Instance Chains: Type Class Programming Without Overlapping Instances

J. Garrett Morris Mark P. Jones
Portland State University
{jgmorris,mpj}@cs.pdx.edu

Abstract

Type classes have found a wide variety of uses in Haskell programs, from simple overloading of operators (such as equality or ordering) to complex invariants used to implement type-safe heterogeneous lists or limited subtyping. Unfortunately, many of the richer uses of type classes require extensions to the class system that have been incompletely described in the research literature and are not universally accepted within the Haskell community.

This paper describes a new type class system, implemented in a prototype tool called `ilab`, that simplifies and enhances Haskell-style type-class programming. In `ilab`, we replace overlapping instances with a new feature, *instance chains*, allowing explicit alternation and failure in instance declarations. We describe a technique for ascribing semantics to type class systems, relating classes, instances, and class constraints (such as kind signatures or functional dependencies) directly to a set-theoretic model of relations on types. Finally, we give a semantics for `ilab` and describe its implementation.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type structure

General Terms Design, Languages

Keywords Qualified types, Type classes, Overlapping instances, Fucntional dependencies, Haskell

1. Introduction

Type classes are a widely used, studied, and extended feature of the Haskell programming language. Some extensions, such as multi-parameter type classes, functional dependencies, and type functions, have been extensively studied and debated. In contrast, the overlapping instances extension has received relatively little attention, despite its use in several interesting examples of type-level programming.

One example of overlapping instances is the "smart constructors" in Wouter Swierstra's solution to the expression problem in Haskell [18]. We discuss this example in detail in Section 2.2.2, but preview that discussion here. His solution uses a coproduct constructor `t :+: u` and a subtyping relation `f :<: g`, which is implemented with the following overlapping instance declarations:

```
instance f :<: f
instance f :<: (f :+: g)
instance f :<: h ⇒ f :<: (g :+: h)
```

The restrictions on overlapping instances in Haskell constrain the use of this relation in several ways. Most significantly, `:<:` only recurses on the right-hand side of a `:+:`, limiting its use to list-like (rather than tree-like) coproducts. There is also an unresolvable overlap between the first and third instances. This overlap causes Hugs, one implementation of Haskell, to reject the instances outright, and will cause GHC, another Haskell implementation, to issue type errors for some otherwise-valid predicates (see Section 2.2.2 for more details).

These issues are typical of those encountered by Haskell programmers using overlapping instances. We argue that overlapping instances lack modularity, lack specification, and that they significantly complicate reasoning about type-level programming.

This paper proposes an alternative approach to type-class programming, replacing overlapping instances with a new feature called instance chains. Using instance chains, we could rewrite Swierstra's subtyping example as:

```
instance f :<: f
else f :<: (g :+: h) if f :<: g
else f :<: (g :+: h) if f :<: h
else f :<: g fails
```

Our version expresses the alternation between the three instances directly instead of relying on the overlapping instances mechanism. As a result, it recurses on both sides of the `:+:` operator, and resolves the overlap between the first and third instances. We can also close the definition of `:<:` in the last line of the declaration. This example highlights the major features of instance chains: explicit alternation within instance declarations, and explicit failure in both predicates and instance declarations. We argue that reasoning about programs with instance chains is simpler than reasoning about programs with overlapping instances despite the additional syntax.

This paper proceeds as follows. Section 2 describes type classes and some frequently used extensions. In the process, we develop an intuitive semantics for type classes (following the lead of Jones and Diatchki [7]). We then examine several interesting examples of type-class programming that use overlapping instances and functional dependencies. We identify places where existing type-class programming techniques are unclear or could be improved.

Section 3 describes the type class system implemented by our prototype tool `ilab`. In designing `ilab`, we identified some usage patterns that are implemented in Haskell using overlapping instances, and made those patterns expressible directly, simplifying coding and removing the need for overlapping instances. The key

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

features of ilab are: instance chains, allowing programmers to express alternation directly in instance declarations; and explicit failure, allowing programmers to define and test when predicates do not hold. We explain these features and their consequences. We revisit the earlier examples of type-class programming, showing how to simplify and improve them using instance chains.

Of course, this is not the first attempt to simplify type-level programming in Haskell: much entertainment has resulted from the odd or incompletely specified interaction of otherwise-reasonable extensions to the Haskell type-class system. To avoid repeating this experience, Section 4 formalizes a set-theoretic semantics for type classes. We highlight several places where instance chains simplify reasoning about type classes compared to overlapping instances. We also state properties, such as soundness and completeness, that link the implementation of a type-class system to its semantics, and provide a basis for programmers to reason about type-class programs. In the process, we connect the semantics of type classes to Jones' theory of qualified types.

Section 5 discusses the implementation of ilab and describes the algorithms that ilab uses to validate sets of instances and to (attempt to) prove predicates. Section 6 discusses related work, while Section 7 discusses future work and concludes.

2. Background

2.1 Type classes

Type classes [21] provide an extensible mechanism for giving principal types to overloaded functions. For instance, we can define a type class for equality:

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

The type of == is now constrained, or qualified, by a predicate mentioning the Eq class:

```
(==) :: Eq t ⇒ t → t → Bool
```

We can explain this qualified polymorphic type using set notation. The == function can assume types from the set

$$\{t \to t \to \texttt{Bool} \mid t \in \texttt{Eq}\}$$

We conclude that one should view type classes as specifications of sets of types, not just as tools for typing overloaded functions.

Type classes are populated by instance declarations. If we had a primitive integer comparison function primIntEq, then we could write an Eq instance for Int as follows

```
instance Eq Int
   where x == y = primIntEq x y
```

Instance declarations themselves may use qualified polymorphism. For example, the definition of Eq for lists reads:

```
instance Eq t ⇒ Eq [t]
   where []    == []    = True
         (x:xs) == (y:ys) = x == y && xs == ys
         _     == _     = False
```

As type classes correspond to sets of types, instance declarations correspond to assertions about those sets. The first declaration asserts that Int ∈ Eq. The second asserts that $t \in \texttt{Eq} \implies [t] \in \texttt{Eq}$. Together, these assertions require that Eq include the subset $\{\texttt{Int}, [\texttt{Int}], [[\texttt{Int}]], \dots\}$ of the set *Type* of all types.

There have been numerous proposals to extend Haskell's class system. In the next sections, we discuss those relevant to our work.

2.1.1 Multi-parameter type classes

Although Wadler and Blott [21] focus on type classes with a single parameter (which correspond to sets of types with associated operators), they proposed that type classes could also apply to more than one parameter. For example, the multi-parameter type class

```
class Elems c e where ...
```

could describe the relation that the elements of (collection) c have type e. This class might be populated for lists:

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

and for sets:

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

For this example, we assume Sets are implemented by balanced binary trees, and so a type t must have an ordering before we can construct a value of type Set t. The type class Ord captures this constraint.

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). Assuming the instances for Eq above, and that we have an instance of Ord for Int, we would expect Elems to include the following subset of *Type* × *Type*

$$\{([\texttt{Int}], \texttt{Int}),\ (\texttt{Set Int}, \texttt{Int}),\ ([[\texttt{Int}]], [\texttt{Int}]),\ \dots\}$$

2.1.2 Functional dependencies

One of the operations of the Elems class might be an insert function with type

```
insert :: Elems c e ⇒ e → c → c
```

Using it, we could write the 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:

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

While one could imagine a collection type c that satisfied both qualifiers, we may wish to require homogeneity. That constraint can be expressed by adding a functional dependency [6, 10] to the definition of Elems:

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

This requires that the value of parameter c uniquely determines the value of parameter e, or, equivalently, that for any two predicates Elem c e and Elem c' e', if c = c', then e = e'. This would make the definition of insert2 above a type error, because it would require that Char = Bool

The functional dependency is a property of the relation itself, not of the constraints on it; for example, the Elems class from Section 2.1.1 had this functional dependency, even though we had not yet added the constraint. However, were we to add an instance

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

that interpreted characters by their ASCII values, then both the predicates Elems [Int] Int and Elems [Int] Char would hold and the relation would no longer have the functional dependency. With the functional dependency constraint on the Elems class, a program could not contain both this instance and the instance Elems [t] t from the previous section.

2.1.3 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 ...
```

Either of these instances could be used to solve the predicate `C ([a], [b])`. However, the compiler has no guarantee that the class methods are implemented equivalently for both instances, 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. The `Show` class includes types whose values have a textual representation:

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

Haskell's syntax for lists surrounds the elements with brackets and separates them with commas—for example, `[1,2,3]`. We could write a `Show` instance that used this syntax:

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

Haskell also has special syntax to allow lists of characters to be written as strings, as in `"string"`, for example. We might like to add a special instance of `Show` to handle this case:

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

but that would not be allowed because this instance overlaps with the more general instance.

Peyton Jones et al. [13] describe an extension to the Haskell type-class system that allows instances to overlap as long as one of them is more specific than the other, using substitutions to make the notion of "more specific" precise. Given two instances

```
instance Q1 ⇒ P1 where ...
instance Q2 ⇒ P2 where ...
```

the instances overlap if $P_1 \sim P_2$ (i.e., P_1 unifies with P_2) The first instance is more specific than the second if there is a substitution S such that $P_1 = S\ P_2$ but there is no substitution T such that $T\ P_1 = P_2$. This extension would allow the two instances of `Show`, but would prohibit the two instances of `C` at the beginning of this section because neither is a substitution instance of the other.

A full description of how overlapping instances affect the semantics of type classes is beyond the scope of this paper; however, we do mention some of the difficulties in Section 2.3.

2.2 Type-class programming

This section describes two (simplified) examples from the literature that use the extensions of the Haskell type-class system described earlier. We focus on examples that use overlapping instances because of their relative complexity in both implementation and semantics. We will return to these examples in Section 3 to demonstrate instance chains.

2.2.1 Type-level arithmetic

In this section, we describe the implementation of several mathematical operations at the type level using Peano arithmetic and type classes, based on work by Thomas Hallgren [2].

We begin by representing Peano numbers at the type level using two data types, one for zero and one for successor. We do not provide value-level constructors for these types because we only intend to use them at the type level.

```
data Z; data S n
```

Similarly, we will introduce types to represent Boolean values:

```
data T; data F
```

Hallgren defines a class `Lte` to implement the $\leq$ relation at the type level as follows:

```
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
```

As indicated by the functional dependency, Hallgren has actually defined the characteristic function of the $\leq$ relation, using additional type constructors `T` and `F` to represent the corresponding Boolean values. This allows him more flexibility in using the `Lte` class because he can now determine, not only when one number is less than or equal to another, but also when that property fails.

Hallgren goes on to use the `Lte` class to define insertion sort at the type level. However, the Haskell implementation that he was using (Hugs 98) could not solve the type constraints in his insertion sort example. His code works in `ilab` without modification, so we do not reproduce it here.

Instead, we try to define another operation on Peano numbers: greatest common divisor. This is not an arbitrary choice: for example, work on typing low-level data structures in Haskell has relied on a type-level GCD operator [1]. We begin by defining a (bounded) subtraction operation:

```
class Subt m n p | m n → p   -- p = m - n
instance Subt Z n Z
instance Subt m Z m
instance Subt m n (S p) ⇒ Subt m (S n) p
```

We use this to implement Euclid's algorithm for GCD:

```
class Gcd m n p | m n → p  -- p = gcd(m,n)
instance Gcd m m m
instance (Lte n m T, Subt m n m', Gcd m' n p)
    ⇒ Gcd m n p
instance (Lte n m F, Subt n m n', Gcd m n' p)
    ⇒ Gcd m n p
```

However, both GHC and Hugs reject this trio of instances. While it is true that the conclusions of the second and third instances (trivially) unify, there is no actual overlap between those instances. For both instances to apply to the same predicate `Gcd m n p`, both the predicates `Lte n m T` and `Lte n m F` would have to hold. However, the functional dependency in `Lte` makes it impossible for both predicates to hold.

2.2.2 The expression problem

In this section, we return to the example presented in the introduction. We describe its context, a Haskell solution to the *expression problem* that relies on multi-parameter type classes and overlapping instances, and highlight the difficulties these extensions introduce.

The expression problem [20] is a benchmark for comparing language expressiveness and modularity. The starting point is to define, by cases, a data type for arithmetic expressions, as well as an operation over that data type. For example, the data type might contain integer constants and addition, and the operation might be evaluation, consuming an expression and generating an integer value. The challenge is to extend the data type with both a new case (such as multiplication) and a new operation (such as pretty-printing). This extension should be done without changing or recompiling the original code, and without losing static type safety.

Though definition of types by cases is standard in both functional and object-oriented languages, the expression problem is usually challenging in either paradigm. In many functional languages, adding a new case to an existing data type requires changing the definition of the data type and all the functions that use it; in many object-oriented languages, adding a new operation requires changing the definition of the base class and its subclasses.

Wouter Swierstra proposed a Haskell solution to the expression problem in "Data Types à la Carte" [18]. His solution works by

constructing coproducts of functor type constructors (using a type constructor `:+:`), injecting values into these coproducts (using a `:<:` type class), and then defining operations over coproducts using one type class per operation. We highlight some details that arise in the construction of coproducts.

The type constructor `f :+: g` represents the coproduct of the functor type constructors `f` and `g`, and is defined similarly to the standard Haskell `Either` type:

```
data (f :+: g) e = Inl (f e) | Inr (g e)
```

The type constructors for various possible expressions will also be functors but the use of functors is irrelevant to the remainder of this presentation. Suppose that we have a type constructor `Const` for integer constants and a type constructor `Add` for additions. The type of an expression containing either constants or sums could be built using the coproduct `Const :+: Add`. Constants would be injected into the expression type using the `Inl` value constructor, and sums using the `Inr` constructor. We could extend the expression type by adding a new type to the coproduct. For instance, if the type `Multiply` represents multiplications, we could construct expressions with the coproduct `Const :+: (Add :+: Multiply)`. We would still inject constants into these new expressions using `Inl`, but additions would be injected using `Inl ∘ Inr` and multiplications using `Inr ∘ Inr`.

It is somewhat tiresome to construct different injection functions for each type in each possible coproduct. Swierstra alleviates this by defining a type class `f :<: g` to indicate that there is an injection from `f e` to `g e` for any type `e`. The class is defined by:

```
class f :<: g where inj :: f e → g e
```

Swierstra populates this class using three instances. The first says that `:<:` is reflexive, with the obvious injection:

```
instance f :<: f where inj = id                    (A)
```

The second instance checks the left-hand side of the `:+:` type:

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

The final instance recurses on the right side of the `:+:` type:

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

Swierstra comments on the overlap between instances (B) and (C): because (B) is a substitution instance of (C), predicates will be checked against (C) only if they fail to match (B), and the set of instances will behave as expected. More interestingly, instances (A) and (C) overlap at predicates of the form `(f :+: g) :<: (f :+: g)` but neither is a substitution instance of the other. As a result, Hugs will reject this set of instances outright for having unresolved overlap. GHC accepts the instances, but attempting to use the `inj` function with such a predicate will result in a type error. This error occurs at the usage of `inj`, not at the ambiguous overlap in the definition of `:<:`.

Although the `:+:` type constructs a coproduct of types, the subtype relation `:<:` cannot fully exploit it because the recursive case only descends the right-hand side. Thus, while the predicate

```
Sum :<: (Const :+: (Sum :+: Product))
```

holds, the predicate

```
Sum :<: ((Const :+: Sum) :+: Product)
```

does not, because it requires recursion on the left-hand side of the `:+:` operator. This forces a list-like use of the `:+:` constructor; for instance, if `t` and `u` are already coproducts, we would not be able to inject components of `t` into the coproduct `t :+: u`. To fix this, we could replace instance (B) with

```
instance (f :<: g) ⇒ f :<: (g :+: h)
    where inj = Inl ∘ inj                          (B-2)
```

However, (B-2) and (C) are each substitution instances of the other. The Haskell compiler no longer has a way to order these instances, and so a program containing (B-2) and (C) would be rejected.

2.3 Challenges

Haskell programmers have three main challenges when using overlapping instances. We summarize them here.

Lack of modularity. In "Data Types à la Carte", the three instances of `:<:` are presented together, and in order from most to least specific. However, Haskell imposes no requirement that the instances be presented together, or in any particular order, or even in the same module. A programmer has no way to know whether a particular instance will be overlapped before it is used.

In fact, GHC only attempts to determine which instance is most specific at call sites, and so will accept ambiguously overlapping instances—that is, cases where two instances apply to the same predicate and neither is more specific than the other—and not report an error unless the programmer attempts to use an overloaded function at one of the ambiguous types. Ambiguity could be introduced in a library but not discovered until some client of the library uses one of the types at which the instances are ambiguous.

Logical consistency. The syntax of instance declarations in Haskell suggests logical implication. For example, the `Eq` instance for lists begins:

```
instance Eq t ⇒ Eq [t]
```

and can be read as an implication $t \in \texttt{Eq} \implies [t] \in \texttt{Eq}$. Even in Haskell 98, this interpretation does not completely cover the meaning of the declaration. Because Haskell 98 instances cannot overlap, the only way that `[t]` can be in `Eq` is for `t` to be in `Eq`, so a more accurate interpretation would be $t \in \texttt{Eq} \iff [t] \in \texttt{Eq}$.

The meaning of overlapping instances is more obscured. A particular instance only applies if a more specific instance could not be found. Furthermore, if the preconditions of the most specific instance are not met, the compiler does not check to see if less specific instances might be applicable but instead immediately issues an error. As a result, it is impossible to interpret the meaning of an individual instance declaration without referring to the other instances in the program. While the syntax of instances still suggests a logical interpretation, applying that interpretation gives an incomplete and potentially incorrect meaning.

Lack of specification. Peyton Jones et al. [13] describe some issues introduced by overlapping instances. However, they do not consider the interaction of overlapping instances with some other type-class system features, such as improvement. Research on functional dependencies in class systems [6, 7, 17] generally does not mention overlapping instances, and recent work by Schrijvers et al. [14] related to type classes explicitly excludes overlap. Without a specification to define the correct behavior, code must be tailored to a particular implementation. For example, Kiselyov et al. [9] discover significant incompatibilities between GHC and Hugs and end up tailoring their code to GHC. Similarly, Swierstra's `:<:` instances are not accepted by Hugs.

3. Features of the `ilab` type-class system

As part of the High Assurance Systems Programming[1] project at Portland State University, we are designing a dialect of Haskell (called Habit) for use in low-level systems programming tasks.

[1] `http://hasp.cs.pdx.edu`

One of our goals is to preserve and expand the possibilities of type-class programming while simplifying the underlying model of type classes. As background to this effort, we surveyed type-level programming in Haskell, using both the existing research literature and the Hackage database of Haskell libraries as resources [11]. Based on the results of our survey, we have developed the `ilab` type-class system and prototype implementation. We use `ilab` to experiment with features of the Haskell type-class system; it is not a complete implementation of either Haskell or Habit type classes, but implements features central to both. Despite their history, the features of `ilab` are not tied to other features of Habit; they could just as well be applied to Haskell or other Haskell dialects. The remainder of this section describes the features of `ilab` and shows how they simplify and improve the examples from Section 2.2.

3.1 Design of the `ilab` type-class system

The `ilab` class system is based on the Haskell 98 class system, extended with overlapping instances and functional dependencies. However, rather than support overlapping instances in `ilab`, we added new features based on the usage patterns implemented using overlapping instances in Haskell. These features support and extend Haskell-style type-level programming while avoiding the complexity that would be introduced by overlapping instances.

For the remainder of the paper, we use Habit instance syntax for examples using `ilab` features, and continue to use Haskell syntax for examples that do not rely on any `ilab`-specific functionality. The Habit syntax for instance declarations is given by the following BNF-like grammar, where non-terminals have initial caps, optional elements are surrounded by brackets, and optional repeatable elements are surrounded by braces.

```
Pred    ::= ClassName Type {Type} [fails]
Context ::= Pred {, Pred}
Clause  ::= Pred [if Context] [where Decls]
Chain   ::= instance Clause {else Clause}
```

There are, of course, additional constraints on instance chain declarations—all the clauses in a chain must refer to the same class, and `fails` clauses cannot contain method definitions—as well as other restrictions as in Haskell. Habit's syntax differs from that of Haskell 98 in three ways:

1. Predicates may include the `fails` keyword, indicating that the given type tuple is not in the named class;
2. Clauses may be chained together using the `else` keyword, allowing a programmer to indicate explicit alternation; and,
3. The instance being defined appears to the front of the instance declaration, calling attention to it even in the presence of long or complex preconditions.

Habit uses additional features, such as the functional notation proposed by Jones and Diatchki [7]. For example, the Haskell instance declaration:

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

can be expressed in Habit as:

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

Because it is a prototype tool and functional notation is orthogonal to our other goals, `ilab` does not support rewriting passes such as those used to implement functional notation. In `ilab` we would have to use the following version instead:

```
instance Gcd m n p
    if Lte n m T, Subt m n m', Gcd m' n p
```

This is the form we will use for the remainder of the paper. Next, we explore the new features introduced by `ilab`, and some of their consequences.

3.1.1 Explicit alternation

Many of the examples we found use overlapping instances to implement alternation between instances. This approach is fragile, obscures the programmer's intention, and limits the algorithms the programmer can encode. In `ilab`, a class can be populated by multiple, non-overlapping instance chains, where each chain may contain multiple clauses (separated by the keyword `else`). Unlike between chains, we make no limitation on overlap between the clauses within a single chain. During instance selection, the clauses within an instance chain are checked in order. 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, in Section 2.2.2, we presented the class `:<:` and the three instances used to populate it. These instances implement a simple conditional by making the alternative clause more general than the consequent. `ilab` allows a more direct expression of the conditional:

```
instance f :<: f where ...
else f :<: (f :+: g) where ...
else f :<: (g :+: h) if f :<: h where ...
```

3.1.2 Explicit failure

Some of the examples we found attempted to encode failure of the instance search [9]. However, lacking a mechanism to encode failure directly, the examples used a combination of the class and module system to prevent the user from solving certain constraints. While this approach works, it leads to confusing error messages and cannot be used as a building block for more complex instance schemes. Making failure explicit in both predicates and instance declarations significantly simplifies coding these patterns.

By defining the characteristic function of the $\leq$ relation instead of the relation itself (as discussed in Section 2.2.1), Hallgren could express both properties of the form $m \leq n$ and $\neg(m \leq n)$. With explicit failure, we can define the $\leq$ relation directly:

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

Because we implement the relation directly, we no longer need the third parameter to the `Lte` class or the functional dependency.

One use of explicit failure is in the definition of closed classes. For example, at one point [16], the `crypto` package defined a class `AESKey` and three instances for types `Word128`, `Word192`, and `Word256`, the only valid key lengths for AES encryption. To prevent users from adding invalid types to the class, `AESKey` was not exported. As a consequence, users could not write type signatures such as:

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

In `ilab`, we can close the `AESKey` class with the following instance chain:

```
instance AESKey Word128
else AESKey Word192
else AESKey Word256
else AESKey a fails
```

No additional instances of `AESKey` can be added because they would overlap with the (last clause of the) existing instance, so there is no need to hide the class.

3.1.3 Backtracking search

Haskell instance search never backtracks: if no two instance heads unify, then no predicate could be solved by more than one instance. However, combined with overlapping instances, this complicates reasoning about 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.

`ilab` instance search backtracks when it can disprove the precondition of an instance (either because of a `fails` clause or because of a functional dependency). When backtracking, `ilab` checks clauses within an instance chain in order. The order in which `ilab` checks instance chains is unimportant because clauses in different chains are not allowed to overlap.

3.2 Type-class programming, revisited

In this section, we demonstrate how the examples from Section 2.2 are changed and improved using the features of `ilab`.

Section 2.2.1 includes several examples of implementing type-level arithmetic using type classes. The first example is the characteristic function for the $\leq$ relation. Section 3.1.2 described how this example can be improved using instance chains. Next, we attempted to define a `Gcd` class. We define the class as before, but populate it with a single instance chain that uses the `Lte` relation:

```
1 instance Gcd m m m
2 else Gcd m n p
3    if Lte n m, Subt m n m', Gcd m' n p
4 else Gcd m n p
5    if Lte n m fails, Subt n m n', Gcd m n' p
6 else Gcd m n p fails
```

As the clauses overlap, we have combined them into an instance chain. We also add the clause at line 6, closing the `Gcd` class.

Section 2.2.2 describes a solution to the expression problem. The solution relies on a type constructor `:+:` to construct coproducts of types, and a type class `:<:` for subtypes. However, the implementation of `:<:` is asymmetric—it recurses only on the right-hand side of a `:+:` type. We can implement it symmetrically:

```
1 instance f :<: f
2     where inj = id
3 else f :<: (g :+: h) fails if f :<: g, f :<: h
4 else f :<: (g :+: h) if f :<: g
5     where inj = Inl ∘ inj
6 else f :<: (g :+: h) if f :<: h
7     where inj = Inr ∘ inj
8 else f :<: g fails
```

Lines 1-2 provide a base case, and correspond to instance (A) in the original implementation. Line 8 serves as the other base case. We explicitly close the class to ensure that we have evidence for backtracking in the middle clauses of the instance. Lines 6-7 recurse on the right-hand side of a `:+:` constructor, and correspond to instance (C). Lines 4-5 replace instance (B). Unlike the original implementation, this clause recurses on the left-hand side of the `:+:` constructor. If both `f :<: g` and `f :<: h` hold, `ilab` would select the left injection because of the ordering of the respective clauses. This behavior may be surprising to programmers, so we add the additional (but optional) clause at Line 3 to rule out this kind of injection completely.

4. A semantics for type classes

The history of Haskell type-class research is littered with proposals to extend, enhance, or simplify writing type-class programs. Some of these proposals, while sensible as proposed, have led to unexpected interactions with other features, or have proven difficult for programmers to understand. We hope to avoid a similar fate for instance chains by defining a semantics for type classes and for instance chains, providing a basis for understanding their use and implementation and a foundation for future research in type classes.

Previous work has focused on translating programs in a language with type classes into programs in a language without type classes (for example, by introducing dictionaries of type-specific method implementations and transforming qualifiers into extra parameters [21]). This approach conflates the meaning of type classes with their implementation, making it difficult or impossible to define properties of type classes without reference to a particular implementation, or to prove properties of the implementation itself. This conflation is particularly unfortunate when it comes to understanding the interaction of type-class features or when the implementation itself is suspect, such as in the interaction between functional dependencies and overlapping instances.

This section elaborates the intuitive understanding of type classes as relations on types to give a full semantics for `ilab` type classes. We follow a standard approach from mathematical logic: first, we characterize models of type classes. Then, we define a property that holds when a given model describes a particular type-class program; we use this property to capture properties of implementations such as soundness or completeness. This approach does not attempt to capture the details of type class implementations such as substitutions, improvement, simplification, etc. Rather, it describes the meaning of type classes and provides a basis both for reasoning about programs that use type classes and for evaluating type-class implementations.

4.1 Modeling type classes

Single parameter type classes, such as `Eq` or `Ord`, are naturally modeled by sets of types. Let *Type* refer to the set of all types. Writing $M(\texttt{Eq})$ for the model of the `Eq` class, we can say that $M(\texttt{Eq}) \subseteq \mathit{Type}$ or, equivalently, $M(\texttt{Eq}) \in \mathscr{P}(\mathit{Type})$. This approach extends to multi-parameter type classes by using relations on types instead of sets of types. Just as (the models of) `Eq` and `Ord` are subsets of *Type*, (the model of) a class like `Elems` (see Section 2.1) is a subset of $\mathit{Type} \times \mathit{Type}$, or equivalently, $M(\texttt{Elems}) \in \mathscr{P}(\mathit{Type}^2)$. A three-parameter class would be modeled by an element of $\mathscr{P}(\mathit{Type}^3)$, and so forth. The number of arguments to a class is called its arity, and we will write $\mathrm{arity}(C)$ (where C ranges over the set of class names *ClassName*) to refer to the arity of class C. For example, we have $\mathrm{arity}(\texttt{Eq}) = 1$ and $\mathrm{arity}(\texttt{Elems}) = 2$. Using the arity function, we can write a general rule that captures the examples so far: for a class C, we have $M(C) \in \mathscr{P}(\mathit{Type}^{\mathrm{arity}(C)})$

A program will typically contain a number of type classes. To model an entire program, we use a function from *ClassName* to models of the individual classes. We can then describe a model of a program as a dependently typed function

$$M : (C : \mathit{ClassName}) \to \mathscr{P}(\mathit{Type}^{\mathrm{arity}(C)})$$

This is not the only possible structure for M; we will discuss some of the design choices further when describing the handling of constraints.

Next, we define a family of relations $M \models x$ that hold if "M models x". We develop this family of relations "bottom-up", starting from single predicates and working towards full programs.

Predicates. Predicates are the simplest parts of a type-class system. We define predicates with the following grammar

$$f ::= \mathsf{holds} \mid \mathsf{fails} \qquad \text{Flags}$$
$$\pi ::= C\ \vec{\tau}\ f \qquad \text{Predicates}$$

$$\varepsilon@\vec{\tau} = \varepsilon$$

$$((\forall \vec{x}.\, P \Rightarrow C\ \vec{v}\, f)\ ;\ \alpha)@\vec{\tau} = \begin{cases} (S\, P \Rightarrow C\ (S\ \vec{v})\, f)\ ;\ \alpha@\vec{\tau} & \text{if } \exists S.\ \mathrm{dom}(S) \subseteq \vec{x} \vee S\ \vec{v} = \vec{\tau} \\ \alpha@\vec{\tau} & \text{otherwise} \end{cases}$$

Figure 1. The restriction of an axiom α to the type tuple $\vec{\tau}$

Here $\vec{\tau}$ is an (arbitrary-size) tuple of types. As Haskell predicates cannot express failure, the Haskell predicate $C\,\vec{\tau}$ is equivalent to the `ilab` predicate $C\ \vec{\tau}$ holds.

Predicates correspond directly to the model of type classes:

$$M \models (C\ \vec{\tau}\ \mathsf{holds}) \iff \vec{\tau} \in M(C)$$
$$M \models (C\ \vec{\tau}\ \mathsf{fails}) \iff \vec{\tau} \notin M(C)$$

The presence of flags within predicates makes possible a simple syntactic definition of the negation of a predicate:

$$\overline{C\ \vec{\tau}\ \mathsf{holds}} = C\ \vec{\tau}\ \mathsf{fails} \qquad \overline{C\ \vec{\tau}\ \mathsf{fails}} = C\ \vec{\tau}\ \mathsf{holds}$$

Contexts. Contexts, or lists of predicates, occur frequently:

$$P ::= \vec{\pi} \qquad \text{Contexts}$$

As above, $\vec{\pi}$ is an arbitrary-size tuple of predicates. We model contexts as conjunctions. A context is modeled if all of its predicates are modeled:

$$M \models P \iff \forall \pi \in P.\, M \models \pi$$

We write the negation of a context P as $\overline{P}$. The negation of a context is modeled if the negation of one of its predicates is modeled:

$$M \models \overline{P} \iff \exists \pi \in P.\, M \models \overline{\pi}$$

Axioms. We turn to the axioms of `ilab`, instance chains. The syntax of instance chains is given by:

$$\alpha ::= (\forall \vec{x}.\, P \Rightarrow \pi)\ ;\ \alpha \mid \varepsilon \qquad \text{Axiom schemes}$$

Because instance chains may contain polymorphic clauses, we refer to them as axiom schemes. Rather than attempting to model axiom schemes directly, we first specialize them to concrete axioms, removing any polymorphism in the process. Intuitively, we specialize an axiom scheme α by enumerating each type tuple that matches the arity of the class mentioned in α, and then attempting to restrict each clause to that tuple.

Some examples may clarify specialization. Consider the instance chain

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

which corresponds to the axiom

$$(() \Rightarrow \mathtt{C\ Int}\ \mathsf{holds})\ ;\ (() \Rightarrow \mathtt{C\ Bool}\ \mathsf{holds})\ ;$$
$$(\forall t.\, (\mathtt{D}\ t\ \mathsf{holds}) \Rightarrow \mathtt{C}\ t\ \mathsf{holds})\ ;\ \varepsilon$$

where we have omitted empty quantifiers. Note that the last clause contains qualified polymorphism. If our set of types were limited to $\{\mathtt{Bool}, \mathtt{Int}, \mathtt{Float}\}$, we would generate the following concrete axioms from this instance chain:

$(() \Rightarrow \mathtt{C\ Int}\ \mathsf{holds})\ ;\ ((\mathtt{D\ Int}\ \mathsf{holds}) \Rightarrow \mathtt{C\ Int}\ \mathsf{holds})\ ;\ \varepsilon$

$(() \Rightarrow \mathtt{C\ Bool}\ \mathsf{holds})\ ;\ ((\mathtt{D\ Bool}\ \mathsf{holds}) \Rightarrow \mathtt{C\ Bool}\ \mathsf{holds})\ ;\ \varepsilon$

$((\mathtt{D\ Float}\ \mathsf{holds}) \Rightarrow \mathtt{C\ Float}\ \mathsf{holds})\ ;\ \varepsilon$

Note particularly the lack of quantifiers: there is no polymorphism in concrete axioms. Alternatively, consider the instance chain

```
instance Eq t ⇒ Eq [t]
```

With the type constructors $\{\mathtt{Int}, \mathtt{[]}\}$, where `[]` constructs the list type, we would generate the following concrete axioms

$$((\mathtt{Eq\ Int}\ \mathsf{holds}) \Rightarrow \mathtt{Eq\ [Int]}\ \mathsf{holds})\ ;\ \varepsilon$$
$$((\mathtt{Eq\ [Int]}\ \mathsf{holds}) \Rightarrow \mathtt{Eq\ [[Int]]}\ \mathsf{holds})\ ;\ \varepsilon$$
$$\vdots$$

We begin formalizing specialization by defining the syntax of concrete axioms, which follows the syntax of axiom schemes closely, but omits the quantifiers.

$$\gamma ::= (P \Rightarrow \pi)\ ;\ \gamma \mid \varepsilon \qquad \text{Concrete axioms}$$

Whether ε denotes a concrete axiom or axiom scheme should be obvious from context. Next, we define the restriction of an axiom α to a particular type tuple $\vec{\tau}$, written $\alpha@\vec{\tau}$. This operation removes the polymorphism from α by attempting to instantiate the type variables in each clause so that the instance head matches $\vec{\tau}$; when that is not possible, the clause is dropped. The definition of $\alpha@\vec{\tau}$ is shown in Figure 1. (We refer to the variables mentioned by a substitution S as $\mathrm{dom}(S)$, and abuse notation by treating the vector of quantified variables as a set.)

We can now give the concrete axioms generated from a given axiom scheme. An empty axiom ε generates exactly one concrete axiom, also ε. If all the clauses in a (non-empty) axiom scheme α are for class C, then the set of concrete axioms generated from α is

$$\{\alpha@\vec{\tau} \mid \vec{\tau} \in \mathit{Type}^{\mathrm{arity}(C)}\}$$

The set of concrete axioms for a given program may be infinite, but because we can determine whether a concrete axiom was specialized from a particular axiom scheme by unification, it is still recursive.

We now describe the modeling of concrete axioms. The empty axiom is trivially modeled:

$$M \models \varepsilon$$

The concrete axiom $(P \Rightarrow \pi)\ ;\ \gamma$ is modeled by the two disjuncts that it represents: if P is modeled, then π must be modeled; alternatively, if $\overline{P}$ is modeled, then γ must be modeled:

$$M \models ((P \Rightarrow \pi)\ ;\ \gamma) \iff M \models P \implies M \models \pi \wedge$$
$$M \models \overline{P} \implies M \models \gamma$$

Axioms correspond to statements about the inclusion or exclusion of particular tuples within the model of a class. Other aspects of type-class systems can be modeled as properties of all the tuples. We describe several such properties next.

Functional dependencies. Our implementation supports the use of functional dependencies, both to constrain instance declarations and to introduce improvement into the deduction algorithm. Functional dependencies were originally proposed for class systems as a mechanism to induce improving substitutions [6]; these improvements, in turn, are only valid because of properties of the underlying relations [10]. Here, we formalize functional dependencies as properties of the models of classes.

The `Elems` class from Section 2.1

```
class Elems c e | c → e
```

has a functional dependency stating that the parameter `c` determines the parameter `e`. We can phrase this with the same language used to describe functions: given two predicates `Elems c e` and `Elems c' e'`, if `c` = `c'` then `e` = `e'`.

We generalize the syntax of functional dependency constraints as follows:

$$X, Y \subseteq \mathbb{N} \qquad \text{Index sets}$$
$$\delta ::= C \ : \ X \rightsquigarrow Y \qquad \text{Functional dependencies}$$

The `Elems` class would generate the constraint $\texttt{Elems} : \{0\} \rightsquigarrow \{1\}$, indicating that the 0^{th} parameter of the class determines the 1^{st} parameter. The class:

```
class F t u v | t v → u
```

would generate the constraint $\texttt{F} : \{0, 2\} \rightsquigarrow \{1\}$.

Modeling these constraints is a straightforward extension of the single-parameter version given above.

$$M \models C : \{X\} \rightsquigarrow \{Y\} \iff \forall \vec{\tau}, \vec{v} \in M(C). \\ \vec{\tau}|_X = \vec{v}|_X \implies \vec{\tau}|_Y = \vec{v}|_Y$$

If $\vec{z}$ is a tuple and X is a subset of $\mathbb{N}$, then we write $\vec{z}|_X$ to refer to the tuple consisting of those elements of $\vec{z}$ indexed by the elements of X. Appealingly, this is exactly the definition of a functional dependency used in the theory of relational databases [10].

Functional dependencies are not the only possible use of the constraint mechanism; for example, it could also be used to model class arities, kind signatures, or Haskell-style superclasses. We describe two of those applications next.

Arities. We have chosen to bake the arity of classes into the definition of models. Alternatively, we could have chosen models over arbitrary sequences of types, with the following structure:

$$M : \mathit{ClassName} \to \mathscr{P}(\mathit{Type}^*)$$

This definition would allow the model of a single class to contain tuples of various lengths. We could then enforce separate arity constraints on classes. An arity constraint of the form $\text{arity}(C) = x$ would require that any tuple in the model of C have length x. We could model this constraint by:

$$M \models \text{arity}(C) = x \iff \forall \vec{\tau} \in M(C). \, \text{length}(\vec{\tau}) = x$$

Note that, unlike the definition of $\models$ heretofore, this relation expresses a property of all tuples in the model of a class.

Kinds. A similar approach could be used to capture the kind signature of a type class. Suppose that the type system were equipped with some set of kinds ranged over by k, and that, for any kind k, the set $\mathit{Type}_k \subseteq \mathit{Type}$, is all the types of kind k. In this setting, classes are assigned kind signatures

$$C : \vec{k}$$

where the n^{th} element of the kind signature is the kind of the n^{th} argument to the class. Kind signatures are validated by:

$$M \models C : \vec{k} \iff \forall \vec{\tau} \in M(C). \, \forall i. \, \tau_i \in \mathit{Type}_{k_i}$$

Programs. We model (the classes and instances of) a program with a pair $A|\Delta$, consisting of a set of axioms A and a set of constraints Δ. In `ilab`, the constraint set will only contain functional dependencies; however, an application to Haskell or Habit would include additional constraints such as kind signatures, superclass constraints, etc. A program is modeled when all of its axioms and all of its dependencies are modeled:

$$M \models A|\Delta \iff (\forall \alpha \in A. \, M \models \alpha) \wedge (\forall \delta \in \Delta. \, M \models \delta)$$

These rules are not generative. A given program $A|\Delta$ may have one, many, or no models. A program with conflicting instance declarations has no models. On the other hand, we do not constrain predicates that are not mentioned in the program. For example, if a program contains neither an assertion that `C Bool holds` nor an assertion that `C Bool fails`, then that program could admit (at least) two models: one in which $\texttt{Bool} \in M(\texttt{C})$ and another in which $\texttt{Bool} \notin M(\texttt{C})$. We say that $A|\Delta$ is consistent if it has at least one model. A predicate π is a theorem of $A|\Delta$ if it holds in all models of the program; that is:

$$\pi \text{ is a theorem of } A|\Delta \iff (M \models A|\Delta \implies M \models \pi)$$

Informally, a particular implementation of a type-class system is sound if it proves only theorems and complete if it proves all theorems. To formalize this, we must formalize our notion of the implementation of a type-class system.

4.2 Predicates, evidence and proof

The preceding section focusses the meaning of type classes; this section builds upon Jones' theory of qualified types to begin describing their implementation.

Jones [4] describes the extension of the polymorphic λ-calculus with qualified types. He uses a notion of 'evidence' to close the gap between the qualifiers in a type and their implementation in a term. For example, the evidence for a type-class predicate `Eq Int` is a function that implements the equality check for integers, while the evidence for a subtype predicate $t \subseteq t'$ is a function embedding values of type t into values of type t'. To capture the use of evidence in computations, Jones extends the term language of the polymorphic λ-calculus with expressions for evidence abstraction, application, and construction. For our purposes, we only need to consider evidence construction; the remainder of Jones' theory can be applied to `ilab` intact.

Jones represents evidence construction with a three-place relation $P \Vdash e : \pi$ indicating that e is evidence for predicate π, given evidence for the predicates in P. He assumes a set of base axioms such as $\emptyset \Vdash \texttt{Eq Int}$ and $\texttt{Eq t} \Vdash \texttt{Eq [t]}$. We will use an alternative relation $A|\Delta \vdash e : \pi$ that diverges from his in two ways:

- His set of base axioms corresponds to our model of a program, so we augment the evidence relation with the program $A|\Delta$; and,
- We will omit the set of assumptions P, as it is trivial to reintroduce and will play no further role in our discussion. It would be valuable for implementing features of Haskell beyond the scope of this paper, such as existential types or GADTs.

Evidence construction does not precisely model the process of proving that a predicate exists. There are a number of predicates in `ilab` that generate no evidence, such as classes without methods or negative predicates. However, we do not wish for all negative predicates to be trivially provable simply because their evidence can always be constructed. Also, evidence construction involves details that are irrelevant for our purposes, such as the implementations of class methods. To avoid these difficulties, we will use proof expressions instead of evidence. Proof expressions capture the reasoning steps made by the deduction algorithm, and there must be a translation from a proof for a predicate π to evidence for π. Proofs may also capture details that are not observable from the generated evidence, such as recursion, naming of common subexpressions, etc. To connect differences in proof expressions to differences in evidence, we introduce a notion of equivalence for proof expressions, written $p \cong p'$. We require that, if $p \cong p'$, then the evidence generated from p is not observably different from the evidence generated from p'. Note that this relation is one-way; it is not likely that $p \cong p'$ for arbitrary proofs p and p' that generate observably equivalent evidence. Equivalence is a statement only about the evidence

generated from proofs, not about what they prove; that is:

$$A|\Delta \vdash p : \pi \quad \wedge \quad p \cong p' \quad \not\Longrightarrow \quad A|\Delta \vdash p' : \pi.$$

Section 5 discusses the details of `ilab` proof expressions and proof equivalence.

We will refer to an algorithm for finding p such that $A|\Delta \vdash p : \pi$ for given $A|\Delta$ and π as a deduction algorithm. The rest of this section will discuss deduction algorithms in general; Section 5 discusses the particular deduction algorithm implemented in `ilab` and the details of its proof expressions.

We can now formalize the notions of soundness and completeness. A deduction algorithm is sound if it only proves predicates that are theorems:

$$\exists p.\, A|\Delta \vdash p : \pi \implies \pi \text{ is a theorem of } A|\Delta \quad \text{(SOUNDNESS)}$$

A deduction algorithm is complete if it can prove any theorem:

$$\pi \text{ is a theorem of } A|\Delta \implies \exists p.\, A|\Delta \vdash p : \pi \quad \text{(COMPLETENESS)}$$

Soundness is an essential property of type-class systems because it connects the implementation to the programmer's model of type classes. We can ensure completeness with sufficient syntactic restrictions on class and instance declarations: for example, Haskell 98's type-class system is complete. However, these syntactic restrictions make expressing many type-class programs difficult or impossible. Alternatively, some implementations use pragmatic measures to ensure termination, such as a (programmer-adjustable) limit to the total number of deduction steps. In `ilab`, we make no effort to ensure completeness or termination, ensuring greater expressiveness as a result. We hope to return to this issue in future work, and find a set of restrictions that ensure completeness while allowing more programs than are allowed by other class systems.

The evidence generated from a deduction algorithm is used in translating programs with type classes. If the deduction algorithm could generate different evidence to prove the same predicate, then the translated program could have multiple meanings. To avoid this incoherence, any two pieces of evidence generated for the same predicate must be semantically indistinguishable, a property Jones calls Uniqueness of Evidence [5]. The notion of evidence being semantically indistinguishable corresponds to proof equivalence, so we restate this for our purposes as Equivalence of Proof:

$$A|\Delta \vdash p : \pi \ \wedge \ A|\Delta \vdash p' : \pi \implies p \cong p' \quad \text{(EOP)}$$

`ilab` type classes are open: new axioms or constraints may be added to existing programs, adding to or refining the meaning of classes. To formalize this, we call a program $A*|\Delta*$ an extension of program $A|\Delta$ if:

1. $A|\Delta$ and $A*|\Delta*$ are consistent; and,
2. $A \subseteq A*$ and $\Delta \subseteq \Delta*$.

Our definition differs from the standard definition of extension in logic in that we require that the exact axioms from A be included in $A*$, not just that $A*|\Delta*$ prove all the theorems of $A|\Delta$. As a consequence, we might hope that, if a predicate is a theorem in both programs, then the proofs in each program will be equivalent. We call this property Stability of Proofs:

$$A|\Delta \vdash p : \pi \ \wedge \ A*|\Delta* \vdash p' : \pi \implies p \cong p' \quad \text{(SOP)}$$

Because the Haskell 98 class system permits no overlap between instances, its proofs are stable. Overlapping instances preclude stable proofs: when more specific overlapping instances are added, the proofs of some predicates will change to use the new instances. By restricting overlap to instance chains, `ilab` restores stability while still allowing many of the programs that could be written using overlapping instances.

We will discuss proofs of `ilab`'s properties following the discussion of the `ilab` deduction algorithm in Section 5.

4.3 Application to other type-class systems

As `ilab`'s extensions to the type-class mechanism could be applied to other languages, the techniques used in the previous subsections to model type classes and to reason about type-class implementations could be applied to other languages, other implementations of Habit, or other type-class systems.

Among the goals of `ilab` was to avoid the complexity of overlapping instances; by applying our modeling techniques to overlapping instances, we can see to what extent we achieved our goal. Overlapping instances are not as modular as `ilab`'s axioms: to determine whether an axiom applies to a predicate, we must determine whether it is the most specific axiom that matches the predicate. Making this determination requires knowing the axioms in the remainder of the program, so it would not be possible to define the meaning of an axiom without the remainder of the program as context. Overlapping instances also preclude the stability of proofs: because a program can be extended with more specific axioms, the proofs of theorems of the original program may change in the extension. While we expected that making implicit aspects of overlapping instances explicit would reduce complexity from the beginning of the `ilab` design process, comparing the models and properties of `ilab` with those of systems with overlapping instances gives a solid basis for this intuition.

5. Mechanics

This section describes our prototype implementation of `ilab`. The presentation is divided into two subsections: Section 5.1 discusses the validation of source axioms and Section 5.2 describes proof expressions and a deduction algorithm for `ilab`.

Functional dependencies will play a larger role in this section than heretofore. Some preliminaries will simplify the remaining discussion. Section 4 used a set Δ to refer to all the functional dependencies in a program. In this section, we will usually only be interested in the functional dependencies that apply to a particular predicate, and so will use the following (overloaded) function:

$$\mathrm{fundeps}_\Delta(C) = \{X \rightsquigarrow Y \mid C : X \rightsquigarrow Y \in \Delta\} \cup \{\mathbb{N} \rightsquigarrow \emptyset\}$$
$$\mathrm{fundeps}_\Delta(C\ \vec{\tau}\ f) = \mathrm{fundeps}_\Delta(C)$$

The set Δ will be omitted when it is obvious from context. To ensure that $\mathrm{fundeps}_\Delta(C)$ is never empty, we have added the dependency $\mathbb{N} \rightsquigarrow \emptyset$ to the functional dependencies for any class. This dependency treats all positions as determining, so it will give the behavior expected were there no functional dependencies at all. Later rules will be able to assume that all classes have at least one functional dependency constraint. Of course, any relation satisfies the dependency $\mathbb{N} \rightsquigarrow \emptyset$, so adding it does not affect the modeling of programs.

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

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

overlap, it is not enough to unify `Elems [t] t` with `Elems [Int] Char`. Rather, we must take the functional dependency for `Elems` into account, and attempt to unify `Elems [t]` with `Elems [Int]`, which succeeds, in this particular case, showing that the instances do overlap. We can generalize this idea to any relation on predicates R and index set Y by writing $\pi R \pi'$ mod Y to indicate the result of $\pi R \pi'$ without considering the elements indexed by Y. Formally, we

define

$$(C\ \vec{\tau}\ f)R(C'\ \vec{v}\ f') \bmod Y \iff (C\ (\vec{\tau}|_{\mathbb{N}\setminus Y})\ f)R(C\ (\vec{v}|_{\mathbb{N}\setminus Y})\ f').$$

The name of this operation is chosen by analogy with modular arithmetic: as arithmetic modulo x does not consider powers of x, so operations modulo the index set Y do not consider the elements indexed by Y.

5.1 Validation

There are two tasks in validating `ilab` axioms: ensuring that there are no overlaps, and checking that the relevant functional dependencies are respected.

To determine whether two instances overlap, we apply a variation of the scheme used in Haskell 98. We say that two instance clauses $\forall\vec{x}.\ P \Rightarrow \pi$ and $\forall\vec{y}.\ P' \Rightarrow \pi'$ overlap if

$$\exists(X \rightsquigarrow Y) \in \mathrm{fundeps}(\pi).\ \exists U.\ \mathrm{dom}(U) \subseteq \vec{x} \cup \vec{y} \\ \wedge\ (\pi \overset{U}{\sim} \pi' \bmod Y \vee \overline{\pi} \overset{U}{\sim} \pi' \bmod Y)$$

where we write $\pi \overset{U}{\sim} \pi'$ to indicate that U is the most general unifier of π and π'. Note that if π and π' mention the same class name C, then $\mathrm{fundeps}(\pi) = \mathrm{fundeps}(C) = \mathrm{fundeps}(\pi')$. Otherwise, π and π' cannot unify, so the choice to quantify over $\mathrm{fundeps}(\pi)$ is irrelevant. Our definition of overlap differs from the Haskell definition in two ways. First, `ilab` axioms have explicit quantifiers, whereas all free type variables in Haskell axioms are implicitly quantified. Second, we take account of the various ways in which predicates can contradict each other. Predicates may overlap if their flags disagree (having proofs that both $C\,\vec{\tau}$ holds and $C\,\vec{\tau}$ fails would be difficult to model, even though the two predicates do not unify). Predicates may also overlap even if they differ in the determined parameters of some functional dependency, as in the example of `Elems [t] t` and `Elems [Int] Char`.

Two axioms α and α' overlap if some clause from α overlaps some clause from α'. This is as strict as the Haskell 98 restriction on overlap; however, because clauses within a single instance chain are free to overlap, `ilab` still offers greater expressivity.

The overlap check is not enough to ensure that instances do not violate functional dependencies; we must also ensure that any quantified variables in determined positions are actually determined. To do this, we make use of the theory of functional dependencies [7, 10].

Let $TV(\pi)$ be all the free type variables mentioned in the type tuple of predicate π. The induced functional dependencies, F_π, of a predicate π are the dependencies

$$\{TV(\pi|_X) \rightsquigarrow TV(\pi|_Y) \mid X \rightsquigarrow Y \in \mathrm{fundeps}(\pi)\}$$

Note that, unlike the class constraints, these are functional dependencies over sets of type variables, not over index sets. 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 $X \rightsquigarrow Y \in F$ and $X \subseteq J_F^+$, then $Y \subseteq J_F^+$.

Now, consider an instance clause $\forall\vec{x}.\ P \Rightarrow C\ \vec{\tau}\ f$. We can ensure that all variables in $\vec{\tau}$ are properly determined if:

$$\forall X \rightsquigarrow Y \in \mathrm{fundeps}(C).\ TV(\vec{\tau}|_Y) \subseteq (TV(\vec{\tau}|_X))^+_{F_P}.$$

We require that all clauses in `ilab` instance chains pass this check.

In previous work on type classes and functional dependencies [7, 17], the process of validating instances against functional dependencies is broken into multiple independent checks. The "consistency" check is incorporated into `ilab`'s expanded overlap check. The "covering" check is implemented as described in the last few paragraphs.

A final note: while a functional dependency does not inherently include or exclude any tuples from a class, each tuple in a class with a dependency excludes all other tuples that would violate the dependency. This can create multiple avenues to prove that a tuple is excluded from a class: either via a negative axiom, or via a (non-overlapping) positive axiom combined with a functional dependency. Luckily, as these proofs generate the same evidence, we can allow both without jeopardizing Equivalence of Proofs.

5.2 Solving

This section describes the inference algorithm used to prove predicates in `ilab`. Intuitively, to prove a predicate π, we try each of the available axioms in sequence. At each step, we compare the current axiom $(\forall\vec{x}.\ P \Rightarrow \pi')\ ;\ \alpha$ to the target predicate π. There are three cases in which we might be able to prove π: either π and π' do not match, so the current clause cannot apply to π, but we can prove π from α; or π and π' match but we can disprove one of the preconditions in P and prove π from α; or π and π' match and we can prove the preconditions.

This intuition is somewhat complicated by the presence of functional dependencies. Recall the `Elems` class from Section 2.1 and suppose we are trying to prove `Elems T U` for some types `T` and `U`. If we can prove `Elems T U'` for some type `U'`$\neq$`U`, then the functional dependency assures us that we will not be able to prove `Elems T U`. Similarly, if we are trying to prove that `Elems T U` fails and can prove that `Elems T U'` holds, then the functional dependency assures us that `Elems T U` fails.

Before formally describing the `ilab` deduction algorithm, we will describe its proof expressions. The structure of `ilab` proof expressions matches the possible reasoning steps mentioned above. To avoid noise, our proof expressions omit steps in which the current axiom does not match the target predicate. Let n range over some countably infinite source of names. We assume that each axiom clause has a unique identifying name (because these are an artifact of the proof expressions, they would not need to be provided by the programmer), and so we will use the following syntax for axiom schemes:

$$\alpha ::= (n : \forall\vec{x}.\ P \Rightarrow \pi)\ ;\ \alpha \qquad \text{Axiom schemes}$$

This differs from the previous syntax only by adding the name n; because names are irrelevant outside construction of proof expressions, this change does not affect the other sections of this paper.

A predicate is usually proved because it matches some axiom clause and the preconditions of that axiom are provable. We describe this case with the proof expression $n(\vec{p})$ where n is the name of the axiom clause that matched, and $\vec{p}$ are the proofs of that clause's preconditions. Alternatively, as discussed above, a negative predicate $C\ \vec{\tau}$ fails may be proven by proving some $C\ \vec{v}$ holds such that for some functional dependency, $\vec{\tau}$ and $\vec{v}$ agree on the determining parameters but disagree on the determined. We capture this case with the proof expression $\mathsf{excl}p$ where p is the proof of the excluding predicate. Finally, axioms that match the target predicate may not apply because their preconditions can be contradicted. We capture that with the proof expression $[n, i, p]p'$ where n identifies the axiom being skipped, i is the index of the contradicted precondition, p is the proof expression for the contradiction, and p' is the remainder of the proof.

Intuitively, only the positive portion of the proof contributes to the construction of evidence, so we can define equivalence for `ilab` proofs inductively by ignoring skip steps. Additionally as mentioned in the last section, it may be possible to prove some

$$\frac{S\,\pi' = \pi \qquad \forall i.\, A \vdash p_i : S\,P_i}{((n : \forall \vec{x}.\, P \Rightarrow \pi')\ ;\ \alpha) \vdash n(\vec{p_i}) : \pi}\ (\textsc{Match})$$

$$\frac{\exists (X \rightsquigarrow Y) \in \text{fundeps}(\pi).\ S\,\pi' = \pi \bmod Y \qquad \exists i.\, A \vdash p_i : \overline{S\,P_i} \qquad \alpha \vdash p : \pi}{((n : \forall \vec{x}.\, P \Rightarrow \pi')\ ;\ \alpha) \vdash [n, i, p_i]p : \pi}\ (\textsc{Step-Contra})$$

$$\frac{\exists (X \rightsquigarrow Y) \in \text{fundeps}(\pi).\ S\,\pi' = \overline{\pi} \bmod Y \qquad S\,\pi' \neq \pi \qquad \forall i.\, A \vdash p_i : S\,P_i \qquad \pi \text{ is negative}}{((n : \forall \vec{x}.\, P \Rightarrow \pi')\ ;\ \alpha) \vdash \mathsf{excl}\ n(\vec{p_i}) : \pi}\ (\textsc{Match-Excl})$$

$$\frac{\forall (X \rightsquigarrow Y) \in \text{fundeps}(\pi).(\pi' \nsim \pi \bmod Y\ \wedge\ \pi' \nsim \overline{\pi} \bmod Y) \qquad \alpha \vdash p : \pi \qquad \pi' \text{ is positive}}{((n : \forall \vec{x}.\, P \Rightarrow \pi')\ ;\ \alpha) \vdash p : \pi}\ (\textsc{Step-Pos})$$

$$\frac{\pi' \nsim \pi \qquad \pi' \nsim \overline{\pi} \qquad \alpha \vdash p : \pi \qquad \pi' \text{ is negative}}{((n : \forall \vec{x}.\, P \Rightarrow \pi')\ ;\ \alpha) \vdash p : \pi}\ (\textsc{Step-Neg})$$

Figure 2. The `ilab` deduction system

predicates either via a negative axiom or via exclusion by a (non-overlapping) positive axiom. To account for this, we make $\mathsf{excl}\ p$ equivalent to any other proof. Equivalence for `ilab` proofs is given by the following assertions:

$$\begin{aligned} p = q &\implies p \cong q \\ p \cong q &\implies [n, i, p']p \cong [n, i', q']q \\ \mathsf{excl}\ p &\cong p' \end{aligned}$$

We can prove a predicate from an axiom set if we can prove it from some axiom in the set:

$$\frac{\exists \alpha \in A.\ \alpha \vdash p : \pi}{A, \Delta \vdash p : \pi}$$

Because `ilab` prohibits overlap between clauses in separate instance chains, there cannot be more than one axiom in the set that matches a given predicate, let alone more than one that proves it.

The deduction rules for $\alpha \vdash p : \pi$ are given in Figure 2. We continue to use the $\text{fundeps}(\pi)$ shorthand instead of passing the constraint set Δ in to all the inference rules. We also omit the regular side condition that substitutions must only mention the quantified variables.

Rule MATCH is intuitive: if an axiom matches the target predicate, and we can prove the preconditions of the axiom, then we can prove the predicate.

Rule STEP-CONTRA is similarly intuitive. In a regular pattern, we require only that the axiom and rule match modulo (the determining parameters of) some functional dependency.

Rule MATCH-EXCL captures the case where we can prove a negative predicate by showing a positive predicate that agrees modulo a functional dependency. We gave an example of this case at the beginning of this section.

Finally, there are two rules for skipping an axiom because it does not match the target predicate. The positive version (STEP-POS) makes the usual allowance for functional dependencies. The negative version (STEP-NEG) does not need to make this allowance because a negative predicate cannot be excluded by functional dependencies.

5.3 Properties of the `ilab` deduction algorithm

Section 4.2 describes several properties of deduction algorithms. We are currently developing formal proofs of those properties for `ilab`; we sketch some of them in this section.

Equivalence and Stability of Proofs are relatively easy to establish because `ilab` does not allow instances to overlap. To generate two inequivalent proofs of the same predicate would require two different axiom clauses that both unify with the predicate being proved. Were these clauses in separate axioms, `ilab` would reject the axioms as overlapping. Were they in the same clause, they would be ordered such that, once the first (whichever it happened to be) applied, `ilab` would not proceed to the second. A similar argument shows stability of `ilab` proofs.

The proof of soundness is along the same lines. If a set of axioms are valid, then none of the conclusions of the clauses in one axiom overlap the conclusions of the other axioms. As a result, the only sources of unsoundness must originate within a single axiom. However, `ilab` will prove at most one conclusion from any single axiom. This rules out all sources of unsoundness.

In this work, we have focussed on expressiveness of `ilab` at the cost of formal termination or completeness properties. We imagine that an approach similar to the one taken by Volpano and Smith [19] to show the undecidability of ML typeability with overloading could be applied to the `ilab` deduction algorithm. We hope to return to issues of completeness and termination in future work.

6. Related work

Although they have been implemented in both Haskell and other languages, such as BitC [15], overlapping instances do not appear to have received much attention in prior research. Peyton Jones et al. [13] consider some of the issues with overlapping instances and other features of Haskell 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 paper.

The use of overlapping instances is not quite as sparse. We have already discussed Swierstra's [18] use of overlapping instances. Kiselyov et al. [9] use overlapping instances and functional dependencies to define a library for heterogeneous lists in Haskell, and Kiselyov and Lämmel [8] take a similar approach in defining an object system in Haskell. In both cases, the authors find ways to avoid overlapping instances, but at the cost of additional code complexity. The Hackage collection of Haskell libraries also includes a number of examples that use overlapping instances.

Heeren and Hage [3] 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. While specifying type-class directives separately allows them to be applied to existing Haskell code, it also limits their usability. 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. Their directives do include some of the uses of explicit exclusion, such as closing classes or ensuring that classes are disjoint.

Maier [10] summarizes the theory of functional dependencies as used in the database community. Jones [6] originally proposed the use of functional dependencies in type-class systems. Hallgren [2] describes some uses of functional dependencies for type-level computation, which we used for examples in Section 2.2.1. Alternative notation for functional dependencies was discussed by Neubauer et al. [12] and by Jones and Diatchki [7].

Sulzmann et al. [17] describe an alternative approach to implementing functional dependencies. In the course of describing their implementation, they establish properties of classes with functional dependencies to make type inference sound, complete, and terminating but do not discuss the soundness of the class system directly. They also do not consider the interaction between overlapping instances and functional dependencies. Later work by Schrijvers et al. [14] proposes an alternative to functional dependencies called type functions and describes an implementation. They explicitly exclude any overlap between type functions.

7. Conclusion and future work

This paper has explored a new type-class feature, instance chains. We have motivated its development from existing Haskell type-level programming, and demonstrated how type-level programming can be simplified and enhanced with instance chains. We have described a semantic framework for reasoning about type classes and their implementations, showed how we can model a type class system with instance chains and functional dependencies, and presented a deduction algorithm for such a type system. There is also significant opportunity for future work in this area; we outline some possibilities next.

Overlap check. The overlap check as implemented in `ilab` is significantly more restrictive than it needs to be. As discussed in Section 2.2.1, the preconditions of instances may prevent them from applying to the same predicate. We would like to improve the `ilab` overlap check so that it takes account of semantic overlap—that is, when two axioms actually cover the same cases—as opposed to the purely syntactic notions of overlap used in this paper. To do so, we will need to determine not just when the hypotheses of two axioms contradict, but also when the possible conclusions of two hypotheses contradict—a potentially expensive search. We hope to apply existing refutation methods to limit this search.

Default implementations. We have discussed coding alternatives using overlapping instances at some length; another use of overlapping instances in existing Haskell code, particularly serialization and generic programming libraries, is to provide default implementations of classes while allowing type-specific implementations to be defined later. We have developed a pattern that encodes default implementations using instance chains instead of overlapping instances. We anticipate testing this pattern against examples of default instances, and hope to report on the results in the future.

Greatest and least models. Section 4 effectively uses the least model of a set of instances to determine its consequences. As it includes failure and functional dependencies, the greatest model of a set of `ilab` instances, unlike in Haskell 98, need not include all predicates. We hope that further study of greatest models will inform alternative approaches to recursive instances and the termination and completeness of deduction algorithms.

Integration into Habit. As discussed in Section 3, the development of `ilab` was an intermediate step in the development of a dialect of Haskell called Habit. Habit includes many features omitted by `ilab`, including Haskell-style superclasses, type-level naturals, explicit representation of binary formats, etc. We hope to extend the techniques used in the modeling and implementation of `ilab` in developing the Habit type-class system. We are also interested to see how features like type-level naturals affect the `ilab` type-class system, and how much we can implement using `ilab` features without baking operations into the compiler. We also believe that instance chains would be a valuable addition to Haskell, or to other Haskell dialects besides Habit.

Acknowledgements. We would like to thank Tim Chevalier, James Hook, Justin Bailey, Andrew Tolmach, and the rest of the HASP group for comments and suggestions on drafts of this paper.

References

[1] I. S. Diatchki and M. P. Jones. Strongly typed memory areas: programming systems-level data structures in a functional language. In *Haskell '06*, pages 72–83, Portland, Oregon, USA, 2006. ACM.

[2] T. Hallgren. Fun with functional dependencies, or (draft) types as values in static computations in Haskell. In *Proc. of the Joint CS/CE Winter Meeting*, 2001.

[3] B. Heeren and J. Hage. Type class directives. In *PADL '05*, pages 253–267. Springer-Verlag, 2005.

[4] M. P. Jones. A theory of qualified types. In B. K. Bruckner, editor, *ESOP '92*, volume 582. Springer-Verlag, London, UK, 1992.

[5] M. P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, 1994.

[6] M. P. Jones. Type classes with functional dependencies. In *ESOP 2000*, pages 230–244, London, UK, 2000. Springer-Verlag.

[7] M. P. Jones and I. S. Diatchki. Language and program design for functional dependencies. In *Haskell Symp.*, pages 87–98, Victoria, BC, Canada, 2008. ACM.

[8] O. Kiselyov and R. Lämmel. Haskell's overlooked object system. Draft; Submitted for publication; online since 10 Sept. 2005.

[9] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell '04*, pages 96–107, Snowbird, Utah, USA, 2004. ACM Press.

[10] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.

[11] J. G. Morris. Experience report: Using Hackage to inform language design. In *Haskell '10*, Baltimore, Maryland, USA, 2010. ACM.

[12] M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. A functional notation for functional dependencies. In *Haskell '01*, Firenze, Italy, September 2001.

[13] S. Peyton Jones, M. P. Jones, and E. Meijer. Type classes: an exploration of the design space. In *Haskell '97*, Amsterdam, The Netherlands, June 1997.

[14] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *IFCP '08*, pages 51–62, Victoria, BC, Canada, 2008. ACM.

[15] J. Shapiro, S. Sridhar, and S. Doerrie. BitC (0.11 transitional) language specification. `http://www.bitc-lang.org/docs/bitc/spec.html`. Last accessed June 15, 2010.

[16] D. Steinitz. Exporting a type class for type signatures. `http://www.haskell.org/pipermail/haskell-cafe/2008-November/050409.html`, November 2008.

[17] M. Sulzmann, G. J. Duck, S. Peyton Jones, and P. J. Stuckey. Understanding functional dependencies via constraint handling rules. *JFP*, 17(1):83–129, 2007.

[18] W. Swierstra. Data types à la carte. *JFP*, 18(04):423–436, 2008.

[19] D. M. Volpano and G. S. Smith. On the complexity of ML typeability with overloading. In *FPCA '91*, pages 15–28, Cambridge, Massachusetts, USA, 1991. Springer-Verlag.

[20] P. Wadler. The expression problem. `http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt`, 1998.

[21] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, pages 60–76, Austin, Texas, USA, 1989. ACM.