Habit, these orderings must be made explicit. For example, the `HasOne` class can be populated using the following declaration.

```
instance HasOne t (Cons t ts) if HasNone t ts
    where project (t :*: _) = t
else HasOne t (Cons u ts) if HasOne t ts
    where project (_ :*: ts) = project ts
```

This clarifies the intention of the programmer. We can also use instance chains to express instances directly that would require multiple classes in Haskell. For example, the insertion sort class (§3.2.1) could be coded in Habit:

```
instance Insert x Nil = Cons x Nil
instance Insert x (Cons y ys) = Cons x (Cons y ys)
    if Lte x y = T
else Insert x (Cons y ys) = Cons y (Insert x ys)
    if Lte x y = F
```

The use of an instance chain for the second instance declaration avoids the needs for the auxiliary `InsertCons` class, as was required in Haskell. The order of clauses in the chain is irrelevant: as the hypotheses of the two instances are inconsistent, there are no predicates to which both could apply. Alternatively, we could have taken advantage of the ordering to omit the hypothesis on the second clause:

```
instance Insert x (Cons y ys) = Cons x (Cons y ys)
    if Lte x y = T
else Insert x (Cons y ys) = Cons y (Insert x ys)
```

These examples help motivate the remaining two features of instance chains:

- The `HasOne` class might seem to suffer from the same modularity problems as overlapping instances: because Haskell type classes are open, `HasNone t ts` may not be provable in one module, but provable in other modules in the same program. Thus, the meaning of a `HasOne` predicate might seem to depend on the other instances in scope. This problem is addressed by the introduction of exclusion (described next): we will distinguish failure to prove a predicate, which may change depending on instances, from the proof that a predicate does not hold. In this case, to choose the second clause, it is not enough that `HasNone t ts` cannot be proved; instead, `HasNone t ts fails` must be proved.

- The two clauses of the `Insert` instance for non-empty lists are not distinguished by syntax: given the functional dependency on `Insert`, we must be able to select a clause from the first two parameters. Instead, they are distinguished by whether the predicate `Lte x y` = `T` can be proven. Thus, instance selection must be based on the provability, not just the syntactic form, of instance clauses. We implement this via a backtracking search, such that disproving the hypotheses of a particular clause can result in trying the next clause in the chain.

3.3.3 Exclusion

As discussed in the previous section, allowing alternatives without introducing modularity concerns requires a way to distinguish the proof that a predicate cannot hold from the lack of a proof that it does hold. Additionally, some examples found in our survey attempt to encode failure in the instance selection process: for example, the `HasNone` class (§3.2.2) used a class `Fail` and type constructor `TypeExists`, both hidden using the Haskell module system, to ensure that particular instances could not hold. While this approach is effective, it is fragile, leads to confusing error messages, and is difficult to use as a building block for more complex instance schemes.

In Habit, we extend the syntax of predicates to include an optional trailing keyword `fails`. As the predicate $C\,\vec{\tau}$ is satsified if types $\vec{\tau}$ are in class C, the predicate $C\,\vec{\tau}\,\mathsf{fails}$ is satisfied if types $\vec{\tau}$ cannot be in class C. This provides a mechanism to invert predicates: the inverse of a predicate $C\,\vec{\tau}$ is $C\,\vec{\tau}\,\mathsf{fails}$, and the inverse of $C\,\vec{\tau}\,\mathsf{fails}$ is $C\,\vec{\tau}$. Predicates of the form $C\,\vec{\tau}\,\mathsf{fails}$ can appear as hypotheses and conclusions in instance declarations and as qualifiers in type signatures The presence of such instances introduces the requirement that Habit programs must be consistent: that is, that it should not be possible to prove both a predicate and its inverse from the instances in a program.

The `HasNone` class provides an obvious opportunity for `fails` predicates; in Habit, we could populate it by the instances:

```
instance HasNone t Nil
instance HasNone t (Cons t ts) fails
else HasNone t (Cons u ts) if HasNone t ts
```

We can also use fails predicates to simplify classes that might otherwise have relied on type functions. For example, Hallgren's `Gte` class (§3.2.1) actually defines the characteristic function of the $\leq$ relation rather than defining the $\leq$ relation itself. This allows Hallgren to use both the $\leq$ relation and its inverse. We can do this directly in Habit:

```
class Lte m n
instance Lte Z n
instance Lte (S m) (S n) if Lte m n
else Lte (S m) n fails
```

The uses of the `Lte` class would be updated similarly. For example, the `Gcd` class could be defined:

```
1  instance Gcd m m = m
2  else Gcd m n = Gcd (Subt m n) n if Lte n m
3  else Gcd m n = Gcd m (Subt n m) if Lte n m fails
```

As in the `Insert` example above, the clauses at Lines 2 and 3 could be presented in the opposite order. We could omit the final hypotheses (Line 3) as well; even if we did so, the `Gcd` relation's meaning depends on proving that inverse of the `Lte m n` predicate at Line 3.

Another use of explicit failure is in the definition of *closed classes*. For example, the AES cryptographic standard provides for keys of three possible lengths, either 128, 192, or 256 bits long. To abstract over the key length in their definitions, the `crypto` package [60], an implementation of various cryptographic primitives in Haskell, defined a class `AESKey` and instances for types `Word128`, `Word192`, and `Word256`. To assure that these were the only instances of the `AESKeyLength` class, the package authors hid the class using the module system. An unfortunate consequence is that users of the package could not write down the signatures of functions that used the `AESKey` class, such as

```
AESKey a ⇒ a → ByteString → ByteString
```

as the `AESKey` class itself was hidden. In Habit, we might similarly define a specific class to capture AES key lengths, and use a `fails` instance to close this class without hiding it.

```
class AESKeyLength n
instance AESKeyLength 128
else AESKeyLength 192
else AESKeyLength 256
else AESKeyLength t fails
```

No other instances of `AESKeyLength` could be added to the program, as they would overlap with the final clause in the instance. However, this would still allow the use of the `AESKeyLength` class in predicates, either as type qualifiers or hypotheses of other instances.

Note that `fails` instances are not the only mechanism for proving that a Habit predicate does not hold. Functional dependencies can also rule out predicates. For example, in a predicate `Subt m n p` (see Figure 3.5, the parameters `m` and `n` determine parameter p. Thus, if we can prove some particular instance of `Subt m n p`—such as `Subt 3 1 2`—we can also prove `Subt m n p' fails` for all types `p'` distinct from `p`—such as `Subt 3 1 1 fails` or `Subt 3 1 4 fails`. This is not new to Habit; however, we believe that ours is the first class system that makes use of these inferred exclusions in predicate simplification.

3.3.4 Backtracking Search

The Haskell instance search mechanism never backtracks. This is reasonable because the Haskell standard requires that no two instance conclusions in a given program unify. As a result, no predicate could be solved by more than one instance. However, this significantly complicates reasoning about overlapping instances. Even if an instance could apply to a predicate, it will not be checked if a more specific instance exists anywhere in the program, and failure to prove the preconditions of the most specific instance causes instance search to fail rather than to attempt to use less specific instances. This behavior is essential to some uses of overlapping instances: for example, the proper behavior of the `HasOne` and `HasNone` classes (§3.2.2) depends on the Haskell compiler only checking the most specific instance.

Habit instance search backtracks when it can disprove the precondition of an instance (either because of a `fails` clause or because of a functional dependency). This allows us to use provability, rather than syntactic matching, to choose between instances. For example, the Habit prelude includes a class `Index n`, used to generalize over indexing operations for arrays of length `n`. Some of these operations, such as reducing an arbitrary value to a valid index, can be performed more efficiently if `n` is a power of 2; thus, the Habit prelude includes an instance chain

```
instance Index n if 2^p = n
    where ...
else Index n
    where ...
```

Note that there is no syntactic distinction between the two clauses—both apply to arbitrary types `n`—and so the choice of a particular clause depends upon the compiler checking the provability of the constraint `2^p = n`.

3.3.5 Default Implementations

The features of instance chains are designed to clarify and extend many of the uses of overlapping instances. However, one of the usage patterns we identified (§3.2) is conspicuous by its absence: the default implementation pattern that was (arguably) the original intention of overlapping instances. We have argued (§2.4.4) that the default instances pattern introduces modularity problems that we have designed instance chains to avoid. This section presents an alternative scheme for providing default implementations. We do not claim this is necessarily the best way to provide default implementations in a system with instance chains; however, it demonstrates that default implementations are still feasible in practice, even without direct language support.

We return to our earlier example of overlapping instances and modularity concerns (§2.4.4), which provided some default behavior, with overrides for certain types. To implement this pattern in Habit, we will use two classes, one that indicates whether or not the default behavior is overridden at a given type, and another that selects between the overridden and default behavior. We show this

```
module A where
  class C t where f :: t → Int
  class COverride t where foverride :: t → Int

  instance C t if COverride t
      where f = foverride
  else C t
      where f _ = 0

  x = f 'c'

module B where
  import A
  instance COverride Char where foverride _ = 2
  b = x == f 'c'
```

Figure 3.10: Instance chains and default implementations

pattern in Figure 3.10. Class `COverride` is used to provide type-specific overrides of the default behavior. We can then define and populate class `C`, providing the default behavior in cases when it is not overridden. In module `A`, there is neither an instance of `COverride Char`, nor an instance of `COverride Char fails`; thus, the constraint `C Char` arising in the definition of `x` cannot be discharged. In module `B`, where an instance for `COverride Char` is provided, both `x` and `f 'c'` will use that instance, and `b` will be `True`, as we would expect. Alternatively, if an instance `COverride Char fails` were added to module `A`, the conflicting instance in module `B` would prevent the program from compiling. There are two inherent complications in this scheme. First, it introduces the additional override class. Second, to use the default implementation for a particular instance, it is not sufficient simply not to declare an override instance; the override must be explicitly ruled out by a `fails` instance. However, it is this latter complication that avoids the modularity problems inherent in overlapping instances: it allows the definition of `x` to use the default implementation of `f` only if there can be no overriding instance anywhere in the program.

3.4 CASE STUDY: EXTENSIBLE DATA TYPES

This section presents a case study in instance chains, using them to define extensible data types with flexible injection and projection operators. The problem of extending abstract data types is well known [2, 37, 53]. The formulation we choose, similar to that of Wadler [69], is to enable both the extension of data types and the definition of new operations over existing data types, without changing the meaning of existing code or losing static type safety. The particular data type we use is a simple abstract syntax tree (AST) for expressions; Wadler called this version of extensible data types the *expression problem*.

In addition to being commonly used as a benchmark for the expressiveness of programming languages, the expression problem illustrates a practical use of extensible data types. For example, the Habit compiler includes a number of small passes to desugar various source-level constructs into intermediate representations. While conceptually simple, these passes pose a software engineering dilemma: defining new versions of the AST for each pass would provide static guarantees of the transformations the passes implement, but would result in an explosion of similar syntax tree types and a corresponding duplication of utility functions. Alternatively, using a common representation of the syntax tree results in later passes having “junk” cases to handle constructs that were (supposedly) removed earlier in the pipeline. Extensible data types enable a third approach, incorporating the advantages of both the first and second approaches, as individual cases could be eliminated from the type without having to repeat the remaining cases or utility functions.

Our approach to the expression problem is based on existing Haskell approaches to extensible unions [38, 64], and has three components: a mechanism to define extensible data types, a mechanism to construct new values of these types, and a mechanism to use (or deconstruct) values of these types.

- To define new extensible types, we will use a generic coproduct (i.e. disjoint union) constructor, written `(:+:)`; individual cases will be defined independently, using open recursion (or two-level types [58]) for modular treatment of recursion. For a simplified example, if two cases were implemented by types `A` and `B`, we could construct their coproduct `A :+: B`

- We must be able to construct values of coproduct types from values of their component types; for example, we should be able to construct a value of type `A :+: B` from values of type `A` or values of type `B`. Attempting to use the constructors of the coproduct type directly is fragile and inextensible; for example, the function to inject a value of type `A` into `A :+: B` is different from that to inject a value of type `A` into `B :+: A` or into `(A :+: B) :+: C`. Previous approaches have used type classes to provide a generic injection operation, but this they have limited the forms of coproducts that can be used. We will demonstrate more flexible injection functions that do not restrict the form of coproducts.

- Finally, we must be able to use values of coproduct types. Again, using the primitive Haskell matching functionality is fragile—as the functions to inject values of type `A` into values of types `A :+: B` and `B :+: A` differ, so the case expressions to match values of type `A` in values of type `A :+: B` and `B :+: A` differ as well. Previous approaches have required that each use of a coproduct type be defined by a new type class, making the use of coproducts awkward and verbose. We will demonstrate a flexible projection operator,

```
1  data Const e t       = Const t
2  data Sum e t         = Plus (e t) (e t)         if Num t
3  data Product e t     = Times (e t) (e t)        if Num t
4  data Conjunction e t = And (e t) (e t)          if t == Bool
5  data Disjunction e t = Or (e t) (e t)           if t == Bool
6  data Conditional e t = If (e Bool) (e t) (e t)
7
8  data (f :+: g) (e :: ∗ → ∗) (t :: ∗)
9                       = Inl (f e t) | Inr (g e t)
10
11 data Expr e t        = In (e (Expr e) t)
```

Figure 3.11: Extensible expression types

allowing the uses of coproduct types to be defined at the value, rather than at the class, level.

These mechanisms are described in the detail in the following subsections.

3.4.1 Types with Open Recursion

Our first problem is to define the type of expressions. To make this type extensible, we will declare each case individually, combining different cases with a generic coproduct constructor, and leaving the type of sub-expressions parametric. Finally, we will use a fixpoint construction to fix the sub-expression type parameters. The declarations are given in Figure 3.11; we explain them in the following paragraphs.

We begin by defining the cases of our expression type. We include constructions for constants (Line 1), numeric expressions (Line 2-3), Boolean expressions (Line 4-5), and conditionals (Line 6). Each construct is parameterized by the result type `t` of evaluating the construct, and the type `e` of subexpressions. In some cases, the result of evaluating an expression is constrained by the expression constructors: for example, a sum can only evaluate to a numeric type. We capture these constraints using generalized algebraic data types (GADTs).[2] For example, the constraint on the `Plus` constructor requires that the type `t` produced by its subexpressions be numeric. Similarly, the constraint on the `And` constructor requires that the type produced by its subexpressions be `Bool`. The `If` constructor requires that its condition generate a Boolean value. However, as this does not affect the type of the conditional expression as a whole, we do not need a GADT for this case.

Next, we define the type of coproducts (Line 8), represented by the `(:+:)` operator. As in the individual constructors, the type of coproducts is parameterized by a result type `t` and subexpression type `e`. In turn, the members of the coproduct evaluate to the same type, and have the same subexpressions.

Finally, we define a type for expressions (Line 10), given a result type `t` and an expression constructor `e`. The `Expr` type instantiates the subexpression arguments of the expression constructors, "tying the knot" of the recursion.

Figure 3.12 includes the definition of several types, using these constructors. First, we define types `NumExpr` and `BoolExpr`, combining the individual constructors of numeric and Boolean expressions. Note that these still have the form of cases, not of expressions. Next, we define two types of expressions, `E1` and `E1'`, using the `Const` and `NumExpr` case types. Note the use of the `Expr` type to close the recursion.

3.4.2 Injection

Having defined the types of expressions, we next attempt to define some values of those types; Figure 3.12 shows a first attempt. We then inject several simple values into these types: `x` is a integer constant, and `y` is the integer expression `1 + 2`.

[2]The syntax we use for GADTs (similar to that of Sheard and Pasalic [59]) uses trailing constraints, rather than requiring the full type signature of constructors (as in GHC).

```
type NumExpr = Sum :+: Product
type BoolExpr = Conjunction :+: Disjunction

type E1 = Expr (Const :+: NumExpr)
type E1' = Expr (NumExpr :+: Const)

x :: E1 Int
x = In (Inl (Const 1))

y :: E1' Int
y = In (Inl (Inl (Sum (In (Inr (Const 1)))
                      (In (Inr (Const 2))))))
```

Figure 3.12: First attempt to define expression values.

Already, we can see that defining these values is becoming unwieldy. In particular, even though the types `E1` and `E1'` are equivalent (i.e. isomorphic), we can neither use an `E1` value where an `E1'` value is expected, nor use the same injectors. For example, the injector for a constant into `E1` is `In` ∘ `Inl`, but the corresponding injector for `E1'` is `In` ∘ `Inr`.

We can address some of these difficulties, following Liang et al. [38] and Swierstra [64], by using a type class to define an overloaded injection function, as shown in Figure 3.13. The predicate `f :<: g` holds if values of types constructed from `f` can be injected into types likewise constructed from `g`. However, difficulties arise in populating this class. Liang's instances, also given in the figure, rely on overlapping instances: the first instance is more specific than the second, and thus is checked first. However, these instances can only inject values into types directly on the left-hand side of the coproduct constructor. For example, they can inject values of type `Const` or `Sum` into type `E1`, but not values of type `Product`, and can

```
class f :<: g
    where inj :: f (e :: * → *) t → g e t

instance f :<: (f :+: g)
    where inj = Inl

instance (f :<: h) ⇒ f :<: (g :+: h)
    where inj = Inr ∘ inj
```

Figure 3.13: Overloaded injection function.

only inject values of type `NumExpr` into type `E1'`.

Swierstra addresses these difficulties by adding a third instance:

```
instance f :<: f
    where inj = id
```

With this instance, values of type `Product` can be injected into type `E1`, above. However, this instance does not make type `E1'` more useful. Further, this instance overlaps with the second instance in Figure 3.13: either one could apply to a predicate of the form `(t :+: u) :<: (t :+: u)`. Neither one is more specific than the other, so the overlapping instances mechanism does not provide a way to resolve this ambiguity. As a result, Hugs rejects this instance. GHC does not immediately reject the program, thanks to its lazier approach to detecting overlap; however, it will indicate a type error should such an ambiguous predicate appear in the program.

Using instance chains, we can more completely populate the `:<:` class, as shown in Figure 3.14. We begin with an identity instance (Lines 1–2); we avoid the difficulties that occur with Swierstra's instance, as the instance chain eliminates ambiguity among clauses. The next two clauses (Lines 3–6) recurse on either side

```
instance f :<: f
    where inj = id
else f :<: (g :+: h) if f :<: g
    where inj = Inl ∘ inj
else f :<: (g :+: h) if f :<: h
    where inj = Inr ∘ inj
else f :<: g fails
```

Figure 3.14: Overloaded injection function with instance chains.

```
class f :<<: g where inj' :: f (e :: * → *) t → g e t
instance f :<<: f
    where inj' = id
else f :<<: (g :+: h) if f :<<: g, f :<: h fails
    where inj' = Inl ∘ inj'
else f :<<: (g :+: h) if f :<<: h, f :<: g fails
    where inj' = Inr ∘ inj'
else f :<<: g fails
```

Figure 3.15: Stricter version of injection function.

of the coproduct constructor, avoiding the left-biased nature of the approaches of Swierstra and Liang et al. The final clause closes the class, providing a mechanism to prove when one type cannot be injected into another, and thus satisfying the conditions needed to bypass the second and third clauses).

One possible objection to this definition is that it (arbitrarily) chooses the left-most occurrence of a type in a coproduct. It might be desirable instead to reject cases where there are multiple, distinct injections for a given type into a given coproduct. We can define such a “stricter” notion of injection as shown

```
type E1 = Expr (Const :+: NumExpr)
type E2 = Expr ((BoolExpr :+: Const) :+: (Conditional :+: NumExpr))
inj_ x = In (inj x)

x :: E1 Int
x = inj_ (Const 1)
y :: E2 Int
y = inj_ (If (inj_ (And (inj_ (Const True)) (inj_ (Const False))))
             (inj_ (Plus (inj_ (Const 1)) (inj_ (Const 2))))
             (inj_ (Times (inj_ (Const 2)) (inj_ (Const 3)))))
```

Figure 3.16: Defining expression values

in Figure 3.15. The instance is structured as before; however, we add additional side conditions to ensure that, when injecting a type on the left-hand side of a coproduct, there is no possible (even ambiguous) injection on the right hand side, and vice versa.

We can now return to the task of defining values of our expression type (Figure 3.16). As before, we define several expression types; we also define a shorthand `inj_` for the composition of the overloaded injector and the `Expr` value constructor. Finally, we build several example values. Note that the treatment of different constructors is uniform, and not dependent on the particular ordering of types in the coproducts.

In this section, we have demonstrated how instance chains improve upon existing approaches to coding injectors in Haskell. Our approach is more powerful—that is to say, handles a greater variety of sum types—without complicating either the subtyping constraint `(:<:)` or the use of the injection function `inj`. Further, we have demonstrated how the features of instance chains can be used to build a new notion, the strict injector, on top of the existing notion of injection without having to change or redefine that existing notion.

3.4.3 Projection

The final part of the expression problem is to define extensible functions over the already-defined extensible types. While it is possible to do so using only existing features of Haskell, as Swierstra does, this relies on implementing all operations over sums as type classes themselves. In this section, we take an alternate approach, inspired by the treatment of extensible variants in TREX, a row-based record system for Haskell [12, 13]. We define a generic combinator, suitable for building projections over sum types, and provide two examples of its use, one to evaluate expressions and one to generate pretty-printed versions.

We begin by defining the projection combinator `(?)`. Intuitively, an expression `m ? n` defines a projection operator where `m` describes its action on one component of the sum, and `n` describes its actions on the remainder of the sum. As with the injection operator, we want to define `(?)` to work uniformly on isomorphic sum constructions, regardless of the order and nesting of the components.

Figure 3.17 gives our definition of `(?)`, as the method of a class `Without`. The predicate `Without t u = v` holds if `t` is a sum containing type `u`, and `v` describes the remaining components of `t` after removing `u`. For example, we can show that:

```
Without (Int :+: Bool) Bool = Int
Without ((Int :+: Char) :+: Bool) Char = Int :+: Bool
```

This class describes the remainder operated on by the second argument of the projection combinator; that is, if the expression `m ? n` operates on a sum `f e t`, and `m` operates on `g e t`, then `n` operates on `(without f g) e t`. We have chosen to use a strict definition of `Without`: each component type must appear in the sum exactly once, and each case must correspond to a component type in the sum. The clauses at Lines 3-5 and 6-8 define the base cases, each eliminating one of the two

```
1  class Without f g = h
2      where (?) :: (g e t → a) → (h e t → a) → f e t → a
3  instance Without (f :+: g) f = g
4      where (m ? n) (Inl x) = m x
5            (m ? n) (Inr x) = n x
6  else Without (f :+: g) g = f
7      where (m ? n) (Inl x) = n x
8            (m ? n) (Inr x) = m x
9  else Without (f :+: g) h = (Without f h :+: g) if h :<: g fails
10     where (m ? n) (Inl x) = (m ? n ∘ Inl) x
11           (m ? n) (Inr x) = n (Inr x)
12 else Without (f :+: g) h = (f :+: Without g h) if h :<: f fails
13     where (m ? n) (Inl x) = n (Inl x)
14           (m ? n) (Inr x) = (m ? n ∘ Inr) x
```

Figure 3.17: Overloaded projection combinator

summands. The implementation of `(?)` in these cases is obvious: the left-hand argument `m` is applied to the eliminated component, and the right-hand argument `n` to the remaining component. The recursive cases are more complex; we will describe the left-recursive case (Lines 9-11), as the two cases are mirror images of each other. First, we use the `(:<:)` class to ensure that the eliminated type `h` is only present on one side of the sum. Second, the argument type of `n` is no longer simply one side of the sum, but a reconstructed sum with one fewer component; thus, the calls to `n` must be composed with constructors for the new sum type.

Use of the `(?)` operator is broadly parallel to use of the `(:+:)` type constructor. We begin by defining projections for individual types; the evaluation projections are shown in Figure 3.18. As these cases can be used in multiple sum constructions, we handle recursive cases via a parameter `r`. The generic typing of these functions

```
evalConst   :: (e t → t) → Const e t       → t
evalSum     :: (e t → t) → Sum e t         → t
evalProduct :: (e t → t) → Product e t     → t
evalConj    :: (e t → t) → Conjunction e t → t
evalDisj    :: (e t → t) → Disjunction e t → t
evalCond    :: (e t → t) → Conditional e t → t

evalConst   r (Const x)   = x
evalSum     r (Plus x y)  = r x + r y
evalProduct r (Times x y) = r x ∗ r y
evalConj    r (And x y)   = r x && r y
evalDisj    r (Or x y)    = r x || r y
evalCond    r (If x y z)  = if r x then r y else r z
```

Figure 3.18: Individual evaluation cases

```
evalCases1 = evalConst ? evalSum ? evalProduct
evalCases2 = evalConj ? evalDisj ? evalCond ? evalCases1
eval1      = fix (λr (In e) → evalCases1 r e)
eval2      = fix (λr (In e) → evalCases2 r e)
```

Figure 3.19: Evaluation functions

(such as the result of `evalConj` being type `t`, not type `Bool`) is justified by the GADT constraints in the definitions of the component types.

Next, we can define the evaluation functions themselves by constructing the fixed points of combinations of cases. Figure 3.19 shows two such evaluation functions: the first (`eval1`) handles types such as `E1` that contain constants and numeric expressions; the second (`eval2`) handles types such as `E2` that additionally contain

```
showConst   r (Const x)   = show x
showSum     r (Plus x y)  = r x ++ "+" ++ r y
showProduct r (Times x y) = r x ++ "*" ++ r y
showConj    r (And x y)   = r x ++ "&&" ++ r y
showDisj    r (Or x y)    = r x ++ "||" ++ r y
showCond    r (If x y z)  = "if " ++ r x ++
                            " then " ++ r y ++
                            " else " r z

showCases1 = showProduct ? showConst ? showSum
showCases2 = showCond ? showDisj ? showConj ? showCases1
showExpr1  = fix (λr (In e) → evalCases1 r e)
showExpr2  = fix (λr (In e) → evalCases2 r e)
```

Figure 3.20: Expression printers

Boolean expressions. Note that the order of arguments to `(?)` is not related to the construction of the sum: `eval1` is equally applicable to arguments of type `E1` or `E1'`, despite the differing order of summands in those types.

Finally, Figure 3.20 demonstrates printers for expressions, using the `show` method to print constant values. The construction is similar to that for evaluation (albeit without the dependency on the GADT constraints) and has the same benefits: cases are defined independently, can be arbitrarily combined, and require neither modification of data type definitions, modification of existing functions, nor the introduction of new types or type classes.

3.5 RELATED WORK

Although they have been implemented in both Haskell and other languages, such as BitC [57], overlapping instances do not appear to have received much attention in prior research. Peyton Jones et al. [50] consider some of the issues with overlapping instances and other features of Haskell that were current at the time, such as context reduction. However, as the combination of functional dependencies and type classes had not yet been proposed, they do not anticipate many of the interactions that motivated the work in this dissertation.

The use of overlapping instances is not quite as sparse. We have already discussed the mechanisms that Swierstra [64] and Liang et al. [38] used to support type-level coproducts, and the approach that Kiselyov et al. [35] use to define a library for heterogeneous lists in Haskell. Kiselyov and Lämmel [34] take a similar approach in defining an object system in Haskell. In the latter two cases, the authors describe methods to avoid the use of overlapping instances, but at the cost of additional code complexity.

We also mechanically collected and analyzed uses of overlapping instances in the Hackage repository. We believe this is one of the first uses of Hackage to answer language design questions. However, there have been several similar projects. We were guided, for example, by Andrew Wright's study of the value restriction in Standard ML [71], which examined a wide variety of ML programs to determine whether a language design choice was justified, and by Duncan Coutts' description of using Hackage for regression testing [4].

Heeren and Hage [19] describe a technique for providing additional information to the type checker in the form of *type-class directives*, specified separately from the Haskell source code. These directives include types excluded from classes, such as excluding functions from the `Eq` class, and disjoint classes, such as requiring that the `Integral` and `Fractional` classes be disjoint. While specifying type-class directives separately allows them to be applied to existing Haskell code, it also limits their usability and generality. In particular, while they can specify that a particular predicate is excluded from a class, or that a class is closed, they cannot use that information in an instance precondition or qualified type. Heeren and Hage's directives do address some of the uses of explicit exclusion, such as closing classes or ensuring that classes are disjoint.

Jones [32] originally proposed the use of functional dependencies in type-class systems. Hallgren [17] describes some uses of functional dependencies for type-level computation, which we used for examples in Section 3.2.1. Alternative notation for functional dependencies was discussed by Neubauer et al. [46] and by Jones and Diatchki [33].

Much of the work described in this section has previously been described in our publications on instance chains [45] and on Hackage [44].

4. A LOGIC OF INSTANCE CHAINS

To this point, we have had two views of type classes. The first is as mappings from types to implementations of class methods: this explains the use of type classes to implement overloading. The second is as systems of predicates: this explains the role of type classes in typing and type inference. This chapter relates these two ideas. We begin by defining the syntax of predicates, instances, and class constraints (§4.1). We formalize the intuitive notion of classes as mappings from types to implementations (§4.2), and extend this notion to provide a Kripke-style model of class predicates and axioms (§4.3). Finally, we provide two judgments on predicates. We define a notion of acceptability (§4.4), and show that all acceptable programs have models, and we define a notion of entailment (§4.5), and show that it is sound and suitable for Jones's system of qualified types.

4.1 SYNTAX

Figure 4.1 gives a summary of the Habit class system syntax [51]. We have omitted some of the special cases of the full language—for example, we have omitted Habit's label and area kinds, and corresponding types—and we have simplified the surface syntax in some ways that do not compromise the expressivity of the class system—for example, we have omitted the optional "$= \tau$" suffix on predicates. We overload the symbol ε to refer to empty axioms and to the empty (non-`fails`) flag; its meaning will be unambiguous in context.

Types. Details of the type system are not significant to our semantics; we have fixed a particular concrete syntax purely to ease presentation and examples. We

Type variables	$t^\kappa \in \mathit{TVar}^\kappa$	Naturals	n
Type constants	K^κ	Class names	$C \in \mathit{ClassName}$
Index sets	$Y, Z \subseteq \mathbb{N}$		

$$
\begin{array}{lrcl}
\text{Kinds} & \kappa & ::= & \star \mid \mathsf{nat} \mid \kappa \to \kappa \\
\text{Types} & \tau^\kappa, \upsilon^\kappa \in \mathit{Type}^\kappa & ::= & t^\kappa \mid n \mid K^\kappa \mid \tau^{\kappa' \to \kappa}\, \tau^{\kappa'} \\
\text{Flags} & f \in \mathit{Flags} & ::= & \varepsilon \mid \mathtt{fails} \\
\text{Predicates} & \pi \in \mathit{Pred} & ::= & C\ \vec{\tau}\ f \\
\text{Contexts} & P, Q & ::= & \vec{\pi} \\
\text{Clauses} & \xi & ::= & d : \forall \vec{t}.\, \pi \Leftarrow P \\
\text{Schematic axioms} & \alpha & ::= & \varepsilon \mid \xi \,;\, \alpha \\
\text{Ground axioms} & \gamma & ::= & \varepsilon \mid (d : \pi \Leftarrow P) \,;\, \gamma \\
\text{Class constraints} & \chi & ::= & C : \vec{\kappa} \mid C : Y \rightsquigarrow Z \mid \forall \vec{t}.\, \pi \Rightarrow \pi'
\end{array}
$$

Figure 4.1: Habit class system syntax (abbreviated)

use $\mathit{TVar} = \bigcup_{\kappa \in \mathit{Kind}} \mathit{TVar}^\kappa$ and $\mathit{Type} = \bigcup_{\kappa \in \mathit{Kind}} \mathit{Type}^\kappa$, and will omit the kind annotations on types and type variables when they are irrelevant, or can be inferred from context. Habit has a number of built-in kinds, all treated identically for the purposes of this presentation; we have chosen to include types n of kind nat as we have already used type-level naturals in a number of examples. We write GType^κ for the set of *ground types* of kind κ, that is, those elements of Type^κ that contain no type variables, and we write GType for $\bigcup_{\kappa \in \mathit{Kind}} \mathit{GType}^\kappa$.

Predicates. We will refer to predicates of the form $C\ \vec{\tau}$ as *positive* predicates, and those of the form $C\ \vec{\tau}$ `fails` as *negative predicates*. Unlike many logics, we do not have a distinct negation operator; instead negation is treated as part of the syntax of predicates. However, we can give a syntactic definition of the negation of a predicate π, written $\overline{\pi}$:

$$\overline{C\ \vec{\tau}} = C\ \vec{\tau}\ \texttt{fails} \qquad\qquad \overline{C\ \vec{\tau}\ \texttt{fails}} = C\ \vec{\tau}$$

Tuples. Given some set A we will write products $A \times A$ as A^2, $A \times A \times A$ as A^3 and so forth. We will write $\vec{a} \in A^n$ for tuple values (avoiding confusion with the notation for negation). We will assume that tuples are indexed by naturals and will write $|\vec{a}|$ for the number of elements in $\vec{a}$; for example, if $\vec{a} \in A^2$, then we have that $|\vec{a}| = 2$ and write a_0 for the first element of the tuple and a_1 for the second.

Axioms. We distinguish between schematic axioms, in which each clause may itself be quantified over some set of type variables, and ground axioms, in which they are not. We will assume that the clauses in axioms are indexed by the naturals. We require that all clauses in an axiom make assertions about the same class, and define:

$$\text{class}(C\ \vec{\tau}\ f) = C$$
$$\text{class}(\alpha) = C \text{ such that } \forall (d : \forall \vec{t}.\, \pi \Leftarrow P) \in \alpha.\ \text{class}(\pi) = C.$$

We will make the same assumption, and use the same notation, for ground axioms γ. Note that we use short versions of implication symbols ($\Rightarrow$ and $\Leftarrow$) for syntactic symbols and longer versions ($\Longrightarrow$ and $\Longleftarrow$) for meta-level implications.

Substitutions. We use a standard notion of substitutions for mappings of type variables to types (of matching kinds). If $\vec{t}$ is a sequence of type variables, we write $\mathit{Subst}(\vec{t})$ for the substitutions with domain $\vec{t}$, and $\mathit{GSubst}(\vec{t})$ for the ground substitutions with domain $\vec{t}$. We define the action of type substitutions on types in the usual fashion.

Class constraints. We include three varieties of class constraints, so called because they restrict the possible membership of classes. Kind constraints $C : \vec{\kappa}$

restrict the tuples of class C to have the kinds specified. Functional dependency constraints $C : Y \rightsquigarrow Z$ require that the tuples in class C preserve the given functional dependency. Superclass constraints $\forall \vec{t}.\, \pi \Rightarrow \pi'$ indicate that whenever predicate π is provable, predicate π' must also be provable; the form of π and π' are limited by the syntactic form of Habit superclasses.

Functional dependencies. We will frequently be interested in those functional dependency constraints that apply to a particular class (or predicate on that class). We use the following (overloaded) function to capture this pattern: if X is some set of class constraints, then we define the functional dependencies of class C in X as:

$$\mathrm{fd}_X(C) = \{\, Y \rightsquigarrow Z \mid (C : Y \rightsquigarrow Z) \in X\} \cup \{\mathbb{N} \rightsquigarrow \emptyset\}$$

and similarly for the functional dependencies of a predicate:

$$\mathrm{fd}_X(C\ \vec{\tau}\ f) = \mathrm{fd}_X(C).$$

The set X will be omitted when it is obvious from context. To ensure that $\mathrm{fd}_X(C)$ is never empty, we have added $\mathbb{N} \rightsquigarrow \emptyset$ to the functional dependencies for all classes. This constraint is trivially satisified, as it does not require any position to be determined from the others. Thus, any relation satisfies the dependency $\mathbb{N} \rightsquigarrow \emptyset$, and so adding it does not affect the modelling of programs. Later rules will be able to assume that all classes have at least one functional dependency constraint.

Relations modulo functional dependencies. When considering predicates and an associated functional dependency, it is useful to consider the predicates without including any of the parameters that are determined by the dependency. For example, to know whether the instances

```
instance Eq t ⇒ Elems [t] t
instance Elems [Int] Char
```

are in conflict, it is not enough just to try unifying the conclusions of the instances `Elems [t] t` and `Elems [Int] Char` (which would fail, suggesting the instances are not in conflict). Rather, we must take the functional dependency for `Elems` into account, and attempt to unify the determining parameters `[t]` with `[Int]`. In this example, the latter unification succeeds, showing that the instances are in conflict. We can generalize this idea to any relation on types R and any index set Z by writing $\pi R \pi' \bmod Z$ to indicate the result of relating only those parameters of π and π' not indexed by Z. Formally we define

$$(C\ \vec{\tau}\ f) R (C'\ \vec{\upsilon}\ f') \bmod Z \iff (C = C' \ \wedge\ f = f' \ \wedge\ \forall i \notin Z.\, \tau_i R \tau'_i),$$

The notation "$\pi R \pi' \bmod Z$" is chosen by analogy with modular arithmetic: as powers of x are not distinguished by arithmetic modulo x, so the elements indexed by Z are not distinguished in relations modulo index set Z.

4.2 CLASSES AND IMPLEMENTATIONS

This section formalizes the intuitive semantics of a type class as a map from types (or tuples of types) to implementations of the class methods at those types. In the remainder of this chapter, we will use this formalization as the basis to define a logic of type classes, and will relate it to different formal elements of programs, such as predicates, instance declarations, and class constraints. In the following chapter, we will use the same formal structure of classes to give semantics to programs with overloading.

The semantics of a single parameter type class, such as `Eq` or `Ord`, can be formalized as a partial mappings from types to evidence values. The domain of such a mapping consists of all types in the class; for example, given the standard `Eq` class with instances for integers and pairs (§2.2), the model of the `Eq` class would have the domain

$$\{\mathit{Int},\ (\mathit{Int},\mathit{Int}),\ (\mathit{Int},(\mathit{Int},\mathit{Int})),\ ((\mathit{Int},\mathit{Int}),\mathit{Int}),\ \ldots\}.$$

The codomain of the mapping depends on the particular class being modelled; for the `Eq` class, it consists of type-specific implementations of equality and inequality operators, while for the `Ord` class we would expect implementations of comparison operators. We will write the implementations for the methods of class C at types $\vec{\tau}$ as $\mathit{Impl}_C(\vec{\tau})$—for example, $\mathit{Impl}_{\mathit{Eq}}(\mathit{Int})$ would be the semantic interpretation of the tuple of equality and inequality tests. Thus, we have that a semantics G_{Eq} of the `Eq` class should be a partial function

$$G_{\mathit{Eq}} : (\tau : \mathit{GType}) \rightharpoonup \mathit{Impl}_{\mathit{Eq}}(\tau).$$

The function is partial because not all types are in the *Eq* class. We have used the dependent notation as a convenient abbreviation, not to indicate a particular type-theoretic framework for our semantics. In this case, an equivalent formulation would have been:

$$G_{\mathit{Eq}} : \mathit{GType} \rightharpoonup \bigcup_{\tau \in \mathit{GType}} \mathit{Impl}_{\mathit{Eq}}(\tau) \text{ such that } \forall \tau \in \mathrm{dom}(G_{\mathit{Eq}}).\ G_{\mathit{Eq}}(\tau) \in \mathit{Impl}_{\mathit{Eq}}(\tau).$$

This approach extends naturally to multi-parameter type classes by extending the domain of the semantics from types to products of types. The domain of the mapping is now a relation on types, capturing the type-level information encoded in the class; the codomain, as before, captures the evidence that particular tuples of type are in the class. The semantics of a two-parameter class, such as the `Elems` class (§2.4.2) would be a function:

$$G_{\mathit{Elems}} : (\vec{\tau} : \mathit{GType}^2) \rightharpoonup \mathit{Impl}_{\mathit{Elems}}(\vec{\tau}).$$

A three-parameter class would have GType^3 as its domain, and so forth. The number of parameters of a class is called its *arity*, and we will write arity(C) to refer to the arity of class C. We can now write a general rule that captures the examples so far: for a class C, we have that:

$$G_C : (\vec{\tau} : \mathit{GType}^{\mathrm{arity}(C)}) \rightharpoonup \mathit{Impl}_C(\vec{\tau}).$$

Finally, we can define a semantics G for all the classes in a program, parameterized by class names:

$$G : (C : \mathit{ClassName}) \rightharpoonup ((\vec{\tau} : \mathit{GType}^{\mathrm{arity}(C)}) \rightharpoonup \mathit{Impl}_C(\vec{\tau})).$$

We call any object of this form a *model structure* for a class system.

This formulation of model structures is quite general: each class has different methods, and a given model structure may contain arbitrary implementations of the class methods. This generality will be useful when extending our semantics of classes to the semantics of programs. In this chapter, however, we are not concerned with the method implementations themselves, and so will introduce an abbreviated notion, which we call *evidence expressions*. Evidence expressions capture the intuition that each method implementation derives from a particular collection of class instances; thus, by identifying the instances, we have enough information to identify the actual implementations. We give the following syntax for evidence expressions:

$$\begin{array}{lrcl} \text{Evidence constructors} & d & \in & \mathit{InstName} \\ \text{Evidence expressions} & e & ::= & d\ e \mid \langle \vec{e} \rangle \mid \bullet \end{array}$$

Evidence constructors d, drawn from some suitable set of names, identify particular instances. Evidence expressions represent combinations of instances. Negative predicates correspond to no method implementations; however, to permit uniform treatment of positive and negative predicates, we introduce a distinguished value $\bullet$ to serve as the evidence expression for negative predicates.

4.3 MODELLING TYPE CLASSES

Next, we extend the semantic view of type classes to model a logic of type class constructs, such as predicates or instances. Our approach follows Kripke's technique to model intuitionistic and modal logics [36]. A Kripke model is a triple $\langle \mathcal{G}, \preceq, \models \rangle$, where

- $\mathcal{G}$ is a set of *nodes* (alternatively called points or possible worlds), each of which corresponds to a particular collection of knowledge about the predicates being modelled.
- $\preceq$ is a relation on elements of $\mathcal{G}$, where for $G, G' \in \mathcal{G}$, $G \preceq G'$ if G' represents an extension of the knowledge represented by G. Different logics impose different constraints on $\preceq$; for our purposes, we will assume that it is reflexive and transitive.
- $\models$, sometimes called the *forcing* relation, relates elements of $\mathcal{G}$ to logical formulae, again subject to constraints depending on the logic being modelled.

There are several reasons that this approach is suited to modelling type classes. First, our notion of proof is constructive: a proof of `Eq Int`, for example, must provide implementations of the equality and inequality predicates. Second, because type classes are open, our notion of refutation is intuitionistic. The typing of an expression, and thus any predicate entailment or semantic interpretation of the expression, must be consistent with any (well typed) use of the expression in the remainder of the program. For example, when typing a particular expression, we may either have evidence for `Eq Int`, evidence for `Eq Int fails`, or neither; we expect any typings or entailments derivable in the third case to hold in environments where either `Eq Int` or `Eq Int fails` are provable.

Typically, a logic is related to Kripke models by leaving the structure of nodes abstract, but by constraining the behavior of the extension and forcing relations. In contrast, we have a fixed model structure—the semantic view of type classes presented in the prior section—and can thus give concrete definitions of the extension and forcing relations. We begin with extension: for any $G, G' \in \mathcal{G}$:

$$G \preceq G' \iff \forall C \in \mathrm{dom}(G). G(C) \subseteq G'(C),$$

or, equivalently

$$G \preceq G' \iff \forall C \in \mathrm{dom}(G), \vec{\tau} \in \mathrm{dom}(G(C)).G(C) = G'(C).$$

To define the forcing relation, we must begin by defining a notion of formulae for type classes, corresponding to the various forms of class expressions and declarations; over the remainder of the section, we shall treat each type of formula in turn.

$$\text{Formulae} \quad \phi \quad ::= \quad \langle \pi, e \rangle \mid \langle P, e \rangle \mid \overline{P} \mid \langle \alpha, X \rangle \mid \chi$$

We associate predicates π and contexts P with the evidence that they hold; we will generally write $G \models e : \pi$ instead of $G \models \langle \pi, e \rangle$, and similarly for contexts. The modelling of an axiom will depend on the class constraints under which it is modelled; we will write $G, X \models \alpha$ for $G \models \langle \alpha, X \rangle$.

4.3.1 Predicates

Predicates are the fundamental part of the Habit class system, corresponding to literals (i.e., atomic formulate and their negations) in logic. As in Haskell, the predicate $C\ \vec{\tau}$ holds if types $\vec{\tau}$ are in class C; the predicate $C\ \vec{\tau}$ `fails`, which holds if types $\vec{\tau}$ are not in class C, has no direct analogue in Haskell.

Predicates map directly to the model of classes. For $G \in \mathcal{G}$, we say that:

$$G \models e : C\ \vec{\tau} \iff G(C)(\vec{\tau}) = e$$
$$G \models \bullet : C\ \vec{\tau}\ \texttt{fails} \iff \forall G' \succeq G.\, \vec{\tau} \notin \mathrm{dom}(G(C))$$

A predicate $C\ \vec{\tau}$ `fails` holds not just if there is no evidence for it in G, but if it is somehow incompatible with G; we capture this by saying that there is no extension G' of G such that semantics of C in G' includes $\vec{\tau}$. This corresponds to the definition of forcing for negation in intuitionistic logics. Observe that, by the definition of extension, if $G \models e : \pi$, then $G' \models e : \pi$ for all $G' \succeq G$. As in intuitionistic logics, it is not necessarily true that, for any predicate π, there is some evidence expression e such that either $G \models e : \pi$ or $G \models e : \overline{\pi}$.

Contexts, or sequences of predicates, occur frequently in Habit programs, both as hypotheses in instance declarations and as qualifiers in qualified types. A context is treated as a conjunction of predicates; writing $|P|$ for the length of context P, we define that:

$$G \models \langle e_i \rangle : P \iff \forall i < |P|.\ G \models e_i : P_i.$$

The negation of a context cannot occur syntactically in a Habit program; however, it will occur in the modelling of instance chains, as we will need to represent not only when the hypotheses of a particular clause hold, but also when they do not. Modelling the negation of a context P, written $\overline{P}$, is inspired by DeMorgan's law:

$$G \models \overline{P} \iff \exists i, e.\ G \models e : \overline{P_i}.$$

We do not associate evidence with a negated context: there many be many suitable indices i with different corresponding evidence values e.

4.3.2 Axioms

We next describe the modelling of instance chains. An individual instance chain includes both logical assertions (for example, an instance for `Eq Bool` asserts that type `Bool` is in class `Eq`) and method implementations (such as the implementation of the equality operator). Clauses within the chain may themselves be polymorphic. We define a schematic axiom corresponding to each instance chain to capture its logical assertions. For example, given a class `C` with method `f`, the schematic axiom corresponding to the instance chain:

```
instance C Int   where f = ...
else C Bool      where f = ...
else C t if D t  where f = ...
```

would be:

$$(\forall.\, \mathrm{C\ Int} \Leftarrow ())\ ;\ (\forall.\, \mathrm{C\ Bool} \Leftarrow ())\ ;\ (\forall t.\, \mathrm{C}\ t \Leftarrow (\mathrm{D}\ t))\ ;\ \varepsilon.$$

We include empty quantifiers "$\forall.$" and qualifiers "$()$" here to emphasize that these are schematic clauses, even if only the last clause actually has multiple instantiations.

Rather than attempting to model schematic axioms directly, we begin with a notion of *specialization*, relating schematic axioms to sets of ground axioms, eliminating any polymorphism present in the original scheme. We will relate ground axioms to models of classes, and then relate schematic axioms to models of classes through their corresponding ground axioms.

Intuitively, we specialize an axiom scheme α for class C by enumerating each well-formed predicate π of class C, and then attempting to restrict each clause in α to that predicate. Consider the example instance chain and schematic axiom for class `C` above. If the set of ground types were limited to $\{\mathrm{Int}, \mathrm{Bool}, \mathrm{Float}\}$, then we could generate the following three specializations of this axiom scheme, one for each positive predicate on C:

$$\begin{aligned}
&(\mathrm{C\ Int} \Leftarrow ())\ ;\ (\mathrm{C\ Int} \Leftarrow (\mathrm{D\ Int}))\ ;\ \varepsilon \\
&(\mathrm{C\ Bool} \Leftarrow ())\ ;\ (\mathrm{C\ Bool} \Leftarrow (\mathrm{D\ Bool}))\ ;\ \varepsilon \\
&(\mathrm{C\ Float} \Leftarrow (\mathrm{D\ Float}))\ ;\ \varepsilon
\end{aligned}$$

This is an entirely syntactic process; for example, in the first specialization, the clause `C Int` $\Leftarrow$ `(D Int)` is included whether or not it is relevant (in this case, it is not), and regardless of whether type `Int` is in class `D`. For another example, consider the standard `Eq` class, along with the instance

```
instance Eq (List t) if Eq t where ...
```

and the set of type constructors $\{\mathrm{Int}, \mathrm{List}\}$. In this case, we would expect to get an infinite, recursive set of ground axioms:

$$\begin{aligned}
&(\mathrm{Eq}\ (\mathrm{List}\,\mathrm{Int}) \Leftarrow (\mathrm{Eq}\ \mathrm{Int}))\ ;\ \varepsilon\\
&(\mathrm{Eq}\ (\mathrm{List}\,(\mathrm{List}\,\mathrm{Int})) \Leftarrow (\mathrm{Eq}\ (\mathrm{List}\,\mathrm{Int})))\ ;\ \varepsilon\\
&\vdots
\end{aligned}$$

The specialization process needs to take contradicting predicates into account. For example, consider a class `C` with instance chain

```
instance C Int fails
else C t
```

When specializing the corresponding schematic axiom to the predicate C Int, we must still include the first clause, as it contradicts the predicate we desire to prove; thus, the specialization to C Int would be

$$(\mathrm{C}\ \mathrm{Int}\ \texttt{fails} \Leftarrow ())\ ;\ (\mathrm{C}\ \mathrm{Int} \Leftarrow ())\ ;\ \varepsilon.$$

The same is true of functional dependencies; for example, given a class

```
class F t u | t → u
```

consider the specialization of the instance chain

```
instance F [Char] Int
else F [t] t
```

When we specialize this instance for the predicate F [Char] Int, we expect the ground axiom:

$$(\mathrm{F}\ [\mathrm{Char}]\,\mathrm{Int} \Leftarrow ())\ ;\ \varepsilon.$$

However, we must be careful when specializing the instance for the predicate F [Char] Char: a naive approach might observe that the first clause does not match the target types, and thus construct the specialized axiom:

$$(\mathrm{F}\ [\mathrm{Char}]\,\mathrm{Char} \Leftarrow ())\ ;\ \varepsilon$$

based on the second clause. Clearly, we cannot model both of these specializations without violating the functional dependency on `F`. Thus, correct specialization depends upon a notion of conflict between predicates. We say that π and π' conflict if either one is the negation of the other, or if they make differing assertions about a parameter determined under a functional dependency. Formally, we define

$$X \vdash \pi \between \pi' \iff \overline{\pi} = \pi' \vee (\exists (Y \rightsquigarrow Z \in \mathrm{fd}_X(\pi)).\, \pi = \pi' \bmod Z \wedge \pi \neq \pi')$$

for some set of class constraints X and any two predicates π and π'.

We are now prepared to define the *restriction* of an axiom α to a predicate π given a set of class constraints X, written $\alpha|_{\pi,X}$. Note that not all quantified variables in a schematic clause necessarily appear in the conclusion of the clause—for example, there may be variables determined by the functional dependencies on the hypotheses—and so the restriction of a single schematic axiom α will, in general, be a set of ground instances γ.

$$\varepsilon|_{\pi,X} = \{\varepsilon\}$$

$$((d : \forall \vec{t}.\, \pi \Leftarrow P)\ ;\ \alpha)|_{\pi',X} = \begin{cases} \{(d : S\,\pi \Leftarrow S\,P)\ ;\ \gamma \mid \\ \quad S \in \mathit{GSubst}(\vec{t}), \\ \quad (S\,\pi = \pi' \vee X \vdash S\,\pi \between \pi', \\ \quad \gamma \in \alpha|_{\pi',X}\} \\ \quad \text{if there is at least one such } S; \\ \alpha|_{\pi,X} \\ \quad \text{otherwise.} \end{cases}$$

Finally, we can extend the forcing relation to ground and schematic axioms. The empty axiom is trivially forced; for any $G \in \mathcal{G}$:

$$G \models \varepsilon.$$

A ground axiom $(\pi \Leftarrow P)\ ;\ \gamma$ corresponds to two conjuncts: for $G \in \mathcal{G}$, if G forces P, then it must also force π; similarly, if it forces $\overline{P}$, then it must force

γ. To formalize this, we begin by abstracting the construction of evidence, taking account of negative predicates:

$$app_\pi(d, e) = \begin{cases} d\ e & \text{if } \pi \text{ is positive;} \\ \bullet & \text{if } \pi \text{ is negative.} \end{cases}$$

Now, we can extend the forcing relation to ground axioms: for $G \in \mathcal{G}$,

$$\begin{aligned} G \models (d : \pi \Leftarrow P)\ ;\ \gamma \iff & (G \models e : P \implies G \models app_\pi(d, e) : \pi) \\ & \wedge (G \models \overline{P} \implies G \models \gamma). \end{aligned}$$

Finally, we can extend the forcing relation to schematic axioms. If we define *Preds*(C) to be the predicates on class C:

$$Preds(C) = \{C\ \vec{\tau}\ f \mid \vec{\tau} \in Type^{\text{arity}(C)}, f \in Flags\},$$

then we can define that, for $G \in \mathcal{G}$ and class constraints X:

$$G, X \models \alpha \iff \forall \pi \in Preds(\text{class}(\alpha)).\forall \gamma \in \alpha|_{\pi,X}.\ G \models \gamma.$$

4.3.3 Class Constraints

This section describes the modelling of the three forms of class constraint present in the Habit class system: kind signatures (which restrict the types that can belong to classes), functional dependencies (which restrict the relations expressed by classes), and superclasses (which restrict the relationships between classes). Unlike the notions in the prior sections, each of these applies to a class as a whole instead of to particular tuples within the class.

Kinds

The simplest class constraints are kind signatures. A kind constraint $C : \vec{\kappa}$, where the length of kind tuple $\vec{\kappa}$ is the arity of class C, specifies that the i^{th} parameter of class C—that is, the i^{th} element in each type tuple in the domain of the semantics of C—be of kind κ_i. We can extend the forcing relation to kind constraints as follows: for $G \in \mathcal{G}$,

$$G \models C : \vec{\kappa} \iff \forall \vec{\tau} \in G(C).\, \forall i. \tau_i \in \mathit{GType}^{\kappa_i}.$$

This illustrates a point of flexibility in the way we have defined the semantics of classes. An alternative approach would have been to express kinds directly in the structure of class semantics, as we did with arities. At the other extreme, we could have modelled classes by mappings of arbitrary sequences of types to evidence, and enforced class arities as a class constraint. We do not believe that any of these approaches is more expressive than the others, but we hope that the approach we have taken captures the intuition of type classes with a minimum of notation.

Functional Dependencies

Functional dependencies were originally proposed for class systems as a mechanism to eliminate ambiguities in multiparameter type classes by inducing improving substitutions [32]. However, these substitutions are only valid because of properties of the underlying relations. This section formalizes the restrictions that functional dependency constraints impose on the models of classes.

Intuitively, functional dependencies require that classes behave as type-level partial functions. For example, the `Elems` class (§2.4.2)

```
class Elems c e | c → e where ...
```

has a functional dependency stating that parameter `c` determines parameter `e`. Equivalently, we can say that given two predicates `Elems c e` and `Elems c' e'`, where `c`,`c'`,`e`,`e'` are arbitrary types, if `c` = `c'`, then we must have that `e` = `e'`. This is precisely the typical notion of a function, and corresponds to the original definition of functional dependencies in database theory [40].

Formally, we capture functional dependency constraints using sets of parameter indices. For example, the `Elems` declaration would give rise to the constraint

$$\mathit{Elems} : \{0\} \rightsquigarrow \{1\}.$$

A single class declaration may give rise zero, one, or many functional dependency constraints; for example, the following declaration

```
class (==) t u | t → u, u → t
```

would give rise to the two constraints $(==) : \{0\} \rightsquigarrow \{1\}$ and $(==) : \{1\} \rightsquigarrow \{0\}$. The `Eq` class, on the other hand, gives rise to no functional dependency constraints. The forcing relation for functional dependency constraints is a straightforward generalization of the intuitive notion. Writing $\vec{\tau}|_Y$ for those elements of $\vec{\tau}$ indexed by the elements of Y, we have that for $G \in \mathcal{G}$

$$\begin{aligned} G \models (C : Y \rightsquigarrow Z) \iff & \forall \vec{\tau} \in \mathrm{dom}(G(C)). \\ & \forall G' \succeq G, \vec{\upsilon} \in \mathrm{dom}(G'(C)). (\vec{\tau}|_Y = \vec{\upsilon}|_Y \implies \vec{\tau}|_Z = \vec{\upsilon}|_Z). \end{aligned}$$

Another approach would be to express functional dependency constraints in terms of negative predicates by observing that, for each tuple in the class, all other tuples that differ only on determined parameters must be excluded from the class. That is:

$$\begin{aligned} G \models (C : Y \rightsquigarrow Z) \iff & \forall \vec{\tau} \in \mathrm{dom}(G(C)). \forall \vec{\upsilon} \in \mathit{Type}^{\mathrm{arity}(C)}. \\ & (\vec{\tau} = \vec{\upsilon} \bmod Z \land \vec{\tau} \neq \vec{\upsilon}) \implies G \models C\ \vec{\upsilon}\ \texttt{fails}. \end{aligned}$$

These definitions are equivalent; the first is better aligned with the intuitive understanding of functional dependencies, and the role that functional dependencies can play in type inference, while the second is better aligned to proving (or disproving) predicates based on a functional dependency.

Superclasses

Habit, like Haskell, allows class declarations to include superclass constraints; for example, the definition of the `Ord` class (for totally ordered types) requires that its members also be members of the `Eq` class:[1]

```
    class Ord t | Eq t where (<) :: t → t → Bool ...
```

We would capture the superclass of the `Ord` class using a class constraint of the form

$$\forall t.\, \text{Ord } t \Rightarrow \text{Eq } t.$$

Note that we can make certain assumptions on the form of superclass constraints because they arise from Habit-style class declarations. In particular, we can assume that, for any superclass constraint $\forall \vec{t}.\, \pi \Rightarrow \pi'$, both π and π' are positive.

Extending the forcing relation to superclasses is pleasingly straightforward; we define that, for $G \in \mathcal{G}$:

$$G \models (\forall \vec{t}.\, \pi \Rightarrow \pi') \iff \forall S \in \mathit{GSubst}(\vec{t}).\, (\exists e.\, G \models e : S\,\pi \implies \exists e'.\, G \models e' : S\,\pi').$$

4.3.4 Programs

We capture the classes and instances of a Habit program with a pair $A \mid X$, where A is a set of axioms and X is a set of class constraints (kind signatures, functional dependencies, or superclasses). We call such a pair a *type class basis* because it specifies the logical content of the class and instance declarations, but not the implementation of the methods. The forcing relation for bases is defined in terms of their components: for $G \in \mathcal{G}$,

$$G \models A \mid X \iff (\forall \alpha \in A.\, G, X \models \alpha) \wedge (\forall \chi \in X.\, G \models \chi).$$

[1]The corresponding Haskell syntax for superclasses `class Eq t => Ord t where ...` uses the implies arrow backwards: being a member of `Eq` does not imply that a type is a member of `Ord`, but being a member of `Ord` does imply that a type is a member of `Eq`.

We say that a tuple $\langle \mathcal{G}, \preceq, \models \rangle$ models the basis $A \mid X$ if $\mathcal{G}$ is a set of model structures, $\preceq$ and $\models$ are defined as over the prior subsections, and $\forall G \in \mathcal{G}.\ G \models A \mid X$. As the extension and forcing relation are constant, we will sometimes refer to a set $\mathcal{G}$, instead of the tuple $\langle \mathcal{G}, \preceq, \models \rangle$, as a model.

A given basis may have zero, one, or many models. A basis with contradictory or incoherent instance declarations, or with instance declarations that conflict with its class constraints, has no models. On the other hand, the forcing relation does not constrain those predicates not mentioned in the program. For example, a program may contain the declaration of the `Eq` class, but neither an instance asserting Eq Bool nor an instance asserting Eq Bool `fails`. The models of such a program could uniformly force `Eq Bool`, uniformly force `Eq Bool fails`, or the treatment of `Eq Bool` could vary in different nodes of the model. We say that a basis $A \mid X$ is *consistent* if it has at least one model.

4.4 ACCEPTABILITY AND MODEL EXISTENCE

The semantics of Habit predicates developed in this chapter, and the semantics of overloaded expressions that we will develop in the next, depend on programs having models. However, as discussed at the end of the last section, there are syntactically valid Habit programs that have no models. This section describes a condition, called *acceptability* that is sufficient to ensure that programs are consistent. Because it is intended to be verified as part of the compilation of Habit programs, acceptability must be a decidable, syntactic criterion; as a result, it is necessarily a conservative approximation of consistency. It is, however, relatively permissive: for example, it admits more programs than the criteria specified in the Haskell report or in some prior work on functional dependencies [63], such as the various Habit examples given in the previous chapter.

We begin this section by describing the acceptability criterion (§4.4.1), identifying a number of component criteria corresponding to the forcing relation for different type class constructs. We then provide a model existence theorem, proving that any program that meets the acceptability criterion has a non-trivial model (§4.4.2). Our proof approach is based on Fitting's proof of model existence for intuitionistic logic [9]; however, we have had to adapt his techniques to our setting, which more intuitively models type classes and assumes a syntactic, rather than semantic, notion of consistency.

4.4.1 Acceptability

As a consistent program has at least one model, we can use the constraints of the model structure and forcing relation to guide our definition of the acceptability criterion. We begin by informally describing several semantic properties of axioms that are necessary for consistency:

- The model structure is a function, guaranteeing a coherent semantics of overloaded expressions. Therefore, there must not be multiple axioms that can prove the same predicate.
- Neither a predicate and its inverse, nor predicates that violate the declared functional dependency constraints, can be modelled.
- The axioms must respect the declared kind constraints and superclass constraints.

To ensure these properties, we will introduce several syntactic notions, conservatively approximating the underlying semantic notions: *overlap*, corresponding to incoherence, *conflict*, corresponding to contradiction, and *preservation*, corresponding to superclasses. We will combine these to define our notion of acceptability, providing a conservative approximation of consistency.

Inter-axiom Overlap and Conflict

In Haskell 98, two instances are said to *overlap* if their conclusions unify. The same notion can be applied to Habit instance clauses. Formally, we define:

$$\begin{aligned}&\mathit{overlaps}(\forall\vec{t}.\,\pi \Leftarrow P, \forall\vec{u}.\,\pi' \Leftarrow Q) \iff \\ &\qquad\qquad \exists S \in \mathit{Subst}(\vec{t}), S' \in \mathit{Subst}(\vec{u}).\, S\pi = S'\pi'. \qquad (4.1)\end{aligned}$$

The notion of *conflict* is new to Habit. We say that two clauses conflict if they may provide proofs of inconsistent predicates; as with overlap, this is a necessarily conservative approximation. In Habit, conflict must take account of both negative predicates (we cannot model a program that asserts both a predicate and its negation) and functional dependencies (we cannot model a program that makes inconsistent assertions about the determined fields of a relation, as discussed in the previous section). We define conflicting axiom clauses by:

$$\begin{aligned}&\mathit{conflicts}_X(\forall\vec{t}.\,\pi \Leftarrow P, \forall\vec{u}.\,\pi' \Leftarrow Q) \iff \\ &\qquad\qquad \exists S \in \mathit{Subst}(\vec{t}), S' \in \mathit{Subst}(\vec{u}).\, X \vdash S\,\pi \lightning S'\,\pi'. \qquad (4.2)\end{aligned}$$

The definitions of conflict and overlap existentially quantify over substitutions; however, they can be implemented directly by unification [55] (following, if necessary, a renaming of quantified variables).

We extend each of these notions to axioms by considering their clauses pairwise:

$$\begin{aligned}\mathit{overlaps}(\alpha,\alpha') &\iff \exists\xi \in \alpha, \xi' \in \alpha'.\, \mathit{overlaps}(\xi,\xi') \\ \mathit{conflicts}_X(\alpha,\alpha') &\iff \exists\xi \in \alpha, \xi' \in \alpha'.\, \mathit{conflicts}_X(\xi,\xi').\end{aligned}$$

This may seem overly strict: indeed, it ignores the additional information at each clause that none of the prior clauses applied. At this time, however, we have found few motivating examples to justify the increased complexity required by more permissive notions of axiom overlap and consistency, and have thus chosen the simpler presentation. We hope to explore more permissive definitions of overlap and conflict as future work (§7.1).

Intra-axiom Overlap and Conflict

The conflict check just described determines when pairs of instances violate functional dependencies; however, it is also possible for different ground instantiations of a single instance clause to violate a functional dependency constraint. For example, the instance

```
instance Elems [t] u where ...
```

could be used to prove both `Elems [Int] Int` and `Elems [Int] Char`, violating the functional dependency constraint on the `Elems` class.

To eliminate such self-conflicting axiom schemes, it is sufficient to require that any type variables appearing in determined positions (under some functional dependency constraint) must appear in determining positions (under the same constraint). To formalize this notion, we make use of some existing theory of functional dependencies [33, 40]. Let $\mathit{ftv}(\pi)$ be all the free type variables of predicate π. The *induced functional dependencies*, F_π, of π are the dependencies:

$$F_\pi = \begin{cases} \{\mathit{ftv}(\vec{\tau}|_Y) \rightsquigarrow \mathit{ftv}(\vec{\tau}|_Z) \mid Y \rightsquigarrow Z \in \mathrm{fd}(\pi)\} & \text{if } \pi \text{ is positive} \\ \emptyset & \text{otherwise} \end{cases}$$

Note that, unlike the class constraints, these are functional dependencies over sets of type variables, not over index sets. Because functional dependencies constrain what is in a class, not what is not in it, a negative predicate induces no relationship among its arguments. By extension, for a context P, let F_P be the union of the induced functional dependencies for each predicate $\pi \in P$. The closure of a set J with respect to a set of functional dependencies F, written J_F^+ is intuitively the set of all elements determinable from J using the functional dependencies in F. Formally, we define J_F^+ as the smallest set such that:

1. $J \subseteq J_F^+$; and,

2. if $Y \rightsquigarrow Z \in F$ and $Y \subseteq J_F^+$, then $Z \subseteq J_F^+$.

We can now define self-conflicting axiom clauses as those in which any variable in a determined position does not appear in some determining position; formally:

$$\textit{self-conflicting}_X(\forall \vec{t}.\ C\ \vec{\tau} \Leftarrow P) \iff \exists (Y \rightsquigarrow Z) \in \mathrm{fd}_X(C).\ \textit{ftv}(\vec{\tau}|_Z) \nsubseteq (\textit{ftv}(\vec{\tau}|_Y))^+_{F_P} \quad (4.3)$$

This definition is only stated for clauses asserting positive predicates; axioms asserting negative predicates are trivially not self-conflicting, as functional dependencies do not constrain the types excluded from relations.

Previous work on type classes and functional dependencies has referred to a similar restriction on instances as the “covering” [33] or “coverage” [63] condition. Our condition is identical to Jones and Diatchki’s covering condition; we believe the name “self-conflict” better captures the motivation for the restriction. It is a relaxed version of the coverage condition of Sulzmann et al.; they do not close the set of determining variables over the functional dependency constraints of the hypotheses.

In a similar way, a single clause may provide distinct evidence values for the same predicate. Consider the following (admittedly pathological) instance, assuming arity-1 classes `C` and `D`:

```
instance C Int if D t
```

with the corresponding schematic axiom

$$(\forall t.\, \mathrm{C}\ \mathrm{Int} \Leftarrow \mathrm{D}\ t)\ ;\ \varepsilon.$$

Specializing this axiom scheme will give one concrete axiom for each type t, each providing distinct evidence expressions, and thus potentially distinct evidence, for C Int. We identify that self-overlap occurs in clauses in which the variables appearing in the hypotheses are not determined by the variables appearing in the conclusion:

$$\textit{self-overlapping}(\forall \vec{t}.\, \pi \Leftarrow P) \iff \textit{ftv}(P) \not\subseteq (\textit{ftv}(\pi))^+_{F_P} \tag{4.4}$$

A stricter version of this check (one that again does not close over the functional dependencies of P) is called the “bound variables” condition by Sulzmann et al.; again, we believe that our name captures the motivation, rather than the implementation, of the restriction.

As with inter-axiom conflict and overlap, these notions of self-conflict and self-overlap can be extended to axioms by considering the clauses individually. We do not need to consider conflict or overlap between clauses in a chain, because at most one clause in a chain can apply to any particular predicate.

$$\begin{aligned}\textit{self-conflicting}(\alpha) &\iff \exists \xi \in \alpha.\ \textit{self-conflicting}(\xi)\\ \textit{self-overlapping}(\alpha) &\iff \exists \xi \in \alpha.\ \textit{self-overlapping}(\xi)\end{aligned}$$

Well-Kindedness

We must ensure that the axioms in the program respect the kind constraints for the classes. This is a straightforward application of common kind-checking techniques from type inference [27]. We formalize the requirement as follows:

$$\begin{aligned}&\textit{well-kinded}(A \mid X) \iff\\ &\qquad \forall \alpha \in A, (\forall \vec{t}.\ C\ \vec{\tau}\ f \Leftarrow P) \in \alpha.\ (C : \vec{\kappa}) \in X \implies \tau_i \in \textit{Type}^{\kappa_i}\end{aligned} \tag{4.5}$$

Superclasses

Finally, we must ensure that each axiom respects the declared superclasses. This is intuitively straightforward—for a basis $A \mid X$ including a superclass constraint $\forall \vec{t}.\pi \Rightarrow \pi'$, we must show that any model of the basis that forces a ground instance of π also forces the corresponding ground instance of π'. It is somewhat less obvious how to syntactically check this (fundamentally semantic) criterion. To do so, we will rely on two approximations. The first is that, if some ground instance of π is forced, then it must be because it is the conclusion of an axiom clause; this allows us to approximate the conditions under which π is forced by the hypotheses of any axiom clause that could prove it. The second is that the entailment relation, developed in the next section (§4.5), provides a mechanical way to verify that ground instances of π' are forced, and the soundness of entailment (Theorem 4.12) will give the semantic condition we desire.

There is, however, a circularity in this approach: the soundness argument for entailment depends on the basis of the entailment being acceptable. Thus, it might seem that accepting a program depends on having already accepted the program. To avoid this difficulty, we will define a relation, which we call *preservation*, that holds if one new axiom preserves all the requirements of an existing, acceptable basis. Using this relation, we can validate a program iteratively, beginning with the (trivially acceptable) empty set of axioms.

We begin by defining preservation for an individual clause $d : \forall\vec{t}.\, \pi \Leftarrow P$ and superclass constraint $\forall\vec{u}.\, \pi'_0 \Rightarrow \pi'_1$: if the conclusion π of the clause matches the hypothesis π'_0 of the superclass constraint, then we require that the hypotheses of the clause P be sufficient to prove the conclusion of the requirement π'_1. Formally, we define:

$$\begin{aligned}
&\mathit{preserves}(d : \forall\vec{t}.\, \pi \Leftarrow P,\ \forall\vec{u}.\, \pi'_0 \Rightarrow \pi'_1,\ A \mid X) \iff \\
&\qquad\qquad \forall S \in \mathit{Subst}(\vec{t}), S' \in \mathit{Subst}(\vec{u}) \text{ such that } S\,\pi = S'\,\pi'_0. \\
&\qquad\qquad\qquad\qquad\qquad\qquad A \mid X \vdash S\,P \Vdash S'\,\pi'_1.
\end{aligned} \tag{4.6}$$

This is equivalent to the superclass check in the Haskell report [49, §4.3.2], extended to multi-parameter type classes. Note that we can find the most general such substitutions S and S' by unification, and that the closure of entailment under substitution (Theorem 4.11) guarantees that showing the most general case is sufficient to show all cases. We can extend the notion of preservation to axioms and to sets of requirements in the obvious fashion: writing X_{SC} for the superclass constraints in X, we define that

$$\mathit{preserves}(\alpha, A \mid X) \iff \forall \xi \in \alpha.\, \forall \chi \in X_{SC}.\, \mathit{preserves}(\xi, \chi, A \mid X).$$

Finally, we can say that a basis $A \mid X$ satisfies its superclasses if there is some ordering of the axioms $\alpha_1, \alpha_2, \ldots$, where $A = \{\alpha_i\}$, such that each α_i preserves the requirements given the preceding axioms. Defining $A_j = \{\alpha_i \mid i < j\}$, we have that:

$$\mathit{satisfies\text{-}superclasses}(A \mid X) \iff \forall i. \mathit{preserves}(\alpha_i, A_i \mid X). \tag{4.7}$$

In practice, we can find such an ordering by topologically sorting the axioms based on their conclusions and on the declared axioms. There are consistent bases that are not accepted by this definition. For example, the following two superclass constraints assert that classes `C` and `D` are equivalent to each other:

$$\forall t.\, \mathrm{C}\ t \Rightarrow \mathrm{D}\ t,$$
$$\forall u.\, \mathrm{D}\ t \Rightarrow \mathrm{C}\ t.$$

Any program in which `C` and `D` have the same instances is consistent with these superclass constraints; however, assuming `C` and `D` are not both empty, there is no ordering of instances that preserves the constraints, as either a `C` instance will be ordered before the corresponding `D` instance, or vice versa. However, we believe these kinds of circular superclasses will be rare in practice (they have never been permitted in Haskell), and so we have chosen a simpler definition of superclass satisfaction that treats axioms one at a time. We believe that further exploration of superclasses, and of the superclass validation mechanism in particular, are valuable directions for future study (§7.2).

Accepting Programs

We are now able to characterize acceptable programs. We say that a program is acceptable if its axioms are non-overlapping and non-conflicting, and if there is an ordering of the axioms such that each axiom preserves the requirements. Formally:

$$\begin{aligned}\mathit{acceptable}(A \mid X) \iff{} & \mathit{satisfies\text{-}superclasses}(A \mid X) \;\wedge\; \mathit{well\text{-}kinded}(A \mid X) \wedge \\ & \forall \alpha \in A.\, (\neg \mathit{self\text{-}overlapping}(\alpha) \;\wedge\; \neg \mathit{self\text{-}conflicting}(\alpha) \wedge \\ & \quad \forall \alpha' \in A.\, (\alpha = \alpha' \vee (\neg \mathit{overlapping}(\alpha, \alpha') \wedge \neg \mathit{conflicting}_X(\alpha, \alpha')))).\end{aligned}$$

4.4.2 Model Existence

To justify the acceptability criterion described in the prior section, we will now prove that any acceptable program can be modelled. Such model existence proofs are well known for intuitionistic logic [9]. However, our setting presents several complications. In particular, because our model structure is based on the intuitive semantics of type classes, it does not explicitly include either negative predicates (as they provide no implementations) or non-atomic formulae.

Theorem 4.1 (Model existence)**.** *Any acceptable program has a non-trivial model. Formally,*

$$\mathit{acceptable}(A \mid X) \implies \exists \mathcal{G}.\, (\mathcal{G} \neq \emptyset) \wedge (\forall G \in \mathcal{G}. G \models A \mid X).$$

The rest of the section is devoted to the proof. We begin by establishing some terminology for model structures, and some lemmas about model structures and extension. This will allow us to define a procedure for constructing a model from a basis, completing the proof.

Definition 4.2. We begin by introducing some terminology to simplify the remainder of the proof. Given a model structure G and a program $A \mid X$, we say that a pair $\langle C\ \vec{\tau}, e\rangle$ is *consistent with* G if $G(C)(\vec{\tau}) = e$. We say that a pair

$\langle C\ \vec{\tau}\ \texttt{fails}, \bullet\rangle$ is consistent with G if $C\ \vec{\tau}$ is inconsistent with G, a term we shall define shortly. We say that a pair $\langle \pi, e\rangle$ is *proved from* G if there is some $\alpha \in A$ and some index i such that, letting $\alpha' = \alpha|_{\pi,X}$:

- For each clause $(d_j : \forall \vec{t_j}.\ \pi_j \Leftarrow P_j) \in \alpha'$ with index $j < i$, there is some predicate $\pi' \in P_j$ such that π' is inconsistent with G; and,
- In clause $(d : \forall \vec{t}.\ \pi \Leftarrow P) \in \alpha'$ with index i, there is some e_k for each predicate P_k such that $\langle P_k, e_k\rangle$ is consistent with G; and,
- If π is positive, then $e = d\ \langle e_k\rangle$; otherwise, $e = \bullet$.

We say that a predicate π is *inconsistent with* G if there is some $\langle \pi', e\rangle$ consistent with, or proved from, G such that $X \vdash \pi \between \pi'$. Finally, we say that axiom α is *relevant to* π if $\alpha|_{\pi,X} \neq \varepsilon$, and that a set of axioms A is relevant to π if some axiom in A is relevant to π.

If G is a model structure, we write $G[C \mapsto x]$ for the model structure G' such that $G'(C) = x$ and $G'(C') = G(C')$ for all $C' \neq C$. We define the pointwise union of model structures G, G' by

$$(G \Cup G')(C) = \begin{cases} G(C) \cup G'(C) & \text{if } C \in \mathrm{dom}(G) \text{ and } C \in \mathrm{dom}(G') \\ G(C) & \text{if } C \in \mathrm{dom}(G) \text{ but } C \notin \mathrm{dom}(G') \\ G(C') & \text{if } C \in \mathrm{dom}(G') \text{ but } C \notin \mathrm{dom}(G). \end{cases}$$

Lemma 4.3. *For $G \preceq G'$, any $\langle \pi, e\rangle$ provable from G is provable from G', and any π inconsistent with G is inconsistent with G'.*

Proof. The first half is immediate from the definition of extension (§4.3); the second follows from the first and the definition of inconsistency (Definition 4.2). □

Lemma 4.4. *Assuming acceptable$(A \mid X)$, and some model structure G, there are no π and distinct evidence values e_1, e_2 such that both $\langle \pi, e_1\rangle$ and $\langle \pi, e_2\rangle$ are proved from G.*

Proof. We proceed by contradiction. If π were negative, $\langle \pi, \bullet \rangle$ would be the only tuple provable about π; as we assume distinct evidence values, we have that π is positive and both $e_1 \neq \bullet$ and $e_2 \neq \bullet$. As each is evidence for a single predicate, we know that neither is a tuple of evidence values. Thus, we must have that $e_1 = d_1\, e_1'$ and $e_2 = d_2\, e_2'$. We distinguish two possibilities: either (1) $d_1 \neq d_2$, identifying distinct clauses that prove π, or (2) $d_1 = d_2$, identifying a single clause $d_1 : \forall \vec{t}. \pi' \Leftarrow P$ proving π, but there are distinct values $e_1 \neq e_2$ proving the hypotheses P. However, in case (1), we would have distinct clauses $d_1 : \forall \vec{t_1}.\, \pi_1 \Leftarrow P_1$ and $d_2 : \forall \vec{t_2}.\, \pi_2 \Leftarrow P_2$ in *Clauses*(A) that could prove π, and thus we would know that, for some $S_1 \in \mathit{Subst}(\vec{t_1}), S_2 \in \mathit{Subst}(\vec{t_2})$, $S_1\, \pi_1 = S_2\, \pi_2 = \pi$, contradicting the inter-axiom overlap check (Equation 4.1) of the acceptability of $A \mid X$. Alternatively, in case (2), we must either have distinct specializations $d_1 : \pi \Leftarrow P_1$, $d_1 : \pi \Leftarrow P_2$ with $P_1 \neq P_2$, or, for specialization $d : \pi \Leftarrow P$, have some element P_i of P with distinct e, e' such that both $\langle P_i, e \rangle$ and $\langle P_i, e' \rangle$ are consistent with G. The first contradicts the intra-axiom overlap check (Equation 4.4) of *acceptable*$(A \mid X)$; the second contradicts that G is a model structure, as the mapping from types to evidence would not be a function. □

Lemma 4.5. *Assuming acceptable*$(A \mid X)$ *and some model structure* G*, there are no* π, π' *and corresponding* e, e' *such that* $X \vdash \pi \lightning \pi'$ *and both* $\langle \pi, e \rangle$ *and* $\langle \pi', e' \rangle$ *are proved from* G*.*

Proof. By contradiction: from $X \vdash \pi \lightning \pi'$, we can conclude that either $\overline{\pi} = \pi'$ or there is some $Y \rightsquigarrow Z$ in $\mathrm{fd}_X(\pi)$ such that $\pi = \pi' \bmod Z, \pi \neq \pi'$. In the first case, we must have distinct clauses $d_1 : \forall \vec{t_1}. \pi_1 \Leftarrow P_1$ and $d_2 : \forall \vec{t_2}. \pi_2 \Leftarrow P_2$ in *Clauses*(A) and substitutions $S_1 \in \mathit{Subst}(\vec{t_1}), S_2 \in \mathit{Subst}(\vec{t_2})$ such that $\overline{\pi} = \overline{S_1\, \pi_1} = S_2\, \pi_2 = \pi'$. However, this contradicts the conflict check (Equation 4.2) of *acceptable*$(A \mid X)$. In the second case, the clauses need not be distinct, but the argument is otherwise identical. □

Definition 4.6. Fix a acceptable program $A \mid X$, and let G_0 be some model structure. We define a sequence of model structures G_n as follows. Enumerate the positive ground predicates to which A is relevant as $\pi_0, \pi_1, \dots,$ where each $\pi_i = C_i\ \vec{\tau}_i$. Given some structure G_n, we define model structures G^i_{n+1}, which extend G_n with $\langle \pi_i, e\rangle$ if $\langle \pi_i, e\rangle$ can be proven from G_n:

$$G^i_{n+1} = \begin{cases} G_n[C_i \mapsto (G_n(C) \cup \{\langle \vec{\tau}_i, e\rangle\})] & \text{if } \langle \pi_i, e\rangle \text{ is proven from } G_n. \\ G_n & \text{otherwise.} \end{cases}$$

Define $G_{n+1} = \biguplus_{i<\omega} G^i_{n+1}$. We say that G_0 is a *valid initial structure* for $A \mid X$ if:

1. For each G_n constructed from G_0, and each pair $\langle \pi, e\rangle$ consistent with G_0, there is neither (a) some evidence e' distinct from e such that $\langle \pi, e'\rangle$ is proven from G_n, nor (b) some predicate π' and evidence e' such that $X \vdash \pi \lightning \pi'$ and $\langle \pi', e'\rangle$ is proven from G_n.

2. For each kind constraint $(C : \kappa_i) \in X$ and each pair $\langle C\ \vec{\tau}, e\rangle$ consistent with $\mathcal{G}_0$, $\tau_i \in \mathit{Type}^{\kappa_i}$.

3. For each functional dependency constraint $(C : Y \rightsquigarrow Z) \in X$ and all pairs $\langle C\ \vec{\tau}, e\rangle, \langle C\ \vec{\upsilon}, e'\rangle$ consistent with G_0, $\vec{\tau} \neq \vec{\upsilon} \bmod Z$.

4. For each superclass $(\forall \vec{t}.\ \pi \Rightarrow \pi') \in X$, if there is a $T \in \mathit{GSubst}(\vec{t})$ such that $\langle T\,\pi, e\rangle$ is consistent with G_0, then there is some evidence e' such that $\langle T\,\pi', e'\rangle$ is consistent with G_0.

Observe that the empty structure is trivially a valid initial structure; that, if G_0 is a valid initial structure for $A \mid X$, then from Condition (1) and Lemma 4.4, each G_n is a well-formed model structure; and that the G_n are monotonically increasing. Define $I(G_0)$ as the limit of the sequence $G, G_1, \dots$ defined from G, and define the *induced model* for $A \mid X$ as the set $\{I(G_0) \mid G_0 \text{ is a valid initial structure for } A \mid X\}$.

Lemma 4.7. *Let $A \mid X$ be a acceptable program, and let $\mathcal{G}$ be the induced model of $A \mid X$. For all $G \in \mathcal{G}$, $G \models A \mid X$.*

Proof. Fix some $G \in \mathcal{G}$; we show that $G \models A \mid X$. As $G = I(G_0)$ for some valid initial structure G_0, we know that G is the limit of some sequence of G_n. We consider each case of the forcing rule for type class bases: G must model each of the axioms, and each of the three varieties of class constraints.

- Fix some $\alpha \in A$. We must show that for all $\pi \in \mathit{Preds}(\text{class}(\alpha))$, $G \models \alpha|_{\pi,X}$. Fix some such π, and let α' be $\alpha|_{\pi,X}$. If $\alpha' = \varepsilon$, then $G \models \alpha'$ by definition. Alternatively, let $\alpha' = \xi_0; \xi_1; ...; \xi_n; \varepsilon$ where each $\xi_i = (d_i : \pi_i \Leftarrow P_i)$. Suppose that for some i, $G \models \overline{P_j}$ for $j < i$, and there is an evidence expression e such that $G \models e : P_i$. Then, for some G_n, we have that: first, for each P_j, some predicate $\pi \in P_j$ is inconsistent with G_n; and, second, that $\langle (P_i)_k, e_k \rangle$ is consistent with G_n for each element $(P_i)_k$ of P_i. Thus, from Definition 4.2, $\langle \pi_i, e_i \rangle$ is provable from G_n, where $e_i = d\ e$ if π_i is positive, and $\bullet$ otherwise. We treat the cases for positive and negative π_i separately.
  - If $\pi_i = C\ \vec{\tau}$, then, we have that $G_{n+1}(C)(\vec{\tau}) = d\ e$; that $G(C)(\vec{\tau}) = d\ e$; that $G \models d\ e : C\ \vec{\tau}$; and thus that $G \models \alpha'$.
  - Alternatively, suppose that $\pi_i = C\ \vec{\tau}$ `fails`. Then, we have that (by Lemma 4.3) for all G' such that $G_n \preceq G'$, π_i is inconsistent with G'; that (by Lemma 4.5) there is no $\langle \pi_i, e \rangle$ consistent with any such G'; that $\vec{\tau} \notin \text{dom}(G')(C)$, so $G \models \bullet : \pi_i$; and, therefore, that $G \models \alpha'$.
- For each kind constraint $(C : \kappa_i) \in X$: if $\vec{\tau} \in \text{dom}(G(C))$, then $\vec{\tau} \in \text{dom}(G_n(C))$, and so $C\ \vec{\tau}$ is either consistent with the initial structure G_0, consistent with the kind constraint by Condition (2) on the valid initial structures, or is the conclusion of some clause in $\mathit{Clauses}(A)$, consistent with the kind constraints by the well-kindedness check (Equation 4.5) of the acceptability of $A \mid X$.

- Fix some functional dependency constraint $(C : Y \rightsquigarrow Z) \in X$ and $\vec{\tau}, \vec{\upsilon} \in \operatorname{dom}(G(C))$. Each type tuple is in the model either because it is proved from some G_n, or because it is in the initial model G_0. We have three cases:
  - If both tuples are consistent with G_0, then Condition (3) on the valid initial structures ensures they do not violate the functional dependency constraint.
  - If one tuple is consistent with G_0 and the other is proven from some G_n, then condition (1) on the initial structure ensures that $X \nVdash \pi \between \pi'$, and so π and π' cannot violate the functional dependency constraints.
  - Finally, if both are proven from some G_n, then by Lemma 4.5 they do not violate the functional dependency constraint.
- Observe that for arbitrary model structure G and axioms A, because

$$G \models A \iff (\forall \alpha \in A.\ G \models \alpha),$$

we have that if $A' \subseteq A$ then $G \models A \implies G \models A'$. Fix some superclass constraint $(\forall \vec{t}.\ \pi \Rightarrow \pi') \in X$, some $S \in \mathit{GSubst}(\vec{t})$, and some evidence value e such that $\langle S\,\pi, e\rangle$ is consistent with G. By definition, $\langle S\,\pi, e\rangle$ is consistent with G either because it is consistent with the initial structure G_0 or because it is proven from some G_n. If $\langle S\,\pi, e\rangle$ is consistent with G_0, then Condition (4) on valid initial structures ensures that there is some e' such that $\langle S\,\pi', e'\rangle$ is consistent with G_0, and thus with G. Alternatively, suppose that $\langle S\,\pi, e\rangle$ is proven from G_n using some clause ξ. From the superclass satisfaction check (Equation 4.7) of the acceptability of $A \mid X$, we have that there is some ordering $\alpha_1, \alpha_2, \ldots, \alpha_i$ of a subset of the axioms in A such that ξ is a clause of α_i; and, writing A' for $\{\alpha_j \mid j < i\}$,

$$\mathit{preserves}(\xi_i, \forall \vec{t}.\ \pi \Rightarrow \pi', A' \mid X).$$

From the definition of the *preserves* predicate (Equation 4.6) and the soundness of the entailment relation (Theorem 4.12), we have that any if $G \models A' \mid X$ and $G \models S\,P$ then $G \models S\,\pi$. we have already shown that $G \models A$, and so, by the observation, $G \models A'$; as we have assumed that $G \models e : S\,\pi$, we can conclude that there is some e' such that $G \models e' : S\,\pi'$, and so $G \models (\forall \vec{t}.\, \pi \Rightarrow \pi')$. □

The proof of Theorem 4.1 is immediate from Lemma 4.7 and from the existence of at least one valid initial structure. This theorem can be seen as a soundness result for the acceptability predicate. The converse, that any program with a model is acceptable, is not true: for example, as mentioned in the discussion of superclasses (§4.4.1), programs with circular requirements can be consistent, but are not accepted by our definition of acceptability.

4.5 ENTAILMENT AND QUALIFIED TYPES

We next present proof rules for an entailment relation, by which the logic of type class predicates (described over the prior sections) is integrated into the Habit type system. Our proof rules are designed to allow efficient proof search during type inference; thus, they are syntactic in nature, but incomplete with respect to the semantics developed in the prior section. We will show, however, that our entailment relation is sound with respect to our semantics, and that it meets the criteria Jones establishes in his system of qualified types [28].

4.5.1 Entailment

The primary judgment of our proof system is the entailment relation

$$A \mid X \vdash P \Vdash Q$$

asserting that any model of $A \mid X$ that forces ground instances of the predicates in P must force the corresponding ground instances of the predicates in Q. When

$$[\textsc{Each}]\ \frac{\forall i.\ (A \mid X \vdash P \Vdash Q_i)}{A \mid X \vdash P \Vdash Q}$$

$$[\textsc{Assume}]\ \frac{\pi \in P}{A \mid X \vdash P \Vdash \pi}$$

$$[\textsc{Super}]\ \frac{(\forall \vec{t}.\ \pi' \Rightarrow \pi) \in X \quad S \in \mathit{Subst}(\vec{t}) \quad A \mid X \vdash P \Vdash S\,\pi'}{A \mid X \vdash P \Vdash S\,\pi}$$

$$[\textsc{Axiom}]\ \frac{\alpha \in A \quad A \mid X, \alpha \vdash P \Vdash \pi}{A \mid X \vdash P \Vdash \pi}$$

Figure 4.2: Top-level deduction rules for predicate entailment.

Q is a singleton set $\{\pi\}$, we will write $A \mid X \vdash P \Vdash \pi$. We present natural deduction rules for this judgment in Figure 4.2. Rule Each allows the proof of a conjunction of predicates by proving each conjunct individually. The remaining rules each provide for proving an individual predicate: either because it is one of the assumptions (Rule Assume), because it is a consequence of one of the superclasses (Rule Super), or a consequence of one of the axioms (Rule Axiom). We introduce a new judgment, $A \mid X, \alpha \vdash P \Vdash \pi$ to capture that predicate π is a consequence of axiom α. Deduction rules for this judgment are given in Figure 4.3, and discussed in the following paragraphs. Note that, to simplify the notation, we avoid repeating the basis $A \mid X$ in each of the axiom judgments.

The first two rules prove the goal predicate from the clause at the head of the axiom. Rule Match is natural: if the goal π matches the conclusion of a clause π', and the assumptions P are sufficient to prove the hypotheses P' of the clause, then the clause proves the goal. Rule Excl-FD gives an alternative way a clause might prove a goal, corresponding to the second interpretation of functional dependency constraints (§4.3.3). While a functional dependency constraint does not of itself include or exclude any particular tuples from a class, each tuple in a class with such

$$[\textsc{Match}]\ \frac{S \in \mathit{Subst}(\vec{t}) \quad S\,\pi' = \pi \quad A \mid X \vdash P \Vdash S\,P'}{((d : \forall \vec{t}.\, \pi' \Leftarrow P')\,;\,\alpha) \vdash P \Vdash \pi}$$

$$[\textsc{Excl-FD}]\ \frac{\begin{array}{c}(Y \rightsquigarrow Z) \in \mathrm{fd}_X(\pi) \quad S \in \mathit{Subst}(\vec{t}) \quad S\,\pi' = \overline{\pi} \bmod Z \\ S\,\pi' \neq \overline{\pi} \quad A \mid X \vdash P \Vdash P' \quad \pi \text{ is negative}\end{array}}{((d : \forall \vec{t}.\, \pi' \Leftarrow P')\,;\,\alpha) \vdash P \Vdash \pi}$$

$$[\textsc{Step-Contra}]\ \frac{\begin{array}{c}(Y \rightsquigarrow Z) \in \mathrm{fd}_X(\pi) \quad S \in \mathit{Subst}(\vec{t}) \quad S\,\pi' = \pi \bmod Z \\ A \mid X \vdash P \Vdash \overline{S\,P'_i} \quad \alpha \vdash P \Vdash \pi\end{array}}{((d : \forall \vec{t}.\, \pi' \Leftarrow P')\,;\,\alpha) \vdash P \Vdash \pi}$$

$$[\textsc{Step-Pos}]\ \frac{\begin{array}{c}\forall (Y \rightsquigarrow Z) \in \mathrm{fd}_X(\pi).(\pi' \not\sim \pi \bmod Z \ \wedge\ \pi' \not\sim \overline{\pi} \bmod Z) \\ \alpha \vdash P \Vdash \pi \quad \pi' \text{ is positive}\end{array}}{((d : \forall \vec{t}.\, \pi' \Leftarrow P')\,;\,\alpha) \vdash P \Vdash \pi}$$

$$[\textsc{Step-Neg}]\ \frac{\pi' \not\sim \pi \quad \pi' \not\sim \overline{\pi} \quad \alpha \vdash P \Vdash \pi \quad \pi' \text{ is negative}}{((d : \forall \vec{t}.\, \pi' \Leftarrow P')\,;\,\alpha) \vdash P \Vdash \pi}$$

Figure 4.3: Axiom-specific deduction rules for predicate entailment. Basis $A \mid X$ is global.

a constraint excludes all tuples that would violate the constraint. For example, the instance

```
instance Elems [Int] Int
```

implicitly excludes predicates such as Elems [Int] Char, and thus provides a mechanism to prove its negation, Elems [Int] Char `fails`. This mechanism is captured by Excl-FD: if the goal π is negative, and the clause proves that the complement $\overline{\pi}$ of the goal is excluded from the class by a functional dependency, then it proves the goal.

The remaining three rules prove the goal from the tail of the axiom. In each case, we must first verify that the head will never prove the goal. Rule Step-Contra handles the case in which the goal matches the conclusion of the current clause up to some functional dependency: to verify that the clause does not apply, we must disprove one of its hypotheses. Rules Step-Pos and Step-Neg address the cases in which the goal does not match the conclusion of the current clause; the need for two cases arises from the treatment of functional dependencies. When trying to prove a positive predicate, such as `Elems [Int] Char`, it is not enough to observe that the predicate does not match an instance like the example instance for `Elems [Int] Int` above; we must ensure that it does not match modulo the functional dependencies. In contrast, as functional dependencies only restrict what is included in classes, not what is excluded, a similar check is not necessary to prove negative predicates.

4.5.2 Properties of Entailment

In his theory of qualified types, Jones treats the entailment relation among predicates abstractly, so long as certain assumptions are valid. In this section, we demonstrate that the proof rules for instance chains built in the previous section meet Jones's criteria, justifying their use in his type system.

The first property we show is that entailment of a set of predicates is equivalent to entailment of each predicate individually. Treating sets of predicates as conjunctions, this has the form of a distributive law: informally, we prove that $P \Vdash (\pi_1 \wedge \cdots \wedge \pi_n)$ if and only if $(P \Vdash \pi_1) \wedge \cdots \wedge (P \Vdash \pi_n)$.

Theorem 4.8 (Distributivity of entailment)**.** *For any basis $A \mid X$, $A \mid X \vdash P \Vdash Q$ if and only if $\forall i. A \mid X \vdash P \Vdash Q_i$.*

Proof. Necessity is direct, by construction with proof rule Each. Sufficiency follows from an inductive argument on the height of the derivation. We give an intuition for the argument; the cases are straightforward. Assume that Q contains more than one predicate, and observe that the rules that discharge predicates—Assume, Super, and Axiom—apply only to singleton sets of predicates. Thus, the derivation of $A \mid X \vdash P \Vdash Q$ must begin with an application of rule Each, with a subderivation of $A \mid X \vdash P \Vdash Q_i$ for $i < |Q|$. $\square$

Second, we show that a set of predicates entails any of its subsets.

Theorem 4.9 (Monotonicity of entailment). *For any basis $A \mid X$, $P \supseteq Q \implies A \mid X \vdash P \Vdash Q$.*

Proof. By construction: from $P \supseteq Q$, we know that each $Q_i \in P$, and thus that we can construct proofs $A \mid X \vdash P \Vdash Q_i$ by rule Assume. Next, we can apply rule Each to conclude that $A \mid X \vdash P \Vdash Q$. $\square$

Third, we show that the entailment relation is transitive.

Theorem 4.10 (Transitivity of entailment). *For any program $A \mid X$, if $A \mid X \vdash P \Vdash Q$ and $A \mid X \vdash Q \Vdash R$, then $A \mid X \vdash P \Vdash R$.*

Proof. We proceed by induction over the derivation of $A \mid X \vdash Q \Vdash R$. Cases Each, Super and Axiom follow immediately from the inductive hypothesis: in each case, the assumptions are irrelevant to the non-inductive hypotheses of the judgment. In case Assume, we must show that some $\pi \in Q$ follows from hypotheses P. This follows from Theorem 4.8 and the assumption that $A \mid X \vdash P \Vdash Q$. $\square$

Finally, we show that entailment respects polymorphism; that is, we show that anything we can prove about predicates containing type variables also holds for any instantiation of those type variables.

Theorem 4.11 (Closure under substitution). *For any program $A \mid X$, if $A \mid X \vdash P \Vdash Q$ and $T \in \mathit{Subst}(P, Q)$, then $A \mid X \vdash T\,P \Vdash T\,Q$.*

Proof. The proof is by induction on the derivation of $A \mid X \vdash P \Vdash Q$.

- Case Each follows from the inductive hypothesis.
- Case Assume is immediate: if $\pi \in P$, then $T\,\pi \in T\,P$.
- In case Super, given some superclass constraint $(\forall \vec{t}.\ \pi' \Rightarrow \pi) \in X$ and $S \in \mathit{Subst}(\vec{t})$, we have that, if T binds variables in P and $S\,\pi$, then $T \circ S \in \mathit{Subst}(\vec{t})$, and that by the inductive hypothesis $A \mid X \vdash T\,P \Vdash T\,(S\,\pi')$.
- In case Axiom, we proceed by induction on the derivation of $A \mid X, \alpha \vdash P \Vdash \pi$. To align notation with the rules in Figure 4.3, we will assume that each step in the derivation is of the form $\alpha' \vdash P \Vdash \pi$, where $\alpha' = ((d : \forall \vec{t}.\ \pi' \Leftarrow P') \,;\, \alpha)$.
  - Case Match. We are given some $S \in \mathit{Subst}(\vec{t})$ such that $S\,\pi' = \pi$, from which we have that $T \circ S \in \mathit{Subst}(\vec{t})$, and that $(T \circ S)\,\pi' = T\,\pi$. From the outer inductive hypothesis, we conclude that $A \mid X \vdash T\,P \Vdash (T \circ S)\,P'$, and thus that $A \mid X, \alpha' \vdash T\,P \Vdash T\,\pi$.
  - Case Excl-FD. The argument is identical to the prior case.
  - Case Step-Contra. As in the prior cases, we are given some $S \in \mathit{Subst}(\vec{t})$ such that $S\,\pi' = \pi \bmod Z$; we can conclude that $T \circ S \in \mathit{Subst}(\vec{t})$ and that $(T \circ S)\,\pi' = T\,\pi \bmod Z$. The outer inductive hypothesis allows us to conclude that $A \mid X \vdash T\,P \Vdash (T \circ S)\,\overline{P'_i}$, and the inner inductive hypothesis provides that $\alpha \vdash T\,P \Vdash T\,\pi$, showing that $\alpha' \vdash T\,P \Vdash T\,\pi$.
  - Case Step-Pos. We are given that, for all functional dependencies $(Y \rightsquigarrow Z) \in \mathrm{fd}_X(\pi)$, $\pi \not\sim \pi' \bmod Z$, that is, that there are no unifying substitutions U_0, U_1 such that $U_0\,\pi = U_1\,\pi' \bmod Z$. From this, we can conclude that there are no U'_0, U'_1 such that $U'_0\,(T\,\pi) = U'_1\,(T\,\pi') \bmod Z$, as if there were then we could construct the original unifying substitutions by $U_0 = U'_0 \circ T$ and $U_1 = U'_1 \circ T$. Thus, we have that $T\,\pi \not\sim$

$T\,\pi'$ mod Z. The inner inductive hypothesis gives that $\alpha \vdash T\,P \Vdash T\,\pi$, and so we can conclude that $\alpha' \vdash T\,P \Vdash T\,\pi$.

– Case Step-Next. The argument is identical to the prior case, but without reference to functional dependencies. $\square$

4.5.3 Soundness of Entailment

We now establish the soundness of the proof system. Intuitively, we wish to show that, for any derivation proving $A \mid X \vdash P \Vdash Q$, any model of the program $A \mid X$ and the hypotheses P must also model the conclusions Q; that is, any theorem of the proof system is a tautology of the logic. For arbitrary formulae ϕ, we will write $\mathcal{G} \models \phi$ to abbreviate $\forall G \in \mathcal{G}.\ G \models \phi$. We will write $\mathcal{G} \models \pi$ to abbreviate $\exists e.\, \mathcal{G} \models e : \pi$, and similarly write $\mathcal{G} \models P$ to abbreviate $\exists e.\, \mathcal{G} \models e : P$

Theorem 4.12 (Soundness of entailment). *If there is a derivation $A \mid X \vdash P \Vdash Q$, then for any model $\mathcal{G}$ and ground substitution $S \in GSubst(ftv(P, Q))$ such that $G \models S\,Q$ whenever $\mathcal{G} \models A \mid X$ and $\mathcal{G} \models S\,P$.*

Proof. Fix some $S \in GSubst(ftv(P, Q))$, and let $\hat{P} = S\,P$ and $\hat{Q} = S\,Q$. Fix a model $\mathcal{G}$ such that $\mathcal{G} \models A \mid X$ and $\mathcal{G} \models \hat{P}$. From Theorem 4.11, we have that $A \mid X \vdash \hat{P} \Vdash \hat{Q}$. We proceed by induction on this latter derivation.

- Case Each: from the definition of the forcing relation for contexts, we have that

$$\mathcal{G} \models \hat{Q} \iff \forall i.(\mathcal{G} \models \hat{Q}_i).$$

  From the inductive hypothesis, we can conclude that $\mathcal{G}$ forces each of the $\hat{Q}_i$; thus, we conclude that $\mathcal{G} \models \hat{Q}$.

- Case Assume: immediate.

- Case Super: we have that $\hat{Q}$ is some singleton $\{\pi\}$, and that there is some superclass $(\forall \vec{t}.\ \pi_0 \Rightarrow \pi_1) \in X$ and substitution $S \in GSubst(\vec{t})$ such that

$S\,\pi_1 = \pi$. Because $\mathcal{G} \models X$, the definition of the forcing relation for superclasses ensures that $\mathcal{G} \models S\,\pi_0 \implies \mathcal{G} \models S\,\pi_1$. As we assume $\mathcal{G} \models \hat{P}$, we can apply the inductive hypothesis to conclude that $\mathcal{G} \models S\,\pi_0$. Thus, we conclude that $\mathcal{G} \models S\,\pi_1$, that is, that $\mathcal{G} \models \pi$.

- Case Axiom: we have that $\hat{Q}$ is a singleton set $\{\pi\}$, and that there is a derivation of $A \mid X, \alpha \vdash \hat{P} \Vdash \pi$. We begin by showing that, for an arbitrary axiom α', if we have a derivation of $A \mid X, \alpha' \vdash \hat{P} \Vdash \pi$ and $\mathcal{G} \models \alpha|_{\pi,X}$, then $\mathcal{G} \models \pi$. We proceed by induction of the length of axiom $\alpha' = ((d : \forall \vec{t}.\, \pi' \Leftarrow P')\ ;\ \alpha'')$:

  - Case Match: we have that there is some $S \in \mathit{GSubst}(\vec{t})$ such that $S\,\pi' = \pi$, and so we know that

    $$\alpha'|_{\pi,X} = (d : \pi \Leftarrow S\,P')\ ;\ \alpha''|_{\pi,X}.$$

    From the assumption that $\mathcal{G} \models \alpha|_{\pi,X}$, we can conclude that if $\mathcal{G} \models S\,P$, then $\mathcal{G} \models \pi$. We can then apply the outer inductive hypothesis to conclude that $\mathcal{G} \models S\,P'$, and thus that $\mathcal{G} \models \pi$.

  - Case Excl-FD: by an identical argument to that in the previous case, we have that there is some S such that

    $$\alpha'|_{\pi,X} = (d : S\,\pi' \Leftarrow S\,P')\ ;\ \alpha''|_{\pi,X},$$

    and that $\mathcal{G} \models S\,\pi'$. However, as $(Y \rightsquigarrow Z) \in \mathrm{fd}_X(\pi)$, $\overline{\pi} = S\,\pi' \bmod Z$, but $\overline{\pi} \neq S\,\pi'$, we have that $X \vdash \overline{\pi} \lightning S\,\pi'$. Finally, as we know that $\pi = C\ \vec{\tau}$ `fails` for some class C and types $\vec{\tau}$, the definition of the forcing relation gives that, for any $G \in \mathcal{G}$,

    $$G \models \pi \iff \forall G' \succeq G.\, \vec{\tau} \notin \mathrm{dom}(G'(C))$$

    and so

    $$\mathcal{G} \models \pi \iff \forall G \in \mathcal{G}.\, \vec{\tau} \notin \mathrm{dom}(G(C)).$$

Because $X \vdash \overline{\pi} \lightning \pi'$ and $\mathcal{G} \models \pi'$, Lemma 4.5 gives that there is no $G \in \mathcal{G}$ and evidence e such that $\langle \overline{\pi}, e \rangle$ is consistent with G, and thus that $\mathcal{G} \models \pi$.

– Case Step-Contra: by an identical argument to that in the previous cases, we have that there is some S such that

$$\alpha'|_{\pi,X} = (d : \pi \Leftarrow S\,P')\ ;\ \alpha''|_{\pi,X}.$$

The outer inductive hypothesis allows us to conclude that $\mathcal{G} \models \overline{S\,P_i}$. Thus, from the assumption that $\mathcal{G} \models \alpha'|_{\pi,X}$ and the definition of the forcing relation for axioms, we have that $\mathcal{G} \models \alpha''|_{\pi,X}$. Finally, applying the inner inductive hypothesis, we have that $\mathcal{G} \models \pi$.

– Case Step-Pos: from $(\pi \not\sim \pi' \wedge \overline{\pi} \not\sim \pi')$ mod Z for all functional dependencies $Y \leadsto Z \in \mathrm{fd}_X(\pi)$, we have that there is no ground substitution $S \in \mathit{GSubst}(\vec{t})$ such that either $S\,\pi' = \pi$ or $X \vdash S\,\pi' \lightning \pi$. Thus, $\alpha'|_{\pi,X} = \alpha''|_{\pi,X}$, and we have that $\mathcal{G} \models \pi$ by the inductive hypothesis.

– Case Step-Neg: From $\pi \not\sim \pi'$ and $\overline{\pi} \not\sim \pi'$, we have that there is no $S \in \mathit{GSubst}(\vec{t})$ such that $S\,\pi' = \pi$. Because π is negative, we have $X \vdash S\,\pi' \lightning \pi$ only if $S\,\pi' = \overline{\pi}$; thus, we have that $\alpha'|_{\pi,X} = \alpha''|_{\pi,X}$, and $\mathcal{G} \models \pi$ follows from the inductive hypothesis.

Because we have that $A \mid X, \alpha \vdash \hat{P} \Vdash \pi$, and such a derivation must end with either a use of rule Match or rule Excl-FD, we have that there is some clause $(d : \forall \vec{t}.\, \pi' \Leftarrow P')$ of α and substitution S such that either $\pi = S\,\pi'$ or $X \vdash \pi \lightning S\,\pi'$. Thus, we can conclude that $\pi \in \mathit{Preds}(\alpha)$, and so from the assumption that $\mathcal{G} \models A \mid X$, we have that $\mathcal{G} \models \alpha|_{\pi,X}$, and, by the above, that $\mathcal{G} \models \pi$. □

The converse (completeness of entailment) is not true. For a simple example, consider a program containing the axioms

```
instance C Int else C Bool else C t fails
instance D Int else D Bool else D t fails
```

The domain of the only semantics of class C is the set $\{\mathrm{Int}, \mathrm{Bool}\}$, and similarly for class D, and thus any model of the program that models a ground instance of C t must model the corresponding ground instance of D t. However, none of the proof rules are sufficient to construct a derivation of C $t \Vdash$ D t.

4.6 RELATED WORK

Much of the previous work on the meaning of type classes has focussed on the meaning of overloaded expressions, rather than the meaning or logical foundations of the predicates themselves. We will discuss this work in the next chapter, in which we construct our own semantics for overloaded expressions.

The form of our models of logic is due to Kripke's models of intuitionistic and modal logics [36]. The model existence proof for intuitionistic logic was originally developed by Fitting [9]. Our proofs are less elegant than his because of our choice of model structure; nevertheless, we were strongly inspired by his approach.

The semantics of type classes that we present is part of the folklore of Haskell, but rarely written down. Harrison refers to such a semantics [18] in his discussion of polymorphic overloading, but does not discuss its logical implications. Jones and Diatcki present a notion of type classes corresponding to relations on types [33], but do not describe the role of evidence, or the relationship between their model and the predicate entailment relation.

Maier's textbook [40] summarizes the theory of functional dependencies as used in the database community, and defines functional dependencies for databases similarly to our extension of the forcing relation to functional dependency constraints. Jones [32] discusses the use of functional dependencies to improve type inference; his improvement rules are justified by, and similar in form to, both the intuitive notion and first formulation of the forcing rule. Sulzmann et al. [63] describe an alternative approach to the interaction of functional dependencies and type inference, via a translation into constraint-handling rules; unfortunately, their presentation conflates properties of their translation, such as termination, with properties of the relations themselves. For example, given a class

```
class F t u | t → u
```

The following two instances both violate GHC's coverage condition, and thus generate the same error message, and are both accepted given GHC's undecidable instances flag:

```
instance F t u ⇒ F (Maybe t) (Maybe u)
instance F Int t
```

However, these instances have dramatically different consequences for the type system. The first is rejected because GHC's coverage check does not take functional dependencies on instance hypotheses into account. Thus, while it may lead to non-termination of their translation, it does not compromise the consistency of the classes. The second instance, on the other hand, introduces arbitrary type equalities, as it provides evidence for the predicates `F Int t` and `F Int u` for arbitrary types `t` and `u`. This demonstrates the hazard of conflating properties of the translation, such as the potential non-termination given the first instances, with properties of the underlying relations, such as the inconsistency caused by the second.

5. SEMANTICS FOR OVERLOADING

The previous chapters have described new type class mechanisms from both intuitive and formal perspectives. This chapter completes the picture, developing the semantics of a simple typed lambda calculus with overloading. We begin by reviewing the type frame semantics for the simply-typed lambda calculus (§5.1). We then describe Ohori's [48] adaptation of type frame semantics to ML-style polymorphism (§5.2), and Harrison's [18] extension of Ohori's approach to polymorphic recursion. We rely on Harrison's extensions, even though polymorphic recursion itself is orthogonal to overloading. Finally, we combine Ohori's semantics for polymorphism with the models of classes developed in the last chapter to give a semantics for overloading (§5.3).

Each of these approaches is type-driven: we will give semantics to typings of terms, not to terms directly. Thus, each section will follow a similar approach: we will begin by giving the terms and types of the language under consideration, and then proceed to a semantic mapping on typings.

5.1 SIMPLY-TYPED LAMBDA CALCULUS

The simply-typed lambda calculus is a simple, higher-order programming language with notions of typed application and abstraction, and provides a foundation for much work in the mathematical semantics of programming languages [16]. In this section, we review *type frames*, one approach to giving semantics to the simply-typed lambda calculus, which serves as the basis for Ohori's semantics for ML-style polymorphism and our semantics for overloading.

Term variable $x \in \mathit{Var}$ Term constants k

Type constants K

$$
\begin{array}{llcl}
\text{Types} & \tau, \upsilon & ::= & K \mid \tau \to \tau \\
\text{Expressions} & M, N & ::= & x \mid k \mid \lambda x : \tau . M \mid M\, N
\end{array}
$$

Figure 5.1: Types and terms of the simply typed lambda calculus

$$
[\textsc{Var}]\ \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}
$$

$$
[\to \mathrm{I}]\ \frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash (\lambda x : \tau . M) : \tau \to \tau'} \qquad [\to \mathrm{E}]\ \frac{\Gamma \vdash M : \tau \to \tau' \quad \Gamma \vdash N : \tau}{\Gamma \vdash (M\, N) : \tau'}
$$

Figure 5.2: Typing rules of the simply typed lambda calculus

The types and terms of the simply-typed lambda calculus are given in Figure 5.1. The expressions are variables, constants, abstractions and applications; the types are constants and functions. We have augmented the system with term constants k and type constants K, and have omitted type constructors, and thus kinds. This simplifies our presentation, and thus allows us to focus on the type rules relevant to polymorphism and overloading; the extension of this system to include additional expression forms, such as conditionals, or more expressive types is straightforward.

The typing rules of the simply-typed lambda calculus are given in Figure 5.2. We assume a type environment Γ mapping variables and constructors to their types; by $\Gamma, x : \tau$ we denote the typing environment identical to Γ for all variables but x, which is now mapped to type τ.

A semantics for the simply-typed lambda calculus must then account for abstraction, application, and constants. An intuitive way to do this is via type frames [11], which are in turn built up from more primitive constructs called pre-frames. A *pre-frame* consists of:

- A function $\mathcal{T}^{\text{type}}[\![\cdot]\!]$ on types, such that $\mathcal{T}^{\text{type}}[\![\tau]\!]$ is a non-empty set providing the interpretation of type τ; and,
- A function $T_{\tau,\upsilon} : \mathcal{T}^{\text{type}}[\![\tau \to \upsilon]\!] \times \mathcal{T}^{\text{type}}[\![\tau]\!] \to \mathcal{T}^{\text{type}}[\![\upsilon]\!]$ providing the interpretation of the application of an element of the interpretation of $\tau \to \upsilon$ to the interpretation of an element of τ.

These components are subject to the following criterion, called the *extensionality property*:

- For any $f, g \in \mathcal{T}^{\text{type}}[\![\tau \to \upsilon]\!]$, if, for all $x \in \mathcal{T}^{\text{type}}[\![\tau]\!]$, $T_{\tau,\upsilon}(f, x) = T^{\tau,\upsilon}(g, x)$, then $f = g$.

A pre-frame $(\mathcal{T}^{\text{type}}, T)$ is extended to a *frame* by the addition of a mapping $\mathcal{T}^{\text{term}}$ providing the meanings of lambda calculus terms, or, more precisely, typings. Formally, we define a Γ-environment η as a mapping from variables to values such that $\eta(x) \in \mathcal{T}^{\text{type}}[\![\tau]\!]$ whenever $(x : \tau) \in \Gamma$. By $\eta[x \mapsto d]$, we mean the Γ-environment η' that agrees with η at all points but x, and such that $\eta'(x) = d$. We can then define the interpretation function $\mathcal{T}^{\text{term}}$ as a mapping from triples (Γ, M, τ) such that $\Gamma \vdash M : \tau$ and Γ-environments η to values of $\mathcal{T}^{\text{type}}[\![\tau]\!]$ such that the following conditions hold:

1. $\mathcal{T}^{\text{term}}[\![\Gamma, x, \tau]\!]\eta = \eta(x)$
2. $\mathcal{T}^{\text{term}}[\![\Gamma, M\,N, \upsilon]\!]\eta = T_{\tau,\upsilon}(\mathcal{T}^{\text{term}}[\![\Gamma, M, \tau \to \upsilon]\!]\eta, \mathcal{T}^{\text{term}}[\![\Gamma, N, \tau]\!]\eta)$
3. $T_{\tau,\upsilon}(\mathcal{T}^{\text{term}}[\![\Gamma, \lambda x : \tau.M, \tau \to \upsilon]\!]\eta, d) = \mathcal{T}^{\text{term}}[\![(\Gamma, x : \tau), M, \upsilon]\!](\eta[x \mapsto d])$

To account for constants in a given frame $(\mathcal{T}^{\text{type}}[\![\cdot]\!], \mathcal{T}^{\text{term}}[\![\cdot]\!], T)$, we additionally define mappings C^{type}, C^{term} such that for all constants k, $\Gamma \vdash k : C^{\text{type}}(k)$ and

Term variable	x	Term constants	k
Type variables	t	Type constants	K

$$
\begin{array}{lrcl}
\text{Types} & \tau, \upsilon & ::= & t \mid K \mid \tau \to \tau \\
\text{Type schemes} & \sigma & ::= & \tau \mid \forall t.\sigma \\
\text{Expressions} & M, N & ::= & x \mid k \mid \lambda x.M \mid M\,N \mid \underline{\texttt{let}}\ x = M\ \underline{\texttt{in}}\ N
\end{array}
$$

Figure 5.3: Types and terms of Core ML

$C^{\text{term}}(k) \in \mathcal{T}^{\text{type}}[\![C^{\text{type}}(k)]\!]$, and we extend the definition of $\mathcal{T}^{\text{term}}[\![\cdot]\!]$ such that

$$\mathcal{T}^{\text{term}}[\![\Gamma, k, C^{\text{type}}(k)]\!]\eta = C^{\text{term}}(k).$$

We will frequently refer to a frame $(\mathcal{T}^{\text{type}}[\![\cdot]\!], \mathcal{T}^{\text{term}}[\![\cdot]\!], T)$ as the frame $\mathcal{T}$, and will omit the superscript identifiers when they are obvious from context.

5.2 A SIMPLE SEMANTICS FOR POLYMORPHISM

Next we discuss Core ML, an extension of the simply-typed lambda calculus to support first-order polymorphism. The principal new features of Core ML are:

- Core ML types include type variables, in addition to the type constants and type constructors of the simply typed lambda calculus. Following Damas [5], but unlike Milner's [41] or Ohori's [48] presentations of Core ML, we introduce type schemes, with explicit quantification of type variables, to type polymorphic values.

- Core ML expressions include a `let`...`in` construct, used to introduce local polymorphic bindings; λ-bound variables continue to be treated monomorphically.

We show the terms and types of Core ML in Figure 5.3

$$[\textsc{Var}]\ \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$[\to \mathrm{I}]\ \frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash (\lambda x.M) : \tau \to \tau'} \qquad [\to \mathrm{E}]\ \frac{\Gamma \vdash M : \tau \to \tau' \quad \Gamma \vdash N : \tau}{\Gamma \vdash (M\, N) : \tau'}$$

$$[\forall\ \mathrm{I}]\ \frac{\Gamma \vdash M : \sigma \quad t \notin \mathit{ftv}(\Gamma)}{\Gamma \vdash M : \forall t.\sigma} \qquad [\forall\ \mathrm{E}]\ \frac{\Gamma \vdash M : \forall t.\sigma}{\Gamma \vdash M : [\tau/t]\sigma}$$

$$[\textsc{Let}]\ \frac{\Gamma, x : \sigma \vdash M : \sigma \quad \Gamma, x : \sigma \vdash N : \tau}{\Gamma \vdash (\underline{\texttt{let}}\ x = M\ \underline{\texttt{in}}\ N) : \tau}$$

Figure 5.4: Typing rules of Core ML

5.2.1 Core ML Typing

We give typing rules for Core ML in Figure 5.4. Again, like Damas and unlike Milner and Ohori, we give separate rules ∀ I and ∀ E to account for the introduction and elimination of quantifiers; this parallels our treatment of overloading in the next section.

Ohori gives an unusual typing rule for `let` expressions, as follows:

$$[\textsc{Let-Ohori}]\ \frac{\Gamma \vdash M : \tau \quad \Gamma \vdash [M/x]N : \upsilon}{\Gamma \vdash (\underline{\texttt{let}}\ x = M\ \underline{\texttt{in}}\ N) : \upsilon}$$

This simplifies his treatment of the Core ML type system: only λ-bound variables appear in the type environment Γ. As λ-bound variables are necessarily monomorphic, the variable environment in his semantics need only contain for monomorphically typed values; similarly, while the bound value in a `let` expression may have a generic type, each of its uses will have a monomorphic type (one of the instances of its generic type). Thus, Ohori's approach specifies the semantics of polymorphic code without having to give a meaning to polymorphic expressions directly.

We could take a similar approach to the typing of overloaded programs, by replacing each use of an overloaded function with a suitably monotyped version of the function, an approach similar to Gaster's semantics of overloading [12]. However, we find this somewhat unsatisfying. As our focus is on the semantics of overloading, we would prefer to give a semantics to class methods directly, rather than by inlining method implementations. Thus, we adapt Harrison's extension of Ohori's semantics (described in the following section), which allows us to preserve much of the simplicity of Ohori's style, while directly providing semantics to polymorphic (and, in the following section, overloaded) expressions.

5.2.2 Type Frames for Polymorphism

Ohori's approach to polymorphism is appealingly simple: the meaning of a polymorphic expression is a map from its ground types to its meaning at each ground type. Its meanings at ground types, in turn, can be given in any frame model of the typed lambda calculus. Harrison [18] extends this framework to describe polymorphic recursion, an extension that we will also use to describe overloading. The remainder of this section reviews Harrison's extension, albeit with minor changes of notation for our setting.

Frame objects contain just enough information to model abstraction and application. To permit the solution of polymorphic recursive equations, Harrison extends the notion of a type frame by requiring that each frame object $\mathcal{T}[\![\tau]\!]$ is also a pointed, complete partial order (pcpo). Having done so, he demonstrates that type-indexed sets, as used in the Ohori semantics, also form a pointed cpo.

We begin by defining a *pcpo frame* as a tuple

$$\langle \mathcal{T}^{\text{type}}[\![\cdot]\!], \mathcal{T}^{\text{term}}[\![\cdot]\!], T_{\tau,\upsilon}, \sqsubseteq_\tau, \sqcup_\tau, \perp_\tau \rangle,$$

where $\langle \mathcal{T}^{\text{type}}[\![\cdot]\!], \mathcal{T}^{\text{term}}[\![\cdot]\!], T_{\tau,\upsilon} \rangle$ is a type frame, and each set $\mathcal{T}[\![\tau]\!]$ is a pcpo with respect to $\sqsubseteq_\tau$, $\sqcup_\tau$ and $\perp_\tau$. We define the ground instances of a type scheme σ, written $\lfloor\sigma\rfloor$, by:

$$\lfloor\sigma\rfloor = \{S\,\tau \mid \sigma = \forall t_0 \ldots \forall t_n.\tau, S \in \mathit{GSubst}(t_0, \ldots, t_n)\}.$$

Alternatively, we could give a definition of the ground instances of scheme type recursively, matching the recursive structure of the syntax of type schemes:

$$\begin{aligned} \lfloor\tau\rfloor &= \{\tau\} \\ \lfloor\forall t.\sigma\rfloor &= \bigcup_{\tau \in \mathit{GType}} \lfloor[\tau/t]\sigma\rfloor. \end{aligned}$$

Now, for a given pcpo frame $\mathcal{T}$, we can define the semantics of a type scheme $\mathcal{T}^{\text{scheme}}[\![\sigma]\!]$ in terms of the ground instances of σ:

$$\mathcal{T}^{\text{scheme}}[\![\sigma]\!] = \Pi(\tau \in \lfloor\sigma\rfloor).\mathcal{T}^{\text{type}}[\![\tau]\!].$$

For example, the identity function $\lambda x.x$ has the type scheme $\forall t.t \to t$. Therefore, we would expect the semantics of the identity function to be a map from the ground instances of its type to the semantics of the simply-typed identity function at each type. We would expect its semantics to include the pair

$$\langle \mathit{Int} \to \mathit{Int}, \mathcal{T}^{\text{term}}[\![\emptyset, \lambda x : \mathit{Int}.x, \mathit{Int} \to \mathit{Int}]\!]\rangle$$

to account for the $\mathit{Int} \to \mathit{Int}$ ground instance of its type scheme; the pair

$$\langle \mathit{Bool} \to \mathit{Bool}, \mathcal{T}^{\text{term}}[\![\emptyset, \lambda x : \mathit{Bool}.x, \mathit{Bool} \to \mathit{Bool}]\!]\rangle$$

to account for the $\mathit{Bool} \to \mathit{Bool}$ ground instance of its type scheme; and so forth. Note that if σ has no quantifiers, and thus $\lfloor\sigma\rfloor = \{\tau\}$ for some type τ, then we have that

$$\mathcal{T}^{\text{scheme}}[\![\sigma]\!] = \{\{\langle\tau, b\rangle\} \mid b \in \mathcal{T}^{\text{type}}[\![\tau]\!]\},$$

and so an element of $\mathcal{T}^{\text{scheme}}[\![\tau]\!]$ is a singleton map, not an element of $\mathcal{T}^{\text{type}}[\![\tau]\!]$. As before, we will write $\mathcal{T}[\![\sigma]\!]$ instead of $\mathcal{T}^{\text{scheme}}[\![\sigma]\!]$ when there is no ambiguity. Harrison then proves the following theorem.

Theorem 5.1 (Harrison)**.** *Let $\mathcal{T}$ be a pcpo frame. Then, for any type scheme σ, $\mathcal{T}[\![\sigma]\!]$ is a pointed cpo where:*

- *For any $f, g \in \mathcal{T}[\![\sigma]\!]$, $f \sqsubseteq^{\sigma} g \iff (\forall \tau \in \lfloor \sigma \rfloor .\, f\,\tau \sqsubseteq_{\tau} g\,\tau)$;*
- *The bottom element $\bot_{\sigma}$ is defined to be $\{\langle \tau, \bot_{\tau} \rangle \mid \tau \in \lfloor \sigma \rfloor\}$; and,*
- *The least upper bound of an ascending chain $\{X_i\} \subseteq \mathcal{T}[\![\sigma]\!]$ is $\{\langle \tau, u_{\tau} \rangle \mid \tau \in \lfloor \sigma \rfloor, u_{\tau} = \sqcup_{\tau}(X_i\,\tau)\}$.*

We can define continuous functions and least fixed points for sets $\mathcal{T}[\![\sigma]\!]$ in the usual fashion:

- A function $f : \mathcal{T}[\![\sigma]\!] \to \mathcal{T}[\![\sigma']\!]$ is continuous if $f(\sqcup_{\sigma} X_i) = \sqcup_{\sigma'}(f(X_i))$ for all directed chains X_i in $\mathcal{T}[\![\sigma]\!]$.
- The fixed point of a continuous function $f : \mathcal{T}[\![\sigma]\!] \to \mathcal{T}[\![\sigma]\!]$ is defined by $\mathit{fix}(f) = \sqcup_{\sigma}(f^n(\bot_{\sigma}))$, and is the least value such that $\mathit{fix}\, f = f\,(\mathit{fix}\, f)$.

5.2.3 Semantics of Polymorphic Expressions

We can now give a semantics for the expressions of Core ML. For some type environment Γ and substitution $S \in \mathit{GSubst}(\mathit{ftv}(\Gamma))$, we define an $S\,\Gamma$-environment η as a mapping from variables to values such that $\eta(x) \in \mathcal{T}^{\text{scheme}}[\![(S\,\sigma)]\!]$ for each assignment $(x : \sigma)$ in Γ. Given a pcpo frame $\mathcal{T}$, a derivation Δ of $\Gamma \vdash M : \sigma$, a ground substitution $S \in \mathit{GSubst}(\mathit{ftv}(\Gamma, \sigma))$, and an $S\,\Gamma$-environment η, we define the interpretation $\mathcal{T}[\![\Delta]\!]S\eta$ by cases, as follows.

- Case Var: we have a derivation of the form

$$\Delta = \frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma}$$

  and define:

$$\mathcal{T}[\![\Delta]\!]S\eta = \eta(x).$$

Because η is a $S\,\Gamma$-environment, and $(x : \sigma) \in \Gamma$, we can conclude that $\eta(x) \in \mathcal{T}[\![S\,\sigma]\!]$ and so $\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}[\![S\,\sigma]\!]$.

- Case $\rightarrow$ I: we have a derivation of the form

$$\Delta = \cfrac{\Delta_1 = \cfrac{\vdots}{\Gamma, x : \tau \vdash M : \tau'}}{\Gamma \vdash (\lambda x.M) : \tau \rightarrow \tau'}$$

Let $\upsilon = S\,\tau$ and $\upsilon' = S\,\tau'$, and define:

$$\begin{aligned}\mathcal{T}[\![\Delta]\!]S\eta = \{\langle \upsilon \rightarrow \upsilon', f\rangle\}& \\ &\text{such that } f \in \mathcal{T}[\![\upsilon \rightarrow \upsilon']\!], \\ &\quad\text{and } \forall d \in \mathcal{T}[\![\upsilon]\!].\ T_{\upsilon,\upsilon'}(f, d) = \mathcal{T}[\![\Delta_1]\!]S(\eta[x \mapsto d]).\end{aligned}$$

We assume that f is representable in $\mathcal{T}^{\text{type}}[\![\upsilon \rightarrow \upsilon']\!]$, and claim its continuity in Lemma 5.2, below. Assuming that $\mathcal{T}[\![\Delta']\!]S\eta \in \mathcal{T}^{\text{scheme}}[\![\upsilon']\!]$, it follows that:

$$\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}^{\text{scheme}}[\![\upsilon \rightarrow \upsilon']\!] = \mathcal{T}[\![S\,\sigma]\!].$$

- Case $\rightarrow$ E: we have a derivation of the form

$$\Delta = \cfrac{\Delta_1 = \cfrac{\vdots}{\Gamma \vdash M : \tau \rightarrow \tau'} \qquad \Delta_2 = \cfrac{\vdots}{\Gamma \vdash N : \tau}}{\Gamma \vdash (M\,N) : \tau'}$$

Let $\upsilon = S\,\tau$ and $\upsilon' = S\,\tau'$, and define

$$\mathcal{T}[\![\Delta]\!]S\eta = \{\langle \upsilon',\, T_{\upsilon,\upsilon'}(\mathcal{T}[\![\Delta_1]\!]S\eta(\upsilon \rightarrow \upsilon'), \mathcal{T}[\![\Delta_2]\!]S\eta(\upsilon))\rangle\}.$$

By construction, $\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}^{\text{scheme}}[\![\upsilon']\!] = \mathcal{T}[\![\sigma]\!]$.

- Case $\forall$ I: we have a derivation of the form

$$\Delta = \cfrac{\Delta_1 = \cfrac{\vdots}{\Gamma \vdash M : \sigma} \quad t \notin \mathit{ftv}(\sigma)}{\Gamma \vdash M : \forall t.\sigma}$$

Intuitively, we interpret a polymorphic expression as the map from ground instances of its type to its interpretations at those types. As the interpretation of the subderivation Δ_1 is already in the form of a such a map, we can interpret Δ as the union of the meanings of Δ_1 for each ground instantiation of the quantified variable t. Formally, we define

$$\mathcal{T}[\![\Delta]\!]S\eta = \bigcup_{\tau \in \mathit{GType}} \mathcal{T}[\![\Delta_1]\!](S[t \to \tau])\eta.$$

Because $\sigma = \forall t.\sigma'$, we have that $\lfloor\sigma\rfloor = \bigcup_{\tau \in \mathit{GType}} \lfloor[\tau/t]\sigma'\rfloor$, and thus that $\mathcal{T}[\![\sigma]\!] = \bigcup_{\tau \in \mathit{GType}}(\mathcal{T}[\![[\tau/t]\sigma']\!])$. Thus, assuming that for ground types τ, $\mathcal{T}[\![\Delta']\!](S[t \mapsto \tau])\eta \in \mathcal{T}[\![S\,\sigma']\!]$ (that is, assuming the soundness of the type system for the derivation Δ_1), we have

$$\mathcal{T}[\![\Delta]\!]S\eta \in \left(\bigcup_{\tau \in \mathit{GType}} \mathcal{T}[\![S\,\sigma']\!]\right) = \mathcal{T}[\![S\,\sigma]\!].$$

- Case $\forall$ E: we have a derivation of the form

$$\begin{array}{r}\Delta_1 = \\ \Delta = \end{array}\dfrac{\dfrac{\vdots}{\Gamma \vdash M : \forall t.\sigma}}{\Gamma \vdash M : [\tau/t]\sigma}$$

By definition, $\lfloor\forall t.\sigma\rfloor = \bigcup_{\tau \in \mathit{GType}} \lfloor\sigma\rfloor$, and so $\lfloor[\tau/t]\sigma\rfloor \subseteq \lfloor\forall t.\sigma\rfloor$. Thus, the interpretation of Δ is a subset of the interpretation of Δ_1; writing $f|_Y$ for the restriction of a function f to some domain Y, we define:

$$\mathcal{T}[\![\Delta]\!]S\eta = (\mathcal{T}[\![\Delta_1]\!]S\eta)|_{\lfloor[\tau/t]\sigma\rfloor}.$$

Assuming that $\mathcal{T}[\![\Delta']\!]S\eta \in \mathcal{T}[\![S\,(\forall t.\sigma')]\!]$, the argument about ground types gives that $\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}[\![S\,\sigma]\!]$.

- Case Let: we have a derivation of the form

$$\Delta = \dfrac{\Delta_1 = \dfrac{\vdots}{\Gamma \vdash M : \sigma} \qquad \Delta_2 = \dfrac{\vdots}{\Gamma, x : \sigma \vdash N : \tau}}{\Gamma \vdash (\underline{\mathtt{let}}\ x = M\ \underline{\mathtt{in}}\ N) : \tau}$$

We begin by computing the value of the binding, taking the possibility of recursion into account. We define a function

$$f : d \mapsto \mathcal{T}[\![\Delta_1]\!]S(\eta[x \mapsto d])$$

whose continuity we assert in Lemma 5.2, below, and then define the value of the binding by

$$b = \bigsqcup\nolimits_{\sigma} f^n(\bot_\sigma)\text{for } n \in \mathbb{N}.$$

Assuming that $\mathcal{T}[\![\Delta_1]\!]S(\eta[x \mapsto d]) \in \mathcal{T}[\![S\,\sigma']\!]$, we have that $b \in \mathcal{T}[\![S\,\sigma']\!]$. Finally, we define:

$$\mathcal{T}[\![\Delta]\!]S\eta = \mathcal{T}[\![\Delta_2]\!]S(\eta[x \mapsto b]).$$

Similarly, assuming that $\mathcal{T}[\![\Delta_2]\!]S(\eta[x \mapsto b] \in \mathcal{T}^{\text{scheme}}[\![\tau]\!]$, we have that $\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}^{\text{scheme}}[\![\tau]\!] = \mathcal{T}[\![\sigma]\!]$.

The definition above relies on the following result to establish continued in cases $\rightarrow$ I and Let:

Lemma 5.2. *For all closed terms M such that Δ is a derivation of $\Gamma, x : \sigma \vdash M : \sigma'$, the function $f : (d \in \mathcal{T}[\![\sigma]\!]) \mapsto \mathcal{T}[\![\Delta]\!]S(\eta[x \mapsto d])$ is continuous.*

The proof is by induction on the term M, and follows Harrison's argument exactly. We conclude this section by arguing that the type system of Core ML accurately approximates its semantics.

Theorem 5.3. *For all derivations Δ of $\Gamma \vdash M : \sigma$, for all ground substitutions $S \in GSubst(ftv(\Gamma, \sigma))$, and for all $S\,\Gamma$-environments η, $\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}[\![S\,\sigma]\!]$.*

The proof is by induction over the height of the derivations; the justification for the cases is included in their definitions above.

Term variable	$x \in \mathit{Var}$	Term constants	k
Type variables	$t \in \mathit{TVar}$	Type constants	K

Types	τ, υ	$::=$	$t \mid K \mid \tau \to \tau$
Qualified types	ρ	$::=$	$\tau \mid \pi \Rightarrow \rho$
Type schemes	$\sigma \in \mathit{Scheme}$	$::=$	$\rho \mid \forall t.\sigma$
Expressions	$M, N \in \mathit{Expr}$	$::=$	$x \mid k \mid \lambda x.M \mid M\ N$
		$\mid$	$\underline{\texttt{let}}\ x = M\ \underline{\texttt{in}}\ N$
Method signatures	Si	$\in$	$\mathit{Var} \rightharpoonup \mathit{Pred} \times \mathit{Scheme}$
Method implementations	Im	$\in$	$\mathit{InstName} \times \mathit{Var} \rightharpoonup \mathit{Expr}$
Programs	Pr	$::=$	$\langle A \mid X, \mathit{Si}, \mathit{Im}, M : \tau \rangle$

Figure 5.5: Types and terms of OML

5.3 A SIMPLE SEMANTICS FOR OVERLOADING

Harrison suggests that his approach to semantics for polymorphic recursion, as described in the prior section, would extend naturally to describe type-class overloading. This section elaborates the intuitive notion he describes, based on Jones's theory of qualified types [28] and our semantics of type classes (from Chapter 4).

Figure 5.5 gives the terms and types of OML, an extension of the Core ML language to include overloading. We extend the types of Core ML with qualified types ρ, capturing the use of predicates in types. We must similarly extend the terms of Core ML to allow the definition of overloaded values. One approach would be to expand the grammar of expressions to include class and instance declarations; such an approach is taken in Wadler and Blott's semantics of type classes [70]. However, this approach makes such definitions local, in contrast to our global approach to type class semantics, and introduces problems with principal typing, as Wadler and Blott indicate in their discussion. We take an alternative approach,

$$[\textsc{Var}]\ \frac{(x : \sigma) \in \Gamma}{P \mid \Gamma \vdash x : \sigma}$$

$$[\to \mathrm{I}]\ \frac{P \mid \Gamma, x : \tau \vdash M : \tau'}{P \mid \Gamma \vdash (\lambda x.M) : \tau \to \tau'} \qquad [\to \mathrm{E}]\ \frac{P \mid \Gamma \vdash M : \tau \to \tau' \quad P \mid \Gamma \vdash N : \tau}{P \mid \Gamma \vdash (M\, N) : \tau'}$$

$$[\Rightarrow \mathrm{I}]\ \frac{P, \pi \mid \Gamma \vdash M : \rho}{P \mid \Gamma \vdash M : \pi \Rightarrow \rho} \qquad [\Rightarrow \mathrm{E}]\ \frac{P \mid \Gamma \vdash M : \pi \Rightarrow \rho \quad P \Vdash \pi}{P \mid \Gamma \vdash M : \rho}$$

$$[\forall\ \mathrm{I}]\ \frac{P \mid \Gamma \vdash M : \sigma \quad t \notin \mathit{ftv}(\Gamma, P)}{P \mid \Gamma \vdash M : \forall t.\sigma} \qquad [\forall\ \mathrm{E}]\ \frac{P \mid \Gamma \vdash M : \forall t.\sigma}{P \mid \Gamma \vdash M : [\tau/t]\sigma}$$

$$[\textsc{Let}]\ \frac{P \mid \Gamma, x : \sigma \vdash M : \sigma \quad P \mid \Gamma, x : \sigma \vdash N : \tau}{P \mid \Gamma \vdash (\underline{\texttt{let}}\ x = M\ \underline{\texttt{in}}\ N) : \tau}$$

Figure 5.6: Expression typing rules of OML

introducing a new top-level construct, which we call programs. Programs contain type class information, such as axioms and class constraints, the type signatures of the class methods, the implementations provided by each instance, and, finally, the body of the program. We leave implicit many syntactic restrictions on Habit programs, such as the requirement that each instance contain a complete set of method definitions, as they do not contribute to the semantics of overloaded values.

5.3.1 OML Typing

We begin with the typing of OML expressions; our treatment is unchanged from Jones's approach [28]. Typing judgments take the form

$$P \mid \Gamma \vdash M : \sigma,$$

where P is a set of predicates restricting the type variables in Γ and σ. The typing rules for OML expressions are given in Figure 5.6. Rules $\Rightarrow$ I and $\Rightarrow$ E describe the interaction between the predicate context P and qualified types ρ; we leave implicit the type class basis $A \mid X$, as it is constant throughout the typing judgments. Otherwise, the rules are minimally changed from the corresponding typing rules of Core ML.

We represent programs by tuples $\langle A \mid X, Si, Im, M : \tau\rangle$, where $A \mid X$, Si, and Im capture the type class components of the program, and M is the program body (see Figure 5.5 for specification of components Si, Im and M, and Figure 4.1 for specification of $A \mid X$). In the source code of a Habit program, type class methods are specified in class and instance declarations, such as the following:

```
class Eq t where (==) :: t → t → Bool
instance Eq (List t) if Eq t where (==) = ...
```

We partition the information in the class and instance declarations into the first three components of the program tuple. The logical content—axioms and class constraints—are captured by the pair $A \mid X$. The method signatures are captured in the mapping Si; for this example, we would have that

$$Si((==)) = \langle \text{Eq } t, t \to t \to \text{Bool}\rangle$$

where we do not quantify over the variables appearing in the class predicate. Note that, as they arise from the class definitions, each predicate in the range of Si will be of the form $C\ \vec{t}$ for some class C and type variables t. The type scheme of a class member may quantify over additional variables, or include additional predicates, beyond those used in the class itself. For example, the `Monad` class has the following definition:

```
class Monad m
  where return :: a → m a
          (>>=)  :: m a → (a → m b) → m b
```

Note that the variable `a` in the type of `return` is not part of the `Monad` constraint. Thus, we would expect that

$$Si(\text{return}) = \langle \text{Monad } m, \forall a. a \to m\ a\rangle.$$

The method implementations themselves are recorded in component Im, which maps pairs of method and instance names to implementing expressions.

To assure that the method implementations are well typed, we must begin by determining what type each such method implementation should have. This is a combination of the defining instance, including its context, and the definition of the method itself. For example, in the instance above, the body of the `(==)` method should compare lists of arbitrary type `t` for equality (this arises from the instance predicate `Eq (List t)` and the signature of `(==)`), given the assumption `Eq t` (arising from the defining instance). That is, we would expect it to have the type

$$\forall t. \mathrm{Eq}\ t \Rightarrow \mathrm{List}\, t \to \mathrm{List}\, t \to \mathrm{Bool}.$$

We introduce abbreviations for the type scheme of each method and of each method at each instance, assuming some program $\langle A \mid X, \mathit{Si}, \mathit{Im}, M : \tau\rangle$. For each method name x such that $\mathit{Si}(x) = \langle \pi, \forall \vec{u}. \rho\rangle$, we define the type scheme for x by:

$$\sigma_x = \forall \vec{t}. \forall \vec{u}.\, \pi \Rightarrow \rho,$$

or, equivalently, writing ρ as $Q \Rightarrow \tau$, we have that

$$\sigma_x = \forall \vec{t}. \forall \vec{u}.\, (\pi, Q) \Rightarrow \tau$$

where, in each case, $\vec{t} = \mathit{ftv}(\pi)$. Similarly, for each method x as above, and each instance d such that $\langle x, d\rangle \in \mathrm{dom}(\mathit{Im})$, where $(d : \forall \vec{t'}.\, \pi' \Leftarrow P) \in \mathit{Clauses}(A)$, and there is some $S \in \mathit{Subst}(\mathit{ftv}(\pi))$ such that $S\, \pi = \pi'$, we define the type scheme for x from d by:

$$\sigma_{x,d} = \forall \vec{t'}. \forall \vec{u}.\, (P, S\ Q) \Rightarrow S\, \tau,$$

where $\vec{t'} = \mathit{ftv}(S\, \pi)$.

We give the typing rule for OML programs in Figure 5.7; the implicit type class specification in the typing derivations of the implementations and of the body is provided by the component $A \mid X$ of the program. Intuitively, program

$$\frac{\forall \langle x, d\rangle \in \mathrm{dom}(\mathit{Im}).\ P \mid (\Gamma, x : \sigma_x) \vdash \mathit{Im}(x,d) : \sigma_{x,d} \qquad P \mid (\Gamma, x : \sigma_x) \vdash M : \tau}{P \mid \Gamma \vdash \langle A \mid X, \mathit{Si}, \mathit{Im}, M : \tau\rangle}$$

Figure 5.7: Program typing rule for OML

$\langle A \mid X, \mathit{Si}, \mathit{Im}, M : \tau\rangle$ is well typed under assumptions P and environment Γ if each method implementation $\mathit{Im}(x, d)$ has the type $\sigma_{x,d}$, and if the main expression has the declared type τ, assuming that each method x has type σ_x.

5.3.2 The Meaning of Qualified Types

To describe the meaning of overloaded expressions, we must begin with the meaning of qualified types. Intuitively, qualifiers in types can be viewed as predicates in set comprehensions—that is, the qualified type $\forall t.\mathrm{Eq}\ t \Rightarrow t \to t \to \mathrm{Bool}$ describes the set of types

$$\{t \to t \to \mathrm{Bool} \mid t \in \mathrm{Eq}\}.$$

However, existing approaches to semantics for overloading typically do not interpret qualifiers in this fashion: Wadler and Blott [70], for instance, translate qualifiers into dictionary arguments, while Jones [28] translates qualified types into a calculus with explicit evidence abstraction and application.

Our approach, by contrast, preserves the intuitive notion of qualifiers. Given some type class basis $A \mid X$, we define the ground instances of an OML type scheme σ by:

$$\lfloor\sigma\rfloor = \{S\,\tau \mid \sigma = (\forall\vec{t}.P \Rightarrow \tau), S \in \mathit{GSubst}(\vec{t}), A \mid X \vdash \emptyset \Vdash S\,P\}.$$

As before, we can give an equivalent, recursive definition as follows:

$$\begin{aligned}
\lfloor\tau\rfloor &= \{\tau\} \\
\lfloor\pi \Rightarrow \rho\rfloor &= \begin{cases} \lfloor\rho\rfloor & \text{if } A \mid X \vdash \emptyset \Vdash \{\pi\} \\ \emptyset & \text{otherwise} \end{cases} \\
\lfloor\forall t.\sigma\rfloor &= \bigcup_{\tau \in \mathit{GType}} \lfloor[\tau/t]\sigma\rfloor.
\end{aligned}$$

In the typing judgments for OML, predicates can appear in both types and contexts. To account for both sources of predicates, Jones introduces *constrained type schemes* $(P \mid \sigma)$, where P is a list of predicates and σ is an OML type scheme; an unconstrained type scheme σ can be treated as the constrained scheme $(\emptyset \mid \sigma)$ (following typical convention, we regard the empty conjunction as true). We can define the ground instances of constrained type schemes by a straightforward extension of the definition for unconstrained schemes:

$$\lfloor(P \mid \sigma)\rfloor = \{S\,\tau \mid \sigma = (\forall\vec{t}.Q \Rightarrow \tau), S \in \mathit{GSubst}(\vec{t}), A \mid X \vdash \emptyset \Vdash (P, S\,Q)\}.$$

We can now define the interpretation of a constrained OML type scheme (or, equivalently, an unconstrained scheme or qualified type) in terms of its ground instances, as we did for Core ML type schemes:

$$\mathcal{T}^{\text{scheme}}[\![(P \mid \sigma)]\!] = \Pi(\tau \in \lfloor(P \mid \sigma)\rfloor).\, \mathcal{T}^{\text{type}}[\![\tau]\!].$$

5.3.3 Semantics for Overloaded Expressions

We can describe the semantics of OML expressions in the same way we described the semantics for Core ML expressions: by giving a semantics $\mathcal{T}[\![\Delta]\!]S\eta$ for each OML typing derivation Δ. As the majority of typing judgments are almost unchanged from Core ML, the majority the cases are very similar. The semantics for the new forms of derivation are as follows.

- Case $\Rightarrow$ I: we have a derivation of the form

  $$\begin{array}{r}\Delta_1 = \dfrac{\vdots}{P, \pi \mid \Gamma \vdash M : \rho} \\ \Delta = \overline{P \mid \Gamma \vdash M : \pi \Rightarrow \rho}\end{array}$$

  This rule does not affect the semantics of an expression; thus, we define:

  $$\mathcal{T}[\![\Delta]\!]S\eta = \mathcal{T}[\![\Delta_1]\!]S\eta.$$

  Observe that $\lfloor(S(P,\pi) \mid S\,\rho)\rfloor = \lfloor(S\,P \mid S\,(\pi \Rightarrow \rho))\rfloor$. As such, if

  $$\mathcal{T}[\![\Delta_1]\!]S\eta \in \mathcal{T}[\![(S\,(P,\pi) \mid S\,\rho)]\!],$$

  then we must also have that

  $$\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}[\![(S\,P \mid S\,(\pi \Rightarrow \rho))]\!].$$

- Case $\Rightarrow$ E: we have a derivation of the form

  $$\begin{array}{r}\Delta_1 = \dfrac{\vdots}{P \mid \Gamma \vdash M : \pi \Rightarrow \rho} \quad P \Vdash \pi \\ \Delta = \overline{\qquad P \mid \Gamma \vdash M : \rho \qquad}\end{array}$$

  As with rule $\Rightarrow$ I, this rule does not affect the semantics of expression M, and so we define:

  $$\mathcal{T}[\![\Delta]\!]S\eta = \mathcal{T}[\![\Delta_1]\!]S\eta.$$

  From Theorem 4.11 and the hypothesis $A \mid X \vdash P \Vdash \pi$, we have that for any $S \in \mathit{GSubst}(\mathit{ftv}(P,\pi))$, $A \mid X \vdash S\,P \Vdash \{S\,\pi\}$; thus, we can conclude that $\lfloor(S\,P \mid S\,(\pi \Rightarrow \rho))\rfloor = \lfloor(S\,P \mid S\,\rho)\rfloor$. Finally, if we assume that $\mathcal{T}[\![\Delta_1]\!]S\eta \in \mathcal{T}[\![(S\,P \mid S\,(\pi \Rightarrow \rho))]\!]$, then we can conclude that $\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}[\![(S\,P \mid S\,\rho)]\!]$.

It is straightforward to extend the soundness result for our semantics of Core ML (Theorem 5.3) to a corresponding result for OML expressions.

Theorem 5.4. *For all derivations $P \mid \Gamma \vdash M : \sigma$, ground substitutions $S \in \mathit{GSubst}(P,\Gamma,\sigma)$ and $S\,\Gamma$-environments η, $\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}[\![(S\,P \mid S\,\sigma)]\!]$.*

The proof is by induction; again, the arguments for each case are included in the definition above.

5.3.4 Semantics for OML Programs

This section builds on the semantics of expressions in the previous section to give meaning to OML programs. Our approach is intuitively simple: as we have defined a program to be a collection of instances and a main expression, we must build the meanings of the methods from the instances, and then use the meanings of the methods to define the meaning of the main expression. Formally, we extend the interpretation function to typing derivations of programs as follows:

- Let Δ be a derivation that program $\langle A \mid X, Si, Im, M : \tau\rangle$ is well-typed given assumptions P and environment Γ. Then we know that Δ must be of the form

  $$\Delta = \frac{\Delta_{x,d} = \dfrac{\vdots}{P \mid (\Gamma, x : \sigma_x) \vdash Im(x,d) : \sigma_{x,d}} \qquad \Delta_M = \dfrac{\vdots}{P \mid (\Gamma, x : \sigma_x) \vdash M : \tau}}{P \mid \Gamma \vdash \langle A \mid X, Si, Im, M : \tau\rangle}$$

  with one derivation $\Delta_{x,d}$ for each pair $\langle x, d\rangle \in \mathrm{dom}(Im)$. Enumerate the methods in the program $x_1, x_2, \ldots, x_m$, and let

  $$\Sigma = \mathcal{T}[\![\sigma_{x_1}]\!] \times \mathcal{T}[\![\sigma_{x_2}]\!] \times \cdots \times \mathcal{T}[\![\sigma_{x_m}]\!].$$

  For each method x_i, we define a function $f_i : \Sigma \to \mathcal{T}[\![\sigma_{x_i}]\!]$, approximating its meaning, as follows:

  $$f_i(\langle b_1, b_2, \ldots, b_m\rangle) = \bigcup_{\langle x_i, d\rangle \in \mathrm{dom}(Im)} \mathcal{T}[\![\Delta_{x_i,d}]\!]\emptyset(\eta[x_j \mapsto b_j]),$$

  and define function $f : \Sigma \to \Sigma$, approximating the meaning of all the methods in the program, as

  $$f(b) = \langle f_1(b), f_2(b), \ldots, f_m(b)\rangle.$$

  We can now define a tuple b, such that the component b_i is the meaning of method x_i, as follows:

  $$b = \bigsqcup\nolimits_{\Sigma} f^n(\bot_\Sigma).$$

Finally, we extend the interpretation function to programs by

$$\mathcal{T}[\![\langle A \mid X, Si, Im, M : \tau\rangle]\!]\eta = \mathcal{T}[\![\Delta_M]\!]\emptyset(\eta[x_i \mapsto b_i]).$$

As M is the entry point of the program, and is used monomorphically, the interpretation of programs is not parameterized by a type substitution S. Assuming that $b \in (\mathcal{T}[\![\sigma_{x_1}]\!] \times \mathcal{T}[\![\sigma_{x_2}]\!] \times \cdots \times \mathcal{T}[\![\sigma_{x_m}]\!])$, Theorem 5.4 gives that $\mathcal{T}[\![\Delta]\!] \in \mathcal{T}^{\text{scheme}}[\![\tau]\!]$.

We must show that $b \in (\mathcal{T}[\![\sigma_{x_1}]\!] \times \mathcal{T}[\![\sigma_{x_2}]\!] \times \cdots \times \mathcal{T}[\![\sigma_{x_m}]\!])$. To do so, we will demonstrate that the interpretation of the type scheme of a method is the union of the interpretation of the type schemes of its instances. This will show that each $f_i(b) \in \mathcal{T}[\![\sigma_{x_i}]\!]$, from which the desired result follows immediately.

Lemma 5.5. *The ground instances of the type scheme of a method x is the union of its ground instances at each instance. That is,*

$$\lfloor \sigma_x \rfloor = \bigcup_{\langle x,d\rangle \in \text{dom}(Im)} \lfloor \sigma_{x,d} \rfloor.$$

Proof. Without loss of generality, assume that $\sigma_x = \forall \vec{t}.(\pi, Q) \Rightarrow \tau$, where x is a method of $class(\pi)$. We prove that

$$\lfloor \sigma_x \rfloor = \bigcup_{\langle d,x\rangle \in \text{dom}(Im)} \lfloor \sigma_{x,d} \rfloor$$

by the inclusions

$$\lfloor \sigma_x \rfloor \subseteq \bigcup_{\langle x,d\rangle \in \text{dom}(Im)} \lfloor \sigma_{x,d} \rfloor,$$

and

$$\lfloor \sigma_x \rfloor \supseteq \bigcup_{\langle x,d\rangle \in \text{dom}(Im)} \lfloor \sigma_{x,d} \rfloor.$$

We will show only the first inclusion; the second is by an identical argument. Fix some $v \in \lfloor \sigma_x \rfloor$. By definition, there is some $S \in GSubst(\vec{t})$ such that $v = S\,\tau$ and $\emptyset \vdash S\,\pi, S\,P$. Because $\emptyset \Vdash S\,\pi$, there must be some $(d : \forall \vec{u}.\ T\,\pi \Leftarrow P) \in$

Clauses(A) and substitution $S' \in \mathit{GSubst}(\vec{u})$ such that $S\,\pi = S'\,\pi'$ and $\emptyset \vdash S'\,P$. Now, we have that $\sigma_{x,d} = \forall \vec{t'}.(P, T\,Q) \Rightarrow T\,\tau$ for some substitution T; thus, there is some $T' \in \mathit{GSubst}(\vec{t'})$ such that $\upsilon = T'\,(T\,\tau)$, $S\,P = T'\,(T\,Q)$, and so $\upsilon \in \lfloor \sigma_{x,d} \rfloor$. □

Lemma 5.6. *The interpretation of the type scheme of a method x is the union of the interpretations of its type scheme at each instance. That is,*

$$\mathcal{T}[\![\sigma_x]\!] = \bigcup_{\langle x,d\rangle \in \mathrm{dom}(\mathit{Im})} \mathcal{T}[\![\sigma_{x,d}]\!].$$

Proof. Recall that

$$\mathcal{T}^{\text{scheme}}[\![\sigma_x]\!] = \Pi(\tau \in \lfloor \sigma_x \rfloor).\mathcal{T}^{\text{type}}[\![\tau]\!].$$

From Lemma 5.5, we have that

$$\mathcal{T}^{\text{scheme}}[\![\sigma_x]\!] = \Pi\left(\tau \in \bigcup_{\langle x,d\rangle \in \mathrm{dom}(\mathit{Im})} \lfloor \sigma_{x,d} \rfloor\right).\mathcal{T}^{\text{type}}[\![\tau]\!].$$

As $\mathcal{T}^{\text{type}}[\![\cdot]\!]$ is a function, this is equivalent to

$$\mathcal{T}^{\text{scheme}}[\![\sigma_x]\!] = \bigcup_{\langle x,d\rangle \in \mathrm{dom}(\mathit{Im})} \Pi(\tau \in \lfloor \sigma_{x,d} \rfloor).\mathcal{T}^{\text{type}}[\![\tau]\!],$$

and finally, again from the definition of $\mathcal{T}^{\text{scheme}}[\![\cdot]\!]$,

$$\mathcal{T}^{\text{scheme}}[\![\sigma_x]\!] = \bigcup_{\langle x,d\rangle \in \mathrm{dom}(\mathit{Im})} \mathcal{T}^{\text{scheme}}[\![\sigma_{x,d}]\!].$$ □

Finally, we can extend the soundness of our semantics of OML from expressions (Theorem 5.4) to programs.

Theorem 5.7. *If Δ is a derivation of $P \mid \Gamma \vdash \langle A \mid X, \mathit{Si}, \mathit{Im}, M : \tau\rangle$, then $\mathcal{T}[\![\Delta]\!] \in \mathcal{T}^{\text{scheme}}[\![\tau]\!]$.*

The proof given in the definition of the meaning function in combination with Lemma 5.6.

5.4 EXAMPLE: POLYMORPHIC IDENTITY FUNCTIONS

We conclude our discussion of the semantics of ad-hoc polymorphism with a brief example. Figure 5.8 gives two definitions of a polymorphic identity function. The first definition (`id1`) is the typical, parametric definition. The second (`id2`) is based on ad-hoc polymorphism; the identity for functions is defined in terms of the identity functions for the domain and range. We might expect that `id1` and `id2` refer to the same function: for any expression `x`, we should expect both `id1 x` and `id2 x` to be well-typed, and that $[\![\text{id1 x}]\!] = [\![\text{x}]\!] = [\![\text{id2 x}]\!]$. However, previous dictionary-passing approaches to semantics for ad-hoc polymorphism provide different denotations for `id1` and `id2`, as `id2` has an additional dictionary argument. In this section, we will argue that our semantics gives these two definitions the same denotation. By doing so, we will show, first, that our semantics is sufficient to conclude non-trivial properties of programs, and, second, that it more closely captures the intuitive meaning of ad-hoc polymorphism than previous dictionary-passing approaches.

We intend to show that $[\![\text{id1}]\!] = [\![\text{id2}]\!]$. We begin by showing that they are defined over the same domain; that is, that $\lfloor \forall t.\ t \to t \rfloor = \lfloor \forall u. \text{Id2}\ u \Rightarrow u \to u \rfloor$. By definition, we have

$$\lfloor \forall t.\ t \to t \rfloor = \{\tau \to \tau \mid \tau \in \mathit{GType}\}$$

and

$$\lfloor \forall u.\ \text{Id2}\ u \Rightarrow u \to u \rfloor = \{\tau \to \tau \mid \tau \in \mathit{GType}, \emptyset \Vdash \text{Id2}\ \tau\}.$$

We show that $\emptyset \Vdash \text{Id2}\ \tau$ for all types τ by induction on the structure of τ. In the base case, we know that $\tau = K$ for some type constructor K. In this case, we have $\tau \nsim (t \to u)$ and $\tau \sim t$, and so, by rules Step-Pos and Match, $\emptyset \Vdash \text{Id2}\ \tau$. In the inductive case, we know that $\tau = \tau_0 \to \tau_1$ for some types τ_0, τ_1. In this case, we have $\tau \sim (t \to u)$ with substitution $[\tau_0/t, \tau_1/u]$ and, by the inductive hypothesis,

```
id1 :: t → t
id1 x = x

class Id2 t
    where id2 :: t → t

instance Id2 (t → u) if Id2 t, Id2 u
    where id2 f = id2 ∘ f ∘ id2
else Id2 t
    where id2 x = x
```

Figure 5.8: Polymorphic identity function, defined using parametric (id1) and ad hoc (id2) polymorphism

that $\emptyset \Vdash \text{Id2}\ \tau_0$ and $\emptyset \Vdash \text{Id2}\ \tau_1$. Thus, by rule Match, we can conclude that $\emptyset \Vdash \text{Id2}\ (\tau_o \to \tau_1)$, that is, that $\emptyset \Vdash \text{Id2}\ \tau$. Because $\emptyset \Vdash \text{Id2}\ \tau$ for all ground types τ, we have

$$\{\tau \to \tau \mid \tau \in \mathit{GType}, \emptyset \Vdash \text{Id2}\ \tau\} = \{\tau \to \tau \mid \tau \in \mathit{GType}\},$$

and so $[\![\text{id1}]\!]$ and $[\![\text{id2}]\!]$ are defined over the same domain.

Next, we show that $[\![\text{id1}]\!]$ and $[\![\text{id2}]\!]$ have the same value at each point in their domain; that is, that for any type $\tau \in \mathit{GType}$,

$$[\![\text{id1}]\!](\tau \to \tau) = [\![\text{id2}]\!](\tau \to \tau).$$

Again, we proceed by induction on the structure of τ. In the base case, we know that $\tau = K$ for some base type K. From the proof of $\emptyset \Vdash \text{Id2}\ K$, we have $[\![\text{id2}]\!](K \to K) = [\![\lambda x : K.x]\!]$. As $[\![\text{id1}]\!](K \to K) = [\![\lambda x : K.x]\!]$, we have

$$[\![\text{id1}]\!](K \to K) = [\![\text{id2}]\!](K \to K).$$

In the inductive case, we know that $\tau = \tau_0 \to \tau_1$ for some types τ_0 and τ_1. From the proof of $\emptyset \Vdash \text{Id2}\ (\tau_0 \to \tau_1)$, we can conclude that

$$[\![\text{id2}]\!](\tau \to \tau) = [\![\lambda f : (\tau_0 \to \tau_1).M \circ f \circ N]\!]$$

for some simply typed expressions M and N such that $[\![M]\!] = [\![\text{id2}]\!](\tau_1)$ and $[\![N]\!] = [\![\text{id2}]\!](\tau_2)$. The inductive hypothesis gives that $[\![\text{id2}]\!](\tau_1) = [\![\text{id1}]\!](\tau_1)$ and that $[\![\text{id2}]\!](\tau_0) = [\![\text{id1}]\!](\tau_0)$, and thus that $[\![M]\!] = [\![\lambda x : \tau_1.x]\!]$ and $[\![N]\!] = [\![\lambda x : \tau_0.x]\!]$. By congruence, we have

$$[\![\text{id2}]\!](\tau \to \tau) = [\![\lambda f : (\tau_0 \to \tau_1).(\lambda x : \tau_1.x) \circ f \circ (\lambda x : \tau_0.x)]\!].$$

Finally, assuming a standard definition of composition, and reducing, we have

$$\begin{aligned}[\![\text{id2}]\!](\tau \to \tau) &= [\![\lambda f : (\tau_0 \to \tau_1).f]\!] \\ &= [\![\lambda f : \tau.f]\!] \\ &= [\![\text{id1}]\!](\tau \to \tau).\end{aligned}$$

We have shown that $[\![\text{id1}]\!]$ and $[\![\text{id2}]\!]$ are defined over the same domain, and that they have the same value at each point in their domain. Thus, we conclude that $[\![\text{id1}]\!] = [\![\text{id2}]\!]$.

5.5 RELATED WORK

The semantics of polymorphism, in its various forms, has been studied extensively over the past 50 years; this section, therefore, will provide an overview of that work most relevant to ours instead of attempting to provide a comprehensive catalog.

Our approach begins with Ohori's semantics of Core ML [48]. His semantics is further developed than ours—in particular, he develops an equational theory of Core ML, and proves the soundness of that theory, not just of the type system, with respect to his semantics. We believe that exploring equational theories for overloading would be a useful extension of this work (§7.3). Ohori's approach to the semantics of Core ML is somewhat unusual; more typical approaches include those of Milner [41] and Mitchell and Harper [43]. Milner's semantics relies on embedding Core ML expressions into an untyped language, including a distinguished value for type errors, and then proving that well-typed programs do not evaluate to the error value. This approach was extended to handle recursive type by MacQueen, Plotkin, and Sethi [39]. Mitchell and Harper embed core ML expressions into a polymorphic lambda calculus with explicit type abstraction and application, in the style of Girard's System F [15] or Reynold's polymorphic lambda calculus [52]. They also provide a treatment of the ML module system, an important consideration in the semantics of Standard ML, but not relevant to this discussion. Ohori identifies reasons to prefer his approach over either that of Milner or that of Harper and Mitchell: both approaches use a semantic domain with far more values than correspond to values of ML, either because (in the untyped case) those values would not be well-typed, or (in the explicit typed case) they differ only in the type-level operations.

While Ohori's approach describes the semantics of polymorphism, he does not represent polymorphic values directly, which leads to an unusual treatment of the typing of `let` expression (§5.2.1). Harrison extends Ohori's approach to treat polymorphic recursion [18]; in doing so, he provides a representation of polymorphic values. Harrison suggests that his approach would be applicable to type class-based overloading, but does not develop the idea further.

The semantics of type class-based overloading has also received significant attention. Wadler and Blott [70] initially described the meaning of type classes using a "dictionary-passing translation", in which overloaded expressions are parameterized by type-specific implementations of class methods. Applying their approach to the full Haskell language, however, requires a target language with more complex types than their source language. For example, the `Num` class, defined in the Haskell prelude, includes a `fromIntegral` method, as follows.

```
class Num t
   where fromIntegral :: Integral u ⇒ u → t
         ...
```

Thus, a dictionary for Num t must itself contain a polymorphic value for the `fromIntegral` method, to allow for different instantiations of `u`. Such values cannot be defined in Haskell 98. A similar problem arises in the extension of type classes to constructor classes [27]. For example, in translating the `Monad` class:

```
class Monad m
   where return :: a → m a
         (>>=)  :: m a → (a → m b) → m b
```

the dictionary for `Monad` τ must contain polymorphic values for the `return` and `(>>=)` methods. In their formal treatment, Wadler and Blott give a form of class and instance declaration that allows local instances. While this does not pose problems with their semantics, it introduces type-system difficulties, including a loss of principal types.

In his semantics of overloading [28], Jones generalized the treatment of evidence by translating from a language with overloading (OML) to a language with explicit evidence abstraction and application. Jones highlights another difficulty of translation-based approaches: that, as evidence abstraction and application are not syntax directed (cf., typing rules $\Rightarrow$ I and $\Rightarrow$ E in our presentation of OML), there can be many distinct translations of a single OML term. He then shows that, given suitable assumptions on the predicate system, if a term has an unambiguous type, then its translations are all equivalent [25]. Jones does not provide a semantics of the language with explicit evidence abstraction and application; indeed, such a semantics could not usefully be defined without choosing a particular form of predicate, and thus a particular form of evidence. Gaster [13] adapts Ohori's approach to provide such a semantics for type classes. In contrast to our approach, Gaster assumes a translation to explicit evidence application and abstraction and still relies on treating dictionaries as values in the target language. Thus, as with Wadler and Blott's semantics, his approach is not directly applicable to classes such as `Num` or `Monad`.

Odersky, Wadler and Wehr [47] propose an alternative formulation of overloading, including a type system and type inference algorithm, and a ideal-based semantics of qualified types. However, their approach requires a substantial restriction to the types of overloaded values—each must be of the form $t \to \tau$, where variable t is the constrained variable. This approach rules out several functions in the Haskell prelude—such as the `fromIntegral` function described above, and obviously does not adapt to multi-parameter type classes. Many of the examples in this dissertation, such as the `inj` and `(?)` functions (§3.4), are not of this form.

While ML modules and type classes serve different purposes in programming, the dictionary-passing implementation of type classes looks similar enough to some uses of modules that to have encouraged several studies of the parallels between modules and type classes themselves. Dreyer et al. formalize such an approach, called modular type classes [8]. Despite the parallels, however, many features of Haskell-style type classes are complicated in this framework. For example, superclasses are implemented by building combinations of structures. This is similar to, if slightly more complex than, the method of embedding superclass dictionaries in subclass dictionaries commonly used in Haskell compilers. However, while the implementation of Haskell superclasses is managed by the compiler, Dreyer et al's encoding requires the programmer to manipulate such embeddings directly. Many properties of type classes are only valid locally with modular type classes—for example, coherence and consistency properties could only established for a particular collection of canonical instances, which is a local property. Finally, this approach effectively defines the semantics of type classes in terms of the semantics of ML modules, arguably a more complex setting.

Devriese and Piessens describe an implicit argument mechanism for Agda [6], which they call instance arguments, and claim that it provides an encoding of type classes. Their encoding of instances is similar to that of Dreyer et al., and shares many of that encoding's infelicities compared to Haskell type classes. Additionally, their mechanism for selecting values for implicit arguments includes no aspect of instance search: thus, for example, each use of a qualified instance (such as that for `Eq (List t)` must be explicitly constructed. Finally, as Agda's semantics have not been formalized, their mechanism currently provides only an intuitive argument for the meaning of values with instance parameters.

6. THE HABIT PREDICATE SOLVER

This chapter describes the Habit predicate solver, a component of our Habit compiler that, through interaction with the typechecker, implements the key type class features described in the previous chapters. We begin by discussing the functionality of the solver. While our semantics provides a sensible basis for reasoning about the OML type system, and the meaning of overloaded expressions, it relies on mathematical structures (in particular, infinite maps) that do not map directly to an implementation. We describe an alternative interpretation of overloading that is both faithful to our semantics and practical for compilation purposes (§6.1). Next, we describe additional functionality provided by the solver: simplification (§6.2), which attempts to reduce the complexity of inferred predicate sets, and improvement (§6.3), which computes types equalities that must hold for given sets of predicates to hold. Finally, we discuss some of the structures and techniques used in the solver, and describe some of the concerns that arose during its implementation (§6.4).

6.1 ENTAILMENT AND EVIDENCE

Our semantics of overloading is based on an enumeration of the meaning of an overloaded value at each of its possible ground types. However, in practice, we would not expect an implementation of Habit to generate such an (infinite) enumeration, either during compilation or at run-time. For example, consider a program containing the typical instances of the `Eq` class for integers and lists (§2.2), and the main expression:

```
main = [1, 2] == [2, 1]
```

Our semantics for the equality function `(==)` in such a program is indexed by every type in the `Eq` class; as we discussed before, such a set is infinite. Thus, we could not hope to include the entire semantics of `(==)` in our compiled program. However, we could observe that only a finite number of those instances are required for the `main` expression, and transform the original program into the following finite, overloading-free version:

```
eqInt x y = isZero (x - y)

eqListInt []     []     = True
eqListInt (x:xs) (y:ys) = eqInt x y && eqListInt xs ys
eqListInt _      _      = False

main = eqListInt [1, 2] [2, 1]
```

In the Habit compiler, we implement this transformation in two steps. First, during type checking, we annotate expressions with evidence abstractions and applications; second, following type inference, a compiler pass called specialization selects type-specific implementations of overloaded values. Our approach thus has the flavor of the dictionary-passing translation that is typically used to describe or implement Haskell type classes, but with an additional compile-time step to eliminate any direct run-time representation of overloading or dictionaries. Dictionary-passing translation has been previously described in detail, by Wadler and Blott [70], Jones [28], and others; we will adopt Jones's treatment, as we did his type system for overloading. Jones has also described a similar scheme for compile-time dictionary elimination via partial evaluation [29]. Our discussion will thus focus on the interpretation of the predicate system necessary to integrate with these existing approaches.

Jones describes his translation-based semantics as follows. He begins by defining a language, called OP, that extends a typed lambda calculus with explicit evidence abstraction ($\lambda_e v.M$) and application ($M\ e$). He then provides a type-based translation from OML to OP, extending the OML typing judgment

$$P \mid \Gamma \vdash M : \sigma$$

to pair an identifier v_i with each predicate in P_i in P, and to include a translation M', in OP, of the source OML term M:

$$P \mid \Gamma \vdash M \hookrightarrow M' : \sigma$$

where M' is an OP term. This translation is largely straightforward; for example, the application rule is extended as follows:

$$[\to E]\ \frac{: P \mid \Gamma \vdash M \hookrightarrow M' : \upsilon \to \tau \quad : P \mid \Gamma \vdash N \hookrightarrow N' : \upsilon}{P \mid \Gamma \vdash M\ N \hookrightarrow (M'\ N') : \tau}$$

The more interesting rules are those that introduce and eliminate predicates in types. Rule $\Rightarrow$ I, for example, corresponds to evidence abstraction:

$$[\Rightarrow \mathrm{I}]\ \frac{P, v : \pi \mid \Gamma \vdash M \hookrightarrow M' : \rho}{P \mid \Gamma \vdash M \hookrightarrow (\lambda_e v.M') : \pi \Rightarrow \rho}$$

Rule $\Rightarrow$ E corresponds to evidence application: for the term M to have a type $\pi \Rightarrow \rho$, there must have been some use of rule $\Rightarrow$ I, and thus, in the translation of M, some evidence abstraction. If we can prove π from P, then we can translate expression M to an application of the translation of M to the evidence for π.

$$[\Rightarrow \mathrm{E}]\ \frac{P \mid \Gamma \vdash M \hookrightarrow M' : \pi \Rightarrow \rho \quad P \Vdash e : \pi}{P \mid \Gamma \vdash M \hookrightarrow (M'\ e) : \rho}$$

To describe the computation of evidence for π, Jones relies on an evidence-annotated version of the entailment relation, of the form:

$$P \Vdash e : \pi,$$

where, as in the typing judgment, each predicate in $P_i \in P$ is paired with some identifier v_i. Intuitively, a derivation of this relation denotes that, if evidence for each of the P_i is substituted for each the of the v_i in e, then the resulting expression is evidence for π.

We might hope to take a similar approach in our setting by similarly annotating our entailment relation (§4.5) based on the evidence expressions we developed as part of our semantics of classes (§4.2). There are two obstacles to this approach. First, our evidence expressions do not account for hypotheses, as they lack any form of variable. More significantly, we do not represent all forms of proof directly in the evidence structure. This is irrelevant in some cases: for example, while there is no evidence expression that explicitly corresponds to excluding a predicate via functional dependences (Rule Excl-FD), such an argument can only prove negative predicates, which already have uniform evidence expressions. On the other hand, superclass constraints can (only) be used to prove positive predicates, but there is no uniform evidence expression in this case. As an illustration, consider the entailment

$$\text{Ord } t \Vdash \text{Eq } t.$$

Assuming the standard superclass relationship between the `Eq` and `Ord` classes, we would expect this entailment to hold, and indeed, we can construct such a derivation (using rules Super and Assume). However, there is no uniform construction of an evidence expression for Eq t from an evidence expression for Ord t.

One approach to these difficulties would be to define extended evidence expressions, augmenting the original form of evidence expressions with notions such as assumptions and superclasses. We could then define both a new set of proof rules, extended with evidence annotations, and a translation from the extended to the original evidence expressions. However, this approach is verbose (as it requires restating the entailment relation). Instead, we observe that each of the new forms of evidence corresponds to a particular proof mechanism. Thus, we can define a meaning function $[\![\cdot]\!]$ on entailment derivations, such that if Δ is a derivation of $P \Vdash Q$, then $[\![\Delta]\!]$ is a function from evidence for P to evidence for Q. This function can then be used in the place of the evidence value in Jones's scheme; that is, rule $\Rightarrow$ E would have the form

$$[\Rightarrow\text{-E}]\ \frac{P \mid \Gamma \vdash M \hookrightarrow M' : \pi \Rightarrow \rho \quad P \Vdash \pi}{P \mid \Gamma \vdash M \hookrightarrow (M'\,[\![\Delta]\!]) : \rho}$$

where Δ is the derivation of $P \Vdash \pi$.

Let Δ be a derivation of $A \mid X \vdash P \Vdash Q$ or of $A \mid X, \alpha \vdash P \Vdash \pi$, where we assume that $\mathit{valid}(A \mid X)$, and let e be evidence for P. We define $[\![\Delta]\!]e$ by cases, as follows. We begin with the cases for $A \mid X \vdash P \Vdash Q$.

- Case Each: we have a derivation of the form:

$$\Delta = \frac{\Delta_1 = \dfrac{\vdots}{A \mid X \vdash P \Vdash Q_1} \quad \cdots \quad \Delta_n = \dfrac{\vdots}{A \mid X \vdash P \Vdash Q_n}}{A \mid X \vdash P \Vdash \{Q_1, \ldots, Q_n\}}$$

  We define

$$[\![\Delta]\!]e = \langle [\![\Delta_1]\!]e, \ldots, [\![\Delta_n]\!]e \rangle.$$

- Case Assume: we have a derivation of $A \mid X \vdash P \Vdash \pi$, where $\pi \in P$. We define $[\![\Delta]\!]e = e_i$ such that π is the i^{th} predicate in P.

- Case Super: we have a derivation of the form

$$\Delta = \frac{\forall \vec{t}.\, \pi' \Rightarrow \pi \in X \quad S \in \mathit{GSubst}(\vec{t}) \quad \Delta_1 = \dfrac{\cdots}{A \mid X \vdash P \Vdash S\,\pi'}}{A \mid X \vdash P \Vdash S\,\pi}$$

  The structure of superclasses requires that π' is positive; thus, we have that $[\![\Delta_1]\!]e$ cannot be $\bullet$ (which proves negative predicates), nor can it be some tuple $\langle e_i \rangle$ (which proves multiple predicates), and so we must have that $[\![\Delta_1]\!]e = d\,e'$ for some evidence constructor d and evidence expression e'. Because $d\,e'$ is evidence for $S\,\pi'$, we can conclude that:

- $(d : \forall \vec{u}.\, \pi_d \Leftarrow P_d) \in \mathit{Clauses}(A)$ (where we assume that the $\vec{u}$ do not appear in $S\,\pi$ or $S\,\pi'$);
- There is some $T \in \mathit{Subst}(\vec{u})$ such that $T\,\pi_d = S\,\pi'$; and,
- e' is evidence for $T\,P_d$.

We can now appeal to the acceptability of $A \mid X$ to construct evidence for π. Observe that, because $S\,\pi' = T\,\pi_d$, there is some most general unifier U such that $U\,\pi' = U\pi_d$, and both S and T factor over U. From the clause preservation check (Equation 4.6), then, we have that there is some derivation Δ_d of $A \mid X \vdash U\,P \Vdash U\,\pi$. Because entailment is closed under substitution (Theorem 4.11), we have that there is some derivation Δ'_d of $A \mid X \vdash T\,P_d \Vdash S\,\pi$. Finally, having described the computation of Δ'_d and e' from our original derivation Δ and evidence e, we can define $[\![\Delta]\!]\,e = [\![\Delta'_d]\!]\,e'$.

- Case Axiom: we have a derivation of the form:

$$\Delta = \cfrac{\alpha \in A \qquad \Delta_1 = \cfrac{\vdots}{A \mid X, \alpha \vdash P \Vdash \pi}}{A \mid X \vdash P \Vdash \pi}$$

  We define $[\![\Delta]\!]\,e = [\![\Delta_1]\!]\,e$.

We continue with the cases for $A \mid X, \alpha \vdash P \Vdash \pi$.

- Case Match: we have a derivation of the form

$$\Delta = \cfrac{S \in \mathit{Subst}(\vec{t}) \quad S\,\pi' = \pi \qquad \Delta_1 = \cfrac{\vdots}{A \mid X \vdash P \Vdash S\,P'}}{((d : \forall \vec{t}.\, \pi' \Leftarrow P')\ ;\ \alpha) \vdash P \Vdash \pi}$$

  If π is positive, we define $[\![\Delta]\!]\,e = d\,([\![\Delta_1]\!]\,e)$; otherwise, we define $[\![\Delta]\!]\,e = \bullet$.

- If derivation Δ is by rule Excl-FD, we have that goal predicate π is negative, and so define $[\![\Delta]\!]\,e = \bullet$.

- A derivation Δ by any of the remaining cases (for rules Step-Contra, Step-Pos, and Step-Neg) has a subderivation Δ_1 of $\alpha \vdash P \Vdash \pi$; in each case, we define $[\![\Delta]\!]e = [\![\Delta_1]\!]e$.

We show that this interpretation of the entailment relation is consistent with our models of type classes, and thus with our semantics of overloading.

Theorem 6.1. *If Δ is a derivation of $A \mid X \vdash P \Vdash Q$, for ground predicates P and Q, and $\mathcal{G}$ is some model such that $\mathcal{G} \models A \mid X$ and $\mathcal{G} \models e : P$, then $\mathcal{G} \models [\![\Delta]\!]e : Q$.*

The proof is by induction over the structure of the derivation of $A \mid X \vdash P \Vdash Q$; each case is immediate from the definitions of the meaning function (in this section) and the forcing relation (§4.3).

6.2 SIMPLIFICATION

The implementation of the Habit predicate solver also provides functionality beyond checking predicate entailments. One such function is context simplification. Type inference for a type system like OML may generate types that include repeated or equivalent predicates. Jones gives a collection of examples in his discussion of context simplification [31], and we reproduce some of them here.

- Given a term such as `(λx y z → x + y + z)` type inference may infer two different copies of the constraint Num t, where t is the type of parameters `x`, `y`, and `z`.

- Alternatively, given a term such as `(λx y → x + y == 1)`, type inference might infer both the constraint Num t, arising from the use of `(+)`, and Eq t, arising from the use of `(==)`. However, there is (typically) a superclass constraint $\forall t.\,\text{Num}\ t \Rightarrow \text{Eq}\ t$, so the context consisting of both constraints is equivalent to the context containing only the `Num` constraint.

Similarly, inferred types may include either trivially provable, or contradicted predicates. Again, we draw on Jones's examples.

- A term such as `(λx → x == 'c')` would give rise to a constraint Eq Char; however, such an instance is (typically) present, and so the constraint could be discharged instead of included in the inferred type.

- Alternatively, a term such as `(λf g x → f == g || f x == g x)` would give rise to a constraint Eq $(a \to b)$ where $a \to b$ is the type of arguments `f` and `g`. However, in Habit, we expect an instance Eq $(a \to b)$ `fails`, as equality for functions is not, in general, decidable, so we could safely indicate a type error instead of inferring an unsatisfiable constraint for this term.

- Finally, a term `(λc → c + 'a')` would give rise to a constraint `Num Char`, for which there is typically no instance. Jones argues that this term could be seen as a type error—as it assumes a concrete instance that does not exist—but could also be seen as being perfectly well typed, if only usable in a context in which such an instance has been defined.

The Haskell Report places restrictions on the form of contexts, ensured by a process called *context reduction*, that can help to resolve some of the above issues [49]. These restrictions require that each predicate in a type must either be of the form $C\ t$ or of the form $C\ (t\,\tau_1\,\tau_2\ \ldots)$, where, in each case, t is some type variable. Applying this approach to Habit would resolve many of the issues raised above: for example, neither the predicates Num Int nor Num Char is of the necessary form, and so must be either discharged (in the first case), or lead to a type error (in the second). This process can also require the application of instances to some contexts. For example, the predicate Eq (a, b) is not of the required form, and so would have to be reduced to the pair of predicates (Eq a, Eq b) (assuming the typical instance of `Eq` for pairs). However, this criterion does not extend naturally to the features of the Habit class system.

- It is not immediately clear how to extend this criterion to multi-parameter type classes, such as the (:<:) class in our example of extensible data types (§3.4). The most obvious approach might be to require that each argument be of one of the approved forms (that is, either a type variable t or the application of a type variable $t\,\tau_1\,\ldots$. However, this would exclude constraints such as Int :<: t, which are integral to the utility of the (:<:) class.

- Context reduction also interacts poorly with instance chains. For example, consider the instance chain

  ```
  instance C [Int]
  else C [t] if C t
  ```

  With this chain, it is not possible to reduce a predicate C $[t]$ to a predicate of the desired form without first knowing that instantiation of type t. Similar problems arise in Haskell extended with overlapping instances.

As a consequence, the definition of the Habit language provides no such restrictions on the form of contexts, and defines no context simplification process.

We believe it is still valuable to provide predicate simplification in our implementation, however, both to (in many cases) improve the presentation of inferred types, and to avoid duplicating solving effort. We define simplification rules for a predicate set P as follows:

1. We eliminate from P any predicates that are either duplicated, or that are consequences of the superclass constraints and other predicates in P, or that can be proven from the axioms and the other predicates in P.

2. We replace a predicate π by the sequence Q if Q are the hypotheses of the only axiom clause that can prove π. Formally, we replace π by Q if there is some axiom $\alpha = \xi_1; \xi_2; \ldots$ and some i such that:

- Each clause $\xi_j = (d_j : \forall \vec{t}_J.\, \pi_j \Leftarrow P_j)$ with $j < i$ cannot apply to π, either because it neither matches nor contradicts π, or because the appropriate instantiation of one of the predicates in P_j is inconsistent with the axioms or the other predicates in P.
- In clause $\xi_i = (d_i : \forall \vec{t}_i.\, \pi_i \Leftarrow P_i)$, there is some $S \in \mathit{Subst}(\vec{t}_i)$ such that $S\,\pi_i = \pi$ and $S\,P_i = Q$.
- Each clause ξ_j with $j > i$ cannot apply to π, because it neither matches nor contradicts π.

We define the simplifications of P, written $\mathit{simpl}(P)$, as the least set such that

- $P \in \mathit{simpl}(P)$; and,
- If $Q \in \mathit{simpl}(P)$ and Q' arises from Q by one of the simplification rules, then $Q' \in \mathit{simpl}(P)$.

We can show that simplification does not change the meaning of type schemes:

Theorem 6.2. *For any valid program $A \mid X$, context P, and $Q \in \mathit{simpl}(P)$, we have both that $A \mid X \vdash P \Vdash Q$ and that $A \mid X \vdash Q \Vdash P$.*

From the definition of $\mathit{simpl}(P)$, we have that, if $Q \in \mathit{simpl}(P)$, then there is some sequence $P, Q_1, Q_2, \ldots, Q$ such that each element of the sequence is in $\mathit{simpl}(P)$ and each element arises from the preceding one by an application of one of the simplification rules. The proof is by induction over the length of this sequence; each step is immediate from the definition of the simplification rules and of the entailment relation (§4.5).

This process may seem complex. However, its implementation is quite direct, as it consists of the same steps that are used to discharge predicates. Thus, our concrete implementation of simplifying a collection of predicates P amounts to attempting to solve predicates P and observing the result. Should the solver have disproved any of the predicates in P, we indicate a type error. Otherwise, we return the final goals the solver was unable to prove (after checking that later clauses cannot apply).

6.3 IMPROVEMENT

Another function of the Habit predicate solver is computing improving substitutions, which capture type equalities (represented by substitutions) that must hold for given predicates to be satisfiable. In our current implementation of the Habit class system, improving substitutions arise exclusively from functional dependencies; we describe an additional source of improvements as future work (§7.4). For example, the Habit prelude includes a class

```
class BitSize t n | t → n where ...
```

where `BitSize t n` holds if a value of type `t` can be represented in `n` bits. Typical instances of this class include

```
instance BitSize Unsigned 32
```

for unsigned integer values, and

```
instance BitSize (Bit n) n
```

for bit vectors of length n. Given these instances, consider an entailment such as $\emptyset \Vdash \text{BitSize Unsigned}\, m$. We cannot discharge this predicate from the given instances by any of the rules for entailment: in particular, there is no substitution for the type variables of the `BitSize` instance for `Unsigned` such that `32` matches m. However, from the functional dependency constraint on the `BitSize` class, and because we can prove $\text{BitSize Unsigned}\, 32$, we know that for any predicate of the form $\text{BitSize Unsigned}\, \tau$ to hold, type τ must be 32. Applying this observation to the given entailment, we can generate the improving substitution $[32/m]$, and can conclude that the predicate $\text{BitSize Unsigned}\, m$ can be discharged under this improving substitution.

The situation can be more complex when multiple predicate are involved. For example, consider the following pair of classes and set of instance declarations:

```
class C t u v | t → u
class D u v | u → v

instance C Int Float Bool
instance C Int Float Char
instance D Float Bool
```

Note that, in a predicate with class `C`, the first parameter determines the second, but not the third. Thus, knowing the first parameter can be sufficient to compute an improving substitution for the second, but cannot be enough to discharge the predicate. As an example, consider the following entailment:

$$\emptyset \Vdash \{\mathrm{C\ Int}\ u\ v, \mathrm{D}\ u\ v\}.$$

As in the prior example, neither of these predicates can be discharged; however, as there are functional dependencies on classes `C` and `D`, we can search for improving substitutions. The functional dependency on class `C` allows us to conclude that the predicate C Int $v\ v$ can only hold under the substitution $[\mathrm{Float}/u]$; note that this is not, by itself, enough to discharge the predicate, as we do not know the instantiation of v. However, by applying this improving substitution to the second predicate, we obtain the predicate D Float v; from the functional dependency on class `D`, we can conclude that this predicate only holds under the substitution $[\mathrm{Bool}/v]$. At this point, we do have enough information to discharge the `D` predicate. Finally, we can apply the latter substitution to the `C` predicate, obtaining C Int Float Bool, which can be discharged.

The previous example demonstrated a case where we can compute improving substitutions from predicates even without necessarily being able to discharge those predicates. On the other hand, the interaction between instance chains and functional dependencies can lead to cases in which such improving substitutions cannot be computed before the predicates are discharged. For example, assume there is some class `C`, type constants `True` and `False`, and the following definition of a class `XC`, representing the characteristic function of class `C`:

```
class XC t b | t → b

instance XC t True if C t
else XC t False
```

The instance of class `XC` does not violate its functional dependency: for any given type τ, we can prove at most one of C τ and C τ `fails`, so either XC τ True or XC τ False may hold, but both cannot. However, given an entailment $\emptyset \Vdash \text{XC}\ \tau\ b$, we cannot conclude any improving substitution for b until we have determined whether C τ can be proven or disproven.

We can give an intuitive description of the improvement rules for a context P as follows:

1. For each pair of predicates $C\ \vec{\tau}$ and $C\ \vec{\upsilon}$ in P, and each functional dependency $(Y \leadsto Z) \in \text{fd}(C)$, such that $\vec{\tau}|_Y = \vec{\upsilon}|_Y$, if there is some most general unifier U such that $U\ \vec{\tau}|_Z = U\ \vec{\upsilon}|_Z$, then U is an improving substitution for P. If there is not such a U, then P is unsatisfiable.

2. For each predicate $C\ \vec{\tau} \in P$, and each functional dependency $(Y \leadsto Z) \in \text{fd}(C)$, suppose there is some axiom $\alpha = \xi_1; \xi_2; \ldots$ and index i such that the following conditions hold.

   - For each clause $\xi_j = (d_j : \forall \vec{t_j}.\ C\ \vec{\tau_j} \Leftarrow P_j)$ with $j < i$, either there is no substitution $S \in \mathit{Subst}(\vec{t_j})$ such that $S\ \vec{\tau_j}|_Y = \vec{\tau}|_Y$, or if there is such an S, then some predicate in $S\ P_j$ is inconsistent with the axioms or with the predicates in P.

- In clause $\xi_i = (d_i : \forall \vec{t_i}.\ C\ \vec{\tau_i} \Leftarrow P_i)$, there is some $S \in \mathit{Subst}(\vec{t_i})$ such that $S\,\vec{\tau_i}|_Y = \vec{\tau}|_Y$, and each predicate in $S\,P_i$ can be proven from the axioms and the assumptions in P.

  In this case, if there is a most general unifier U such that $U\,(S\,\vec{\tau_i}|_Z) = U\,\vec{\tau}|_Z$, then U is an improving substitution for P; otherwise, P is unsatisfiable.

We define the improving substitutions induced from context P, written $\mathit{impr}(P)$, as the substitutions such that:

- The identity substitution is in $\mathit{impr}(P)$; and,
- For an substitution $S \in \mathit{impr}(P)$, if S' arises from $S\,P$ by one of the improvement rules, then $S' \circ S \in \mathit{impr}(P)$.

To formally characterize the soundness of induced improving substitutions, we begin by introducing a notion of the ground instances of a set of predicates, parallel to the notion of the ground instances of a qualified type used in the prior chapter (§5.3.2). Given some basis $A \mid X$, we define:

$$\lfloor P \rfloor = \{S\,P \mid A \mid X \vdash \emptyset \Vdash S\,P\}.$$

We can now state that induced improvements neither strengthen nor weaken predicate sets:

Theorem 6.3. *For any valid program $A \mid X$ and context P, if $S \in \mathit{impr}(P, A \mid X)$, then $\lfloor P \rfloor = \lfloor S\,P \rfloor$.*

From the definition of $\mathit{impr}(P)$, we have that, if $S \in \mathit{impr}(P)$, then $S = S_n \circ S_{n-1} \circ \cdots \circ S_1$, where each $S_i \in \mathit{impr}(P)$ and each S_i arises from the improvement rules applied to $S_{i-1}\,P$. The proof is by induction on the sequence S_i; each step is immediate from the improvement rules and the forcing relation for functional dependencies (§4.3.3).

6.4 IMPLEMENTATION MECHANISMS

In this section, we give a brief overview of our implementation of the Habit predicate solver. Our model of the Habit predicate system relied on our notions of proof and refutation being intuitionistic, and our description of the proof rules is quite similar to resolution or tableaux methods of proof. It may thus be surprising that we have not relied on an existing first-order prover. Our particular implementation was developed during our experimentation and research on instance chains. Nevertheless, we believe that the continued use and development of a specialized prover is justified by the challenges inherent in translating instance chains and predicate entailments to first-order formulae and translating the resulting proofs to class method implementations. In particular:

- The translation of instance chains into first-order formulae is not trivial. For example, for the following instance chain:

  ```
  instance C (Maybe t) if D t
  else C t if E t
  ```

  we could construct an equivalent first-order formula, such as:

  $$\forall t.((\exists u.(t = \mathrm{Maybe}\,u \ \wedge\ \mathrm{D}\,u) \implies \mathrm{C}\,t) \quad \wedge \\ (\forall u.(t \neq \mathrm{Maybe}\,u \ \vee\ \neg\mathrm{D}\,u) \ \wedge\ \mathrm{E}\,t \implies \mathrm{C}\,t)).$$

  We make several observations about this translation process. First, the implementation of the translation itself could have bugs; thus, assurance about the underlying solver would not transfer immediately to assurance about the Habit implementation. Second, we anticipate some need to adapt the translation to any specific prover; for example, the particular translation given here obscures the connection between the hypotheses of the first and second implications, and encodes (decidable) unification and matching in terms of existential quantification and equality, which require an underlying logic that is not (in general) decidable.

- Even given a successful and efficient translation of instance chains and predicate entailments into an underlying logic, there would still be potentially significant effort in translating the resulting proofs into suitable implementations of class methods. As described earlier in this chapter (§6.1), the particular structure of the proof rules that we have given is closely tied to our implementation of ad-hoc polymorphism. On the other hand, given a proof of C τ from the translation above, we would have to determine which clause was used to prove the predicate, and extract the semantically significant portions of its subproofs. This would introduce further complexity and further reduce the assurance derived from the correctness of the underlying prover.

- Finally, while we believe that, modulo the concerns above, predicate entailment and program acceptability could be encoded in a suitable first-order logic, simplification and improvement are less obviously equivalent to typical first-order solving. While we could potentially forego simplification, improvement plays a central role in the Habit type system.

Given these factors, we have continued our development of a specialized Habit predicate solver. However, this has introduced some challenges of its own. We have been able to draw on well-known proof procedures for those parts of the solver not specific to instance chains; however, this does not exclude the possibility of implementation errors. More significantly, our implementation has neither been formally verified, nor (yet) subjected to significant testing, and thus lacks guarantees of its soundness or correctness. We believe that formal specification of our present implementation and continued exploration of the use of external provers are both valuable directions for future work.

The remainder of this section gives a brief overview of the structures and logic of the Habit predicate solver, and discuss some concerns that arose during its implementation. Our implementation effort has been more concerned with the correctness of the solver than its speed; thus, some of our implementation choices preserve the intuition of the solver at the cost of some perhaps-unnecessary effort.

Intuitively, the solver behaves as a function that, given a basis $A \mid X$ and an entailment $P \Vdash Q$, returns either

- A simplification of Q, empty in the case that Q is completely proven, along with an improving substitution; or,
- A proof that Q is unsatisfiable.

The latter case arises if the solver can prove some predicate π such that, for some $\pi' \in Q$, $A \mid X \vdash \pi \lightning \pi'$. However, there may be semantically unsatisfiable predicates for which no such conflicting predicate is provable; in these cases, while the solver will not discharge the predicates, it will also not be able to disprove them.

The solver's operation is defined primarily in terms of two primary data structure: the *forest*, which tracks the proofs the solver is attempting to build; and, the *trail*, which tracks the assumptions the solver has made. The solver's implementation is structured as a collection of local transformations of the forest and trail. We begin our explanation of the solver's mechanism with a worked example (§6.4.1), demonstrating the evolution of the forest and trail over the course of a simple proof. We then describe the data structures that represent the forest and trail (§6.4.2), the structure of the local transformations (§6.4.3), and the domain-specific transformations that implement the solver's proof search (§6.4.4). We conclude with some notes on the concrete implementation (§6.4.5).

As the solver is implemented in Haskell, the latter portions of this section may require more familiarity with Haskell idioms than has been required to this point.

```
class C t
class XC t b | t → b
class D t u | t → u

instance XC t True if C t
else XC t False

instance D Int Bool
```

Figure 6.1: Example program definitions

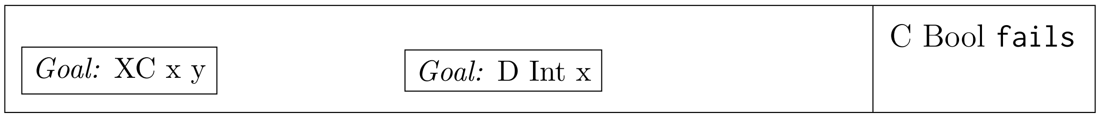
6.4.1 The Solver by Example

We begin with a simple example of the solver's operation; while contrived, it suffices to demonstrate many of the solver's transformations. We assume the simple collection of definitions, shown in Figure 6.1. These include an arbitrary class `C`, a class `XC` implementing the characteristic function of class `C`, and some type function `D`. We shall demonstrate the process the solver follows, given the entailment

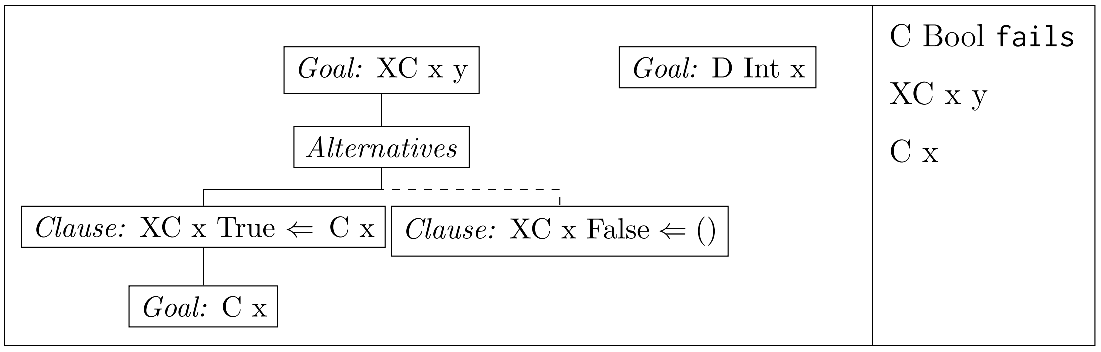
$$\mathrm{C\ Bool}\ \texttt{fails} \Vdash \mathrm{XC\ x\ y}, \mathrm{D\ Int\ x},$$

to conclude that the substitution $[\mathrm{Bool/x}, \mathrm{False/y}]$ improves the given predicates, and that the entailment holds under that assumption.

The solver's initial state is shown in Figure 6.2a. The forest is represented in the left portion of the diagram—in this case, it contains only the two starting goals. The trail is represented in the right portion of the diagram—in this case, it contains only the initial assumption. The solver begins by exploring the goal `XC x y`—we show the state after it has done so in Figure 6.2b. The axiom for class `XC` applies to the goal, modulo the functional dependency on class `XC`. The solver therefore expands the goal node with the two alternatives provided by that



(a) The initial solver state



(b) Solver state after exploring the first goal

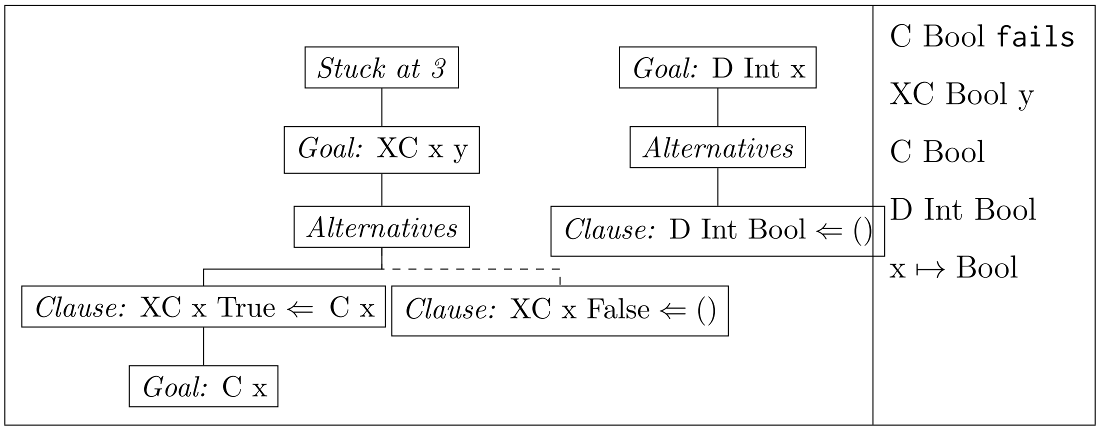
Figure 6.2: Example solver execution (part 1)

axiom, and begins to explore the first alternative. The dashed line to the second alternative represents that the solver is not considering that alternative yet. The first alternative in turn depends upon the goal `C x`; at this point in the process, the solver has no information to prove or disprove that goal, and there are no axioms for class `C`. As the solver encounters goals in the forest, it assumes that they hold; this is to allow for the possibility of improvements arising from the goals (§6.3); thus, both predicates `XC x y` and `C x` appear in the trail.

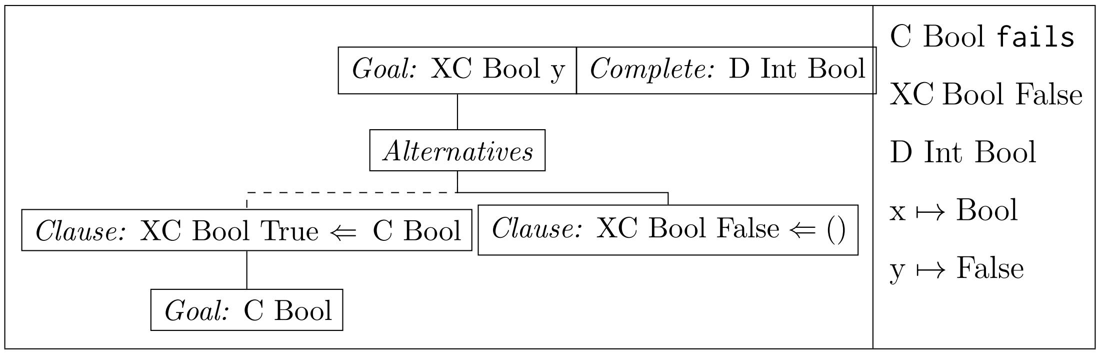
As the solver can make no further progress on the first goal, it moves to the second; we show the solver state after exploring the second goal in Figure 6.3a. The solver has introduced a new root node indicating that it was able to make no further progress on the first goal; this node captures the number of assumptions available when the solver failed to make progress. The axiom for `D Int Bool` applies to the goal `D Int x` modulo the functional dependency on class `D`; thus, the solver can expand the goal node by that axiom, and begin to explore the first (and only) alternative it provides. As that clause has no hypotheses, the solver can then commit to the improvement the clause introduced (that of `Bool` for `x`). Note that at this point, the solver has reached an inconsistent state: the assumption that `C x` held, arising from the use of the first alternative to prove `XC x y`, conflicts with the provided assumption of C Bool `fails`. The solver will discover this inconsistency and backtrack when it re-examines the `C x` goal.

Having proved the `D Int x` goal (under the current improvement), the solver collapses the tree to a proof and returns to the first goal. The stuck node indicates that there were three assumptions available when the solver was unable to make progress; as there are now more assumptions available, the solver removes the stuck node and attempts to explore the remaining goal, `C x`. Under the assumption $x \mapsto \text{Bool}$, this goal is equivalent to `C Bool`; however, we have the assumption that C Bool `fails`. Therefore, the solver attempts to backtrack, invalidating any assumptions based on the skipped alternative. In this case, there is another alternative available, so the solver marks the first alternative as skipped and moves to the second. The second can be trivially discharged, and so the proof can be completed. Figure 6.3b shows the solver state just before completion.

This example has simplified our current implementation in two ways. First, our implementation of backtracking is coarser than in this example: upon discovering that the goal `C x` was unsatisfiable, the solver would revert to the state immediately before the first alternative was explored, also reverting any progress on the `D Int x` goal. However, from that point the exploration of the second goal would proceed exactly as described above. We believe that refining our implementation of backtracking is important future work. Second, our implementation relies on a breadth-first search of the proof space, instead of the depth-first presentation here. In this example, this would not result in any difference in the steps the solver takes, but simply take them in a less immediately intuitive order.



(a) Solver state after exploring the second goal



(b) Solver state before completion

Figure 6.3: Example solver execution (part 2)

6.4.2 Forest and Trail

The previous section relied on intuitive notions of the solver's primary data structures. In this section, we will discuss the Haskell implementations of those structures; this will provide a foundation for the discussion of the solver algorithms in the following sections.

We define the forest through two mutually-recursive data types, `Node` and `Tree`. The `Node` type captures the structure of proofs themselves, (mostly) avoiding details of the solver's implementation. We give a simplified definition of the `Node` type in Figure 6.4, and discuss each of its constructors.

```
data Node = Goal     { goal        :: Pred
                     , solution    :: Maybe Tree }
          | Alternatives
                     { skipped     :: [Tree]
                     , current     :: Tree
                     , remaining   :: [Tree] }
          | Clause   { spin        :: Spin
                     , axiomName   :: AxId
                     , improvement :: Subst
                     , subtrees    :: Either [Pred] [Tree] }
          | Complete { spin        :: Spin
                     , proof       :: Proof }
          | Stuck    { subtree     :: Tree }
```

Figure 6.4: The `Node` datatype

- The `Goal` constructor captures solver goals; for example, given a query $P \Vdash Q$, the initial forest would consist of a `Goal` node for each predicate in Q. The proof of a goal is constructed in the `solution` field of its `Goal` node; initially, this field is `Nothing`, but it is replaced when the solver begins exploring the goal.

- The `Alternatives` constructor captures an ordered collection of ways to prove a goal, corresponding to an instance chain. The `skipped` field contains those alternatives already shown to be inapplicable, the `current` field contains the alternative being explored, and the `remaining` field contains further possibilities should the current alternative be inapplicable. The solver operates only on the `current` alternative; the `remaining` alternatives are not explored concurrently, and do not contribute to the trail, and any assumptions introduced by trees in the skipped alternatives are removed when the alternative is moved to the skipped list.

- The `Clause` constructor corresponds to a single clause within an instance chain. The `spin` field, which can be `Proving` or `Disproving`, captures whether the clause conclusions matched or conflicted with the goal. It may seem odd to include `Disproving` clauses at all. However, this allows us to provide useful programmer feedback—identifying inconsistent predicate sets instead of allowing type inference to generalize over them—while reducing duplicated solving effort. The `improvement` field contains any improving substitution that would be valid were the clause proved, but not before; the instance for the `XC` class (§6.3) would induce such an improvement for a predicate XC τ b, as the improvement for b can only be determined after proving (or disproving) C τ. Finally, the `subtrees` field holds the hypotheses of the clause. Note that `Clause` nodes may be constructed before they are explored—for example, if they are in the `remaining` field of an `Alternatives` node. When a `Clause` node is explored (that is, becomes the `current` alternative), the predicates in `subtrees` are converted to `Goal` nodes.

- The `Complete` constructor captures a finished proof; the `spin` argument is as in the `Clause` constructor, and the `proof` field captures the structure of the proof tree without the solver's internal metadata.

- The `Stuck` node captures a point where the solver could neither prove nor disprove some goal. (Our earlier example included the number of assumptions in the node; in fact, this is stored in the metadata common to all nodes, described below.) The solver may be able to make further progress, however, given additional predicate assumptions or refinement of type variables.

The `Tree` type combines a node with a collection of solver metadata; we provide a simplified definition in Figure 6.5. The metadata for a given node includes:

```
data Metadata = Meta { lastUpdated :: Int
                     , introduces  :: [Int]
                     , saved       :: Maybe SolverState }

data Tree    = Tree { nodeFrom     :: Node
                    , metaFrom     :: Metadata }
```

Figure 6.5: The `Tree` and `Metadata` datatypes

```
data Cursor = Cursor Path Tree
data Path = Forest [Tree] [Tree]
          | NodeP Node [Tree] Path [Tree] Metadata
```

Figure 6.6: The `Path` zipper data type

- A timestamp, `lastUpdated`, capturing the number of assumptions in the trail when this node, or its children, were last updated.
- The set of trail assumptions introduced based on this node. Because the solver may consider an individual node multiple times, it is important to avoid basing its eventual proof on the (circular) assumption that it holds.
- A saved copy of the solver state (consisting of the trail and forest, and discussed further below), for backtracking purposes.

Conceptually, the forest is a set of `Tree` values. However, as the solver is specified by local solver rewrites, it represents the forest by an encoding of a particular location in the forest, not just by the forest itself. We use Huet's zipper data structure [23] to encode paths in the forest, as shown in Figure 6.6. A `Cursor` contains the path to the current subtree (in its first argument) and the current subtree itself (in its second argument). Unlike Huet's path datatype, we do not

```
data Trail = Trail { substitution :: Subst
                   , assumptions  :: [(Int, Pred)]
                   , ignored      :: [Int]
                   , now          :: Int }
```

Figure 6.7: The `Trail` datatype

create a separate constructor for each constructor in the node type; instead, we define two functions, `children` and `withChildren`, with the following types:

```
children     :: Node → [Node]
withChildren :: Node → [Node] → Node
```

These functions are responsible for extracting the children of a node, and for reconstructing a node with new children, respectively. Their implementations are straightforward, based on the structure of nodes, and their use simplifies the definition of many of our cursor manipulation operations. For example, the code to move the cursor up in the tree is:

```
upwards (Cursor (NodeP n left up right meta) here) =
    Cursor up (Tree (withChildren n
                        (reverse left ++ here : right))
                   meta)
```

This definition works regardless of the constructor used to build node `n`. This also simplifies the extension of the `Node` datatype—as long as appropriate cases are added to the `children` and `withChildren` functions, the zipper types and operations remain unchanged.

The second component of the solver's state is the trail, which captures the assumptions made during the proof. A simplified definition is given in Figure 6.7. We distinguish two forms of assumptions: assumed predicates, stored in field

```
data SolverState = St { here  :: Cursor
                      , trail :: Trail }
```

Figure 6.8: The `SolverState` datatype

`assumptions`, and type equalities, stored in field `substitution`. Each assumption, whether a predicate or type equality, is associated with an index, as if they were included in a uniform sequence. These indices identify the assumptions introduced by a given node (in the solver metadata, discussed above). They are also used to identify assumptions whose use would introduce circularity (as they were assumed as a result of the goals the solver is currently trying to prove); this list of assumptions is stored in the `ignored` field in the trail. The `now` field contains the current time—that is, the number of assumptions in the (uniform representation of the) trail.

Finally, the `SolverState` datatype (Figure 6.8) pairs a position in the forest, represented by a `Cursor`, with the current trail.

6.4.3 Generic Tactics

We structure the Habit predicate solver as a series of local transformations to the `SolverState`. These range from simple and generic, such as moving the cursor, to complex and domain-specific, such as applying the available axioms to the current goal. We also define a set of combinators for sequencing, repeating, or choosing among transformations. Collectively, we refer to these transformers as "tactics," as they are the individual components of our general solving strategy, and play a similar, if simplified, role to the tactics in theorem proving tools. This section gives an overview of the structure of tactics, and describes their combinators; the next section provides intuitive descriptions of the more complex, domain-specific tactics.

```
data Tactic t = Tactic { runTactic :: SolverState →
                                       (TacticResult t, SolverState) }
data TacticResult t = Progress t
                    | NoProgress
                    | Exit Reason
data Reason = Done | CantProgress | Failed
```

Figure 6.9: `Tactic`s and associated types

```
instance Monad Tactic
    where return r = Tactic (λst → (Progress r, st))
          t >>= f  = Tactic g
              where g st = case runTactic t st of
                             (Progress r, st') → runTactic (f r) st'
                             (NoProgress, _)   → (NoProgress, st)
                             (Exit r, st')     → (Exit r, st')
```

Figure 6.10: `Monad` instance for the `Tactic` type constructor

Figure 6.9 gives the type of tactics. Each tactic, given the current solver state, generates both a `TacticResult t` and a new solver state. The tactic result indicates whether the tactic was applicable to its input state. A result of `Progress` indicates that the tactic was able to run, and contains an additional return value of type `t`. A result of `NoProgress` indicates that the tactic did not apply to its input state. The `Exit` result is used to terminate the solver; this can be either because the goals are proven (constructor `Done`), because the solver can make no further progress (constructor `CantProgress`), or because one of the goals has been disproved (constructor `Failed`).

Tactics form a monad, similar to the composition of state and error monads [30].

```
orElse :: Tactic a → Tactic a → Tactic a
orElse t0 t1 = Tactic f
    where f st = case runTactic t0 st of
                   (NoProgress, _) → runTactic t1 st
                   r               → r

try :: Tactic () → Tactic ()
try t = t `orElse` return ()

whileProgressing :: Tactic () → Tactic ()
whileProgressing t = try (t >> whileProgressing t)
```

Figure 6.11: Tactic combinators

The first tactic combinator, sequencing, is provided by the bind (`>>=`) method of the `Monad` instance for tactics, given in Figure 6.10. Note that, while a `NoProgress` result resets to the initial state, an `Exit` result does not. Thus, `Exit` does not behave quite like an error in the Error monad.

We define several other tactic combinators for handling progress, some of which are shown in Figure 6.11. The most basic is the `orElse` combinator: the tactic `` t0 `orElse` t1 `` is equivalent to tactic `t0` if `t0` makes progress, or to `t1` otherwise. We define two additional tactics using `orElse`: the tactic `try t` always makes progress, even if `t` does not, while the tactic `whileProgressing t` repeats `t` until it stops making progress. (The definition of `whileProgressing` uses the combinator `>>`, defined by `m >> n = m >> λ_ → n`.)

6.4.4 Domain-Specific Tactics

We also define a number of domain-specific tactics, each implementing particular parts of the solving process. As they are more involved, we will give intuitive descriptions here instead of reproducing their code directly.

- The `applyTrail` tactic updates the current goal with the assumptions in the trail, discharging the goal if it has already been assumed, and applying any improving substitution.

- The `assume` tactic adds a new assumption, and its logical consequences, to the trail. We derive the consequences of an assumption from three sources, handled by the three tactics `improvePairwise`, `improveFromAxioms`, and `applyRequirements`.

- The `improvePairwise` tactic compares a new assumption to the other assumed predicates, searching for any improving substitutions; this corresponds to the first of the improvement rules gives earlier (§6.3).

- The `improveFromAxioms` tactic compares a new assumptions to the axioms, searching for any improvement that can be justified from the axioms and the known assumptions; this corresponds to the second of the improvement rules. Note that this will not compute improvements that depend on discharging axiom hypotheses; those are introduced only once the relevant hypotheses are proven. This corresponds to the second of the improvement rules given earlier.

- The `applyRequirements` tactic determines whether the new assumption is the hypothesis of any superclasses, and, if it is, adds the conclusions of those superclasses to the trail as well. This corresponds to use of the Super rule of entailment (§4.5).

- The `applyAxiom` tactic attempts to apply the given axiom to the current goal node (and makes no progress if the current node is not a goal). Should an axiom apply, the tactic constructs a subtree containing an `Alternatives` node with one `Clause` child for each applicable clause in the axiom.

Additionally, we define several further domain-specific tactics encapsulating proof state manipulations.

- The `expand` tactic attempts to make progress at the current point in the tree. This tactic is unable to progress when applied to `Complete` nodes; for `Clause` and `Alternative` nodes, it moves to the first child node and invokes itself recursively. For `Stuck` nodes, if the node's `lastUpdated` metadata is before the current time, it replaces the node with its (now un-stuck) subtree and attempts to expand that subtree; otherwise, the tactic is makes no progress. Finally, for `Goal` nodes, it begins by attempting to apply the trail to the goal. If that does not discharge the goal, it introduces any assumptions from the current goal, and attempt to apply the axioms. If any axioms apply, it has made progress; otherwise it replaces the goal with a stuck node.

- The `collapse` tactic attempts to collapse subtrees of the current node. In many cases, this is straightforward—for example, if the subtree of a `Goal` node is complete, then the goal is complete as well; or, if one of the subtrees of a node is stuck, then the node itself is also stuck. The `Alternative` case is more complicated. If the `current` subtree is complete, proving the goal, then the `Alternative` node is complete as well. On the other hand, if the `current` subtree was not applicable to the goal (that is, the `current` subtree is a `Clause` node and one of its hypotheses has been disproven), then the proof search must move on to the next clause in the `remaining` list, while backtracking any assumptions that resulted from the unsuccessful `current` proof attempt. This is implemented by restoring to the solver state that was saved before the `current` clause was explored, and then replacing the `Alternatives` node, adding the failed attempt to the `skipped` list and moving the head of the `remaining` list to the `current` subtree. If the remaining list is empty, the current branch is stuck.

- The `advance` tactic attempts to move to the next point in the forest that can be explored. If the current node has right siblings, it moves to the first such sibling. Otherwise, it attempts to move up the tree, collapsing nodes as it goes, until it finds a node that has a right sibling. If there is no such node (that is, the cursor is at the right edge of the forest), the tactic resets to the left-most explorable node in the forest, exiting if there is no such node.

Finally, we can define the overall search strategy of the solver. Simple traversals are quite easy to define. For example, a depth-first traversal can be defined by

```
whileProgressing (whileProgressing expand >> advance)
```

which expands the current node as far as possible before moving to the next node in the tree. Alternatively, a breadth-first traversal can be defined by

```
whileProgressing (try expand >> advance)
```

Either approach will terminate as long as there is no infinite proof, as there will eventually be no node that can be expanded, and so `advance` will not make progress. However, as rule STEP-CONTRA requires that only one hypothesis be contradicted, there may be finite proofs that a depth-first approach does not find. The breadth-first approach is guaranteed to find the contradiction in such cases, and so we take this approach in our implementation.

6.4.5 Notes on the Implementation

This section has given an overview of the solver, and shown some of the data structures and algorithms that make up its implementation. We have omitted much discussion of the syntax of predicates and axioms and the details of unification, as these notions are standard. We have also avoided going into detail about the implementation of instance validation, as it closely follows our earlier formal description (§4.4).

Function	*Lines of code*
Syntax and substitutions	423
Solver implementation	867
Instance validation	321
REPL	443
Total	2054

Figure 6.12: Code size of solver implementation

In addition to these functions, our implementation of the solver also provides a simple interpreter (or REPL) for specifying class definitions and proving entailments. We use this environment both for prototyping new solver features and for testing changes to the solver code.

Figure 6.12 gives aggregate lines of code for various solver functions—syntax, the solver implementation itself, instance validation, and the REPL. We also maintain a small, but growing, array of tests, including all of the examples in this dissertation.

6.5 RELATED WORK

There is much work on translation-based approaches to overloading, which we have summarized in the prior chapter (§5.5). Our approach to superclasses, however, differs from much of the previous work in this area. Wadler and Blott [70] introduce superclasses as an abbreviation—for example, they suggest that the constraint `Ord t` might abbreviate the context `Ord t, Eq t`. Their translation does not account for superclasses; although we can surmise that, in their scheme, any use of a subclass will also include dictionaries for each of its superclasses, whether or not they are necessary. Jones [28] describes including (references to) superclass dictionaries within each of their subclasses; this is the approach taken by most

Haskell compilers. Our technique could describe his approach, but is more flexible. For example, the Habit specializer does not combine subclass and superclass evidence directly, but looks up superclass evidence as needed. We will discuss the advantages to the Habit approach further in the next chapter, when we discuss our generalization of superclasses (§7.2).

Our notion of simplification is similar to the Haskell notion of context reduction, extended to take account of the new features of the Habit class system, and similar systems are implemented by existing Haskell compilers. In their discussion of overlapping instances in Haskell [50], Peyton Jones et al. describe the conflict between overlapping instances and context reduction as specified in the Haskell report. Similar concerns motivated the definition of our simplification rules (§6.2), as multiple clauses in a given instance chain may be used to discharge a particular predicate. Jones characterizes the interaction of simplification and type inference [31], giving a rule that allows type inference to replace contexts with equally strong contexts; Theorem 6.2 allows the use of our simplification rules to compute such equally strong contexts.

Jones proposed improvement as a generalization of the use of the satisfiability of predicate sets to improve the accuracy of inferred principal types [31], and gives a rule allowing the application of improving substitutions during type inference. He later proposed that functional dependencies could provide a general way to compute improvements from type class predicates; our improvement rules (§6.3) are similar to his, but extended to take account of instance chains. Theorem 6.3 shows that substitutions computed using our rules meet Jones's definition of improving substitutions. Sulzmann et al. [63] give an alternative approach to the use of functional dependencies in type inference, based on a translation to constraint handling rules. As we have previous argued (§4.6), however, their translation limits the instances that can populate classes with functional dependencies beyond the requirements of the dependencies themselves.

Automated deduction for intuitionistic logics is a well-studied area, and our proof search follows standard approaches, adapted to the particular interpretation of instance chains. We found Fitting's textbook [10] and Waller and Wallen's summary of tableaux for intuitionistic logics [67] particularly helpful in understanding the domain and the particulars of our solver.

7. FUTURE WORK

Over the course of this dissertation we have described instance chains, a new feature to support type-class programming in languages like Haskell and Habit. In particular:

- We have presented a varied collection of examples, demonstrating both the difficulties encountered in Haskell type-class programming, and the expressiveness of instance chains.
- We have developed a formal semantics of type classes and their predicates. We have built sound and computable acceptability and entailment judgments, describing whether sets of class and instance declarations have models, and which predicates are proved from those declarations if they can be modelled.
- We have extended Ohori's semantics of Core ML to provide a semantics for overloaded expressions, and have proved the OML type system, instantiated with our entailment relation, is a sound approximation of that semantics.
- We have described the implementation of instance chains in the context a prototype compiler for Habit.

In this chapter, we conclude by describing several areas of further exploration that we have identified during our development of instance chains:

- Refinements to our notion of validation, allowing more programs without compromising the consistency or coherence of the class system (§7.1);

- A generalization of superclasses, that allows programmers to better define and enforce the intended semantics of classes (§7.2);
- An extension of our semantics, providing further tools to reason about the meaning of programs with overloading (§7.3); and,
- Some non-parametric proof techniques, extending the expressiveness of the predicate language (§7.4).

7.1 REFINING ACCEPTABILITY

We have described a syntactic notion of acceptability, and shown that it is sufficient to guarantee that programs have models (§4.4). As we intend the acceptability check to be decidable, it must be a conservative approximation of the semantic definitions of consistency and coherence. We believe that there are, however, several ways in which our definition could be relaxed without compromising the model existence proof (Theorem 4.1).

- Our definition of acceptability treats clauses in instance chains independently; that is, we do not draw on the fact that, for a given clause to apply to a predicate, the clauses before it must be inapplicable to that predicate. A more precise approach would be to take the preceding clauses into account, but doing so raises several challenges. The first challenge is to develop a method to represent negative syntactic constraints on types. For example, in the following instance chain

  ```
  instance C (Maybe t)
  else C t
  ```

  the second clause can only apply to types that are not of the form Maybe τ for some type τ. To pursue this approach, we would need to develop representations and proof rules for such limitations on the instantiation of type variables. The second challenge is to find a mechanism to support true disjunctions in clause contexts. For example, in the following instance chain

```
instance C t if D t, E t
else C t if F t
```

the second clause can apply only if the first clause does not—that is, we can think of the second clause as an (independent) axiom of the form

$$\forall t.\mathrm{C}\ t \Leftarrow (\mathrm{F}\ t \land (\mathrm{C}\ t\ \texttt{fails} \lor \mathrm{D}\ t\ \texttt{fails})),$$

where we use a disjunction to capture the two ways that the first clause could be skipped. Such disjunctions do not occur normally in predicate contexts, as giving the solver a free choice of which disjunct to prove could compromise coherence. This example does not give rise to such a concern, however, because the evidence for failing predicates is uniform. A treatment of the second clause that takes the first into account would require some approach to disjunction, either through extension of the solver, or by enumeration of each disjunct individually. The first approach would require more changes to our existing formalization and implementation, but might lead to interesting additional results; the second would be simpler, but might not scale well to larger examples.

- Our definition of acceptability also requires that clauses in distinct chains neither syntactically unify nor conflict. We could relax this constraint by taking the contexts of clauses into account. For example, given the two separate one-clause chains:

  ```
  instance D t if C t
  instance D t if C t fails
  ```

  our current criteria would reject these instances, because their conclusions unify. However, there is no possibility of incoherence resulting from these instances, as we can have no type `t` such that both `C t` and `C t fails` are provable. We have identified two consequences of attempting to take contexts into account in checking overlap and conflict among instances. First, we must adapt our proofs, and our notion of simplification, to reflect that different instance chains may syntactically apply to the same predicate. Second, we must define some notion of when two contexts are provably inconsistent; as we would intend such a criterion to be well defined and decidable, it must necessarily be incomplete. There are obvious notions—such as finding a pair of conflicting predicates. However, this may miss cases that a programmer would expect to be inconsistent. For example, consider the following trio of instances:

```
instance D t fails if C t
instance E t if C t
instance E t if D t
```

The second and third instances do not introduce incoherence, even though their contexts, `C t` and `D t` do not conflict, as, for any ground instance of `C t`, the first instance guarantees that there is a corresponding instance of `D t fails`.

- Finally, there may be cases in which the overlap check can be relaxed, as overlapping proofs would not compromise the coherence of the models. Such cases include instances asserting negative predicates, as negative predicates generate no evidence values, and instances for classes with no methods, such as the `Lt` or `Gcd` classes (§3.2.1).

7.2 GENERALIZING SUPERCLASSES

Our presentation of superclasses (§4.3.3) closely mirrors the functionality of Haskell superclasses. Working on the formalization and implementation of superclasses has suggested a new, more powerful generalization of superclasses, which we call *requirements*. While a superclass constraint has the form

$$\forall \vec{t}.\ C\ \vec{\tau} \Rightarrow D\ \vec{v},$$

for arbitrary classes C and D, such that the $\vec{v}$ are type variables, requirements allow arbitrary contexts in place of the hypothesis $C\ \vec{\tau}$, and arbitrary conclusions $D\ \vec{v}$ without restriction on $\vec{v}$.

We have developed a prototype implementation of requirements, and have begun their formalization. In the remainder of this section, we provide some motivating examples for requirements, and sketch the extensions to acceptability, entailment, and the semantics of overloading that are necessary to support them.

Example requirements. We begin with several motivating examples of requirements; we have used these examples, among others, to validate our prototype implementation. All superclasses fit within this more general framework. For example, we could express the typical superclass relationship between the `Eq` and `Ord` classes with the following requirement declaration

```
require Eq t if Ord t
```

In our prototype implementation, we continue to support the traditional syntax for superclasses; however, as part of the compilation pipeline, superclasses are translated into requirements.

We can also use requirements to capture relationships among classes that cannot be expressed using superclasses. For example, the standard Haskell prelude includes two classes, `Integral` and `Floating`, abstracting integer and floating-point operations. Intuitively, we would expect these classes to be disjoint; however, there is no way to capture such a constraint in Haskell. Thus, an expression such as `(λx → sin x `mod` 2)` would have the type

$$(\text{Integral}\ t, \text{Floating}\ t) \Rightarrow t \rightarrow t,$$

where the constraint should be unsatisfiable in practice, but would not trigger a type error. Using requirements, we could insist that the two classes be disjoint, with declarations such as

```
require Floating t fails if Integral t
require Integral t fails if Floating t
```

Using these declarations, we could conclude that, for any ground instance of `Integral t`, there must be a corresponding ground instance of `Floating t fails`, and thus that the type of `(λx → sin x `mod` 2)` is provably unsatisfiable.

We can also use requirements to reason about intersections of classes. For example, the Habit language uses a predicate of the form `NumLit n t` to indicate that a value of type `t` can represent a numeric literal with value `n`, and a type function `NonZero` to statically eliminate the possibility of division by zero. We might like to ensure that, if there is a numeric literal of type `t` with non-zero value `n`, then there is also a literal of type `NonZero t` with value `n`. We can do this using requirements, as follows:

```
require NumLit n (NonZero t) if NumLit n t, 0 < n
```

The Habit report proposes an instance of this form; however, this has implementation difficulties, as it overlaps with all other instances of `NumLit`, and the `NonZero` class does not include a class method for constructing literals.

Finally, requirements allow us to capture properties of individual classes. For example, given a class `Lt`, implementing the less-than relationship for type-level numbers, we could capture that `Lt` is anti-symmetric, using the requirement

```
require Lt m n fails if Lt n m
```

or that it is transitive, using the requirement

```
require Lt m p if Lt m n, Lt n p
```

These examples demonstrate that requirements are a powerful tool for programmers to document important semantic properties of, and relationships between, classes. In the following sections, we will describe how the compiler can enforce these requirements, and their implications for entailment and the semantics of overloading.

Modelling and Acceptability. The forcing relation for requirements is a direct generalization of that for superclasses. In the case of superclasses, we had constraints of the form $\forall\vec{t}.\,\pi' \Rightarrow \pi$ and the forcing relation

$$G \models (\forall\vec{t}.\,\pi' \Rightarrow \pi) \iff \forall S \in \mathit{GSubst}(\vec{t}).\,(G \models S\,\pi' \implies G \models S\,\pi).$$

Requirements have a more general structure, $\forall\vec{t}.\ P \Rightarrow \pi$, replacing the single hypothesis π' with the hypotheses P, but the extension of the forcing relation is direct

$$G \models (\forall\vec{t}.\,P \Rightarrow \pi) \iff \forall S \in \mathit{GSubst}(\vec{t}).\,(G \models S\,P \implies G \models S\,\pi).$$

On the other hand, the extension of the acceptability relation to requirements is not as simple. As with superclasses, we approximate the conditions under which some $\pi \in P$ is forced by the hypotheses Q of each axiom clause $\forall\vec{t}.\,\pi' \Leftarrow Q$ such that π' matches π. However, with requirements, we must consider combinations of axiom clauses that match the requirement hypotheses, not just single clauses. For example, consider the constraint

$$\forall m, n, p.\,(\mathrm{Lt}\ m\ n, \mathrm{Lt}\ n\ p) \Rightarrow \mathrm{Lt}\ m\ p,$$

corresponding to the transitivity requirement above, along with the instances

```
instance Lt Z (S n)
instance Lt (S m) (S n) if Lt m n
```

As with superclasses, we will check the axioms against the requirements in sequence; in this case, we will start with the axiom for zero, and then the axiom for successors. Validating the zero axiom against the requirement is straightforward, as the zero axiom cannot syntactically match both hypotheses. However, in the case of the successor axiom, there are two possible ways to match the hypotheses of the requirement. Thus, to show that the new axiom respects the requirement, we must show the entailments:

$$\mathrm{Lt}\ m\ n \Vdash \mathrm{Lt}\ \mathrm{Z}\ (\mathrm{S}\ n)$$

$$\mathrm{Lt}\ m\ n, \mathrm{Lt}\ n\ p \Vdash \mathrm{Lt}\ (\mathrm{S}\ m)\ (\mathrm{S}\ n)$$

The first of these is immediate. The second requires an inductive argument. In particular, we must use the requirement that we are validating to conclude from Lt m p from the hypotheses. A development of requirements would thus need to allow this kind of inductive use of requirements without generating specious proofs.

As this example demonstrates, the number of entailments needed to validate a requirement can grow polynomially. We believe, however, that the required effort will be limited in practice. First, we imagine many requirements will arise from Haskell-style superclasses; in this case, validating the requirement involves no more work than would validating a superclass in Haskell. For the remaining cases, we make the observation that most classes are either populated by a large number of specific instances, or by a small number of generic instances. In either case, the potential effort to validate requirements is limited: in the first case, because the requirement is validated at specific types, limiting the number of possible instantiations; and, in the second case, because there are relatively few total instances to be considered. In the example given above, while there are four possible combinations of the two axioms for `Lt`, two can be eliminated syntactically.

Entailment. As with acceptability, extending the entailment rules from superclass constraints to requirements is initially straightforward, but seems to introduce significant complexity to the proof search. The entailment rule for superclasses is

$$[\textsc{Super}]\ \frac{(\forall \vec{t}.\, \pi' \Rightarrow \pi) \in X \quad S \in \mathit{Subst}(\vec{t}) \quad A \mid X \vdash P \Vdash S\,\pi'}{A \mid X \vdash P \Vdash S\,\pi}$$

We could extend this rule to requirements by simply replacing hypothesis π' with a set of hypotheses Q:

$$[\textsc{Require}]\ \frac{(\forall \vec{t}.\, Q \Rightarrow \pi) \in X \quad S \in \mathit{Subst}(\vec{t}) \quad A \mid X \vdash P \Vdash S\,Q}{A \mid X \vdash P \Vdash S\,\pi}$$

However, it is not clear if there is a complete implementation for such a rule; in particular, for requirements such as the earlier transitivity example:

$$\forall m, n, p.\, \mathrm{Lt}\ m\ n, \mathrm{Lt}\ n\ p \Rightarrow \mathrm{Lt}\ m\ p,$$

using this rule to prove a predicate Lt $\tau_1\ \tau_2$ would seem to require searching for a suitable instantion of type variable n. To avoid this search, our implementation uses forward-chaining with requirements, while still using backwards-chaining with axioms. That is, rather than attempt to match the current goals with the conclusions of requirements, we add the conclusion of a requirement to the assumptions if all of its hypotheses have already been assumed. While this avoids search, it is incomplete. For example, consider the program containing an instance

$$(\forall t.\, D\ t \Leftarrow C\ t)\ ;\ \varepsilon$$

and the requirement

$$\forall t.\, (D\ t, E\ t) \Rightarrow F\ t.$$

Given rule Require as above, we could construct a proof of the entailment $C\ \tau, E\ \tau \Vdash F\ \tau$, using rule Require on goal $F\ \tau$, followed by rule Assume to discharge the goal $E\ \tau$ and rule Axiom to discharge goal $D\ \tau$. However, our implementation does not find this proof, because the predicate $D\ \tau$ is never assumed.

Semantics. Dictionary-passing translations of Haskell traditionally implement superclasses by embedding superclass dictionaries in subclass dictionaries. For example, each dictionary for an `Ord t` constraint would contain the corresponding `Eq t` dictionary. This approach does not obviously generalize to handle requirements, however. For example, given the example transitivity requirement, it would seem to require embedding the evidence for Lt m p in either the evidence for Lt m n or the evidence for Lt n p; however, neither of these embeddings is sensible. We give a more generic treatment of superclasses in our translation semantics of overloading (§6.1). Our approach describes the construction of superclass evidence by a function on the proof of its hypothesis. This naturally generalizes to requirements, as requirement evidence could be given by a function of the proofs of its hypotheses. In this view, an inductive proof, such as the one for the `Lt` instances above, would correspond to a recursion in the requirement's evidence function, and the induction being well founded would correspond to the recursion terminating.

7.3 SEMANTICS OF OVERLOADING

Our semantics of overloading serves to relate our models of type classes to the meanings of overloaded expressions, and to prove that the OML type system, in combination with our entailment relation, is an accurate approximation of the dynamic semantics of those expressions. In this section, we discuss two extensions of our semantics to facilitate its use in reasoning about programs with overloading.

The first is to extend our approach to equational theories of OML expressions, as in Ohori's semantics for Core ML. This would simplify reasoning about the meanings of OML expressions, without having to reason directly about sets of simply-typed terms; it would also allow us to investigate whether full abstraction, proved by Ohori for his semantics, could also be proved for ours. We believe that the most significant technical obstacle to this extension would arise from class methods. Ohori's approach relies on uniform substitution of polymorphic expressions. However, we cannot, for example, hope to substitute a uniform expression for the equality predicate, as its meaning differs at different types. Ohori's development also assumes β-equality, which fits languages, like Haskell, that use call-by-need evaluation models, but would not describe languages, like Habit, that use call-by-value evaluation.

The second is to investigate parametricity in our semantics. The idea of parametricity originated with Reynold's abstraction theorem [54]; in the functional programming community, it is most commonly known via Wadler's development of "free theorems" [68]. Free theorems refer to semantic properties of expressions that are derived solely from their types, and from the application of the abstraction theorem. For example, from the types of `reverse` and `map`, Wadler shows that `map f` $\circ$ `reverse` $=$ `reverse` $\circ$ `map f`, for an arbitrary function `f`. Again, we believe that the primary difficulty in adapting Reynolds's approach to our setting would be the non-uniformity introduced by classes and their methods.

7.4 PROOF BY CASES

In the previous section, we discussed several challenges introduced by the non-uniformity of our semantics. This section, by contrast, discusses some ways that we could rely on that non-uniformity to increase the expressiveness of our predicate language. As a motivating example, consider the instance

```
instance C Int
else C t if D t
```

and the entailment $\mathrm{D}\ t \Vdash \mathrm{C}\ t$. It may be surprising that the proof rules we have given are not sufficient to prove such an entailment from the given instance. However, note that our rules for entailment require each predicate to be discharged uniformly. In this case, a proof of $\mathrm{C}\ t$ would depend on the instantiation of t: if

```
class C t
   where f :: t → t

instance C Bool
   where f = not
else C t
   where f = id

g x = f (f x)
```

Figure 7.1: Non-parametric behavior without qualified type

t is instantiated to `Int`, then the proof relies on the first clause of the instance chain; in all other cases, it relies on the second. Similar cases have arisen in practice; for example, a programmer was surprised to discover that the Habit compiler was unable to discharge an entailment $(n < 32) \Vdash (m + n = 32)$ with a suitable improvement for m. However, as in the previous entailment, the proof of $m + n = 32$, and the corresponding improvement for m, differs depending upon the instantiation of n.

We have begun experimenting with proof techniques that would be able to discharge such entailments. Our approach is based on observing that, if there is an axiom that includes a clause matching the goal predicate, then the goal predicate must be proved by clauses in that axiom. Thus, we can consider each clause that could provide the goal separately. For example, in the above example, we would observe that goal C t must be proven either by the clause C Int $\Leftarrow \varepsilon$ or by the clause $\forall t.$ C $t \Leftarrow$ D t. Distinct proofs, and distinct entailments, are computed in each case. Note that we could not apply this technique without the assumption D t, as we could not prove the second case, nor could we use it to simplify C t to D t, as we do not know whether or not D Int holds. This approach does not cause problems with our semantics of overloading, as the semantics of a polymorphic value is defined in terms of its monomorphic specializations, and the proof objects and improvements are well-defined at each monomorphic type. Similarly, it works well with Habit's existing compilation pipeline.

However, this approach introduces challenges with other semantic results. For example, consider the code in Figure 7.1. Given these declarations, we might expect the function `g` to have the type `C t` $\Rightarrow$ `t` $\rightarrow$ `t`. However, we could use a proof by cases to conclude the entailment $\emptyset \Vdash \mathrm{C}\ t$, and so we could assign `g` the unqualified type `t` $\rightarrow$ `t`. This suggests that allowing proof by cases could introduce further challenges with abstraction and parametricity results, along the lines described in the previous section.

Analysis by cases could also contribute to the computation of useful improvements from instance chains. For example, consider the instance chain

```
instance C Int
else C t fails
```

A case-wise analysis of this instance chain would allow us to conclude that the only satisfiable instance of a predicate C t is C Int, and thus that $[\mathrm{Int}/t]$ is an improving substitution for such a predicate. Our prototype implementation of proof by cases does not yet compute such improvements.

REFERENCES

[1] Cabal. `http://haskell.org/cabal`, 2013.

[2] William R. Cook. Object-oriented programming versus abstract data types. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 151–178, London, England, 1991. Springer-Verlag.

[3] Duncan Coutts. Solving the diamond dependency problem. `http://blog.well-typed.com/2008/08/solving-the-diamond-dependency-problem/`, 2008. Last accessed June 8, 2010.

[4] Duncan Coutts. Regression testing with hackage. `http://blog.well-typed.com/2009/03/regression-testing-with-hackage/`, 2009. Last accessed June 8, 2010.

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

[6] Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in Agda. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 143–155, Tokyo, Japan, 2011. ACM.

[7] Iavor S. Diatchki and Mark P. Jones. Strongly typed memory areas: programming systems-level data structures in a functional language. In *Proceedings*

of the 2006 ACM SIGPLAN workshop on Haskell, Haskell '06, pages 72–83, Portland, Oregon, USA, 2006. ACM.

[8] Derek Dreyer, Robert Harper, Manuel M. T. Chakravarty, and Gabriele Keller. Modular type classes. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 63–70, Nice, France, 2007. ACM.

[9] Melvin Fitting. Model existence theorems for modal and intuitionistic logics. *Journal of Symbolic Logic*, 38(4):613–627, Dec 1973.

[10] Melvin Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Secaucus, New Jersey, 1996.

[11] Harvey Friedman. Equality between functionals. In Rohit Parikh, editor, *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 22–37. Springer Berlin Heidelberg, 1975.

[12] Benedict R. Gaster. *Records, variants, and qualified types.* PhD thesis, University of Nottingham, 1998.

[13] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, University of Nottingham, 1996.

[14] GHC. `http://haskell.org/ghc`, 2013.

[15] Jean-Yves Girard. The system F of variable types, 15 years later. In Gerard Huet, editor, *Logical Foundations of Functional Programming*, pages 87–126. Addison-Wesley, 1990.

[16] Carl A. Gunter. *Semantics of Programming Languages.* The MIT Press, Cambridge, Massachusetts, 1992.

[17] Thomas Hallgren. Fun with functional dependencies, or (draft) types as values in static computations in Haskell. http://www.cse.chalmers.se/~hallgren/Papers/wm01.html.

[18] William Harrison. A simple semantics for polymorphic recursion. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, APLAS '05, pages 37–51, Tsukuba, Japan, 2005. Springer-Verlag.

[19] Bastiaan Heeren and Jurriaan Hage. Type class directives. In *Seventh International Symposium on Practical Aspects of Declarative Languages*, PADL '05, pages 253–267. Springer-Verlag, 2005.

[20] David Himmelstrup, Paolo Martini, Bjorn Bringert, Isaac Potoczny-Jones, and Duncan Coutts. cabal-install: The command-line interface for cabal and hackage. http://hackage.haskell.org/package/cabal-install. Last accessed June 7, 2010.

[21] J. Roger Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the AMS*, (146):29–60, 1969.

[22] Ralf Hinze. Generics for the masses. *J. Funct. Program.*, 16(4-5):451–483, July 2006.

[23] Gérard Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, September 1997.

[24] Hugs 98. http://haskell.org/hugs, 2006.

[25] Mark P. Jones. Coherence for qualified types. Technical Report YALEU/DCS/RR-989, Yale University, 1993.

[26] Mark P. Jones. Gofer 2.28 release notes. http://web.cecs.pdx.edu/~mpj/goferarc/gofer230b.zip/docs/release.228, 1993.

[27] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Conference on Functional Programming and Computer Architecture*, FPCA '93, pages 52–61, Copenhagen, Denmark, 1993. ACM.

[28] Mark P. Jones. *Qualified Types: Theory and Practice.* Cambridge University Press, 1994.

[29] Mark P. Jones. Dictionary-free overloading by partial evaluation. *Lisp Symb. Comput.*, 8(3):229–248, September 1995.

[30] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques*, volume 925 of *Lecture Notes in Computer Science*, May 1995.

[31] Mark P. Jones. Simplifying and improving qualified types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 160–169, La Jolla, California, USA, 1995. ACM.

[32] Mark P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, pages 230–244, Berlin, Germany, 2000. Springer-Verlag.

[33] Mark P. Jones and Iavor S. Diatchki. Language and program design for functional dependencies. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 87–98, Victoria, BC, Canada, 2008. ACM.

[34] Oleg Kiselyov and Ralf Lämmel. Haskell's overlooked object system. Draft; Submitted for publication; online since 10 Sept. 2005.

[35] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 96–107, Snowbird, Utah, USA, 2004. ACM Press.

[36] Saul A. Kripke. Semantical analysis of modal logic I. Normal propositional calculi. *Zeitschrift fur mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[37] Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 91–113, London, England, 1998. Springer-Verlag.

[38] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 333–343, San Francisco, California, 1995. ACM.

[39] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 165–174, Salt Lake City, Utah, United States, 1984. ACM.

[40] David Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[41] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, (17):348–375, 1978.

[42] Robin Milner, Mads Tofte, and Robert Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[43] J. C. Mitchell and R. Harper. The essence of ML. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 28–46, San Diego, California, United States, 1988. ACM.

[44] J. Garrett Morris. Experience report: Using Hackage to inform language design. In *Proceedings of the third ACM symposium on Haskell*, Haskell '10, Baltimore, Maryland, USA, 2010. ACM.

[45] J. Garrett Morris and Mark P. Jones. Instance chains: Type-class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, Baltimore, MD, 2010. ACM.

[46] Matthias Neubauer, Peter Thiemann, Martin Gasbichler, and Michael Sperber. A functional notation for functional dependencies. In *Proceedings of The 2001 ACM SIGPLAN Workshop on Haskell*, Haskell '01, Firenze, Italy, September 2001.

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

[48] Atsushi Ohori. A simple semantics for ML polymorphism. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 281–292, London, UK, 1989. ACM.

[49] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.

[50] Simon Peyton Jones, Mark P. Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Proceedings of the 1997 workshop on Haskell*, Haskell '97, Amsterdam, The Netherlands, 1997.

[51] The Hasp Project. The habit programming language: The revised preliminary report. `http://hasp.cs.pdx.edu/habit-report-Nov2010.pdf`, 2010.

[52] John C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, pages 408–423. Springer-Verlag, 1974.

[53] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Advances in Algorithmic Languages*, pages 157–168. Inst. de Recherche d'Informatique et d'Automatique, 1975.

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

[55] John A. Robinson. A Machine-Oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965.

[56] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, IFCP '08, pages 51–62, Victoria, BC, Canada, 2008. ACM.

[57] Jonathan Shapiro, Swaroop Sridhar, and Scott Doerrie. BitC (0.11 transitional) language specification. `http://www.bitc-lang.org/docs/bitc/spec.html`. Last accessed June 15, 2010.

[58] Tim Sheard and Emir Pasalic. Two-level types and parameterized modules. *J. Funct. Program.*, 14(5):547–587, Sep 2004.

[59] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. *Electronic Notes in Theoretical Computer Science*, 199(0):49–65, 2008.

[60] Dominic Steinitz. Exporting a type class for type signatures. `http://www.haskell.org/pipermail/haskell-cafe/2008-November/050409.html`, November 2008.

[61] Don Stewart. Re: [Haskell-cafe] Overlapping/Incoherent instances. `http://www.haskell.org/pipermail/haskell-cafe/2008-October/049155.html`, 2008. Last accessed June 8, 2010.

[62] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1):11–49, Apr 2000.

[63] Martin Sulzmann, Gregory J. Duck, Simon Peyton Jones, and Peter J. Stuckey. Understanding functional dependencies via constraint handling rules. *JFP*, 17(1):83–129, 2007.

[64] Wouter Swierstra. Data types à la carte. *JFP*, 18(04):423–436, 2008.

[65] The Cabal Team. #435 (ban upwardly open version ranges in dependencies on base). `http://hackage.haskell.org/trac/hackage/ticket/435`, 2009. Last accessed June 8, 2010.

[66] Roel van Dijk. Ann: Reverse dependencies in hackage (demo). `http://www.haskell.org/pipermail/haskell/2009-October/021691.html`, 2009. Last accessed June 8, 2010.

[67] Arild Waaler and Lincoln Wallen. Tableaux for intuitionistic logics. In Marcello D'Agostino, Dov M. Gabbay, Reiner H ahnle, and Joachim Posegga, editors, *Handbook of Tableau Methods*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999.

[68] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 347–359, London, England, 1989. ACM.

[69] Philip Wadler. The expression problem. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, 1998.

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

[71] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.