Semantical Analysis of Type Classes

J. Garrett Morris
The University of Edinburgh
Garrett.Morris@ed.ac.uk

Abstract

Type classes play two roles in Haskell: they introduce a system of predicates in types, describing when overloaded terms are well-defined, and they define the implementations of class methods, giving meaning to well-typed overloaded terms. Existing approaches to the semantics of type classes account only for the second role; consequently, they are not useful for specifying typing-related properties of the class system itself, or for comparing class system features or implementations.

We propose a model theoretic semantics for type classes, building on Kripke frame models of intuitionistic predicate logic. Our models capture both the logical interpretation of class predicates (and thus their role in typing) and the semantics of class methods (and thus the meanings of overloaded terms). This reveals connections between type class features and standard logical connectives, and allows us to prove properties both of class systems themselves and of the semantics of class methods and overloaded terms. For example, we show that the Haskell '98 class system is sound and complete with respect to its models. On the other hand, we show that no class system with overlapping instances can guarantee that overloaded terms have unambiguous semantics. Our approach thus gives new understanding of class systems, existing features and implementations, and suggests directions for further development.

1. Introduction

Implicit polymorphism, as provided by Hindley-Milner type systems, provides a balance between the safety guarantees provided by strong typing and the convenience of generic programming. The type system is strong enough to guarantee that the evaluation of well-typed terms will not get stuck, while polymorphism and principal types allow programmers to reuse code and omit explicit type annotations. Type classes [20] play a similar role for overloading: they preserve strong typing (ruling out run-time failures from the use of overloaded symbols in undefined ways) without requiring that programmers explicitly disambiguate overloaded expressions. They are one of the defining features of the Haskell programming language: extensions of the core class system have been the subject of much research and vigorous debate, and uses of type classes range from simple overloading to maintaining complex type-encoded invariants. Type classes have been adopted in several other languages, including Isabelle [2] and Coq [17], and have inspired language features such as C++ concepts.

Two Views of Type Classes. Consider a simple use of type classes: the `elem` function, which determines whether a given value is an element of a list.

```
elem          :: Eq t ⇒ t → [t] → Bool
elem x []      = False
elem x (y:ys) = x == y || elem x ys
```

The class system plays two roles in this definition:

- In the type of `elem`, the predicate `Eq t` constrains the instantiation of type variable `t` to types in the `Eq` class (i.e., those types for which the equality operator is defined).
- In the term `x == y`, the operator `==` is overloaded, with its implementation determined by the type of `x` and `y`.

These two roles are closely related: the witness that a type τ is in class `Eq` (i.e., that the predicate `Eq` τ holds) is the implementation of the `==` method. For example, in the dictionary-passing translation proposed by Wadler and Blott [20], the definition above will be translated to one with an explicit parameter for the implementation of `==`. At each use of `elem`, the argument that supplies that implementation of `==` at a particular type τ is constructed from the proof of `Eq` τ. Despite the connection between the meaning of a term and the proofs of predicates in its typing, existing approaches to the semantics of languages with type classes focus only on the language after translation. With such an approach, semantic properties of class systems and individual classes can only be established indirectly, and the role of classes in typing itself is obscured.

Modeling Type Classes. In this paper, we give a model theory for type classes that captures their logical content without losing their term-level significance. This accounts for both roles of type classes. It reveals parallels between apparently ad-hoc class system features, like functional dependencies and overlapping instances, and familiar logical connectives, like negation and disjunction. It relates syntactic characterizations of class system features with underlying semantic notions of validity and implication. Finally, it allows us to characterize the semantics of overloaded terms, and guarantee that their meanings are unambiguous.

Our model theory is based on Kripke frame models [9], a standard approach to the semantics of intuitionistic and modal logics. Kripke models relate points in a model structure, each of which represents a possible state of knowledge, to sentences of a logic. To model type classes, we will consider model structures which include not only what is true (i.e., which predicates hold), but the witnesses of its truth (i.e., the implementations of the class methods). This will allow us to study both the meanings of classes in individual programs, and the properties of class systems as a whole.

Contributions. We can characterize the contributions of this work along two axes: on the one hand, a general notion of models and properties of class systems and, on the other, concrete instances of those notions for the Haskell '98 class system and several of its extensions. First, we define the structure of class system models, and identify several properties relating models to the syntactic elements (class and instance declarations and predicates) and judgments (entailment and well-formedness) of class systems:

- *Soundness.* The most basic property we expect is soundness of the entailment judgment (the component of the type system that describes when one set of predicates implies another). If we can derive an entailment (say that predicates P entail predicates Q) given some class and instance declarations, we would expect that any model of the declarations and of predicates P is also a model of predicates Q.
- *Completeness.* We might also hope that any conclusions we could draw from the models would be provable using the entailment judgment—i.e., if models of P must also be models of Q, we would hope that the judgment that P entails Q is derivable. This is particularly significant because the mechanics of the entailment relation, and thus its consequences for the semantics of terms, are implicit in Haskell programs.
- *Model existence.* Haskell compilers must validate the sets of instances in programs; for example, a program is not allowed to contain two instances for `Eq Bool`, as the semantics of such a program (i.e., the meaning of the == operator) would be ambiguous. Additional restrictions are frequently imposed to assure that searching for entailment proofs (and thus typing derivations) terminates. We will call a set of declarations that passes these checks well-formed, and expect that well-formedness is sufficient to assure that set of declarations have a model. While it is simple to state the converse, we do not know of any class systems for which it holds.
- *Monotonicity.* Haskell programs are modular, where any module may be included in a larger program. We would like to know that our models are modular as well, so conclusions drawn from the models of individual modules hold for models of whole programs. This allows programmers to reason about modules in isolation and assures the coherence of the semantics of overloaded terms in different modules.

Second, we instantiate our approach for the Haskell '98 class system and several of its popular and proposed extensions. This gives a greater understanding of these features, allowing us to relate them to more familiar logical connectives. It provides a grounds to compare variations of the syntactic judgments, as we can relate different judgment forms to the same model structures. Finally, it shows when current formulations fall short of the properties above, suggesting directions for future class system research.

- We formalize a simple generalization of the Haskell '98 class system. The Haskell '98 class system includes notions of positive assertions (i.e., that some type or tuple of types belongs to a class) and implication (as from instance or superclass declarations). We give some characterization of the expressivity of the Haskell '98 class system, and show that it enjoys all the properties listed above.
- We consider two extensions that introduce ideas of negation: negative predicates and functional dependencies. We give a novel characterization of functional dependencies, showing that they are also modeled by negative assertions. Finally, we show that existing presentations of both approaches are incomplete, but that we can show a limited form of completeness (for entailments with consistent assumptions) for functional dependencies.
- We consider two extensions that introduce ideas of disjunction: overlapping and alternative instances. We give the first (we believe) formal treatment of overlapping instances. We show that we can view overlapping instances as introducing biased disjunction, but that this approach necessarily loses monotonicity. This motivates alternative instances, which give a more direct encoding of biased disjunction while preserving monotonicity.

Outline. We begin with a (necessarily) brief introduction to type classes and qualified types (§2). We then define models and model structures for type classes, and give concrete instances accounting for several notions of evidence (§3). We apply our approach to the Haskell '98 class system (§4); in doing so, we introduce several syntactic ideas that will be reused in the remaining sections. We show how this approach can be extended to account for negation (§5) and disjunction (§6). We conclude by discussing related (§7) and future (§8) work.

2. Qualified Types and Type Classes

This section provides a very brief overview of type classes, including their motivation and their role in the typing and semantics of overloaded terms. This is necessarily an incomplete account; Jones [5], among others, gives a more full description.

Type classes are a mechanism for user-defined, extensible overloading; in the context of Hindley-Milner type systems, they do so while preserving principal typing and type inference. A simple motivating example is the treatment of equality: on the one hand, we cannot hope to have a parametric definition of equality applicable at all types; on the other, we would like to be able to abstract over those types for which equality is defined, instead of having to write specialized code for each such type. We can define a type class to solve this problem, such as the following:

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

This class declaration does two things: first, it introduces a new set of predicates, $\mathsf{Eq}\,\tau$ for arbitrary type τ, that holds if equality is defined at type τ; second, it introduces the overloaded class method (==) that implements the equality test. Initially the class has no inhabitants (that is, there are no τ such that $\mathsf{Eq}\,\tau$ holds); types are added to the class by instance declarations, such as the following:

```
instance Eq Bool where
  x == y  =  x && y || not x && not y
instance (Eq t, Eq u) ⇒ Eq (t, u) where
  (x, y) == (x', y')  =  x == x' && y == y'
```

Logically, we can read these as introducing axioms for the `Eq` predicate. In the context of these instances, we would expect predicates such as `Eq Bool` and `Eq (Bool, Bool)` to hold.

Jones's system of qualified types [5] abstracts the details of the class system from the typing relation. In his system, the typing judgment ($P \mid \Gamma \vdash M : \sigma$) contains, in addition to the expected typing environment (Γ), expression (M) and type (σ), a predicate context P, restricting the free type variables in Γ and σ. Predicate entailment is captured by a secondary judgment $\vdash P \Rightarrow Q$ (our notation differs slightly from Jones's). Note that both P and Q are treated as conjunctions, unlike classical sequent calculi. Jones also relies on an ambient context of class instances, superclass declarations, and so on; we will frequently use an explicit context $A|X$, writing $A|X \vdash P \Rightarrow Q$ (A captures instances while X captures restrictions on instances, such as superclasses or functional dependencies). By varying the forms of predicates and definition of the entailment judgment, Jones's system can be adapted to a variety of class system extensions without changing the typing judgment or type inference algorithms. Building on Jones's approach, we will focus exclusively on the entailment judgment.

In a dictionary-passing semantics of type classes [20], an overloaded term is interpreted as a function from dictionaries, containing the type-specific implementations of class methods, to the meaning of the term. For example, the translation (here called `elemD`) of the `elem` function in the introduction would have an additional parameter `eq` for the implementation of ==:

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

Note that the type of `elemD`

$$(t \to t \to \mathtt{Bool}) \to t \to [t] \to \mathtt{Bool}$$

no longer refers to the `Eq` class (nor, indeed, carries any clue about the meaning of the first parameter). The translation must then provide a suitable argument for `eq` at each well-typed use of `elem`. For example, to use `elem` with type

$$(\mathtt{Bool}, \mathtt{Bool}) \to [(\mathtt{Bool}, \mathtt{Bool})] \to \mathtt{Bool},$$

we must prove `Eq (Bool, Bool)`. Such a proof, given the instances (i.e., axioms) above, would be

$$\dfrac{\dfrac{}{\Rightarrow \mathtt{Eq\,Bool}} \quad \dfrac{}{\Rightarrow \mathtt{Eq\,Bool}}}{\Rightarrow \mathtt{Eq\,(Bool, Bool)}}$$

Similarly, if we name the two method implementations `eqBool` and `eqPair` (where `eqPair` itself has two dictionary parameters, corresponding to the assumptions `Eq t` and `Eq u`, a suitable dictionary argument for `elem` is `eqPair eqBool eqBool`. Note the close connection between these: the structure of the dictionary term follows the structure of the proof exactly. Jones augments his entailment judgment to account for the construction of these dictionary parameters; however, we can equally well imagine their construction as a function of the entailment proofs themselves.

Morris has proposed a denotational semantics of class-based overloading [11], in which polymorphic terms are interpreted by their meaning at each ground instance of their types. This approach addresses many complications that arise in typed dictionary-passing translations, and gives meaning to class methods directly rather than via translation. His approach maintains the connection between entailment proofs and the meanings of terms: the semantics of a class method at some ground type τ is provably equal to that of a term constructed by inlining the implementations of class methods from their instance declarations, following the proof that τ is in the given class.

3. Type Classes and their Models

3.1 Models and Model Structures

In this section, we develop the structure of models and describe how they are related to the concrete syntactic elements of programs (instance declarations, predicates, and so on). In doing so, we address three distinguishing features of type classes:

- First, our notion of proof is constructive: a proof of `Eq Int`, must provide an implementation of the equality operator at type `Int` → `Int` → `Bool`.
- Second, because type classes are open, our models must distinguish between negation and absence of proof. The typing of an expression, and thus its semantics, 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 or may not have evidence for `Eq Int`; even in the latter case, the expression may later be used in a context in which we do have evidence for `Eq Int`, so we cannot assume that the predicate does not hold.
- Finally, we must assure the coherence of programs that use type classes: differing proofs of the same predicates (i.e., differing derivations of the same entailment) must not give differing meanings of the same term.

The first points make type classes intuitionistic in nature. Thus, we base our models of classes on Kripke frame models [9], originally developed to model first-order intuitionistic and modal predicate logics. The final point introduces an additional constraint on model structure beyond that required for intuitionistic logic; to address it, we require that models associate a unique evidence value with any true predicate.

Model structures. We begin by defining model structures, following Kripke. A model structure is a triple $\langle G, H, \preceq \rangle$ where

- G is a set of points (or nodes) K, each of which corresponds to a particular "evidential situation" (i.e., collection of knowledge) about the predicates being modeled;
- H is a particular point in G, denoting a root (or initial) collection of knowledge; and,
- $\preceq$ is a relation on points of G, where $K \preceq K'$ if K' represents an extension of the knowledge represented by K.

Following Kripke's characterization of intuitionistic (as opposed to modal) logic, we require that $\preceq$ be reflexive and transitive.

Models. Intuitively, we can think of a single-parameter type class as representing the subset of the set of all types such that particular operations are defined at each type in the set. For example, we could think of `Eq` as identifying the types at which the equality operator is defined, or `Monad` as identifying the type constructors for which the `return` and `>>=` operators are defined. This view can naturally be extended to multiparameter type classes. For example, Wadler and Blott [20] propose a `Coerce` class:

```
class Coerce t u where coerce :: t → u
```

used to coerce integers to floating-point numbers, or single-precision to double-precision floats. In our view of type classes, we would then think of `Coerce` as identifying those pairs of types for which a coercion function is defined. We also wish our models to take account of the semantics of class methods (which, following Jones, we will term evidence). This allows us to capture important properties of class systems, such as coherence, in the models. We also hope that this would make the models themselves sufficient to reason about the meanings of class methods.

Given this intuition, we define models as follows. (For now, we leave the notions of types and of evidence abstract; we will make them concrete shortly.) A model ϕ for a model structure $\langle G, H, \preceq \rangle$ associates with each point $K \in G$ a partial function ϕ_K from class names C and tuples of types $\vec{\tau}$ to evidence values, such that $\phi_K(C, \vec{\tau})$ is the evidence that types $\vec{\tau}$ are in class C, if they are, and is undefined otherwise. Our notion of a model both extends and restricts Kripke's notion, in which a model is (the characteristic function of) the set of provable predicates. We extend the notion of a model to include evidence. The requirement that ϕ_K be a function ensures coherence, as the meaning of any proof of a predicate must correspond to the unique evidence value in the model. On the other hand, we restrict it refer only to (positive) atoms—that is, statements about a single class and tuple of types. In the remainder of the paper, we will consider how a number of features of the class system can be described without extending our notion of models or model structures. To do so, we will define relations $K \vDash \cdot$, relating nodes K of a model structure to syntactic features of class systems, parametrically in terms of models ϕ.

We will use $\mathcal{G}$ to range over model structures, and write $\mathcal{G} \vDash \cdot$ to denote that $H \vDash \cdot$ where $\mathcal{G} = \langle G, H, \preceq \rangle$.

Extension. We require that later points in a model do not contradict earlier points; that is, if $K \preceq K'$ and $\phi_K(C, \vec{\tau}) = e$, then

$\phi_{K'}(C, \vec{\tau}) = e$. This requirement augments Kripke's notion of extension to include evidence.

3.2 Models and Evidence, Concretely

For showing some properties of class systems, such as soundness and monotonicity, general characterizations of models and model structures are sufficient. However, to show model existence and completeness we will need to exhibit particular models. This section defines a simple notion of model structures sufficient for these purposes. In the course of doing so, we will need to pick concrete instantiations of types and evidence. We show how our approach can be adapted to either the dictionary-passing or specialization-based ideas of evidence.

Intuitively, our approach defines a model structure as a set of modeling functions, ordered by inclusion. That is, in a model structure $\langle G, H, \preceq \rangle$, each point $K \in G$ is a partial function from classes and type tuples to evidence values. The model ϕ_K for a given point K is just K itself. Extension is inclusion: $K \preceq K' \iff K \subseteq K'$.

We assume that we have some collection of *ClassName* of class names, and let $C, D, F \in \mathit{ClassName}$. We introduce a simple grammar for type expressions:

Type variables	t	$\in \mathit{TVar}$
Type constructors	T	$\in \mathit{TCon}$
Types	$\mathit{Type} \ni \tau$	$::= t \mid T \mid \tau\,\tau$

This grammar could be straight-forwardly extended to include type-level naturals, lifted types, or other exotica without changing the remainder of the development. We write *GType* for the set of ground types (i.e., types that contain no type variables), and will use $\mathcal{T}$ to range over subsets of *Type*.

Next, we consider evidence. Wadler and Blott [20] originally proposed that type classes could be implemented by translation, introducing dictionaries (tuples of method implementations) to define the behavior of overloaded terms. In their presentation, the result of this translation was typable in (a subset of) Haskell. However, constructor classes requires dictionaries not expressible in Haskell '98: for example, the dictionary witnessing `Monad []` must itself include polymorphic values (the implementations of `return` and >>=). Thus, to apply their approach to Haskell '98 requires translation to a more expressive calculus, such as System F. Sulzmann [18] observes that in the presence of features like functional dependencies, typed dictionary translation requires explicit treatment of type equalities, requiring a yet more expressive target calculus.. Morris [11] proposed an alternative semantics of overloading, building on work by Ohori [13] and Harrison [3]. In his approach, polymorphic expressions denote maps from ground types to (the interpretations of) simply-typed terms. This approach avoids much of the complexity of typed dictionary-passing translations.

Rather than fixing a particular notion of evidence (and thus a particular semantics of polymorphism) for the remainder of the paper, we introduce a simple grammar of evidence expressions *Evid*, assuming some collection of identifiers *ECon*, as follows:

Evidence constructors	d	$\in \mathit{ECon}$
Evidence expressions	$\mathit{Evid} \ni e$	$::= d\,\vec{e}$

and we assume that models provide an interpretation function η from evidence expressions to the underlying semantics of evidence. For Wadler and Blott's original translation-based approach, we can interpret evidence constructors as naming dictionaries or dictionary constructors, and evidence expressions as being applications of dictionary constructors to dictionaries. The case is more complex for modern translation-based approaches: a dictionary is likely to have type parameters as well as other dictionary parameters, and so an evidence constructor must package a dictionary constructor with suitable type arguments. Finally, in Morris's approach, the denotation of a class method is the fixed point of a mapping of types to method implementations. For our purposes, then, evidence constructors identify particular method implementations, and the constructors appearing in a given evidence expression identify those definitions necessary to compute a (non-trivial) implementation of the needed methods. Finally, observe that, because all Haskell types include divergence, we can construct trivial evidence for any predicate in any of these schemes, and will assume that there is some evidence expression corresponding to such trivial evidence should we need it.

Syntax.

Predicates	$\mathit{Pred} \ni \pi, \psi ::= C\,\vec{\tau}$
Contexts	$P, Q ::= \vec{\pi}$
Axioms	$\alpha, \gamma ::= \forall \vec{t}.\, P \Rightarrow \pi$
Class restrictions	$\chi, \xi ::= \mathit{Super}(\forall \vec{t}. \psi \Leftarrow \pi)$
Formulae	$F ::= \langle \pi, e \rangle \mid \langle P, \vec{e} \rangle \mid \langle \alpha, d \rangle \mid \langle \chi, d \rangle$

Modeling.

$$\begin{aligned}
K \vDash e : C\,\vec{\tau} &\iff \phi_K(C, \vec{\tau}) = \eta(e) \\
K \vDash \vec{e} : P &\iff \bigwedge K \vDash e_i : P_i \\
K \vDash d : P \Rightarrow \pi &\iff \forall K' \succeq K. \\
&\qquad (K' \vDash \vec{e} : P) \implies (K' \vDash d\,\vec{e} : \pi) \\
K \vDash d : \alpha &\iff \forall \gamma \in \lfloor \alpha \rfloor.\ K \vDash d : \gamma \\
K \vDash d : \mathit{Super}(\psi \Leftarrow \pi) &\iff \forall K' \succeq K. \\
&\qquad (K' \vDash e : \pi) \implies (K' \vDash e' : \psi) \\
K \vDash d : \chi &\iff \forall \xi \in \lfloor \chi \rfloor.\ K \vDash d : \xi \\
K \vDash A|X &\iff \forall \alpha \in A. K \vDash \alpha \wedge \forall \chi \in X. K \vDash \chi
\end{aligned}$$

Entailment.

$$(\textsc{Atom})\ \frac{Q \subseteq P}{A|X \vdash P \Rightarrow Q} \qquad (\textsc{Conj})\ \frac{\bigwedge \{A|X \vdash P \Rightarrow \pi \mid \pi \in Q\}}{A|X \vdash P \Rightarrow Q}$$

$$(\textsc{Inst})\ \frac{Q \Rightarrow \pi \in \lfloor A \rfloor_{\mathit{Type}} \quad A|X \vdash P \Rightarrow Q}{A|X \vdash P \Rightarrow \pi}$$

$$(\textsc{Super})\ \frac{\mathit{Super}(\pi \Leftarrow \psi) \in \lfloor X \rfloor_{\mathit{Type}} \quad A|X \vdash P \Rightarrow \psi}{A|X \vdash P \Rightarrow \pi}$$

Figure 1: The Haskell '98 Class System

4. Haskell '98

We begin by considering a slight generalization of the class system defined in the Haskell '98 language specification [14]. While relatively straightforward, it includes all the syntactic classes that we will encounter in its more complex extensions, and will allow us to demonstrate the key properties of the modeling relation and model existence proofs. Finally, it is well-behaved: the meta-theoretic properties identified earlier all hold for this system.

4.1 Syntax

This system is based closely on the Haskell '98 class system; we allow multiparameter type classes, and have relaxed restrictions the Haskell report places on the form of instances and superclass declarations, replacing them with a more general syntactic characterization of decreasing axioms and superclasses. The syntax is given at the top of Figure 1. Predicates π pair a class name C and a sequences of types. Contexts P are sequences of predicates, interpreted conjunctively; we will write P_i for the i^{th} predicate in P, and write $\pi \in P$ to indicate that there is an index i such that π is P_i. Axioms capture the logical content of class instances; we assume that the quantified variables $\vec{t}$ are precisely the free type variables of P and π. Finally, we include superclasses, which restrict possible class instances. In Haskell, superclasses form part of class declarations; for example, the declaration

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

requires that any type that is an instance of `Ord` also be an instance of `Eq`. We would represent this superclass as

$$Super(\forall t.\mathtt{Eq}\,t \Leftarrow \mathtt{Ord}\,t).$$

We have made an effort to get the implication the right way around: the superclass requires that if $\mathtt{Ord}\,\tau$ holds, then $\mathtt{Eq}\,\tau$ must also hold, but not vice versa. Finally, we will let A range over sets of axioms, and X range over sets of class restrictions.

We will frequently need to refer to the substitution instances of a given axiom or class restriction. If $\alpha = \forall\vec{t}.\ P \Rightarrow \pi$, $|\vec{t}| = n$, and $\mathcal{T}$ is some set of types, we define

$$\lfloor\alpha\rfloor_{\mathcal{T}} = \{[\vec{\tau}/\vec{t}]P \Rightarrow [\vec{\tau}/\vec{t}]\pi \mid \vec{\tau} \in \mathcal{T}^n\},$$

and similarly, if $\chi = Super(\forall\vec{t}.\psi \Leftarrow \pi)$ and $|\vec{t}| = n$,

$$\lfloor\chi\rfloor_{\mathcal{T}} = \{Super([\vec{\tau}/\vec{t}]\psi \Leftarrow [\vec{\tau}/\vec{t}]\pi) \mid \vec{\tau} \in \mathcal{T}^n\}.$$

Finally, we define $\lfloor A\rfloor_{\mathcal{T}} = \bigcup_{\alpha\in A}\lfloor\alpha\rfloor_{\mathcal{T}}$ and $\lfloor X\rfloor_{\mathcal{T}} = \bigcup_{\chi\in X}\lfloor\chi\rfloor_{\mathcal{T}}$. We will write $\lfloor\cdot\rfloor$ to abbreviate $\lfloor\cdot\rfloor_{GType}$ (that is, to denote the ground instances of the given scheme).

4.2 Modeling and Entailment

The modeling relation $K \vDash F$ relates points in model structures K to formulae F. We begin by defining a notion of a formula for the Haskell '98 class system. As we are concerned not only with which predicates are provable, but with the witnesses of those proofs, our formulae F pair predicates (or contexts) with evidence expressions (or sequences thereof). An axiom or superclass gives rise not to a single witness but to a family of witnesses; therefore, axioms and superclasses are paired with evidence constructors d, not evidence expressions e. We will write $K \vDash e : \pi$ for $K \vDash \langle\pi, e\rangle$, and similarly for the other formulae. We will assume that formulae contain no free type variables; for example, in a judgment $K \vDash e : C\,\vec{\tau}$, the τ_i are all ground types.

We define the relation $K \vDash F$ by cases on F; the cases are given in the center of Figure 1. The cases for predicates and contexts are straightforward: $\langle\pi, e\rangle$ is modeled if $\eta(e)$ (that is, the interpretation of evidence expression e) is the witness for π in the model, and a context corresponds to a conjunction. The modeling relation for axioms and class restrictions are defined in terms of their ground instances. (We write $P \Rightarrow \pi$ to indicate an axiom with no quantified variables, and similarly for superclasses.) In each case, the modeling relation for the ground instances relates syntactic implication to semantic implication. A point K models an instance $P \Rightarrow \pi$ if, at any subsequent point such that P is true, π is also true; the case for superclasses is parallel. Note that while instances define evidence (and thus the evidence is determined by the identifier of the axiom), superclasses only assert that evidence must exist (and so neither e or e' is related to the name d of the superclass). Finally, we extend the modeling relation to sets of axioms and sets of restrictions in the obvious fashion.

Even before considering the entailment judgment, we can already use the structure of models and the modeling relation to start characterizing the expressivity of the Haskell '98 class system.

Lemma 1. *For any declarations $A|X$ and predicate π, if $A|X$ has any models, there is some $\mathcal{G}$ and e such that $\mathcal{G} \vDash A|X \wedge \mathcal{G} \vDash e : \pi$.*

Intuitively, this means that there is no set of axioms that excludes a given predicate, nor any predicate or context that holds only if another predicate does not.[1]

The entailment judgment for the Haskell '98 class system is given at the bottom of Figure 1. It is derived directly from the definition of entailment given by Jones [5]; the rules should be unsurprising. It is straightforward to see that entailment is sound.

Theorem 2 (Soundness). *Suppose $A|X \vdash P \Rightarrow \pi$, where P is a ground context, π is a ground predicate, and let $\mathcal{G} \vDash A|X$. If for each $\pi' \in P$ there is some e' such that $\mathcal{G} \vDash e' : \pi'$, then there is an e such that $\mathcal{G} \vDash e : \pi$.*

Proof. By structural induction on the derivation of $A|X \vdash P \Rightarrow \pi$, relying on the definition of the modeling function. □

As the entailment judgment is closed under substitution, the previous result generalizes to account for (all instances of) entailments with non-ground contexts and goals.

Corollary 3. *Suppose $A|X \vdash P \Rightarrow \pi$; then for any ground instance $P' \Rightarrow \pi'$ of $P \Rightarrow \pi$ and any model $\mathcal{G} \vDash A|X$, if there is some evidence $\vec{e}'$ such that $\mathcal{G} \vDash \vec{e}' : P'$ then there is some e such that $\mathcal{G} \vDash e : \pi'$.*

4.3 Well-formedness and Model Existence

The soundness result (and thus the entailment judgment itself) is only meaningful for sets of declarations that have models. This section introduces a well-formedness condition on declarations $A|X$, and shows that it is sufficient to guarantee model existence.

To assure that a model exists, we must guarantee that there is at most one way to prove any predicate. As the class system is not expressive enough to exclude any predicates, any pair of distinct ground instances with the same conclusion could produce distinct proofs. We define a syntactic coherence criterion that holds for ground instances (paired with their evidence constructors):

$$coh(d : P \Rightarrow \pi, d' : P' \Rightarrow \pi') \iff \pi \neq \pi' \vee (d = d' \wedge P = P').$$

We extend this condition to sets of axiom schemes by

$$coh(A) \iff \forall\gamma,\gamma' \in \lfloor A\rfloor.coh(\gamma,\gamma').$$

Note that we do not assume that γ, γ' are ground instances of different axiom schemes; the conditions for coh account for either overlap between different instances (that is, where d and d' differ) or overlap among instantiations of the same instance ($d = d'$, but P and P' differ). The coh predicate thus accounts for two concrete checks in implementations of the Haskell class system: whether the conclusions of instances unify, and whether there are type variables in the hypotheses of an instance that do not appear in its conclusion.

The Haskell report severely restricts the form of types that appear in instance and superclass declarations. Superclass declarations must contain only predicates of the form $C\,t$ for class C and type variable t, and instance declarations are required to have conclusions of the form $C\,(T\,t_1 \ldots t_n)$ and hypotheses $D\,t_i$ for classes C, D, type constructor T, and type variables t_i. These restrictions assure that the search for an entailment proof must be finite (as the types in the goal must shrink at each application of an instance). We present a relaxed version of this restriction, achieving the same goal but allowing many more instances and accounting for multi-parameter type classes. We begin by defining a notion of the size of

[1] The details of a number of definitions and the proofs of several properties, including this one, are available in the non-anonymized appendix.

types and predicates; this reflects how deeply type constructors are nested.

$$\|t\| = 1 \quad \|T\,\vec{\tau}\| = 1 + \max\{\|\tau_i\|\} \quad \|C\,\vec{\tau}\| = \max\{\|\tau_i\|\}$$

where max of an empty sequence is 0. We now define

$$decr(\forall\vec{t}.\,P \Rightarrow \pi) \iff \|\pi\| > \max\{\|P_i\|\}$$
$$decr(Super(\forall\vec{t}.\psi \Leftarrow \pi)) \iff \|\psi\| \geq \|\pi\|,$$

and write $decr(A|X)$ if $decr(\alpha)$ for all $\alpha \in A$ and $decr(\chi)$ for all $\chi \in X$. The *decr* predicate is still stricter than necessary to avoid infinite proofs or unbounded proof search; we believe this is an interesting area for further research into class systems and their expressivity.

Finally, we must account for superclass restrictions. We begin by characterizing when a single axiom α respects the restrictions X. Suppose that $\alpha = \forall\vec{t}.\,P \Rightarrow \pi$; we define

$$resp(\alpha, A|X) \iff \forall(Super(\forall\vec{u}.\psi' \Leftarrow \psi)) \in X.$$
$$\exists\vec{\tau}, \vec{\upsilon}.\ [\vec{\tau}/\vec{t}]\pi = [\vec{\upsilon}/\vec{u}]\psi \implies A|X \vdash [\vec{\tau}/\vec{t}]P \Rightarrow [\vec{\upsilon}/\vec{u}]\psi'.$$

Intuitively, this means that for each superclass that applies to the conclusion of α, we can prove the conclusions of the superclass from the hypotheses of α and the axioms $A|X$. Note that this condition is only meaningful if $\alpha \notin A$; otherwise, it holds trivially. We can extend this definition to sets of axioms as follows:

$$resp(A|X) \iff \forall\alpha \in A.\exists A' \subseteq (A \setminus \{\alpha\}).$$
$$resp(A'|X) \wedge resp(\alpha, A'|X).$$

A surprising consequence of this definition is that no non-trivial axiom set respects a set of cyclic superclass restrictions. Cyclic superclasses are prohibited by the Haskell report; however, as future work, we hope to consider conditions that would not exclude them.

Finally, we can define the well-formedness condition for axioms A under restrictions X as the conjunction of the above criteria:

$$wf(A|X) \iff resp(A|X) \wedge decr(A|X) \wedge coh(A).$$

Theorem 4 (Model existence)**.** *If $wf(A|X)$, then there is some model $\mathcal{G}$ such that $\mathcal{G} \vDash A|X$.*

The proof depends on the iterative construction of a coherent model structure $\mathcal{I}_{A|X}$ for $A|X$. We show that, assuming $wf(A|X)$, the result of such a construction is defined, and that it models $A|X$.

4.4 Completeness

We can now show the completeness of the Haskell '98 class system. Intuitively, completeness means that any semantic implication should correspond to a derivation of the entailment judgment. To avoid triviality, we insist that the axioms must have at least one non-trivial model, which we can ensure using the $wf(\cdot)$ predicate from the previous section.

Theorem 5 (Completeness)**.** *If $wf(A|X)$, P and Q are ground contexts, and, for all $\mathcal{G} \vDash A|X$, if there is some $\vec{e}$ such that $\mathcal{G} \vDash \vec{e} : P$ then there is some $\vec{e}'$ such that $\mathcal{G} \vDash \vec{e}' : Q$, then there is a derivation of $A|X \vdash P \Rightarrow Q$.*

The proof is via the contrapositive: if there is no derivation, we construct a model of $A|X$ and P that is not a model of Q. We use the induced model $\mathcal{I}_{A|X}$, extending it to model P and restricting it to not model Q. That this is well-defined can be seen following Lemma 1, and that it is still a model of $A|X$ is guaranteed by the lack of a derivation of Q.

4.5 Modularity and Monotonicity

Haskell programs are composed of modules, each with its own collection of classes and instances. We would like to guarantee that the models of an individual module are somehow related to the models of the whole program. In particular, we would hope that an overloaded expression does not change meaning in different modules. We capture this with a property we call monotonicity. We begin by defining extension of programs: for well-formed $A|X$ and $A'|X'$, we define

$$A|X \sqsubseteq A'|X' \iff A \subseteq A' \wedge X \subseteq X'.$$

We would then expect that if $A|X \sqsubseteq A'|X'$, then any model of $A'|X'$ is also a model of $A|X$.

Theorem 6 (Monotonicity)**.** *Given well-formed $A|X, A'|X'$, if $A|X \sqsubseteq A'|X'$ and $\mathcal{G} \vDash A'|X'$, then $\mathcal{G} \vDash A|X$.*

We might expect that monotonicity would be straightforward for the Haskell '98 class system: as predicates cannot be excluded, a model of a given $A|X$ may include (almost) arbitrary additional information. We can confirm this intuition as follows. It is immediately apparent that if $A|X \vdash P \Rightarrow Q$ then (by a derivation with identical structure) $A'|X' \vdash P \Rightarrow Q$. Theorem 2 alone does not guarantee that the evidence be the same. However, if we further observe that since $A \subseteq A'$ and $wf(A'|X')$, there cannot be any new axioms in A' that prove any predicate provable from A, we can conclude that $A'|X'$ cannot introduce new evidence for any predicate provable from $A|X$.

Syntax.

Flags	$Flag \ni f$	$::= \ \varepsilon \mid \texttt{fails}$
Predicates	$Pred \ni \pi, \psi$	$::= \ C\,\vec{\tau}\,f$

Modeling.

$$K \vDash e : C\,\vec{\tau} \iff \phi_K(C, \vec{\tau}) = \eta(e)$$
$$K \vDash e : C\,\vec{\tau}\,\texttt{fails} \iff \forall K' \succeq K.\ \langle C, \vec{\tau}\rangle \notin \mathsf{dom}(\phi_{K'})$$

Figure 2: Class System with Negative Predicates

5. Negation in Predicates and Instances

We next consider two extensions to the Haskell class system that introduce negative information, allowing predicates to be excluded from models. The first extension we consider adds a new form of predicate, $C\,\vec{\tau}$ `fails` which denotes that $\vec{\tau}$ cannot appear in C. This is intuitively simple, and is an important component of previous work on increasing the expressivity of class systems [12]. Studying this system will allow us to introduce the intuitionistic treatment of negation, and to discuss extending the well-formedness mechanisms to account for contradiction between predicates. The second extension we consider is functional dependencies, a (much maligned) mechanism for the satisfiability of predicates to inform typing and type inference. We will show that functional dependencies can be modeled by the negation of (sets of) predicates.

5.1 Negative Predicates

The system with negative predicates is a simple extension of the Haskell '98 class system. The new syntax is given at the top of Figure 2. We introduce flags, which can be either empty or `fails`; each predicate now has a flag in addition to a class name and list of types. A predicate $C\,\vec{\tau}$ (a positive predicate) has the same meaning as such a predicate in Haskell '98; a predicate $C\,\vec{\tau}$ `fails` (a negative predicate) holds only if $\vec{\tau}$ is not in class C. We will write $\overline{\pi}$ to indicate the result of flipping the flag of π; that is:

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

We also introduce a simple syntactic characterization of contradiction: two predicates π and π' contradict (written $\pi \lightning \pi'$) if they have the same parameters but opposite flags:

$$(C\,\vec{\tau} f) \lightning (C'\,\vec{\tau}' f') \iff C = C' \wedge \vec{\tau} = \vec{\tau}' \wedge f \neq f'.$$

The remaining syntactic elements are as defined before (Figure 1), but taking account of the new form of predicates.

In the previous section, we argued that the Haskell class system was not expressive enough to exclude predicates: for any $A|X$ and predicate π, if $A|X$ has any models, it has a model that includes π. Our notion of refutation can be viewed as reading that intuition in reverse: a predicate $C\,\vec{\tau}\,\texttt{fails}$ holds at some point in a model if there is no later point in the model at which $C\,\vec{\tau}$ holds. This is stated symbolically at the bottom of Figure 2. The remaining cases of the modeling relations are defined as before. Just as ambiguous sets of instances cannot be modeled, we can see that this definition does not allow contradictory predicates to be modeled. That is, for any class C, types $\vec{\tau}$, and point K, at most one of $K \vDash e : C\,\vec{\tau}$ and $K \vDash e : C\,\vec{\tau}\,\texttt{fails}$ holds.

The entailment judgment is changed only to incorporate the new syntax of predicates. The statement and proof of soundness are also identical. However, we can see immediately that this entailment judgment is not complete: for example, while it includes implication, it does not account for the contrapositive. For a simple example, the following entailment is not derivable

$$\{\forall t.\ C\,t, D\,t \Rightarrow E\,t\}|\emptyset \vdash E\,\tau\,\texttt{fails}, C\,\tau \Rightarrow D\,\tau\,\texttt{fails}$$

but the semantic implication must hold in any model of the axioms. Extending the entailment relation to account for this case is not entirely trivial: we must either support some form of negated context directly, or introduce general disjunction among predicates, but neither option has clear implications for the meanings of class methods. We discuss disjunction further in the next section.

5.1.1 Well-formedness with Negative Predicates

We can significantly generalize our notion of well-formedness in the presence of negative predicates. For Haskell '98, because we could not exclude any predicates, any pair of instances with unifying conclusions could introduce ambiguity. However, this is no longer the case:

```
instance C t ⇒ D t where ...
instance C t fails ⇒ D t where ...
```

These instances cannot introduce ambiguity, regardless of the other instances in the program, as there can be no model structure $\mathcal{G}$ and type τ such that both $\mathcal{G} \vDash e : \texttt{C}\,\tau$ and $\mathcal{G} \vDash e' : \texttt{C}\,\tau\,\texttt{fails}$.

To account for such contradictory sets of predicates, we introduce a new syntactic judgment, $A|X \vdash P \lightning Q$ that holds when predicates P and Q are mutually inconsistent. This judgment will be based on our entailment judgment, and is thus sound but not complete. We define

$$A|X \vdash P \lightning P' \iff (A|X \vdash P \Rightarrow \overline{P'}) \vee (A|X \vdash P' \Rightarrow \overline{P}),$$

where $A|X \vdash P \Rightarrow \overline{Q}$ denotes that there is some i such that $A|X \vdash P \Rightarrow \overline{Q_i}$. We can use this judgment to improve our definition of overlap: we allow axioms to overlap if their hypotheses are mutually inconsistent. We begin with the case for ground axioms:

$$coh_{A|X}(d : P \Rightarrow \pi, d' : P' \Rightarrow \pi') \iff \\ \pi \neq \pi' \vee (d = d' \wedge P = P') \vee A|X \vdash P \lightning \overline{P'}.$$

and give the expected generalizations for sets of axiom schemes:

$$coh(A|X) \iff \forall \gamma, \gamma' \in \lfloor A \rfloor . coh_{A|X}(\gamma, \gamma')$$

Ambiguity is no longer the only reason that a set of axioms might not have a model; the axioms also might be inconsistent. To exclude such cases, we introduce a consistency check for instances. The definition closely parallels that of the coherence check; however, instead of ruling out pairs of instances that prove the same predicate, it rules out pairs of instances that prove contradictory predicates. We begin with the case for ground axioms:

$$cons_{A|X}(P \Rightarrow \pi, P' \Rightarrow \pi') \iff \overline{\pi} \neq \pi' \vee A|X \vdash P \lightning P'$$

and extend to sets of axiom schemes:

$$cons(A|X) \iff \forall \gamma, \gamma' \in \lfloor A \rfloor . cons_{A|X}(\gamma, \gamma').$$

Finally, we can state the well-formedness condition, using the obvious extensions of *decr* and *resp* to this setting:

$$wf(A|X) \iff resp(A|X) \wedge decr(A|X) \wedge coh(A|X) \wedge cons(A|X).$$

5.1.2 Model Existence

Showing model existence for the system with negative predicates is a straightforward extension of the argument for Haskell '98.

Theorem 7 (Model existence)**.** *If $wf(A|X)$, then there is some model $\mathcal{G}$ such that $\mathcal{G} \vDash A|X$.*

Where for Haskell '98 we iteratively constructed a coherent model structure $\mathcal{I}_{A|X}$, here we must assure that the model structure is both coherent and consistent. Again, the assumption of $wf(A|X)$ is sufficient to assure that the result of such a construction is well-defined and models $A|X$.

The monotonicity of this class system can be shown by an easy extension of the similar argument for the Haskell '98 class system, relying on the coherence and consistency components of $wf(A|X)$.

5.2 Functional Dependencies

We next discuss functional dependencies, an extension of the Haskell class system that captures dependencies among the arguments of multi-parameter type classes. Functional dependencies were originally introduced to address several problems that arose in the use of multi-parameter type classes [7]. For a simple example, consider the following class definition, which abstracts over container types

```
class Collects c e where
  empty :: c
  insert :: e → c → c
```

Intuitively, the predicate $\texttt{Collects}\,\tau\upsilon$ holds if type τ is a collection of elements of type υ; `empty` and `insert` provide for the construction of τ values from υ values. However, this class is not usable in practice. For example, the type of `empty` is problematic:

```
empty :: Collects c e ⇒ c
```

Note that type variable `e` is unconstrained by the type `c`; as different instantiations of `e` might give rise to different witnesses of `Collects c e`, the meaning of `empty` is potentially ambiguous. Functional dependencies resolve this problem by allowing the programmer to require that the element type `e` is determined by the collection type `c`:

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

Formally, the declaration `c → e` requires that, for any two provable predicates $\texttt{Collects}\,\tau\upsilon, \texttt{Collects}\,\tau'\upsilon'$, if $\tau = \tau'$, then $\upsilon = \upsilon'$; that is, the set of tuples $\langle \tau, \upsilon \rangle$ such that $\texttt{Collects}\,\tau\,\upsilon$ holds is a (partial) function $GType \rightharpoonup GType$.

We represent declared functional dependencies by restrictions $C : Y \rightsquigarrow Z$, where Y and Z are sets of indices identifying parameters of class C. Abstract syntax is given at the top of Figure 3, extending the syntax in Figure 1. For example, the declaration above would be represented by the restriction $\texttt{Collects} : \{0\} \rightsquigarrow \{1\}$. If $\vec{\tau}$ and $\vec{\upsilon}$ are sequences of types and Y is an index set, we write $\vec{\tau} =_Y \vec{\upsilon}$ to indicate that $\tau_i = \upsilon_i$ for each $i \in Y$.

Syntax.

$$\begin{array}{lrcl} \text{Index sets} & Y, Z & \subseteq & \mathbb{N} \\ \text{Class restrictions} & \chi & ::= & C : Y \rightsquigarrow Z \mid \ldots \end{array}$$

Modeling.

$$K \vDash C : Y \rightsquigarrow Z \iff \forall K_2 \succeq K_1 \succeq K. \langle C, \vec\tau \rangle \in \mathsf{dom}(\phi_{K_1}) \wedge \langle C, \vec\upsilon \rangle \in \mathsf{dom}(\phi_{K_2}) \wedge \vec\tau =_Y \vec\upsilon \implies \vec\tau =_Z \vec\upsilon$$

Entailment.

$$(\text{IMPR})\ \frac{A|X \vdash S\,P \Rightarrow S\,Q \quad A|X \vdash S \text{ improves } P}{A|X \vdash P \Rightarrow Q}$$

$$(\text{FUNDEP})\ \frac{\begin{array}{c} A|X \vdash P \Rightarrow C\,\vec\tau, C\,\vec\upsilon \\ (C : Y \rightsquigarrow Z) \in X \quad \vec\tau =_Y \vec\upsilon \quad \vec\tau \overset{S}{\sim}_Z \vec\upsilon \end{array}}{A|X \vdash S \text{ improves } P}$$

Figure 3: Class System with Functional Dependencies

5.2.1 Modeling and Entailment

We can formalize our intuitive description to give a modeling rule for functional dependency declarations, shown in the center of Figure 3. Intuitively, $C : Y \rightsquigarrow Z$ is modeled at point K if, for any subsequent point K_1 such that K_1 models $C\,\vec\tau$, no later point K_2 models a predicate $C\,\vec\upsilon$ that violates the functional dependency with respect to $\vec\tau$.

The connection to negation may not be immediately apparent. However, we can restate the modeling rule as excluding all predicates that would violate the functional dependency:

$$K \vDash C : Y \rightsquigarrow Z \iff \forall K_2 \succeq K_1 \succeq K. \langle C, \vec\tau \rangle \in \mathsf{dom}(\phi_{K_1}) \wedge \vec\tau =_Y \vec\upsilon \wedge \vec\tau \neq_Y \vec\upsilon \implies \langle C, \vec\upsilon \rangle \notin \mathsf{dom}(\phi_{K_2}).$$

This follows the form of intuitionistic refutation exactly.

Jones's original presentation of functional dependencies does not extend the entailment judgment at all. Rather, he relies on improving substitutions in typing and type inference [6]. Intuitively, given some context P, S is an improving substitution for P if every satisfiable ground instance of P is a satisfiable ground instance of $S\,P$. Jones permits the use of improving substitutions in type inference: he shows that if type inference were to have inferred the type $P \Rightarrow \tau$ for some term M, and S improves P, then the type $S\,P \Rightarrow S\,\tau$ could also be inferred for M without compromising principality. This approach is not immediately helpful to us, as we do not rely on the typing judgment. However, we can adapt Jones's approach to entailment, as shown at the bottom of Figure 3. Rule (IMPR) allows us to apply improving substitutions in entailment: if every satisfiable ground instance of P is also a satisfiable ground instance of $S\,P$, then showing that $S\,P \Rightarrow S\,Q$ is sufficient to show that $P \Rightarrow Q$. (Intuitively, this is an application of the transitivity of entailment.) Rule (FUNDEP) introduces improving substitutions: if P entails $C\,\vec\tau$ and $C\,\vec\upsilon$, such that for some functional dependency restrictions $C : Y \rightsquigarrow Z$, $\vec\tau =_Y \vec\upsilon$, then any unifying substitution S of the Z-indexed types in $\vec\tau$ and $\vec\upsilon$ must be an improving substitution for P.

Lemma 8. *Let $A|X \vdash S$ improves P and $\mathcal{G} \vDash A|X$. If there is some $Q \in \lfloor P \rfloor$ and $\vec e$ such that $G \vDash \vec e : Q$, then $Q \in \lfloor S\,P \rfloor$.*

The proof follows from the condition $C : Y \rightsquigarrow Z \in \lfloor X \rfloor$ of rule (FUNDEP) and the assumption $\mathcal{G} \vDash A|X$. Soundness of the entailment judgment is shown by induction as before, relying on Lemma 8 for case (IMPR).

5.2.2 Well-formedness and Model Existence

As in the system with negative predicates, we must verify that in declarations $A|X$ the axioms A do not violate functional dependency restrictions in X. We can also generalize our overlap check to account for functional dependencies; for example, the axiom $\forall tu. F\,t\,u \Rightarrow C\,t$ is unambiguous if there is a functional dependency requirement $F : \{0\} \rightsquigarrow \{1\}$.

We start by defining a notion of contradiction among predicates. Two ground predicates are contradictory if they violate some functional dependency requirement:

$$C\,\vec\tau \;\lightning_X\; C'\,\vec\upsilon \iff C = C' \wedge \exists (C : Y \rightsquigarrow Z) \in X. \vec\tau =_Y \vec\upsilon \wedge \vec\tau \neq_Z \vec\upsilon.$$

We extend this to sequences of ground predicates in the obvious fashion:

$$A|X \vdash P \lightning Q \iff \exists \pi, \pi'. A|X \vdash P \Rightarrow \pi \wedge A|X \vdash Q \Rightarrow \pi' \wedge \pi \lightning_X \pi'.$$

Two ground axioms are inconsistent if their conclusions violate some functional dependency restriction and their hypotheses are not inconsistent:

$$\mathit{cons}_X(P \Rightarrow C\,\vec\tau, Q \Rightarrow C'\,\vec\upsilon) \iff C \neq C' \vee (\forall (C : Y \rightsquigarrow Z) \in X. \vec\tau =_Y \vec\upsilon \implies \vec\tau =_Z \vec\upsilon) \vee (A|X \vdash P \lightning Q).$$

We can extend this to sets of axioms by

$$\mathit{cons}(A|X) \iff \forall \gamma, \gamma' \in \lfloor A \rfloor. \mathit{cons}_X(\gamma, \gamma').$$

Paralleling our specification of the Haskell '98 overlap check, this predicate unifies two checks performed by concrete implementations: if γ and γ' are ground instances of the same axiom, it corresponds to Jones's "covering" check; if they are ground instances of different axioms, it corresponds to his "consistency" check [8]. Note that our consistency check is more permissive than Jones's: his consistency check does not allow distinct instances to have inconsistent conclusions, even if their hypotheses are inconsistent.

Similarly, we can extend the coherence check to allow overlapping instances if their hypotheses are inconsistent

$$\mathit{coh}_X(d : P \Rightarrow \pi, d' : P' \Rightarrow \pi') \iff \pi \neq \pi' \vee (d = d' \wedge P = Q) \vee (A|X \vdash P \lightning Q),$$

with the obvious extension to sets of axioms

$$\mathit{coh}(A|X) \iff \forall \gamma, \gamma' \in \lfloor A \rfloor. \mathit{coh}_X(\gamma, \gamma').$$

Finally, we define well-formedness using our new formulations of inconsistency and overlap. The remaining predicates are defined as for the Haskell '98 class system (§4.3):

$$\mathit{wf}(A|X) \iff \mathit{resp}(A|X) \wedge \mathit{decr}(A|X) \wedge \mathit{coh}(A|X) \wedge \mathit{cons}(A|X).$$

In showing model existence for the system with negative predicates, we defined an iterative construction of a coherence, consistent model structure $\mathcal{I}_{A|X}$ for declarations $A|X$. We can apply the same approach now, replacing the notion of contradiction with that appropriate for functional dependencies.

Theorem 9 (Model existence)**.** *If $\mathit{wf}(A|X)$ then there is some model $\mathcal{G}$ such that $\mathcal{G} \vDash A|X$.*

5.2.3 Completeness

Unfortunately, our entailment judgment for functional dependencies is not complete; as a simple example, consider:

$$\{\forall t.\, \mathtt{I}\, t\, t\} | \{\mathtt{F} : \{0\} \rightsquigarrow \{1\}\} \vdash \mathtt{F}\, t\, \mathtt{Int}, \mathtt{F}\, t\, \mathtt{Bool} \Rightarrow \mathtt{I}\, t\, \mathtt{Int}.$$

From the functional dependency on F, we can conclude that there can be no τ such that both $\mathtt{F}\,\tau\,\mathtt{Int}$ and $\mathtt{F}\,\tau\,\mathtt{Bool}$ holds, and thus the entailment is trivially modeled. Unlike the similar example for negative predicates above, however, a class system with just functional dependencies cannot express negation directly. This means that we can show a kind of pseudo-completeness: non-trivial semantic implications (i.e., those which do not depend on inconsistent assumptions) are syntactically derivable.

Theorem 10. *Let $A|X$ be well-formed, and P and Q be sets of ground predicates. If there exists some $\mathcal{G}_0$ such that $\mathcal{G}_0 \vDash A|X \wedge \mathcal{G}_0 \vDash P$, and for all $\mathcal{G} \vDash A|X$, $\mathcal{G} \vDash P \implies \mathcal{G} \vDash Q$, then $A|X \vdash P \Rightarrow Q$.*

5.2.4 Functional Dependencies and Negative Predicates

We have shown that functional dependencies are modeled similarly technique to negative predicates. This section briefly outlines other connections between the two ideas.

In a system with both functional dependencies and negative predicates, we can express the modeling relation for functional dependencies in terms of negative predicates, as follows:

$$K \vDash C : Y \rightsquigarrow Z \iff \forall K' \succeq K. K' \vDash C\,\vec{\tau} \implies (\forall \vec{\upsilon}.(\vec{\tau} =_Y \vec{\upsilon} \wedge \vec{\tau} \neq_Z \vec{\upsilon} \implies K' \vDash C\,\vec{\upsilon}\,\mathtt{fails})).$$

Similarly, a system with both features admits additional entailment rules, such as using functional dependencies to prove negative predicates:

$$\frac{A|X \vdash P \Rightarrow C\,\vec{\tau}' \quad C : Y \rightsquigarrow Z \in X \quad \vec{\tau} =_Y \vec{\tau}' \quad \vec{\tau} \neq_Z \vec{\tau}'}{A|X \vdash P \Rightarrow C\,\vec{\tau}\,\mathtt{fails}}$$

This is a specific instance of a more general rule

$$\frac{A|X \vdash P \Rightarrow \pi' \quad \pi \lightning \pi'}{A|X \vdash P \Rightarrow \overline{\pi}}$$

This rule is trivial for systems with only negative predicates, as the only case where $\pi \lightning \pi'$ is $\overline{\pi} = \pi'$. Alternatively, systems without negative predicates have no use for a rule of this form, since they cannot express the negation of predicates even should they be excluded from their models.

5.2.5 Some Thoughts on History

One benefit of our approach is that we can use it to compare different implementations of the same or similar ideas. We conclude our discussion of functional dependencies by discussing two such applications: comparing different well-formedness conditions and comparing functional dependencies and type functions.

There have been several, non-equivalent statements of the well-formedness conditions for functional dependencies. The one we have given is adapted from Jones's original criteria [7]. Jones gives an informal specification of the use of functional dependencies to compute improvements during type inference; significantly, however, he indicates that this may be an iterative process. Subsequently, Sulzmann et al. [19] give a translation of functional dependencies into constraint handling rules (CHRs), and evaluate the resulting sets of CHRs to compute improving substitutions during type inference. They identify sets of declarations accepted by Jones's criteria but which cause divergence under their translation, and introduce stronger well-formedness conditions to rule out such declarations. One might reasonable ask whether this divergence is a consequence of their translation, or whether it captures an underlying property of the declarations that would cause divergence in any approach. We hope that our approach can provide an answer. That is, as there are only finitely many applications of (IMPR) for a given set of predicates, and our entailment judgment is complete (for non-explosive assumptions), we hope that we can give finite derivations of all improving substitutions. We believe that formalizing this argument is significant future work.

Type families [16] provide an alternative to functional dependencies. The application of our approach to type families is not immediate: type families appear in types, rather than as predicates, and their meaning is defined by rewriting. However, our approach could be applied to type families by interpreting each n-argument type family F as an $n+1$-place predicate F' such that $F'\,\tau_1 \ldots \tau_n\,\tau$ holds if and only if $F\,\tau_1\, \ldots\, \tau_n$ reduces to τ and interpreting each use of F by a type variable constrained by F'. We believe that investigating such an approach would be valuable future work.

6. Disjunction in Instances

Finally, we consider two extensions to the Haskell class system that incorporate limited notions of disjunction. In general, disjunction seems to introduce serious complications for the meanings of terms. For example, what would witness a proof of the predicate $\mathtt{Eq}\,t \vee \mathtt{Show}\,t$? Which method implementations would such a witness provide? The extensions in this section address these issues by allowing disjunction in the introduction of class instances, permitting piecewise definition of instances so long as the individual cases are, in some sense, well-ordered. That is, they allow disjunction of the form $\pi \vee (\neg\pi \wedge \psi)$ (which is not intuitionistically equivalent to $\pi \vee \psi$). The first extension, called the overlapping instances extension, is well-known and has a long history in Haskell implementations. Its theoretical properties are less well-studied. We will show that it is incompatible with monotonicity. Morris and Jones [12] proposed the second extension, alternative instances, in an attempt to permit the kinds of programs written with overlapping instances without compromising monotonicity.

6.1 Overlapping Instances

Haskell '98 prohibits any overlap among instances, as it could lead to semantic ambiguity. However, this restriction is frequently inconvenient in practice. For example, Haskell's concrete syntax provides special-case handling of lists of characters (strings); we might want the Prelude-defined Read and Show classes to provide similar handing. However, doing so would seem to require giving instances such as the following:

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

The overlapping instances extension permits such instances, so long as the conclusion of one is a substitution instance of the other; when solving a given predicate, the most specific instance is used [15]. For example, with this extension, the two instances above would be allowed (as Show [Char] is the image of Show [t] under [Char/t]); the compiler would use the first instance to witness Show [Char] and the second for any other Show [t] predicate. The remainder of the section will attempt to formalize this idea and describe some of its consequences.

Modeling. An appealing aspect of the overlapping instances extension is that it requires no extension to the existing syntax of Haskell '98 predicates and instances, and only requires small changes to notions of entailment. We therefore assume the syntax from Figure 1, and begin by describing what it means for one instance α to be more specific than another α' (written $\alpha \lesssim \alpha'$). Intuitively, it means that any conclusion of the more specific instance should also be a conclusion of the less specific instance; that is, writing $\llbracket\alpha\rrbracket_{\mathcal{T}}$ for $\{\pi \mid (P \Rightarrow \pi) \in \lfloor\alpha\rfloor_{\mathcal{T}}\}$, we define

$$\alpha \lesssim_{\mathcal{T}} \alpha' \iff \llbracket\alpha\rrbracket_{\mathcal{T}} \subsetneq \llbracket\alpha'\rrbracket_{\mathcal{T}}.$$

We can now refine our notion of the ground instances of an axiom α: they should only include those instances where there is not some more specific instance that could prove the same conclusion. That is, if $\alpha = \forall\vec{t}.\, P \Rightarrow \pi$, $|\vec{t}| = n$, and $\mathcal{T}$ is some set of types,

$$\begin{aligned}\lfloor\alpha\rfloor^{A}_{\mathcal{T}} = \{[\vec{\tau}/\vec{t}]P \Rightarrow [\vec{\tau}/\vec{t}]\pi \mid \vec{\tau} \in \mathcal{T}^{n},\\ \forall\alpha' \in A, \alpha' \lesssim_{\mathcal{T}} \alpha \implies \lfloor[\vec{\tau}/\vec{t}]\pi\rfloor_{\mathcal{T}} \# \llbracket\alpha'\rrbracket_{\mathcal{T}}\}.\end{aligned}$$

Our notion of modeling a single ground axiom is no different than before; however, our notion of modeling sets of axioms is refined to use our new definition of substitution instances:

$$K \vDash A \iff \forall\alpha \in A, \forall\gamma \in \lfloor\alpha\rfloor^{A}_{GType}. K \vDash \gamma.$$

Entailment. Entailment must take account of overlapping instances. For example, consider the two axioms $\alpha = d : \forall t.\, C\,t$, $\alpha' = d' : \forall t.\, D\,t \Rightarrow C\,[t]$, and the sets $A = \{\alpha\}$, $A' = \{\alpha, \alpha'\}$. We can trivially conclude that $A|\emptyset \vdash C\,[\mathtt{Int}]$, but cannot conclude that $A'|\emptyset \vdash C\,[\mathtt{Int}]$, as in the latter case we must use axiom α' rather than α, and cannot prove the hypothesis $D\,\mathtt{Int}$. However, note that the (INST) rule (Figure 1) is defined in terms of the substitution instances of an axiom; restating this rule in terms of $\lfloor\alpha\rfloor^{A}_{Type}$ instead of $\lfloor\alpha\rfloor_{Type}$ gives an entailment judgment for overlapping instances. The soundness of this system is immediate.

Well-formedness. We still cannot allow arbitrary overlapping instances. For example, given both $\alpha = \forall t.\, P \Rightarrow C\,(t, \mathtt{Int})$ and $\alpha' = \forall t.\, Q \Rightarrow C\,(\mathtt{Int}, t)$, we have neither $\llbracket\alpha\rrbracket \subseteq \llbracket\alpha'\rrbracket$ nor $\llbracket\alpha\rrbracket \supseteq \llbracket\alpha'\rrbracket$, but both instances can be used to prove $C\,(\mathtt{Int}, \mathtt{Int})$, and thus allowing both axioms in the same program would introduce ambiguity. Therefore, we replace our previous coherence check with a notion of instances being well-ordered. This constraint is vacuously satisfied if they have no overlap; otherwise, it requires that one instance conclusion be a proper substitution instance of the other.

$$ord(\alpha, \alpha') \iff \llbracket\alpha\rrbracket \# \llbracket\alpha'\rrbracket \lor \llbracket\alpha\rrbracket \subsetneq \llbracket\alpha'\rrbracket \lor \llbracket\alpha\rrbracket \supsetneq \llbracket\alpha'\rrbracket.$$

We can extend this notion to sets of axioms:

$$ord(A) \iff \forall\alpha, \alpha' \in A, ord(\alpha, \alpha')$$

Otherwise, well-formedness is defined as for Haskell '98:

$$wf(A|X) \iff resp(A|X) \land decr(A|X) \land ord(A)$$

Model existence. We can define the induced model $\mathcal{I}_{A|X}$ following the same steps as we did for Haskell '98, but making use of our refined notion of substitution instances.

Theorem 11 (Model existence)**.** *If $wf(A|X)$ then there is some model $\mathcal{G}$ such that $\mathcal{G} \vDash A|X$.*

Completeness. The completeness of this system follows by an identical argument to that for Haskell '98 (§4.4).

Monotonicity. Unfortunately, the system with overlapping instances violates monotonicity, and as a consequence translations of programs with overlapping instances can demonstrate incoherence. Consider the pair of axiom sets A, A' given earlier, and observe that $C\,[\mathtt{Int}] \in \lfloor A\rfloor$, $(D\,\mathtt{Int} \Rightarrow C\,[\mathtt{Int}]) \in \lfloor A'\rfloor$. Pick some model $\mathcal{G} = \langle G, H, \preceq\rangle$ such that $\mathcal{G} \vDash A'$ and there is some point $K \succeq H$ such that $K \nvDash D\,\mathtt{Int}$. K then need not model $C\,[\mathtt{Int}]$. On the other hand, if it does not, then $K \nvDash A$, and so monotonicity fails.

In their discussion of the overlapping instances extensions, Peyton Jones et al. [15] identify that context reduction (i.e. predicate simplification) can interact with overlapping instances to introduce incoherence. They propose an alternative approach, delaying context reduction for non-ground predicates. However, as the preceding discussion makes clear, the overlapping instances extension allows incoherence at any use of a non-ground instance, not only when they are used for simplification and so no modification of context reduction can restore coherence.

Syntax.

$$\begin{array}{lrcl}\text{Clauses} & \kappa & ::= & \forall\vec{t}.\, P \Rightarrow \pi\\ \text{Axioms} & \alpha & ::= & \varepsilon \mid \kappa;\alpha\end{array}$$

Modeling.

$$\begin{aligned}&K \vDash \varepsilon\\ K \vDash (d : P \Rightarrow \pi\ ;\ \alpha) &\iff \forall K' \succeq K.\\ &\qquad (K' \vDash e : P \implies K' \vDash d(e) : \pi)\\ K \vDash A &\iff \forall\alpha \in A, \forall\gamma \in \lfloor\alpha\rfloor. K \vDash \gamma\end{aligned}$$

Entailment.

$$(\textsc{Inst})\ \frac{\alpha \in A \quad \alpha \vdash P \Rightarrow \pi}{A|X \vdash P \Rightarrow \pi}$$

$$(\textsc{Match})\ \frac{(Q \Rightarrow \pi) \in \lfloor\forall\vec{t}.\, Q' \Rightarrow \pi'\rfloor_{Type} \quad A|X \vdash P \Rightarrow Q}{(\forall\vec{t}.\, Q' \Rightarrow \pi'\ ;\ \alpha) \vdash P \Rightarrow \pi}$$

$$(\textsc{Step})\ \frac{\forall Q.(Q \Rightarrow \pi) \notin \lfloor\forall\vec{t}.\, Q' \Rightarrow \pi'\rfloor_{Type} \quad \alpha \vdash P \Rightarrow \pi}{(\forall\vec{t}.\, Q' \Rightarrow \pi'\ ;\ \alpha) \vdash P \Rightarrow \pi}$$

Figure 4: Class System with Alternative Instances

6.2 Alternative Instances

Instance chains were proposed by Morris and Jones [12] as a mechanism to support more expressive type classes than could be defined using overlapping instances, while avoiding the incoherence that comes from overlapping instances. Their system combines functional dependencies, negative predicates, and a form of syntactically-linked alternative instance declarations. In this paper, we treat each of these features separately. This section will formalize their alternative instance declarations in isolation.

Intuitively, alternative instances play a similar role to overlapping instances. However, where overlapping instances can occur anywhere in a program, and are disambiguated automatically by the compiler based on the instances in scope, alternative instances are linked syntactically, and their ordering is specified by the programmer. For example, returning the Show example from the previous section, we could provide special treatment for lists of characters using the following declaration:

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

The first instance clause would be used to show lists of characters, while all other lists would use the second. Programs may contain multiple instance declarations for the same class, so long as they are non-overlapping. The given declaration, for example, would not preclude separate instances for `Show Int` or `Show (a, b)`, but would preclude an instance `Show [Int]`.

6.2.1 Syntax and Modeling

We describe the abstract syntax of alternative instances at the top of Figure 4; an axiom α now consists of a sequence of individual clauses κ, where ε is the empty sequence. We impose the syntactic restriction that all clauses in a single declaration must assert predicates of the same class. In formulae, we will assume that an axiom is paired with a sequence of names, one for each clause, rather than a single instance name; we will write $d : \forall \vec{t}.\, P \Rightarrow \pi \;;\; \alpha$ to indicate that d is the name paired with the head clause in the axiom.

The ground instances of axioms are more complex than before; for example, we would expect the ground instance of the axiom `C Char ; ∀t. D t ⇒ C t ; ε` to include `C Char ; D Char ⇒ C Char ; ε` and `D Int ⇒ C Int ; ε`, but neither `C Char ; D Int ⇒ C Int ; ε` nor `D Char ⇒ C Char ; ε`. We describe the substitution instances of an axiom as follows. The substitution instances of an individual clause $\kappa = \forall \vec{t}.\, P \Rightarrow \pi$ (where $|\vec{t}| = n$) restricted to predicate ψ are

$$\lfloor \kappa \rfloor^{\psi}_{\mathcal{T}} = \{ d : [\vec{\tau}/\vec{t}]P \Rightarrow [\vec{\tau}/\vec{t}]\pi \mid \vec{\tau} \in \mathcal{T}^n, [\vec{\tau}/\vec{t}]\pi = \psi \}.$$

We next define the substitution instances of an axiom restricted to a given predicate ψ:

$$\lfloor \varepsilon \rfloor^{\psi}_{\mathcal{T}} = \{\varepsilon\}$$

$$\lfloor \kappa;\alpha \rfloor^{\psi}_{\mathcal{T}} = \begin{cases} \{\kappa';\alpha' \mid \kappa' \in \lfloor \kappa \rfloor^{\psi}_{\mathcal{T}}, \alpha' \in \lfloor \alpha \rfloor^{\psi}_{\mathcal{T}}\} & \text{if } \lfloor \kappa \rfloor^{\psi}_{\mathcal{T}} \neq \emptyset. \\ \lfloor \alpha \rfloor^{\psi}_{\mathcal{T}} & \text{otherwise} \end{cases}$$

Finally, we can define the ground instances of an axiom as its ground instance for each predicate to which it applies. If the conclusion of each clause in α is for class C with arity n, then

$$\lfloor \alpha \rfloor_{\mathcal{T}} = \bigcup \{ \lfloor \alpha \rfloor^{C\,\vec{\tau}}_{\mathcal{T}} \mid \vec{\tau} \in \mathcal{T}^n \}.$$

We can now extend the modeling relation to alternative instances, as shown in the center of Figure 4. We begin by defining the modeling relation for a single axiom. The empty axiom is trivially modeled. The ground axiom $\kappa;\alpha$ is modeled if κ is modeled; as the first syntactically matching clause always may apply, we need not consider later clauses in the chain. Finally, a set of axioms schemes is modeled if all the ground instances of its axioms are modeled.

6.2.2 Entailment

We define a new entailment rule for alternative instances, given at the bottom of Figure 4; the remaining rules are unchanged from Figure 1. The entailment rule for instances must consider each clause in a chain in sequence. We describe this process using an auxiliary judgment $\alpha \vdash P \Rightarrow \pi$, describing the conditions under which the axiom α proves an entailment. (We leave the declarations $A|X$ implicit in describing this judgment to decrease notational overload.) There are two cases. Rule (MATCH) applies when the head clause in a chain matches the goal predicate; in this case, the axiom proves the given predicate if the hypotheses of the clause hold. As we are considering alternative instances in isolation, we have no way to refute the hypotheses of the clause, and so no need to consider later clauses in the chain. Rule (STEP) applies when the head clause does not match the goal. There is no rule for the empty chain ε, as it proves no predicates.

Soundness for the entailment judgment is shown by induction. The new case is in the treatment of alternative instances. The proof follows from the observation that, letting $\alpha = \forall \vec{t}.\, P \Rightarrow \pi' \;;\; \alpha'$, if there is a Q such that $(Q \Rightarrow \pi) \in \lfloor \forall \vec{t}.\, P \Rightarrow \pi' \rfloor$, then $\lfloor \alpha \rfloor_{\pi} \ni (Q \Rightarrow \pi \;;\; \gamma)$ for some $\gamma \in \lfloor \alpha' \rfloor_{\pi}$, and, conversely, if there is no Q such that $(Q \Rightarrow \pi) \in \lfloor \forall \vec{t}.\, P \Rightarrow \pi' \rfloor$, then $\lfloor \alpha \rfloor_{\pi} = \lfloor \alpha' \rfloor_{\pi}$.

6.2.3 Well-formedness and Model Existence

We must extend our well-formedness conditions to account for sequences of clauses; our task is simplified because, as only the first clause that matches a given predicate can prove it, we need only consider the head clause of ground instances. Alternative instances limit overlap to clauses in the same instance. We begin by defining coherence for ground instances:

$$coh((d : P \Rightarrow \pi \;;\; \alpha), (d' : P' \Rightarrow \pi' \;;\; \alpha')) \iff \pi \neq \pi' \vee (d = d' \wedge P = P').$$

We can extend this check to sets of axioms as expected:

$$coh(A) \iff \forall \gamma, \gamma' \in \lfloor A \rfloor, coh(\gamma, \gamma').$$

The remaining well-formedness predicates are extended to chains clause-wise, and we can define the well-formedness predicate as expected.

Model existence and completeness follows by an almost identical arguments to those for Haskell '98, since we need only consider the head clause of each ground instance.

Theorem 12 (Model existence)**.** *If $wf(A|X)$ then there is some model $\mathcal{G}$ such that $\mathcal{G} \vDash A|X$.*

Theorem 13 (Completeness)**.** *If $wf(A|X)$, P and Q are ground contexts, and, for all $\mathcal{G} \vDash A|X$, if there is some $\vec{e}$ such that $\mathcal{G} \vDash \vec{e} : P$ then there is some $\vec{e}'$ such that $\mathcal{G} \vDash \vec{e}' : Q$, then there is a derivation of $A|X \vdash P \Rightarrow Q$.*

Intuitively, overlapping instances are non-monotonic because overlap may be introduced by any instance in a program. In contrast, alternative instances must be syntactically connected, and so are modular.

Theorem 14 (Monotonicity)**.** *For well-formed $A|X$ and $A'|X'$, if $A|X \sqsubseteq A'|X'$, $G \vDash A|X$ and $G' \vDash A'|X'$, then for each $K' \in G'$ there is a $K \in G$ such that $K \subseteq K'$.*

The proof is direct, relying on the assumptions $A \subseteq A'$ and the component $coh(A')$ of $wf(A'|X')$.

6.3 Alternatives and Negation

Recall our example overlapping instances α and α'. Suppose that $\mathcal{G}$ is some model of $\{\alpha, \alpha'\}$; we see that for any τ,

$$\mathcal{G} \vDash C\,\tau \iff (\exists \upsilon.\tau = [\upsilon] \wedge G \vDash D\,\upsilon) \vee (\forall \upsilon.\tau \neq [\upsilon]).$$

There is, of course, a third possibility: there is an υ such that $\tau = [\upsilon]$, but $G \nvDash D\,\upsilon$. However, as we have argued, the Haskell '98 class system is not strong enough to express such a possibility.

The original presentation of instance chains included both negative predicates and alternative instance. With access to negation, we can extend entailment to allow clauses to be skipped if their hypotheses can be disproved. For example, we might expect the following addition to the previous entailment rules

$$\frac{(Q \Rightarrow \pi) \in \lfloor \forall \vec{t}.\, Q' \Rightarrow \pi' \rfloor \quad A|X \vdash P \Rightarrow \overline{Q} \quad \alpha \vdash P \Rightarrow \pi}{\forall \vec{t}.\, Q' \Rightarrow \pi' \;;\; \alpha \vdash P \Rightarrow \pi}$$

where, as before, we write $A|X \vdash P \Rightarrow \overline{Q}$ to indicate that there is some i such that $A|X \vdash P \Rightarrow \overline{Q_i}$. This shows that these extensions are more expressive in combination than either is in isolation.

7. Related Work

The model theory of intuitionistic logic was originally developed by Kripke [9], including both the structure of models and the treatment of negation. We use his formulations almost exactly.

Type classes were originally proposed by Wadler and Blott [20]. Several generalizations of their system have been proposed; we have relied on Jones's system of qualified types [5]. These systems have all treated the semantics of class methods by dictionary-passing translation, and have not addressed the logical content of type classes.

Heeren and Hage [4] propose type class directives, auxiliary declarations which can exclude types from classes, require that classes be disjoint, and provide additional feedback to programmers about type errors. As with our presentation of negative instances, this mechanism can assure that particular predicates do not appear in the models of well-typed programs; however, negated predicates cannot appear in the hypotheses of instances or the predicates in type qualifiers.

Functional dependencies are foundational to the study of relational databases. Their application to type classes was originally proposed by Jones [7]. Jones and Diatchki [8] present an intuitive description of multi-parameter type classes as relations on types; they also formalize "covering" and "consistency" conditions for validating instances against functional dependency restrictions. Sulzmann et al. [19] interpret functional dependencies via constraint handling rules, deriving explicit equality constraints from predicates. They use a operational semantics based on rewriting sets of predicates, rather than the denotational approach we pursue in this paper; their approach also seems to require ruling out some instances that are acceptable in our presentation and that of Jones.

Sulzmann [18] gives a formal account of type class solving via translation into Constraint Handling Rules, and describes dictionary construction as proof extraction from CHR programs. He observes a number of challenges in typed dictionary construction, particularly to do with superclasses and functional dependencies, and proposes a system that includes both conventional class dictionaries and witnesses of type equality to account for them. Finally, he shows soundness of both a simpler, untyped dictionary translation algorithm and of his approach using type equalities.

Type families, originally introduced by Schrijvers et al. [16], provide an alternative to functional dependencies based on explicitly introducing equality constraints and rewriting rules to the type system. We believe that our approach could also be applied to type families, although we have not tried to do so. In particular, as type families do not determine method implementations, the coherence requirements could be relaxed for type family declarations.

The overlapping instances extension has a long history in Haskell compilers (we can find references to it in release notes dating to 1993), but has received relatively little formal study. Peyton Jones et al. [15] discuss the overlapping instances extension informally and describe some of its consequences. Eisenberg et al. [1] describe an extension that allows type families to be defined by overlapping equations, but only for closed type families.

Morris [10] presents a model theory for functional dependencies and instance chains, also based on Kripke frames. He does not consider completeness or monotonicity properties of the system he presents, and his treatment does not include overlapping instances.

8. Conclusions and Future Work

We have presented a model theory for type classes, used it to characterize several key semantic properties of class systems, and applied it to the Haskell '98 class system and several of its extensions. In doing so, we have identified several directions for future work.

Negation. Proper support for negation significantly increases the expressivity of class systems; however, the existing approaches are either implicit (in the case of functional dependencies) or incomplete (in the case of negative predicates). We believe that finding a complete treatment of negation, particular in combination with the approaches to alternative or overlapping instances, would represent a significant advance in the expressivity of type classes.

Stronger notions of completeness. The notion of completeness we have presented in this paper applies only to entailments among sets of ground predicates. We hope to expand this to a stronger notion of completeness, deriving non-ground entailments when their ground instances are necessary semantically. However, this will introduce challenges for several of the extensions in this paper. In particular, such a properties does not hold for the formulations of overlapping instances nor alternative instances we have presented.

Functional dependencies and equality. Our presentation of functional dependencies relies on a non-standard treatment of equality, based on computing and applying improving substitutions. We would like to relate this to standard axiomatizations of equality and their models; we are particularly interested in whether the judgments given in our presentation are complete, and, if not, whether they can be repaired to be so.

References

[1] R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *POPL '14*, pages 671–683, San Diego, California, USA, 2014. ACM.

[2] F. Haftmann and M. Wenzel. Constructive type classes in Isabelle. In *TYPES 2006*, pages 160–174, Nottingham, UK, 2006. Springer.

[3] W. 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.

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

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

[6] M. P. Jones. Simplifying and improving qualified types. In *FPCA '95*, pages 160–169, La Jolla, California, USA, 1995. ACM.

[7] M. P. Jones. Type classes with functional dependencies. In *ESOP '00*, pages 230–244, Berlin, Germany, 2000. Springer-Verlag.

[8] M. P. Jones and I. S. Diatchki. Language and program design for functional dependencies. In *Haskell '08*, pages 87–98, Victoria, BC, Canada, 2008. ACM.

[9] S. A. Kripke. Semantical analysis of modal logic I. Normal propositional calculi. *Zeitschrift fur mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.

[10] J. G. Morris. *Types Classes and Instance Chains: A Relational Approach*. PhD thesis, Portland State University, 2013.

[11] J. G. Morris. A simple semantics for Haskell overloading. In *Haskell '14*, pages 107–118. ACM, 2014.

[12] J. G. Morris and M. P. Jones. Instance chains: Type-class programming without overlapping instances. In *ICFP '10*, Baltimore, MD, 2010. ACM.

[13] A. 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.

[14] S. Peyton Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.

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

[16] 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.

[17] M. Sozeau and N. Oury. First-class type classes. In *TPHOLs 2008*, pages 278–293, Montreal, Canada, 2008. Springer.

[18] M. Sulzmann. Extracting programs from type class proofs. In *PPDP '06*, pages 97–108, Venice, Italy, 2006. Springer.

[19] 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.

[20] 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.

A. Proofs

A.1 Haskell '98 Modeling and Entailment

Lemma 1. *For any declarations $A|X$ and predicate π, if $A|X$ has any models, there is some $\mathcal{G}$ and e such that $\mathcal{G} \models A|X \wedge \mathcal{G} \models e : \pi$.*

Proof. Let $\mathcal{G}_0 = \langle G_0, H_0, \preceq \rangle$, $\pi = C\,\vec{\tau}$, and suppose that $\mathcal{G}_0 \models A|X$. There are two cases. First, suppose that there is some e and some $K \succeq H_0$ such that $K \models e : \pi$. Then let $\mathcal{G}_1 = \langle G_0, K, \preceq \rangle$, and we have that $\mathcal{G}_1 \models e : \pi$. Alternative, suppose there are no such e and K. Let $e_\perp^\psi$ be the trivial evidence value for a predicate ψ, and let P be the least set such that $\pi \in P$ and if $\mathit{Super}(\psi' \Leftarrow \psi) \in \lfloor X \rfloor$ and $\psi \in P$ then $\psi' \in P$. Finally, for $K_0 \in G_0$ define

$$K_1 = K_0 \cup \{\langle\langle C, \vec{\tau}\rangle, e_\perp^{C\,\vec{\tau}}\rangle \mid C\,\vec{\tau} \in P, \langle C, \vec{\tau}\rangle \notin \mathsf{dom}(K_0)\},$$

and let $\mathcal{G}_1 = \langle \{K_1 \mid K_0 \in G_0\}, H_1, \preceq \rangle$. We trivially have that $\mathcal{G}_1 \models e_\perp^\pi : \pi$ and that $\mathcal{G}_1 \models A$, and we have that $\mathcal{G}_1 \models X$ from the construction of each K_1 and the definition of P. ☐

A.2 Haskell '98 Model Existence

Theorem 4. *If $\mathit{wf}(A|X)$, then there is some model $\mathcal{G}$ such that $\mathcal{G} \models A|X$.*

The remainder of the section is dedicated to the proof of this result. We begin by defining, for any $A|X$, a notion of an induced model $\mathcal{I}_{A|X}$. We then show that, assuming $A|X$ is well-formed, $\mathcal{I}_{A|X} \models A|X$. This latter step requires a brief detour: to simplify the proof that $\mathcal{I}_{A|X} \models X$, we will show that, for ground contexts P and Q, if there is a derivation of $A|X \vdash P \Rightarrow Q$, then there is a derivation that does not make use of the superclasses X.

Definition 15. Let M range over assignments of evidence expressions to predicates; that is, $M : \mathit{Pred} \rightharpoonup \mathit{Evid}$. Given a set of axioms A, we define a function $I(\cdot)$ as follows. Enumerate the possible predicates $\pi_1, \ldots, \pi_j, \ldots$. Define

$$I_j(M) = \begin{cases} M \uplus \{\langle \pi_j, d\,\vec{e}\rangle\} & \\ \quad \text{if } (d : P \Rightarrow \pi_j) \in \lfloor A \rfloor, \langle P_i, e_i\rangle \in M \text{ for each } i; \\ M \quad \text{otherwise.} \end{cases}$$

where $M \uplus N$ is $M \cup N$ if $M(\pi) = N(\pi)$ for each $\pi \in \mathsf{dom}(M) \cap \mathsf{dom}(N)$, and undefined otherwise. We define $I(M) = \bigcup_j I_j(M)$, and define $I^\omega(M)$ to be the limit of the iteration of $I(\cdot)$ on M. (That is, $I^\omega(M)$ is the limit of the sequence $M, M_1, M_2, \ldots$ where each $M_{n+1} = I(M_n)$.) Note that, by the definition of $\uplus$, $I^\omega(M)$ is undefined if some iteration of I would introduce ambiguity. Finally, we define an interpretation function $[\![-]\!]$ by

$$[\![M]\!] = \{\langle\langle C, \vec{\tau}\rangle, \eta(e)\rangle \mid \langle C\,\vec{\tau}, e\rangle \in M\}.$$

Unfortunately, we cannot expect $[\![I^\omega(M)]\!] \models X$ for arbitrary starting assignment M: in particular, M may contain the hypotheses of superclasses but not their conclusions. We say that a context P respects the superclasses X if

$$\mathit{resp}(P, X) \iff \forall \mathit{Super}(\psi \Leftarrow \pi) \in \lfloor X \rfloor . \pi \in P \implies \psi \in P.$$

and extend this notion to assignments $\mathit{resp}(M, X)$ in the obvious fashion. Now, we can define the induced model $\mathcal{I}_{A|X}$ of $A|X$ by

$$G_{A|X} = \{[\![I^\omega(M)]\!] \mid \mathit{resp}(M, X) \text{ and } I^\omega(M) \text{ is defined}\}$$
$$\mathcal{I}_{A|X} = \langle G_{A|X}, [\![I^\omega(\emptyset)]\!], \preceq \rangle.$$

In the course of showing that $\mathcal{I}_{A|X} \models A|X$, we will rely on the guarantees provided by $\mathit{wf}(A|X)$. To simplify this reasoning, we begin by showing that derivations of ground entailments from well-formed declarations do not require superclasses.

Lemma 16. *If $\mathit{wf}(A|X)$, $\mathit{resp}(P, X)$ and $A|X \vdash P \Rightarrow \pi$, where P is a ground context and π a ground predicate, then there is a derivation $A|\emptyset \vdash P \Rightarrow \pi$ (i.e., one which does not make use of superclasses).*

Proof. By induction on the size of A and the structure of the derivation. The only interesting case is for uses of superclasses. Suppose that we have a derivation

$$\frac{\mathit{Super}(\pi \Leftarrow \pi') \in \lfloor X \rfloor \qquad \dfrac{\vdots}{A|X \vdash P \Rightarrow \pi'}}{A|X \vdash P \Rightarrow \pi}$$

There are then three cases.

- In the first case, we have $\pi' \in P$. By the assumption $\mathit{resp}(P, X)$, we must also have $\pi \in P$, and so $A|\emptyset \vdash P \Rightarrow \pi$ trivially.
- In the second case, we have some further superclass $\mathit{Super}(\pi' \Leftarrow \pi'') \in \lfloor X \rfloor$ and $A|X \vdash P \Rightarrow \pi''$; this case reduces to one of the others by the induction hypothesis.
- In the final case, we have some axiom $Q \Rightarrow \pi' \in \lfloor A \rfloor$ such that $A|X \vdash P \Rightarrow Q$. From the assumption $\mathit{wf}(A|X)$ we have $\mathit{resp}(A|X)$, and thus that there must be some derivation of $A'|X \vdash Q \Rightarrow \pi$, where $A' \subset A$, $Q \Rightarrow \pi' \notin \lfloor A' \rfloor$. Is it trivial to see that $\mathit{wf}(A|X) \implies \mathit{wf}(A'|X)$, and so by induction there is a derivation of $A'|\emptyset \vdash Q \Rightarrow \pi$, and thus a derivation of $A|\emptyset \vdash Q \Rightarrow \pi$ and finally, since (again by induction) $A|\emptyset \vdash P \Rightarrow Q$, a derivation of $A|\emptyset \vdash P \Rightarrow \pi$. ☐

Lemma 17. *If $\mathit{wf}(A|X)$, then $\mathcal{I}_{A|X}$ is defined and $\mathcal{I}_{A|X} \models A|X$.*

Proof. Let $\mathcal{I}_{A|X} = \langle G, H, \preceq \rangle$. From $\mathit{wf}(A|X)$ we have that $H = I^\omega(\emptyset)$ is defined.

We show that, for all $K \in G$ and $\alpha \in A$, $K \models \alpha$. Let $\alpha = d : \forall \vec{t}.\, P \Rightarrow \pi$ and $\gamma \in \lfloor \alpha \rfloor$. Then we have that $\gamma = d : P' \Rightarrow \pi'$, where $\pi' = [\vec{\tau}/\vec{t}]\pi$, $P' = [\vec{\tau}/\vec{t}]P$ for some ground types $\vec{\tau}$. Suppose that, for each k there is some e_k such that $K \models e_k : P'_k$. (If not, that $K \models \gamma$ is trivial.) By the definition of $\mathcal{I}_{A|X}$, we know that $K = [\![I^\omega(M_0)]\!]$ for some initial evidence assignment M_0. By the definition of $[\![-]\!]$ and I^ω, we know that there is some n such that $\{\langle P'_k, e_k\rangle\} \subseteq I^n(M_0)$. Finally, from the definition of $I(M)$, we have that $\langle \pi, d(e_k)\rangle \in I^{n+1}(M_0)$ and so $K \models d(e_k) : \pi'$.

Finally, we show that, for all $K \in G$ and $\chi \in X$, $K \models \chi$. Let $\chi = d : \mathit{Super}(\forall \vec{t}.\psi \Leftarrow \pi)$, and $\xi \in \lfloor \chi \rfloor$; then we know that $\xi = d : \mathit{Super}(\psi' \Leftarrow \pi')$, where $\pi' = [\vec{\tau}/\vec{t}]\pi$ and $\psi' = [\vec{\tau}/\vec{t}]\psi$ for some types $\vec{\tau}$. Suppose that $K \models e : \pi'$. (Otherwise $K \models \xi$ trivially.) As before, we know that $K = [\![I^\omega(K_0)]\!]$. We have two cases.

- If $\langle \pi', e\rangle \in M_0$ then, by the assumption $\mathit{resp}(M_0, X)$, we must have $\langle \psi', e'\rangle \in M_1$ and therefore $K \models e' : \psi'$.
- If $\langle \pi', e\rangle \notin M_0$, then there must be some $(Q \Rightarrow \pi) \in \lfloor A \rfloor$ where, for each i, there is some e_i such that $K \models e_i : Q_i$. From the assumption of $\mathit{wf}(A|X)$ and Lemma 16, we must have a derivation of $A|\emptyset \vdash Q \Rightarrow \psi'$. Finally, as $K \models A$ (by the argument above), $K \models A|\emptyset$ and by Theorem 2, there is some e' such that $K \models e' : \psi'$. ☐

Finally, Theorem 4 follows immediately from Lemma 17.

A.3 Haskell '98 Completeness

Theorem 5. *If $\mathit{wf}(A|X)$, P and Q are ground contexts, and, for all $G \models A|X$, if there is some $\vec{e}$ such that $G \models \vec{e} : P$ then there is some $\vec{e}'$ such that $G \models \vec{e}' : Q$, then there is a derivation of $A|X \vdash P \Rightarrow Q$.*

We begin by introducing a notion of restriction for concrete model structures.

Definition 18. Let K be a point in some concrete model structure G, and P be any set of predicates. We define the restriction $K|_P$ by

$$K|_P \triangleq \{\langle C, K_C|_P\rangle \mid \langle C, K_C\rangle \in K\}$$
$$K_C|_P \triangleq \{\langle \vec{\tau}, e\rangle \mid \langle \vec{\tau}, e\rangle \in K_c, C\,\vec{\tau} \notin P\}$$

We will write $K \nvDash \pi$ to denote that there is no e such that $K \vDash e : \pi$.

Proof of Theorem 5. Without loss of generality, assume that Q is a singleton set. We show the contrapositive: that if there is no derivation of $A|X \vdash P \Rightarrow Q$, then we can construct a model of $A|X$ and P that is not a model of Q. We begin by defining a set E as the least set such the that following conditions holds:

- $Q \subseteq E$;
- If $\pi \in E$ and $(P \Rightarrow \pi) \in \lfloor A \rfloor$, then $P \subseteq E$; and,
- If $\pi \in E$ and $Super(\pi \Leftarrow \pi') \in \lfloor X \rfloor$, then $\pi' \in E$.

Let $\mathcal{I}_{A|X} = \langle G_0, H_0, \preceq\rangle$ be the induced model structure of $A|X$. We define a new model structure $\langle G, H, \preceq\rangle$ where $G = \{K|_P \mid K \in G_0\}$ and H is the least point in G such that $H \vDash P$. We must show that H exists and that $G \vDash A|X$; that $G \nvDash Q$ is immediate from its construction.

We begin by showing that H exists. Intuitively, this follows from Lemma 1 and the assumption that there is no derivation $A|X \vdash P \Rightarrow Q$. Pick some $\pi \in P$. If $\pi \in E$, then there is a derivation of Q from P. If not, but $A|X \vdash \pi$, then by construction $G_0 \vDash \pi$. Finally, if $A|X \nvdash \pi$, then for any point K such that for all e, $K \nvDash e : \pi$, we can trivially extend K to a point K' such that, for arbitrary e, $K \vDash e : \pi$. Observe that by construction G_0, and thus G, includes such extensions. Thus, we have some suitable H in G.

Finally, we show that $G \vDash A|X$. Suppose that it did not; then there must be some point $K \succeq H$ such that $K \nvDash A|X$. We know that $K = K_0|_P$ for some $K_0 \in G_0$, and that $K_0 \vDash A|X$. From the definition of the modeling relation, and De Morgan's laws, we have two possibilities. Either there is some $P \Rightarrow \pi \in \lfloor A \rfloor$ such that $K \vDash P$ but $K \nvDash \pi$, or there is some $Super(\pi \Leftarrow \pi') \in \lfloor X \rfloor$ such that $K \vDash \pi'$ but $K \nvDash \pi$. In either case, since $K_0 \vDash A|X$, we know that there is some e such that $K_0 \vDash e : \pi$ and, as K and K_0 differ at π, that $\pi \in E$. But then both cases contradict the construction of $K_0|_E$. So we know that there is no such K, and thus $G \vDash A|X$. □

A.4 Haskell '98 Monotonicity

Theorem 6. *Given well-formed $A|X, A'|X'$, if $A|X \sqsubseteq A'|X'$ and $\mathcal{G} \vDash A'|X'$, then $\mathcal{G} \vDash A|X$.*

Proof. We have trivially that $\lfloor A \rfloor \subseteq \lfloor A' \rfloor$ and $\lfloor X \rfloor \subseteq \lfloor X' \rfloor$. From $\mathcal{G} \vDash A'|X'$ we know that $\forall \alpha \in \lfloor A' \rfloor, \mathcal{G} \vDash \alpha$, thus $\forall \alpha \in \lfloor A \rfloor, \mathcal{G} \vDash \alpha$ and finally $\mathcal{G} \vDash A$. An identical argument holds for X, thus $\mathcal{G} \vDash A|X$. □

A.5 Soundness with Negative Predicates

Lemma 19. *If $wf(A|X)$, then $\mathcal{I}_{A|X} \vDash A|X$.*

Proof. Let $K \in \mathcal{I}_{A|X}$, $\alpha \in A$; we show that $K \vDash \alpha$. We will assume that the conclusion of α is negative; if it is positive, the argument is identical to that in Lemma 17. Let $\alpha = d : \forall \vec{t}.\, P \Rightarrow C\,\vec{\tau}\,\texttt{fails}$ and $\gamma \in \lfloor \alpha \rfloor$; we know that $\gamma = d : P' \Rightarrow C\,\vec{\tau}'\,\texttt{fails}$ where $P' = [\vec{v}/\vec{t}]P$ and $\vec{\tau}' = [\vec{v}/\vec{t}]\vec{\tau}$. Suppose that for each P'_k there is an e_k such that $K \vDash e_k : P'_k$ (if not, $K \vDash \gamma$ trivially). By definition of $\mathcal{I}_{A|X}$, we know that $K = [\![I^\omega(M_0)]\!]$ for some evidence assignment M_0; as $K \vDash e_k : P'_k$, we have that there is some n such that $\{\langle P'_k, e_k\rangle\} \subseteq I^n(M_0)$. We consider two possibilities:

- Suppose that there is some e such that $\langle C\,\vec{\tau}', e\rangle \in I^n(M_0)$. By definition of $I(M)$, we have that $I^{n+1}(M) = I^n(M_0) \uplus \{\langle C\,\vec{\tau}'\,\texttt{fails}, d(e_k)\rangle\}$ for some evidence assignment M'. But, as $\langle C\,\vec{\tau}', e\rangle \in M'$, $I^{n+1}(M)$ is undefined, a contradiction.
- Suppose that there is some $m > n+1$ and some e such that $I^m(M_0) = M' \uplus \{\langle C\,\vec{\tau}', e\rangle\}$. But then we know that $\langle C\,\vec{\tau}'\,\texttt{fails}, d(e_k)\rangle \in M'$, and so $I^m(M)$ is undefined, a contradiction.

We conclude that there is no $K \in G$ and e such that $K \vDash e : C\,\vec{\tau}'$, and thus that $K \vDash d(\vec{e}_k) : C\,\vec{\tau}'\,\texttt{fails}$.

The argument that $G \vDash X$ is identical to that in Lemma 17. □

A.6 Model Existence with Negative Predicates

Theorem 7. *If $wf(A|X)$, then there is some model $\mathcal{G}$ such that $\mathcal{G} \vDash A|X$.*

Showing model existence for the system with negative predicates is a straightforward extension of the argument for Haskell '98. The key difference is extending the definition of $\uplus$ to exclude both ambiguous and contradictory evidence assignments.

Definition 20. Given a set of axioms A, we define a function $I(\cdot)$ as follows. Enumerate the possible predicates $\pi_1, \ldots, \pi_j, \ldots$. We define $I(\cdot)$ following the previous definition:

$$I_j(M) = \begin{cases} M \uplus \{\langle \pi_j, d\,\vec{e}\rangle\} \\ \quad \text{if } (d : P \Rightarrow \pi_j) \in \lfloor A \rfloor, \langle P_i, e_i\rangle \in M \text{ for each } i \\ M \quad \text{otherwise.} \end{cases}$$

where $M \uplus N$ is the union of M and N, so long as the union is consistent and unambiguous:

$$M \uplus N = \begin{cases} M \cup N, \\ \quad \text{if } \forall \pi \in \mathsf{dom}(M) \cap \mathsf{dom}(N), M(\pi) = N(\pi) \\ \quad \text{and } \forall \pi \in \mathsf{dom}(M), \pi' \in \mathsf{dom}(N), \neg(\pi \between \pi') \\ \text{undefined, otherwise.} \end{cases}$$

We define $I(\cdot)$ and $I^\omega(\cdot)$ identically to Definition 15. The interpretation function for evidence assignments is given by

$$[\![M]\!] = \{\langle\langle C, \vec{\tau}\rangle, \eta(e)\rangle \mid \langle C\,\vec{\tau}, e\rangle \in M\}.$$

Note that the interpretation $[\![M]\!]$ includes only evidence for the positive predicates. Finally, the induced model $\mathcal{I}_{A|X}$ is defined exactly as before:

$$G_{A|X} = \{[\![I^\omega(M)]\!] \mid resp(M, X) \text{ and } I^\omega(M) \text{ is defined}\}$$
$$\mathcal{I}_{A|X} = \langle G_{A|X}, [\![I^\omega(\emptyset)]\!], \preceq\rangle.$$

Lemma 21. *If $wf(A|X)$, then $\mathcal{I}_{A|X} \vDash A|X$.*

Proof. The only new case is for axioms asserting negative predicates; the argument in this case follows from the consistency requirement of $\uplus$. □

Lemma 22. *Given $A|F$ such that that $wf(A|X)$, and $\alpha, \alpha' \in A$, $\gamma \in \lfloor \alpha \rfloor$, $\gamma' \in \lfloor \alpha' \rfloor$, where $\gamma = P \Rightarrow \pi$ and $\gamma' = P' \Rightarrow \pi'$. If $A|X \vdash P \between P'$, then, because of the assumption $decr(A)$, we also have that $(A \setminus \{\alpha, \alpha'\})|X \vdash P \between P'$.*

The proof is direct from the definition of $decr(A)$: γ and γ cannot contribute to the derivation of $P \between P'$ as we must have $\|\pi\| > \|\psi\|$ for any $\psi \in P \cup P'$, and similarly for π'.

Proof of Theorem 7. As before, we show that if $wf(A|X)$ then $\mathcal{I}_{A|X}$ is defined and models $A|X$. From the assumption of $wf(A|X)$ and by induction on the size of A, we can conclude that the axioms of A are non-overlapping, non-contradictory, and respect the restrictions X. Thus $I^\omega(\emptyset)$ is defined. The only new case in showing that, for

$K \in G_{A|X}$, $K \vDash A|X$ is for axioms asserting negative predicates. The argument in this case follows from the consistency requirement of $\uplus$; see Lemma 21 in the appendix for details. $\square$

A.7 Model Existence with Functional Dependencies

Theorem 9. *If $wf(A|X)$ then there is some model $\mathcal{G}$ such that $\mathcal{G} \vDash A|X$.*

In showing model existence for the system with negative predicates (§A.6), we introduced a consistent union operator $\uplus$ based on the definition of $\lightning$. Our definition of model construction $(I(\cdot), I^{\omega}(\cdot), \mathcal{I}_{A|X})$ with functional dependencies then follows Definition 20 exactly, but replacing $\lightning$ with $\lightning_X$ in the definition of $\uplus$.

Proof of Theorem 9. We show that $\mathcal{I}_{A|X}$ is defined and models $A|X$.

From the assumption of $wf(A|X)$, we have that the axioms of A are non-overlapping and consistent with respect to the restrictions in X; thus, $I^{\omega}(\emptyset)$ and $\mathcal{I}_{A|X}$ are defined.

The only new case in showing $\mathcal{I}_{A|X} \vDash A|X$ is for functional dependency restrictions. We must show that if $C : Y \rightsquigarrow Z \in X$, then $\mathcal{I}_{A|X} \vDash C : Y \rightsquigarrow Z$. Suppose it does not; then there must be some $K \in \mathcal{I}_{A|X}$ such that $\vec{\tau}, \vec{\upsilon} \in K(C)$, $\vec{\tau} =_Y \vec{\upsilon}$, $\vec{\tau} \neq_Z \vec{\upsilon}$. By definition of $\mathcal{I}_{A|X}$, we must have some assignment M and index n such that $\langle C\,\vec{\tau}, e\rangle, \langle C\,\vec{\upsilon}, e'\rangle \in I^n(M)$; but $C\,\vec{\tau} \lightning_X C\,\vec{\upsilon}$, contradicting the definition of $\uplus$. $\square$

A.8 Completeness with Functional Dependencies

Theorem 10. *Let $A|X$ be well-formed, and P and Q be sets of ground predicates. If there exists some $\mathcal{G}_0$ such that $\mathcal{G}_0 \vDash A|X \wedge \mathcal{G}_0 \vDash P$, and for all $\mathcal{G} \vDash A|X$, $\mathcal{G} \vDash P \implies \mathcal{G} \vDash Q$, then $A|X \vdash P \Rightarrow Q$.*

Proof. We follow the same approach we used to show the completeness of the Haskell '98 class system (Theorem 5). In particular, as we assume that P and Q are ground predicates, we do not need to account for (IMPR) in derivations.

We construct a model structure $\langle G, H, \preceq\rangle$ that includes predicates P and does not include predicates Q, identically to the construction for Theorem 5. The only new case is to show that G models the functional dependencies in X. Suppose it does not. Since removing predicates cannot introduce functional dependency violations, P must be either inconsistent itself, or inconsistent with the axioms A. But in either case, we would have no model structure $\mathcal{G}_0$ that models $A|X$ and P, a contradiction. Thus, we conclude that $G \vDash A|X$, $G \vDash P$ but $G \nvDash Q$. $\square$

A.9 Model Existence with Alternative Instances

Definition 23. As before, let M range over predicate assignments $Pred \rightharpoonup Evid$, and enumerate the possible predicates $\pi_0, \ldots, \pi_i, \ldots$. We define

$$I_j(M) = \begin{cases} M \uplus \{\langle \pi_j, d\,\vec{e}\rangle\} & \\ \quad \text{if } (d : P \Rightarrow \pi_j \;;\; \alpha) \in \lfloor A \rfloor, & \\ \quad \langle P_k, e_k\rangle \in M \text{ for each } k; & \\ M \quad \text{otherwise.} & \end{cases}$$

Again, we are concerned only with the first clause in any chain that matches a given predicate. Finally, we can define $I(\cdot), I^{\omega}(\cdot), \mathcal{I}_{A|X}$ as in Definition 15, but using our updated definition of $I_j(\cdot)$.

Theorem 12. *If $wf(A|X)$ then there is some model $\mathcal{G}$ such that $\mathcal{G} \vDash A|X$.*

Proof. From $wf(A|X)$, we know that the axioms are non-overlapping, and so $I^{\omega}(\emptyset)$ and $\mathcal{I}_{A|X}$ are defined. In showing $\mathcal{I}_{A|X} \vDash A|X$, the only case that differs from previous systems is that for alternative instances; the argument for that case follows directly from the definitions of the modeling relation and $I(\cdot)$. $\square$