A simple semantics for Haskell overloading

A Simple Semantics for Haskell Overloading

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

Abstract

As originally proposed, type classes provide overloading and ad-hoc definition, but can still be understood (and implemented) in terms of strictly parametric calculi. This is not true of subsequent extensions of type classes. Functional dependencies and equality constraints allow the satisfiability of predicates to refine typing; this means that the interpretations of equivalent qualified types may not be interconvertible. Overlapping instances and instance chains allow predicates to be satisfied without determining the implementations of their associated class methods, introducing truly non-parametric behavior. We propose a new approach to the semantics of type classes, interpreting polymorphic expressions by the behavior of each of their ground instances, but without requiring that those behaviors be parametrically determined. We argue that this approach both matches the intuitive meanings of qualified types and accurately models the behavior of programs.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Denotational semantics

Keywords overloading; type classes; semantics

1. Introduction

Implicit polymorphism (as provided by the Hindley-Milner type systems in ML and Haskell) provides a balance between the safety guarantees provided by strong typing, and the convenience of generic programming. The Hindley-Milner 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 excessive type annotation. Type classes [16] 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. Since their introduction, type classes have seen numerous extensions, such as multi-parameter type classes, functional dependencies [5], and overlapping instances [13]; a variety of practical uses, from simple overloading to capturing complex invariants and type-directed behavior; and, the adoption of similar approaches in other strongly-typed programming languages, including Isabelle and Coq.

Haskell '14, September 6, 2014, Gothenburg, Sweden.

ACM 978-1-4503-3041-1/14/09…$15.00.
http://dx.doi.org/10.1145/2633357.2633364

1.1 Dictionary-Passing and its Disadvantages

The semantics of type classes has primarily been given by translations from instance declarations and (implicit) overloading to dictionaries and (explicit) dictionary arguments. This parallels the treatment of implicit polymorphism by translation to systems with explicit polymorphism (such as System F), and shares similar challenges. For a simple example, in Haskell, the `map` function has the polymorphic type scheme $(t \to u) \to [t] \to [u]$. In translating to System F, this could be interpreted as either

$$\forall t.\forall u.(t \to u) \to [t] \to [u] \quad \text{or} \quad \forall u.\forall t.(t \to u) \to [t] \to [u].$$

But these types are not equivalent: they express different orders of passing type arguments. There are various ways of addressing this discrepancy: for example, Mitchell [8] shows that, for any two translations of an implicitly typed scheme, there is a term (which he calls a retyping function) which transforms terms of one translation to terms of the other, while only manipulating type abstractions and applications. Similar issues arise in the semantics of type classes. For example, a function to compare pairs (t, u) for equality could be given either the type scheme

$$(\texttt{Eq}\ t, \texttt{Eq}\ u) \Rightarrow (t, u) \to (t, u) \to \texttt{Bool}$$

or the type scheme

$$(\texttt{Eq}\ u, \texttt{Eq}\ t) \Rightarrow (t, u) \to (t, u) \to \texttt{Bool}.$$

In a dictionary-passing translation, type classes are interpreted by tuples, called dictionaries, containing the type-specific implementations of each of the class methods. Class instances correspond to dictionary definitions, while predicates in types correspond to dictionary arguments. In the case of the `Eq` class, which has equality and inequality methods, we could define `Eq` dictionaries by

$$\texttt{EqDict}\ t = (t \to t \to \texttt{Bool}, t \to t \to \texttt{Bool}).$$

Even though the two types for pair equality above are equivalent in the implicitly overloaded setting, their dictionary-passing translations are not: the first corresponds to a function of type

$$\texttt{EqDict}\ t \to \texttt{EqDict}\ u \to (t, u) \to (t, u) \to \texttt{Bool},$$

while the second corresponds to

$$\texttt{EqDict}\ u \to \texttt{EqDict}\ t \to (t, u) \to (t, u) \to \texttt{Bool},$$

Again, approaches exist to address this discrepancy: for example, Jones shows [3] that there are conversion functions, similar to Mitchell's retyping functions, to convert between different translations of the same overloaded term.

Our own work began by exploring instance chains [10], a proposed extension to Haskell-like type class systems. In the course of this exploration, we discovered several difficulties with existing approaches to the semantics of overloading.

Mismatch in expressivity. System F typing is significantly more expressive than the Hindley-Milner type systems it is used to

model. In particular, even within the translation of an ML or Haskell type scheme, there are arbitrarily many expressions that do not correspond to any expressions of the source language. The problem is compounded when considering dictionary-passing translations of type classes. For example, there is no notion in Haskell of class instances depending on terms; on the other hand, there is no difficulty in defining a term of type $\texttt{Int} \to \texttt{EqDict Int}$. Uses of such a term cannot be equivalent to any use of the methods of `Eq`. As a consequence, there are properties of source programs (for example, that any two instances of $==$ at the same type are equal) that may not be provable of their dictionary-passing translation without reference to the specific mechanisms of translation.

Predicates refine typing. Second, the notions of equivalence of System F and Haskell types diverge once the satisfiability of predicates can refine typing. For example, functional dependencies allow programmers to declare that some parameters of a class depend upon others; in the declaration

```
class Elems c e | c → e where
  empty :: c
  insert :: e → c → c
```

the dependency $\texttt{c} \to \texttt{e}$ captures the intuition that the type of a container's elements are determined by the type of the container. Concretely, given two predicates $\texttt{Elems}\ \tau\ \upsilon$ and $\texttt{Elems}\ \tau'\ \upsilon'$, if we know that $\tau = \tau'$, then we can conclude $\upsilon = \upsilon'$. This property is lost in the dictionary-passing translation. Dictionaries for `Elems` contain just their methods:

$$\texttt{ElemsDict}\ c\ e = (c, e \to c \to c)$$

As a consequence, there are types that are equivalent in Haskell, but are not interconvertible in the dictionary-passing interpretation. For example, the type $(\texttt{Elems}\ c\ e, \texttt{Elems}\ c\ e') \Rightarrow e \to e' \to c$ is equivalent to the (simpler) type $(\texttt{Elems}\ c\ e) \Rightarrow e \to e \to c$ as we must have that $e = e'$ for the qualifiers in the first type to be satisfiable. However, there is no corresponding bijection between terms of type $\texttt{ElemsDict}\ c\ e \to \texttt{ElemsDict}\ c\ e' \to e \to e' \to c$ and terms of type $\texttt{ElemsDict}\ c\ e \to e \to e \to c$. While we can construct a term of the second type given a term of the first, there is no parametric construction of a term of the first type from a term of the second.

Non-parametric behavior. Finally, other extensions to class systems make it possible to define terms which have no translation to parametric calculi. For example, we could define a function `invBool` that negated booleans and was the identity on all other types. We begin by introducing a suitable class:

```
class Univ t where
  invBool :: t → t
```

There are several approaches to populating the class, using different extensions of the Haskell class system. Using overlapping instances [13], we could simply provide the two desired instances of the class, relying on the type checker to disambiguate them based on their specificity:

```
instance Univ Bool where
  invBool = not
instance Univ t where
  invBool = id
```

Using instance chains, we would specify the ordering directly:

```
instance Univ Bool where
  invBool = not
else Univ t where
  invBool = id
```

With either of these approaches, we might expect that the type of the class method `invBool` is $(\texttt{Univ}\ t) \Rightarrow t \to t$. However, the predicate $\texttt{Univ}\ \tau$ is provable for arbitrary types τ. Thus, the above type is intuitively equivalent to the unqualified type $t \to t$; however, there is no term of that type in a purely parametric calculus that has the behavior of method `invBool`. (In practice, this is avoided by requiring that `invBool`'s type still include the `Univ` predicate, even though it is satisfied in all possible instantiations; while this avoids the difficulties in representing `invBool` in a parametric calculus, it disconnects the meaning of qualified types from the satisfiability of their predicates.)

1.2 Specialization-based Semantics

We propose an alternative approach to the semantics of type-class based implicit overloading. Rather than interpret polymorphic expressions by terms in a calculus with higher-order polymorphism, we will interpret them as type-indexed collections of (the interpretations of) monomorphic terms, one for each possible ground instantiation of their type. We call this a specialization-based approach, as it relates polymorphic terms to each of their (ground-typed) specializations. We believe this approach has a number of advantages.

- First, our approach interprets predicates directly as restrictions of the instantiation of type variables, rather than through an intermediate translation. Consequently, properties of the source language type system—such as the type refinement induced by the `Elems` predicates—are immediately reflected in the semantics, without requiring the introduction of coercions.
- Second, our approach naturally supports non-parametric examples, such as class `Univ`, and avoids introducing artificial distinction between the semantics of expressions using parametric and ad-hoc polymorphism.
- Third, because our approach does not need to encode overloading via dictionaries, it becomes possible to reason about class methods directly, rather than through reasoning about the collection of dictionaries defined in a program.

Our approach builds on Ohori's simple semantics for ML polymorphism [12], extended by Harrison to support polymorphic recursion [1].

In this paper, we introduce a simple overloaded language called H^- (§2), and give typing and equality judgments in the presence of classes and class methods. We apply our specialization-based approach to give a denotational semantics of H^- (§3), and show the soundness of typing and equality with respect to the denotational semantics (§4). We also develop two examples, to demonstrate the advantages of our approach. First, we consider a pair of definitions, one parametric and the other ad-hoc, defining operational equivalent terms. We show that the defined terms are related by our equality judgment (§2.3) and have the same denotations (§3.5). This demonstrates the flexibility of our approach, and the ability to reason about class methods directly (the second and third advantages listed above). Second, we extend H^- with functional dependencies (§5), and establish the soundness of the (extended) typing and equality judgments, all without having to augment the models of terms. This demonstrates the extensibility of our approach, and the close connection between properties of source terms and properties of their denotations (the first advantage listed above).

2. The H^- Language

Figure 1 gives the types and terms of H^-; we write $\overline{x}$ to denote a (possibly empty) sequence of x's, and if π is a predicate $C\ \overline{\tau}$, we will sometimes write $\mathit{class}(\pi)$ for C. As in Jones's theory of qualified types [2], the typical Hindley-Milner types are extended with

Term variable	$x \in \mathit{Var}$	Term constants	k
Type variables	$t \in \mathit{TVar}$	Type constants	K
Class names	C	Instance names	$d \in \mathit{InstName}$

Types		τ, υ	$::= t \mid K \mid \tau \to \tau$
Predicates	$\mathit{Pred} \ni \pi$		$::= C\ \overline{\tau}$
Contexts		P, Q	$::= \overline{\pi}$
Qualified types		ρ	$::= \tau \mid \pi \Rightarrow \rho$
Type schemes	$\mathit{Scheme} \ni \sigma$		$::= \rho \mid \forall t.\sigma$
Expressions	$\mathit{Expr} \ni M, N$		$::= x \mid k \mid \lambda x.M \mid M\,N$
			$\mid \mu x.M \mid \underline{\texttt{let}}\ x = M\ \underline{\texttt{in}}\ N$
Class axioms	$\mathit{Axiom} \ni \alpha$		$::= d : \forall \bar{t}.\, P \Rightarrow \pi$
Axiom sets		A	$\subset \mathit{Axiom}$
Methods:			
Signatures		Si	$\in \mathit{Var} \rightharpoonup \mathit{Pred} \times \mathit{Scheme}$
Implementations		Im	$\in \mathit{InstName} \times \mathit{Var} \rightharpoonup \mathit{Expr}$
Class contexts		Ψ	$::= \langle A, \mathit{Si}, \mathit{Im} \rangle$

Figure 1: Types and terms of H^-.

qualified types ρ, capturing the use of predicates. We must also account for the definition of classes and their methods. One approach would be to expand the grammar of expressions to include class and instance declarations; such an approach is taken in Wadler and Blott's original presentation [16]. However, this approach makes such definitions local, in contrast to the global nature of subsequent type class systems (such as that of Haskell), and introduces problems with principal typing (as Wadler and Blott indicate in their discussion). We take an alternative approach, introducing new top level constructs (axioms A, method signatures Si, and method implementations Im) to model class and instance declarations. We refer to tuples of top level information as class contexts Ψ, and will give versions of both our typing and semantic judgments parameterized by such class contexts. Note that this leaves implicit many syntactic restrictions that would be present in a full language, such as the requirement that each instance declaration provide a complete set of method implementations.

2.1 H^- Typing

We begin with the typing of H^- expressions; our expression language differs from Jones's only in the introduction of μ (providing recursion) . Typing judgments take the form

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

where P is a set of predicates restricting the type variables in Γ and σ, and A is the set of class axioms (the latter is the only significant difference between our type system and Jones's). The typing rules for H^- expressions are given in Figure 2. We write $\mathit{ftv}(\tau)$ for the free type variables in τ, and extend ftv to predicates π, contexts P, and environments Γ in the expected fashion. Rules ($\Rightarrow$ I) and ($\Rightarrow$ E) describe the interaction between the predicate context P and qualified types ρ. Otherwise, the rules are minimally changed from the corresponding typing rules of most Hindley-Milner systems.

We continue with the rules for predicate entailment in H^-, given in Figure 3. The judgment $d : P \Vdash_A \pi$ denotes that the axiom named d proves predicate π, given assumptions P and class axioms A. We use a dummy instance name, written -, in the case that the goal is one of the assumptions. We will omit the instance name if (as in the typing rules) the particular instance used is irrelevant. We write $P \Vdash_A Q$ if there are $d_1 \ldots d_n$ such that $d_i : P \Vdash Q_i$, and $\Vdash_A P$ to abbreviate $\emptyset \Vdash_A P$. Our entailment relation differs from Jones's entailment relation for type classes and from our prior systems [10] in two respects. First, our system is intentionally simplified (for example, we omit superclasses and instance chains).

$$(\textsc{Var})\ \frac{(x:\sigma) \in \Gamma}{P \mid \Gamma \vdash_A x : \sigma} \qquad (\to I)\ \frac{P \mid \Gamma, x:\tau \vdash_A M : \tau'}{P \mid \Gamma \vdash_A (\lambda x.M) : \tau \to \tau'}$$

$$(\to E)\ \frac{P \mid \Gamma \vdash_A M : \tau \to \tau' \quad P \mid \Gamma \vdash_A N : \tau}{P \mid \Gamma \vdash_A (M\,N) : \tau'}$$

$$(\mu)\ \frac{P \mid \Gamma, x:\tau \vdash_A M : \tau}{P \mid \Gamma \vdash_A \mu x.M : \tau} \qquad (\Rightarrow I)\ \frac{P, \pi \mid \Gamma \vdash_A M : \rho}{P \mid \Gamma \vdash_A M : \pi \Rightarrow \rho}$$

$$(\Rightarrow E)\ \frac{P \mid \Gamma \vdash_A M : \pi \Rightarrow \rho \quad P \Vdash_A \pi}{P \mid \Gamma \vdash_A M : \rho}$$

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

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

Figure 2: Expression typing rules of H^-.

$$(\textsc{Assume})\ \frac{\pi \in P}{\text{-} : P \Vdash_A \pi}$$

$$(\textsc{Axiom})\ \frac{(d : \forall \bar{t}.Q' \Rightarrow \pi') \in A \quad S\,\pi' = \pi \quad P \Vdash_A S\,Q'}{d : P \Vdash_A \pi}$$

Figure 3: Predicate entailment rules of H^-.

Second, we do not attempt to capture all the information that would be necessary for an dictionary-passing translation; we will show that having just the first instance name is sufficient to determine the meanings of overloaded expressions.

In the source code of a Haskell program, type class methods are specified in class and instance declarations, such as the following:

```
class Eq t where (==) :: t → t → Bool
instance Eq t ⇒ Eq [t] where xs == ys = ...
```

We partition the information in the class and instance declarations into class context tuples $\langle A, \mathit{Si}, \mathit{Im} \rangle$. The logical content is captured by the axioms A; in this example, we would expect that there would be some instance name d such that

$$(d : \forall t.\, \texttt{Eq}\ t \Rightarrow \texttt{Eq}\ [t]) \in A.$$

Haskell's concrete syntax does not name instances; for our purposes, we assume that suitable identifiers are generated automatically. The method signatures are captured in the mapping Si; we distinguish the class in which the method is defined (along with the corresponding type variables) from the remainder of the method's type scheme. For this example, we would have

$$\mathit{Si}(==) = \langle \texttt{Eq}\ t, t \to t \to \texttt{Bool} \rangle.$$

Note that we have not quantified over the variables appearing in the class predicate, nor included the class predicate in the type scheme $t \to t \to \texttt{Bool}$. Each predicate in the range of Si will be of the form $C\ \bar{t}$ for some class C and type variables $\bar{t}$, as they arise from class definitions. The type scheme of a class member may quantify over variables or include predicates beyond those used in the class itself. For example, the `Monad` class has the following definition:

$$\text{(CTXT)}\ \frac{\begin{array}{c}\{\pi \nsim \pi' \mid (d : P \Rightarrow \pi), (d' : P' \Rightarrow \pi') \in A\} \\ \{(P \mid \Gamma, \overline{x_i : \sigma_{x_i}} \vdash_A Im(y,d) : \sigma_{y,d}) \mid \langle y, d\rangle \in \mathsf{dom}(Im)\} \\ P \mid \Gamma, \overline{x_i : \sigma_{x_i}} \vdash_A M : \sigma\end{array}}{P \mid \Gamma \vdash_{\langle A, Si, Im\rangle} M : \sigma}$$

Figure 4: H^- typing with class contexts.

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

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

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

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

To describe the typing of methods and method implementations, we begin by describing the type of each method implementation. This is a combination of the defining instance, including its context, and the definition of the method itself. For example, in the instance above, the body of the $==$ method should compare lists of arbitrary type `t` for equality (this arises from the instance predicate `Eq [t]` and the signature of $==$), given the assumption `Eq t` (arising from the defining instance). That is, we would expect it to have the type

$$\forall t.\mathsf{Eq}\ t \Rightarrow [t] \to [t] \to \mathtt{Bool}.$$

We introduce abbreviations for the type scheme of each method, in general and at each instance, assuming some class context $\langle A, Si, Im\rangle$. For each method name x such that $Si(x) = \langle \pi, \forall \overline{u}.\rho\rangle$, we define the type scheme for x by:

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

or, equivalently, writing ρ as $Q \Rightarrow \tau$:

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

where, in each case, $\overline{t} = ftv(\pi)$. Similarly, for each method x as above, and each instance d such that

- $\langle x, d\rangle \in \mathsf{dom}(Im)$;
- $(d : \forall \overline{t}.\ P \Rightarrow \pi') \in A$; and,
- there is some substitution S such that $S\,\pi = \pi'$

we define the type scheme for x in d by:

$$\sigma_{x,d} = \forall \overline{t}, \overline{u}.\ (P, S\,Q) \Rightarrow S\,\tau.$$

Finally, we give a typing rule parameterized by class contexts in Figure 4; in $\overline{x_i : \sigma_{x_i}}$, the x_i range over all methods defined in the program (i.e., over the domain of *Si*). Intuitively, an expression M has type τ under $\langle A, Si, Im\rangle$ if:

- None of the class instances overlap. More expressive class systems will require more elaborate restrictions; we give an example when extending H^- to support functional dependencies (§5).
- Each method implementation $Im(x,d)$ has the type $\sigma_{x,d}$ (methods are allowed to be mutually recursive).
- The main expression has the declared type σ, given that each class method x_i has type σ_{x_i}.

$$\{\beta\}\ \frac{P \mid \Gamma, x : \tau \vdash_\Psi M : \tau' \quad P \mid \Gamma \vdash_\Psi N : \tau}{P \mid \Gamma \vdash_\Psi (\lambda x.M)N \equiv [N/x]M : \tau'}$$

$$\{\eta\}\ \frac{P \mid \Gamma \vdash_\Psi M : \tau \to \tau' \quad x \notin fv(M)}{P \mid \Gamma \vdash_\Psi \lambda x.(Mx) \equiv M : \tau \to \tau'}$$

$$\{\mu\}\ \frac{P \mid \Gamma, x : \tau \vdash_\Psi M : \tau}{P \mid \Gamma \vdash_\Psi \mu x.M \equiv [\mu x.M/x]M : \tau}$$

$$\{\text{LET}\}\ \frac{P \mid \Gamma \vdash_\Psi M : \sigma \quad P \mid \Gamma, x : \sigma \vdash_\Psi N : \tau}{P \mid \Gamma \vdash_\Psi (\underline{\texttt{let}}\ x = M\ \underline{\texttt{in}}\ N) \equiv [M/x]N : \tau}$$

$$\{\text{METHOD}\}\ \frac{Si(x) = \langle \pi, \sigma\rangle \quad d : P \Vdash S\,\pi}{P \mid \Gamma \vdash_{\langle A, Si, Im\rangle} x \equiv Im(x,d) : S\,\sigma}$$

$$\{\forall \text{I}\}\ \frac{t \notin ftv(P, \Gamma) \quad \{(P \mid \Gamma \vdash_\Psi M \equiv N : [\tau/t]\sigma) \mid \tau \in GType\}}{P \mid \Gamma \vdash_\Psi M \equiv N : \forall t.\sigma}$$

$$\{\forall \text{E}\}\ \frac{P \mid \Gamma \vdash_\Psi M \equiv N : \forall t.\sigma}{P \mid \Gamma \vdash_\Psi M \equiv N : [\tau/t]\sigma}$$

$$\{\Rightarrow I\}\ \frac{P, \pi \mid \Gamma \vdash_\Psi M \equiv N : \rho}{P \mid \Gamma \vdash_\Psi M \equiv N : \pi \Rightarrow \rho}$$

$$\{\Rightarrow E\}\ \frac{P \mid \Gamma \vdash_\Psi M \equiv N : \pi \Rightarrow \rho \quad P \Vdash \pi}{P \mid \Gamma \vdash_\Psi M \equiv N : \rho}$$

Figure 5: Equality for H^- terms.

2.2 Equality of H^- Terms

In this section, we give an axiomatic presentation of equality for H^- terms. Our primary concerns are the treatment of polymorphism and class methods; otherwise, H^- differs little from standard functional calculi. As described in the introduction, our intention is to permit reasoning about class methods directly, without relying on either a dictionary-passing translation or a preliminary inlining step that resolves all method overloading. This results in two unusual aspects of our rules:

- While our presentation gives equality for expressions, it relies critically on components of the class context $\langle A, Si, Im\rangle$—the axioms A to determine which instance solves given constraints, and the method implementations *Im* to determine the behavior of methods.
- The treatment of polymorphism cannot be completely parametric, and different equalities may be provable for the same term at different types; for example, we cannot hope to have uniform proofs of properties of the $==$ method when it is defined differently at different types.

Equality judgments take the form $P \mid \Gamma \vdash_\Psi M \equiv N : \sigma$, denoting that, assuming predicates P, variables typed as in Γ, and class context Ψ, expressions M and N are equal at type σ. To simplify the presentation, we have omitted equational assumptions; however, extending our system with assumptions and a corresponding axiom rule would be trivial. The rules are those listed in Figure 5, together with rules for reflexivity, symmetry, and transitivity of equality, and the expected α-equivalence and congruence rules for each syntactic form. Rules $\{\beta\}$, $\{\eta\}$, $\{\mu\}$ and $\{\text{LET}\}$ should

be unsurprising. Rules $\{\Rightarrow\text{I}\}$ and $\{\Rightarrow\text{E}\}$ mirror the corresponding typing rules, assuring that we can only conclude equalities about well-typed expressions. Rule $\{\forall\text{E}\}$ should also be unsurprising: if we have proved that two expressions are equal at a quantified type, we have that they are equal at any of its instances. Rule $\{\forall\text{I}\}$ is less typical, as it requires one subproof for each possible ground type (*GType* ranges over ground type expressions). Note that this is only non-trivial for terms involving overloading. Finally, rule $\{$METHOD$\}$ provides (one step of) method resolution. Intuitively, it says that for some class method x at type σ, if instance d proves that x is defined at σ, then x is equal to the implementation of x provided by instance d.

2.3 Polymorphic Identity Functions

In the introduction, we gave an example of a polymorphic function (`invBool`) that could be instantiated at all types, yet did not have parametric behavior. In this section, we will consider a function which does have parametric behavior, but is defined in an ad-hoc fashion. We will demonstrate that our treatment of equality allows us to conclude that it is equal to its parametric equivalent.

Our particular example is the identity function. First, we give its typical definition:

```
id1 :: t → t
id1 x = x
```

For our second approach, we intend an overloaded definition that is provably equal to the parametric definition. We could produce such a definition using instance chains:

```
class Id2' t where
  id2' :: t → t
instance (Id2' t, Id2' u) ⇒ Id2' (t → u) where
  id2' f = id2' ∘ f ∘ id2'
else Id2' t where
  id2' x = x
```

This gives an ad-hoc definition of the identity function, defined at all types but defined differently for function and non-function types. Reasoning about this definition would require extending the entailment relation to instance chains, introducing significant additional complexity. We present simpler instances, but restrict the domain of types to achieve a similar result.

```
class Id2 t where
  id2 :: t → t
instance Id2 Int where
  id2 x = x
instance (Id2 t, Id2 u) ⇒ Id2 (t → u) where
  id2 f = id2 ∘ f ∘ id2
```

We will use `Int` to stand in for all base (non-function) types.

It should be intuitive that, while they are defined differently, `id1 x` and `id2 x` should each evaluate to `x` for any integer or function on integers `x`. Correspondingly, given a class context Ψ that describes (at least) `Id2`, we can prove that $\vdash_\Psi$ `id1` $\equiv$ `id2` : τ (we omit the empty context and empty assumptions) for any such type τ. The case for integers is direct: one application of $\{$METHOD$\}$ is sufficient to prove $\vdash_\Psi$ `id2` $\equiv \lambda x.x$: `Int` $\to$ `Int`. For functions of (functions of...) integers, the proof has more steps, but is no more complicated. For the simplest example, to show that

$$\vdash_\Psi \texttt{id2} \equiv \lambda x.x : (\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int}),$$

we use $\{$METHOD$\}$ to show

$$\vdash_\Psi \texttt{id2} \equiv \lambda f.(\texttt{id2} \circ f \circ \texttt{id2}) : (\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int}).$$

Relying on the usual definition of composition and $\{\beta\}$, we show

$$\vdash_\Psi \lambda f.(\texttt{id2} \circ f \circ \texttt{id2}) \equiv \lambda f.\lambda x.\texttt{id2}(f(\texttt{id2}\,x)) : (\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int})$$

Finally, by two uses of $\{$METHOD$\}$ for `id2` on integers, and $\{\eta\}$, we have

$$\vdash_\Psi \lambda f.\lambda x.\texttt{id2}(f(\texttt{id2}\,x)) \equiv \lambda f.f : (\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int})$$

and thus the desired result.

We cannot expect to prove that `id1` $\equiv$ `id2` at all types (i.e., $\vdash_\Psi$ `id1` $\equiv$ `id2` : $\forall t.t \to t$) without limiting the domain of types. For example, there is no instance of `Id2` at type `Bool`; therefore, we cannot prove any non-trivial equalities $\vdash_\Psi$ `id2` $\equiv M$: `Bool` $\to$ `Bool`. However, if we were to restrict the grammar of types to those types for which `Id2` is defined (that is, if we define that $\tau ::= \texttt{Int} \mid \tau \to \tau$), then we could construct such an argument. To show that $\vdash_\Psi$ `id2` $\equiv \lambda x.x : \forall t.t \to t$, we begin by applying $\{\forall\text{E}\}$, requiring a derivation $\vdash_\Psi$ `id2` $\equiv \lambda x.x : \tau \to \tau$ for each ground type τ. We could construct such a set of derivations by induction on the structure of types, using the argument for `Int` above as the base case, and a construction following the one for `Int` $\to$ `Int` for the inductive case.

A similar approach applies to the formulation using instance chains (class `Id2'`): we could show that the first clause applied to functions, the second clause applied to any non-function type, and use induction over the structure of types with those cases.

3. A Simple Semantics for Overloading

Next, we develop a simple denotational semantics of H^- programs, extending an approach originally proposed by Ohori [12] to describe the implicit polymorphism of ML. As with the presentation of equality in the previous section, the primary new challenges arise from the definition of class methods and the treatment of overloading. We will demonstrate that the specialization-based approach is well-suited to addressing both challenges. In particular, it allows expressions to have different interpretations at each ground type without introducing additional arguments or otherwise distinguishing qualified from unqualified type schemes.

3.1 The Meaning of Qualified Types

To describe the meaning of overloaded expressions, we must begin with the meaning of qualified types. Intuitively, qualifiers in types can be viewed as predicates in set comprehensions—that is, a class `Eq` denotes a set of types, and the qualified type $\forall t.\texttt{Eq}\ t \Rightarrow t \to t \to \texttt{Bool}$ describes the set of types $\{t \to t \to \texttt{Bool} \mid t \in \texttt{Eq}\}$. However, most existing approaches to the semantics of overloading do not interpret qualifiers in this fashion: Wadler and Blott [16], for instance, translate qualifiers into dictionary arguments, while Jones [2] translates qualified types into a calculus with explicit evidence abstraction and application.

Our approach, by contrast, preserves the intuitive notion of qualifiers. Given some class context $\Psi = \langle A, Si, Im\rangle$, we define the ground instances $\lfloor\sigma\rfloor_\Psi$ of an H^- type scheme σ by:

$$\lfloor\tau\rfloor_\Psi = \{\tau\}$$

$$\lfloor\pi \Rightarrow \rho\rfloor_\Psi = \begin{cases} \lfloor\rho\rfloor_\Psi & \text{if } \Vdash_A \pi \\ \emptyset & \text{otherwise} \end{cases}$$

$$\lfloor\forall t.\sigma\rfloor_\Psi = \bigcup_{\tau \in GType} \lfloor[\tau/t]\sigma\rfloor_\Psi.$$

Equivalently, if we define $GSubst(\bar{t})$ to be substitutions that map t to ground types and are otherwise the identity, we have

$$\lfloor\forall\bar{t}.P \Rightarrow \tau\rfloor_\Psi = \{S\tau \mid S \in GSubst(\bar{t}, \Vdash_A SP\}.$$

We will omit annotation Ψ when it is unambiguous.

In the typing judgments for H^-, predicates can appear in both types and contexts. To account for both sources of predicates, we adopt Jones's constrained type schemes $(P \mid \sigma)$, where P is a list of predicates and σ is an H^- type scheme; an unconstrained type scheme σ can be treated as the constrained scheme $(\emptyset \mid \sigma)$ (as an empty set of predicates places no restrictions on the instantiation of the variables in σ). We can define the ground instances of constrained type schemes by a straightforward extension of the definition for unconstrained schemes: if $\Psi = \langle A, Si, Im \rangle$, then

$$\lfloor (P \mid \forall \bar{t}.Q \Rightarrow \tau) \rfloor_\Psi = \{S\,\tau \mid S \in \mathit{GSubst}(\bar{t}), \Vdash_A (P, S\,Q)\}.$$

3.2 Type Frames for Polymorphism

We intend to give a semantics for H^- expressions by giving a mapping from their typing derivations to type-indexed collections of monomorphic behavior. We begin by fixing a suitable domain for the monomorphic behaviors. Ohori assumed an underlying type-frame semantics; his translations, then, were from implicitly polymorphic terms to the interpretations of terms in the simply-typed λ-calculus. Unfortunately, we cannot apply his approach without some extension, as type classes are sufficient to encode polymorphic recursion. However, we can adopt Harrison's extension [1] of Ohori's approach, originally proposed to capture polymorphic recursion, and thus also sufficient for type class methods.

We begin by defining *PCPO frames*, an extension of the standard notion of type frames. A PCPO frame is a tuple

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

(where we will omit the type and term annotations when they are apparent from context) subject to the following six conditions.

1. For each ground type τ, $\mathcal{T}^{\mathsf{type}}[\![\tau]\!]$ is a non-empty set providing the interpretation of τ.
2. For each typing derivation Δ of $\Gamma \vdash M : \tau$ and Γ-compatible environment η, $\mathcal{T}^{\mathsf{term}}[\![\Delta]\!]\eta$ is the interpretation of M in $\mathcal{T}^{\mathsf{type}}[\![\tau]\!]$.
3. $T_{\tau,\upsilon} : \mathcal{T}^{\mathsf{type}}[\![\tau \to \upsilon]\!] \times \mathcal{T}^{\mathsf{type}}[\![\tau]\!] \to \mathcal{T}^{\mathsf{type}}[\![\upsilon]\!]$ provides the interpretation of the application of an element of $\tau \to \upsilon$ to an element of τ.
4. For any $f, g \in \mathcal{T}^{\mathsf{type}}[\![\tau \to \upsilon]\!]$, if, for all $x \in \mathcal{T}^{\mathsf{type}}[\![\tau]\!]$, $T_{\tau,\upsilon}(f,x) = T_{\tau,\upsilon}(g,x)$, then $f = g$.
5. $\mathcal{T}^{\mathsf{term}}[\![\cdot]\!]$ and $T_{\tau,\upsilon}$ respect the semantics of the simply-typed λ-calculus. In particular:
   - If Δ derives $\Gamma \vdash x : \tau$, then $\mathcal{T}[\![\Delta]\!]\eta = \eta(x)$;
   - If Δ derives $\Gamma \vdash MN : \upsilon$, Δ_M derives $\Gamma \vdash M : \tau \to \upsilon$ and Δ_N derives $\Gamma \vdash N : \tau$, then $\mathcal{T}[\![\Delta]\!]\eta = T_{\tau,\upsilon}(\mathcal{T}[\![\Delta_M]\!]\eta, \mathcal{T}[\![\Delta_N]\!]\eta)$; and,
   - If Δ_λ derives $\Gamma \vdash \lambda x : \tau.M : \tau \to \upsilon$ and Δ_M derives $\Gamma, x : \tau \vdash M : \upsilon$, then $T_{\tau,\upsilon}(\mathcal{T}[\![\Delta_\lambda]\!]\eta, d) = \mathcal{T}[\![\Delta_M]\!](\eta[x \mapsto d])$.
6. Each set $\mathcal{T}[\![\tau]\!]$ is a PCPO with respect to $\sqsubseteq_\tau$, $\sqcup_\tau$ and $\bot_\tau$.

The first five conditions are the standard requirements for type frames; the final condition relates the type frame and PCPO structures of a PCPO frame. Given a PCPO frame $\mathcal{T}$, we can define the interpretation of a polymorphic type scheme σ as the mappings from the ground instances τ of σ to elements of $\mathcal{T}[\![\tau]\!]$. That is:

$$\mathcal{T}^{\mathsf{scheme}}[\![\sigma]\!]_\Psi = \Pi(\tau \in \lfloor \sigma \rfloor_\Psi).\mathcal{T}^{\mathsf{type}}[\![\tau]\!].$$

where we will omit the scheme and Ψ annotations when it is not ambiguous. For example, the identity function $\lambda x.x$ has the type scheme $\forall t.t \to t$. Therefore, the semantics of the identity function is a map from the ground instances of its type (i.e., the types $\tau \to \tau$) to the semantics of the simply-typed identity function at each type. We would expect its semantics to include the pair

$$\langle \mathtt{Int} \to \mathtt{Int}, \mathcal{T}^{\mathsf{term}}[\![\vdash \lambda x : \mathtt{Int}.x : \mathtt{Int} \to \mathtt{Int}]\!] \rangle$$

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

$$\langle \mathtt{Bool} \to \mathtt{Bool}, \mathcal{T}^{\mathsf{term}}[\![\vdash \lambda x : \mathtt{Bool}.x : \mathtt{Bool} \to \mathtt{Bool}]\!] \rangle$$

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

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

and so an element of $\mathcal{T}^{\mathsf{scheme}}[\![\tau]\!]$ is a singleton map, not an element of $\mathcal{T}^{\mathsf{type}}[\![\tau]\!]$. Harrison proves that $\mathcal{T}[\![\sigma]\!]$ is itself a pointed CPO, justifying solving recursive equations in $\mathcal{T}[\![\sigma]\!]$.

Theorem 1 (Harrison)**.** *Let $\mathcal{T}$ be a PCPO frame. Then, for any type scheme σ, $\mathcal{T}[\![\sigma]\!]$ is a pointed CPO where:*

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

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

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

3.3 Semantics for Overloaded Expressions

We can now give denotations for (typing derivations of) H^- expressions. For some type environment Γ and substitution $S \in \mathit{GSubst}(\mathit{ftv}(\Gamma))$, we define an $S - \Gamma$-environment η as a mapping from variables to values such that $\eta(x) \in \mathcal{T}[\![(S\,\sigma)]\!]$ for each assignment $(x : \sigma)$ in Γ. Given a PCPO frame $\mathcal{T}$, a derivation Δ of $P \mid \Gamma \vdash_A M : \sigma$, a ground substitution S, and an environment η, we define the interpretation $\mathcal{T}[\![\Delta]\!]S\eta$ by cases. We have included only a few, representative cases here.

- Case ($\to$E): we have a derivation of the form

$$\Delta = \dfrac{\Delta_1 = \dfrac{\vdots}{P \mid \Gamma \vdash_A M : \tau \to \tau'} \qquad \Delta_2 = \dfrac{\vdots}{P \mid \Gamma \vdash_A N : \tau}}{P \mid \Gamma \vdash_A (M\,N) : \tau'}$$

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

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

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

$$\Delta = \dfrac{\Delta_1 = \dfrac{\vdots}{P, \pi \mid \Gamma \vdash_A M : \rho}}{P \mid \Gamma \vdash_A M : \pi \Rightarrow \rho}$$

  This rule excludes those cases in which the predicate does not hold; thus, we define:

$$\mathcal{T}[\![\Delta]\!]S\eta = \begin{cases} \mathcal{T}[\![\Delta_1]\!]S\eta & \text{if } S\,P \Vdash S\,\pi; \\ \emptyset & \text{otherwise.} \end{cases}$$

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

$$\Delta_1 = \dfrac{\vdots}{P \mid \Gamma \vdash_A M : \pi \Rightarrow \rho} \quad P \Vdash \pi$$
$$\Delta = \dfrac{}{P \mid \Gamma \vdash_A M : \rho}$$

This rule does not affect the semantics of expression M, and so we define:

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

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

$$\Delta_1 = \dfrac{\vdots}{P \mid \Gamma \vdash_A M : \sigma} \quad t \notin ftv(P, \Gamma)$$
$$\Delta = \dfrac{}{P \mid \Gamma \vdash_A M : \forall t.\sigma}$$

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

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

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

$$\Delta_1 = \dfrac{\vdots}{P \mid \Gamma \vdash_A M : \forall t.\sigma}$$
$$\Delta = \dfrac{}{P \mid \Gamma \vdash_A M : [\tau/t]\sigma}$$

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

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

3.4 Expressions with Class Contexts

To complete our semantics of H^- programs, we must account for the meaning of class methods. Our approach is intuitively simple: we collect the meanings of the class methods from the method implementations in each instance, and use the meanings of the methods to define the meaning of the main expression. Formally, we extend the interpretation function from derivations of $P \mid \Gamma \vdash_A M : \sigma$ to derivations of $P \mid \Gamma \vdash_\Psi M : \sigma$ as follows:

- Let Δ be a derivation of $P \mid \Gamma \vdash_\Psi M : \tau$. Then we know that Δ must begin with an application of (CTXT) (Figure 4) with one subderivation

$$\Delta_{y,d} = \dfrac{\vdots}{P \mid \Gamma, \overline{x_i : \sigma_{x_i}} \vdash_A Im(y,d) : \sigma_{y,d}}$$

for each pair $\langle y, d \rangle \in \mathsf{dom}(Im)$ and a subderivation

$$\Delta_M = \dfrac{\vdots}{P \mid \Gamma, \overline{x_i : \sigma_{x_i}} \vdash_A M : \tau}$$

for the main expression M. We enumerate the methods in the program as $x_1, x_2, \ldots, x_m$, and let

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

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

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

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

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

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

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

Finally, we extend the interpretation function to programs by

$$\mathcal{T}[\![\Delta]\!]S\eta = \mathcal{T}[\![\Delta_M]\!]S(\eta[\overline{x_i} \mapsto \overline{b_i}]).$$

3.5 Polymorphic Identity Functions Revisited

We return to our earlier example of polymorphic identity functions (§2.3). As before, we consider two definitions of identity functions, one given parametrically (`id1`) and one given by overloading (`id2`). In this section, we will show that the denotations of `id1` and `id2` agree at all types for which `id2` is defined. By doing so, we provide an intuitive demonstration that our denotational semantics captures the meaning of ad-hoc polymorphic and agrees with our definition of equality for H^- terms.

We show that $\mathcal{T}[\![\texttt{id1}]\!]$ and $\mathcal{T}[\![\texttt{id2}]\!]$ have the same value at each point in the domain of $\mathcal{T}[\![\texttt{id2}]\!]$; that is, that for any type $\tau \in GType$ such that $\Vdash$ `Id2` τ,

$$\mathcal{T}[\![\texttt{id1}]\!](\tau \to \tau) = \mathcal{T}[\![\texttt{id2}]\!](\tau \to \tau).$$

We proceed by induction on the structure of τ. In the base case, we know that $\tau = K$ for some non-functional type K. As we have assumed $\Vdash$ `Id2` τ, we must have that $K =$ `Int`, and, from the instances for `Id2`, we have

$$\begin{aligned}\mathcal{T}[\![\texttt{id2}]\!](K \to K) &= \mathcal{T}[\![\texttt{id2}]\!](\texttt{Int} \to \texttt{Int}) \\ &= \mathcal{T}[\![\vdash \lambda x : \texttt{Int}.x : \texttt{Int} \to \texttt{Int}]\!].\end{aligned}$$

As $\mathcal{T}[\![\texttt{id1}]\!](\texttt{Int} \to \texttt{Int}) = \mathcal{T}[\![\vdash \lambda x : \texttt{Int}.x : \texttt{Int} \to \texttt{Int}]\!]$, we have $\mathcal{T}[\![\texttt{id1}]\!](K \to K) = \mathcal{T}[\![\texttt{id2}]\!](K \to K)$. In the inductive case, we know that $\tau = \tau_0 \to \tau_1$ for some types τ_0 and τ_1. From the assumption that $\Vdash$ `Id2` $(\tau_0 \to \tau_1)$ and the instances for `Id2`, we can assume that `Id2` τ_0, `Id2` τ_1, and that

$$\mathcal{T}[\![\texttt{id2}]\!](\tau \to \tau) = \mathcal{T}[\![\vdash \lambda f : (\tau_0 \to \tau_1).M \circ f \circ N : \tau \to \tau]\!]$$

for some simply typed expressions M and N such that $\mathcal{T}[\![M]\!] = \mathcal{T}[\![\texttt{id2}]\!](\tau_0 \to \tau_0)$ and $\mathcal{T}[\![N]\!] = \mathcal{T}[\![\texttt{id2}]\!](\tau_1 \to \tau_1)$. The induction hypothesis gives that $\mathcal{T}[\![\texttt{id2}]\!](\tau_0 \to \tau_0) = \mathcal{T}[\![\texttt{id1}]\!](\tau_0 \to \tau_0)$ and that $\mathcal{T}[\![\texttt{id2}]\!](\tau_1 \to \tau_1) = \mathcal{T}[\![\texttt{id1}]\!](\tau_1 \to \tau_1)$, and thus that $\mathcal{T}[\![M]\!] = \mathcal{T}[\![\vdash \lambda x : \tau_1.x : \tau_1 \to \tau_1]\!]$ and $\mathcal{T}[\![N]\!] = \mathcal{T}[\![\vdash \lambda x : \tau_0.x : \tau_0 \to \tau_0]\!]$. By congruence, we have

$$\mathcal{T}[\![\texttt{id2}]\!](\tau \to \tau) = \mathcal{T}[\![\lambda f : (\tau_0 \to \tau_1).(\lambda x : \tau_1.x) \circ f \circ (\lambda x : \tau_0.x)]\!].$$

Finally, assuming a standard definition of composition, and reducing, we have

$$\begin{aligned}\mathcal{T}[\![\texttt{id2}]\!](\tau \to \tau) &= \mathcal{T}[\![\lambda f : (\tau_0 \to \tau_1).f]\!] \\ &= \mathcal{T}[\![\lambda f : \tau.f]\!] \\ &= \mathcal{T}[\![\texttt{id1}]\!](\tau \to \tau).\end{aligned}$$

In our previous discussion of this example, we argued that if the set of types were restricted to those types for which `Id2` held, then `id1` and `id2` were equal. We can show a similar result here, by showing that if we define that $\tau ::= \texttt{Int} \mid \tau \to \tau$, then $\mathcal{T}[\![\texttt{id1}]\!] = \mathcal{T}[\![\texttt{id2}]\!]$. We begin by showing that they are defined over

the same domain; that is, that $\lfloor \forall t.\, t \to t \rfloor = \lfloor \forall u.\mathtt{Id2}\ u \Rightarrow u \to u \rfloor$. By definition, we have

$$\lfloor \forall t.\, t \to t \rfloor = \{\tau \to \tau \mid \tau \in \mathit{GType}\}$$

and

$$\lfloor \forall u.\, \mathtt{Id2}\ u \Rightarrow u \to u \rfloor = \{\tau \to \tau \mid \tau \in \mathit{GType}, \Vdash \mathtt{Id2}\ \tau\}.$$

We show that $\Vdash \mathtt{Id2}\ \tau$ for all types τ by induction on the structure of τ. In the base case, we know that $\tau = \mathtt{Int}$, and by the first instance of `Id2` we have $\Vdash \mathtt{Id2}\ \tau$. In the inductive case, we know that $\tau = \tau_0 \to \tau_1$ for some types τ_0, τ_1. In this case, we have that $[\tau_0/t, \tau_1/u]\tau = t \to u$ and by the induction hypothesis, that $\Vdash \mathtt{Id2}\ \tau_0$ and $\Vdash \mathtt{Id2}\ \tau_1$. Thus, from the second instance of `Id2`, we can conclude that $\Vdash \mathtt{Id2}\ (\tau_o \to \tau_1)$, that is, that $\Vdash \mathtt{Id2}\ \tau$. Because $\Vdash \mathtt{Id2}\ \tau$ for all ground types τ, we have

$$\{\tau \to \tau \mid \tau \in \mathit{GType}, \Vdash \mathtt{Id2}\ \tau\} = \{\tau \to \tau \mid \tau \in \mathit{GType}\},$$

and so $\mathcal{T}[\![\mathtt{id1}]\!]$ and $\mathcal{T}[\![\mathtt{id2}]\!]$ are defined over the same domain. We have already shown that $\mathcal{T}[\![\mathtt{id1}]\!]$ and $\mathcal{T}[\![\mathtt{id2}]\!]$ agree at all points at which they are defined, and so we conclude $\mathcal{T}[\![\mathtt{id1}]\!] = \mathcal{T}[\![\mathtt{id2}]\!]$.

4. Formal Properties

The previous sections have outlined typing and equality judgments for H^- terms, and proposed a denotational semantics for H^- typings. In this section, we will relate these two views of the language. We begin by showing that the denotation of a typing judgment falls into the expected type. This is mostly unsurprising; the only unusual aspect of H^- in this respect is the role of the class context. We go on to show that the equational judgments are sound; again, the unusual aspect is to do with polymorphism ($\{\forall \mathrm{I}\}$ and $\{\forall \mathrm{E}\}$) and class methods ($\{\textsc{Method}\}$). The H^- type system follows Jones's original formulation of OML; we rely on several of his metatheoretical results, such as the closure of typing under substitution.

Theorem 2 (Soundness of typing)**.** *Given a class context Ψ, if Δ is a derivation of $P \mid \Gamma \vdash_\Psi M : \sigma$, S is a substitution, and η is an $(S\,\Gamma)$-environment, then $\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}[\![(S\,P \mid S\,\sigma)]\!]_\Psi$.*

We will divide the proof into three pieces. First, we show the soundness of the judgment $P \mid \Gamma \vdash_A M : \sigma$. Then, we will argue that the union of the implementations of a method has the type of the method itself. Finally, we can combine these results to argue the soundness of $P \mid \Gamma \vdash_\Psi M : \sigma$.

Lemma 3. *Given a class context $\Psi = \langle A, Si, Im\rangle$ where A is non-overlapping, if Δ is a derivation of $P \mid \Gamma \vdash_A M : \sigma$, S is a substitution, and η is a $(S\,\Gamma)$-environment, then $\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}[\![(S\,P \mid S\,\sigma)]\!]_\Psi$.*

Proof. The proof is by induction over the structure of derivation Δ. The cases are straightforward; we include several representative examples. (Meta-variables Δ_n are as in the definition of $\mathcal{T}[\![\cdot]\!]$ above.)

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

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

  then we must also have that

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

- *Case* ($\Rightarrow$ E). As entailment is (trivially) closed under substitution, $P \Vdash \pi$ implies that $S\,P \Vdash S\,\pi$ for any substitution S; thus, we can conclude that $\lfloor (S\,P \mid S\,(\pi \Rightarrow \rho)) \rfloor = \lfloor (S\,P \mid S\,\rho) \rfloor$. Finally, assuming that $\mathcal{T}[\![\Delta_1]\!]S\eta \in \mathcal{T}[\![(S\,P \mid S\,(\pi \Rightarrow \rho))]\!]$, we can conclude that $\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}[\![(S\,P \mid S\,\rho)]\!]$.

- *Case* ($\forall$I). Because $\sigma = \forall t.\sigma'$, we have that

  $$\lfloor \sigma \rfloor = \bigcup_{\tau \in \mathit{GType}} \lfloor [\tau/t]\sigma' \rfloor,$$

  and thus that

  $$\mathcal{T}[\![\sigma]\!] = \bigcup_{\tau \in \mathit{GType}} (\mathcal{T}[\![[\tau/t]\sigma']\!]).$$

  Thus, assuming that for ground types τ, $\mathcal{T}[\![\Delta_1]\!](S[t \mapsto \tau])\eta \in \mathcal{T}[\![(S\,P \mid S\,\sigma')]\!]$, we have

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

- *Case* ($\forall$E). Assuming that $\mathcal{T}[\![\Delta_1]\!]S\eta \in \mathcal{T}[\![(S\,P \mid S\,(\forall t.\sigma'))]\!]$, the same argument about ground types as in the previous case gives that $\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}[\![(S\,P \mid S\,\sigma)]\!]$. □

The interpretation of typings $P \mid \Gamma \vdash_\Psi M : \sigma$ depends on the interpretations of the class methods. We will begin by showing that the interpretation of each method is in the denotation of its type. To do so, we will demonstrate that the interpretation of the type scheme of a method is the union of the interpretation of the type schemes of its instances. This will show that the union of the implementations is in the type of the method, from which the desired result follows immediately.

Lemma 4. *The ground instances of the type scheme of a method x are the union of its ground instances at each of its instances. That is,*

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

Proof. Let $\sigma_x = \forall \overline{t}.(\pi, Q) \Rightarrow \tau$, where x is a method of $\mathit{class}(\pi)$. We prove that

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

by the inclusions

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

and

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

We will show only the first inclusion; the second is by an identical argument. Fix some $\upsilon \in \lfloor \sigma_x \rfloor$. By definition, there is some $S \in \mathit{GSubst}(\overline{t})$ such that $\upsilon = S\,\tau$ and $\Vdash S\,\pi, S\,Q$. Because $\Vdash S\,\pi$, there must be some $(d : \forall \overline{u}.\ P \Rightarrow \pi') \in A$ and substitution $S' \in \mathit{GSubst}(\overline{u})$ such that $S\,\pi = S'\,\pi'$ and $\Vdash S'\,P$. Now, we have that $\sigma_{x,d} = \forall \overline{t'}.(P, T\,Q) \Rightarrow T\,\tau$ for some substitution T; thus, there is some $T' \in \mathit{GSubst}(\overline{t'})$ such that $\upsilon = T'\,(T\,\tau)$, $S\,P = T'\,(T\,Q)$, and so $\upsilon \in \lfloor \sigma_{x,d} \rfloor$. □

Lemma 5. *The interpretation of the type scheme of a method x is the union of the interpretations of its type scheme at each instance. That is,*

$$\mathcal{T}[\![\sigma_x]\!] = \bigcup_{\langle x,d\rangle \in \mathsf{dom}(\mathit{Im})} \mathcal{T}[\![\sigma_{x,d}]\!].$$

Proof. Recall that

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

From Lemma 4, we have that

$$\mathcal{T}^{\mathsf{scheme}}[\![\sigma_x]\!] = \Pi\left(\tau \in \bigcup_{\langle x,d\rangle \in \mathsf{dom}(Im)} \lfloor \sigma_{x,d} \rfloor\right).\mathcal{T}^{\mathsf{type}}[\![\tau]\!].$$

As $\mathcal{T}^{\mathsf{type}}[\![\cdot]\!]$ is a function, this is equivalent to

$$\mathcal{T}^{\mathsf{scheme}}[\![\sigma_x]\!] = \bigcup_{\langle x,d\rangle \in \mathsf{dom}(Im)} \Pi(\tau \in \lfloor \sigma_{x,d} \rfloor).\mathcal{T}^{\mathsf{type}}[\![\tau]\!],$$

and finally, again from the definition of $\mathcal{T}^{\mathsf{scheme}}[\![\cdot]\!]$,

$$\mathcal{T}^{\mathsf{scheme}}[\![\sigma_x]\!] = \bigcup_{\langle x,d\rangle \in \mathsf{dom}(Im)} \mathcal{T}^{\mathsf{scheme}}[\![\sigma_{x,d}]\!]. \qquad \square$$

Proof of Theorem 2. Finally, we can extend the soundness of our semantics to include class contexts. From Lemmas 4 and 5, we know that the interpretations of the methods fall in the interpretations of their type schemes, and so if η is a $S-\Gamma$-environment, then $\eta[\overline{x_i} \mapsto \overline{b_i}]$ is a $S-(\Gamma, \overline{x_i : \sigma_{x_i}})$-environment. From Theorem 3, we have that $\mathcal{T}[\![\Delta_M]\!]S(\eta[\overline{x_i} \mapsto \overline{b_i}) \in \mathcal{T}[\![(SP \mid S\sigma)]\!]_\Psi$, and thus that $\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}[\![(SP \mid S\sigma)]\!]_\Psi$. $\square$

We would like to know that the meaning of an expression is independent of the particular choice of typing derivation. Unfortunately, this is not true in general for systems with type classes. A typical example involves the `read` and `show` methods, which have the following type signatures

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

We can construct an expression `show`∘`read` of type

$$(\mathsf{Read}\ t, \mathsf{Show}\ t) \Rightarrow \mathtt{String} \to \mathtt{String},$$

where variable t can be instantiated arbitrarily in the typing, changing the meaning of the expression. To avoid this problem, we adopt the notion of an unambiguous type scheme from Jones's work on coherence for qualified types [3].

Definition 6. A type scheme $\sigma = \forall\vec{t}.P \Rightarrow \tau$ is unambiguous if $ftv(P) \subseteq ftv(\tau)$.

As long as we restrict our attention to unambiguous type schemes, we have the expected coherence result. For example, suppose that Δ is a derivation of $P \mid \Gamma \vdash_A \lambda x.M : \sigma$. We observe that Δ must conclude with an application of ($\to$I), say at $P_0 \mid \Gamma \vdash_A \lambda x.M : \tau \to \tau'$, followed by a series of applications of ($\Rightarrow$I), ($\Rightarrow$E), ($\forall$I) and ($\forall$E). While these latter applications determine σ, we can see intuitively that each $\upsilon \in \lfloor\sigma\rfloor$ must be a substitution instance of $\tau \to \tau'$, and that the interpretation of Δ at each ground type must be the interpretation of an instance of the subderivation ending with ($\to$I). We can formalize these two observations by the following lemma.

Lemma 7. *If $\sigma = \forall\vec{t}.Q \Rightarrow \tau$, and $\Delta_1 \ldots \Delta_n$ is a sequence of derivations such that:*

- *Δ_1 is a derivation of $P_1 \mid \Gamma \vdash_A M : \tau_1$;*
- *Δ_n is a derivation of $P \mid \Gamma \vdash_A M : \sigma$;*
- *Each of $\Delta_2 \ldots \Delta_n$ is by ($\Rightarrow$I), ($\Rightarrow$E), ($\forall$I) or ($\forall$E); and,*
- *Each Δ_i is the principal subderivation of Δ_{i+1}*

then

(a) There is a substitution S such that $\tau = S\tau_1$ and $P \cup Q \dashv\vdash SP_1$; and,

(b) For all ground substitutions S, for all $\upsilon \in \lfloor S\sigma \rfloor$, there is a unique S' such that $\mathcal{T}[\![\Delta_n]\!]S\eta\upsilon = \mathcal{T}[\![\Delta_1]\!]S'\eta\upsilon$.

The proof is by induction on n; the cases are all trivial. We can now characterize the relationship between different typings of M.

Theorem 8 (Coherence of $\mathcal{T}[\![\cdot]\!]$)**.** *If Δ derives $P \mid \Gamma \vdash_A M : \sigma$ and Δ' derives $P' \mid \Gamma' \vdash_A M : \sigma'$, where σ and σ' are unambiguous, then for all substitutions S and S' such that $SP \dashv\vdash S'P'$, $S\Gamma = S'\Gamma'$, and $S\sigma = S'\sigma'$, and for all ground substitutions U, $\mathcal{T}[\![\Delta]\!](U \circ S) = \mathcal{T}[\![\Delta']\!](U \circ S')$.*

The proof is by induction over the structure of M. In each case, use of the inductive hypothesis is justified by Lemma 7(a), and the conclusion derived from the definition of $\mathcal{T}[\![\cdot]\!]$ and Lemma 7(b). As an immediate corollary, we have that if Δ and Δ' are two derivations of the same typing judgment, then $\mathcal{T}[\![\Delta]\!] = \mathcal{T}[\![\Delta']\!]$. We can also show that, if $P \mid \Gamma \vdash_A M : \sigma$ is a principal typing of M, with derivation Δ, and Δ' derives $P \mid \Gamma \vdash_A M : \sigma'$ for any other σ', then for each substitution S' there is a unique S such that, for all environments η, $\mathcal{T}[\![\Delta]\!]S\eta \supseteq \mathcal{T}[\![\Delta']\!]S'\eta$.

Theorem 9 (Soundness of $\equiv$)**.** *Given a class context Ψ, if σ is unambiguous, $P \mid \Gamma \vdash_\Psi M \equiv N : \sigma$, and Δ_M, Δ_N are derivations of $P \mid \Gamma \vdash_\Psi M : \sigma$, $P \mid \Gamma \vdash_\Psi N : \sigma$, then $\mathcal{T}[\![\Delta_M]\!] = \mathcal{T}[\![\Delta_N]\!]$.*

Proof. The proof is by induction over the derivation of $P \mid \Gamma \vdash_\Psi M \equiv N : \sigma$. The interesting cases are to do with polymorphism and overloading.

- *Case* $\{\Rightarrow \mathrm{I}\}$. We have a derivation concluding

$$\frac{P, \pi \mid \Gamma \vdash_\Psi M \equiv N : \rho}{P \mid \Gamma \vdash_\Psi M \equiv N : \pi \Rightarrow \rho}$$

  Let Δ_M, Δ_N be typing derivations of $P \mid \Gamma \vdash_A M : \pi \Rightarrow \rho$ and $P \mid \Gamma \vdash_A N : \pi \Rightarrow \rho$; without loss of generality (because of Theorem 8), assume that each is by ($\Rightarrow$I), with subderivations Δ'_M, Δ'_N of $P, \pi \mid \Gamma \vdash_\Psi M : \rho$ and $P, \pi \mid \Gamma \vdash_\Psi N : \rho$. From the definition of $\mathcal{T}[\![\cdot]\!]$, we have $\mathcal{T}[\![\Delta_M]\!] = \mathcal{T}[\![\Delta'_M]\!]$ and $\mathcal{T}[\![\Delta_N]\!] = \mathcal{T}[\![\Delta'_N]\!]$. The induction hypothesis gives that $\mathcal{T}[\![\Delta'_M]\!] = \mathcal{T}[\![\Delta'_N]\!]$, and so we can conclude $\mathcal{T}[\![\Delta_M]\!] = \mathcal{T}[\![\Delta_N]\!]$.
- *Case* $\{\Rightarrow \mathrm{E}\}$. We have a derivation concluding

$$\frac{P \mid \Gamma \vdash_\Psi M \equiv N : \pi \Rightarrow \rho \quad P \Vdash_A \pi}{P \mid \Gamma \vdash_\Psi M \equiv N : \rho}$$

  where $\Psi = \langle A, Si, Im\rangle$. As in the previous case, the interpretation of the typing derivations for $P \mid \Gamma \vdash_\Psi M : \rho$ and $P \mid \Gamma \vdash_\Psi M : \pi \Rightarrow \rho$ are equal, and similarly for the typing derivations for N, and thus the induction hypothesis is sufficient for the desired conclusion.
- *Case* $\{\forall \mathrm{I}\}$. We have a derivation concluding

$$\frac{\{(P \mid \Gamma \vdash_\Psi M \equiv N : [\tau/t]\sigma) \mid \tau \in GType\}}{P \mid \Gamma \vdash_\Psi M \equiv N : \forall t.\sigma}$$

  From the induction hypothesis, we can conclude that, given derivations Δ^τ_M of $P \mid \Gamma \vdash_\Psi M : [\tau/t]\sigma$ and Δ^τ_N of $P \mid \Gamma \vdash_\Psi N : [\tau/t]\sigma$, $\mathcal{T}[\![\Delta^\tau_M]\!] = \mathcal{T}[\![\Delta^\tau_N]\!]$. Let Δ_M derive $P \mid \Gamma \vdash_\Psi M : \forall t.\sigma$ (and, without loss of generality, assume Δ_M is by ($\forall$I)); we know that $\mathcal{T}[\![\Delta_M]\!] = \bigcup_{\tau \in GType} \mathcal{T}[\![\Delta^\tau_M]\!]$. We argue similarly for derivations Δ_N of $P \mid \Gamma \vdash_\Psi N : \forall t.\sigma$, and conclude that $\mathcal{T}[\![\Delta_M]\!] = \mathcal{T}[\![\Delta_N]\!]$.
- *Case* $\{\forall \mathrm{E}\}$. We have a derivation concluding

$$\frac{P \mid \Gamma \vdash_\Psi M \equiv N : \forall t.\sigma}{P \mid \Gamma \vdash_\Psi M \equiv N : [\tau/t]\sigma}$$

Let Δ_M, Δ_N be derivations that M and N have type $[\tau/t]\sigma$; without loss of generality, assume they are by ($\forall$E), with subderivations Δ'_M, Δ'_N that M and N have type $\forall t.\sigma$. From the induction hypothesis, we know $\mathcal{T}[\![\Delta'_M]\!] = \mathcal{T}[\![\Delta'_N]\!]$, and from the definition of $\mathcal{T}[\![\cdot]\!]$ we know that $\mathcal{T}[\![\Delta_M]\!] \subseteq \mathcal{T}[\![\Delta'_M]\!]$ and $\mathcal{T}[\![\Delta_N]\!] \subseteq \mathcal{T}[\![\Delta'_N]\!]$. Thus, we can conclude that $\mathcal{T}[\![\Delta_M]\!] = \mathcal{T}[\![\Delta_N]\!]$.

- *Case* {METHOD}. We have a derivation of the form

$$\frac{Si(x) = \pi, \sigma \quad d : P \Vdash_A S\,\pi}{P \mid \Gamma \vdash_{\langle A, Si, Im\rangle} x \equiv Im(x,d) : S\,\sigma}$$

Let Δ_M be the derivation of $P \mid \Gamma \vdash_\Psi x : S\,\sigma$. From the definition of $\mathcal{T}[\![\cdot]\!]$, we know that $\mathcal{T}[\![\Delta_M]\!]S\eta = \mathcal{T}[\![\Delta'_M]\!]S(\eta[\overline{x_i} \mapsto \overline{b_i}])$ where the x_i are the class methods, the b_i are their implementations, and Δ'_M is the derivation of $P \mid \Gamma, x_i : \sigma_i \vdash_A x : S\,\sigma$. Since x is a class method, we know that $\eta[\overline{x_i} \mapsto \overline{b_i}]$ maps x to some method implementation b_j, and therefore that $\mathcal{T}[\![\Delta'_M]\!] \subseteq b_j$. We also know that b_j is the fixed point of a function $f_j(\langle b_1, \ldots, b_n\rangle)S\eta = \bigcup_d \mathcal{T}[\![\Delta_{x,d'}]\!]S(\eta[\overline{x_i} \mapsto \overline{b_i}])$, where $\Delta_{x,d'}$ derives $P \mid \Gamma \vdash_A Im(x,d') : \sigma_{x,d'}$ and d is one of the d_i. Thus, we know that if Δ_N derives $P \mid \Gamma \vdash_\Psi Im(x,d) : S\,\sigma$, then $\mathcal{T}[\![\Delta_N]\!] \subseteq b_j$. Finally, as $\mathcal{T}[\![\Delta_M]\!]$ and $\mathcal{T}[\![\Delta_N]\!]$ are defined over the same domain, we have that $\mathcal{T}[\![\Delta_M]\!] = \mathcal{T}[\![\Delta_N]\!]$. □

5. Improvement and Functional Dependencies

In the introduction, we set out several ways in which extensions of type class systems went beyond the expressiveness of existing semantic approaches to overloading. In this section, we return to one of those examples, demonstrating the flexibility of our specialization-based approach to type-class semantics.

Functional dependencies [5] are a widely-used extension of type classes which capture relationships among parameters in multi-parameter type classes. Earlier, we gave a class `Elems` to abstract over common operations on collections:

```
class Elems c e | c → e where
  empty :: c
  insert :: e → c → c
```

The functional dependency `c → e` indicates that the type of a collection (`c`) determines the type of its elements (`e`). Practically speaking, this has two consequences:

- A program is only valid if the instances in the program respect the declared functional dependencies. For example, if a program already contained an instance which interpreted lists as collections:

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

  the programmer could not later add an instance that interpreted strings (lists of characters in Haskell) as collections of code-points (for simplicity represented as integers):

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

- Given two predicates `Elems` $\tau\,\upsilon$ and `Elems` $\tau'\,\upsilon'$, if we know $\tau = \tau'$, then we must have $\upsilon = \upsilon'$ for both predicates to be satisfiable.

We now consider an extension of H$^-$ to support functional dependencies. Following Jones [4], we introduce a syntactic characterization of improving substitutions, one way of describing predicate-induced type equivalence. We then extend the typing and equality judgments to take account of improving substitutions. Finally, we show that the extended systems are sound with respect to our semantics. Importantly, we do not have to extend the models of terms, nor do we introduce coercions, or other intermediate translations. We need only show that our characterization of improving substitutions is sound to show that the resulting type equivalences hold in the semantics.

5.1 Extending H$^-$ with Functional Dependencies

To account for the satisfiability of predicates in qualified types, Jones introduces the notion of an improving substitution S for a set of predicates P [4]. Intuitively, a S improves P if every satisfiable ground instance of P is also a ground instance of SP. Jones uses improving substitutions to refine the results of type inference while still inferring principal types. We will adopt a similar approach, but in typing instead of type inference.

Syntax. We begin by extending the syntax of class axioms to include functional dependency assertions:

$$\begin{array}{lll} \text{Index sets} & X, Y & \subseteq \mathbb{N} \\ \text{Class axioms} & \alpha & ::= C : X \rightsquigarrow Y \mid d : \forall \overline{t}.\, P \Rightarrow \pi \end{array}$$

In the representation of functional dependency axioms, we treat the class parameters by index rather than by name. If A were the axioms for the example above, we would expect to have a dependency

$$\texttt{Elems} : \{0\} \rightsquigarrow \{1\} \in A.$$

Any particular class name may appear in many functional dependency assertions, or in none at all. We adopt some notational abbreviations: if X is an index set, we write $\pi =_X \pi'$ to indicate that π and π' agree at least on those parameters with indices in X, and similarly write $\pi \overset{S}{\sim}_X \pi'$ to indicate that S is a unifier for those parameters of π and π' with indices in X.

Improvement. To account for improvement in typing, we need a syntactic characterization of improving substitutions. In the case of functional dependencies, this can be given quite directly. We can give an improvement rule as a direct translation of the intuitive description above:

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

For example, if we have some Q such that $Q \Vdash$ `Elems` $\tau\,\upsilon$ and $Q \Vdash$ `Elems` $\tau\,\upsilon'$, then (FUNDEP) says that the any unifying substitution U such that $U\,\upsilon = U\,\upsilon'$ is an improving substitution for Q. If S is an improving substitution for P, then the qualified type schemes $(P \mid \sigma)$ and $(SP \mid S\,\sigma)$ are equivalent, and we should be able to replace one with the other at will in typing derivations. One direction is already possible: if a term has type σ, then it is always possible to use it with type $S\,\sigma$ (by a suitable series of applications of ($\forall$I) and ($\forall$E)). On the other hand, there is not (in general) a way with our existing typing rules to use a term of type $S\,\sigma$ as a term of type σ. We add a typing rule to support this case.

$$(\text{IMPR})\ \frac{SP \mid S\Gamma \vdash_A M : S\,\sigma \quad A \vdash S \text{ improves } P}{P \mid \Gamma \vdash_A M : \sigma}$$

As in the case of ($\Rightarrow$I) and ($\Rightarrow$E), (IMPR) has no effect on the semantics of terms. Thus, if we have a derivation

$$\Delta = \frac{\Delta_1 = \dfrac{\vdots}{SP \mid S\Gamma \vdash_A M : S\,\sigma} \quad A \vdash S \text{ improves } P}{P \mid \Gamma \vdash_A M : \sigma}$$

we define that $\mathcal{T}[\![\Delta]\!]S'\eta = \mathcal{T}[\![\Delta_1]\!]S''\eta$, where $S'' \circ S = S'$ (the existence of such an S'' is guaranteed by the soundness of (FUNDEP)). Finally, we add a rule to the equality judgment allowing us

to use improving substitutions in equality proofs.

$$\{\text{IMPR}\}\ \frac{S\,P \mid S\,\Gamma \vdash_{\langle A,Si,Im\rangle} M \equiv N : S\,\sigma \quad A \vdash S \text{ improves } P}{P \mid \Gamma \vdash_{\langle A,Si,Im\rangle} M \equiv N : \sigma}$$

Validating Functional Dependency Axioms. We must augment the context rule to check that the axioms respect the declared dependencies. This can be accomplished by, first, refining the overlap check to assure that no axioms overlap on the determining parameters of a functional dependencies, and second, requiring that, for each dependency $C : X \rightsquigarrow Y$ and each instance $P \Rightarrow \pi$ of class C, any variables in the positions Y are determined by the functional dependencies of P. Our formalization of the latter notion follows Jones's development [6]. We define the closure of a set of variables J with respect to the functional dependencies F as the least set J_F^+ such that

- $J \subseteq J_F^+$; and
- If $U \rightsquigarrow V \in F$ and $U \subseteq J_F^+$, then $V \subseteq J_F^+$.

We write $ftv_X(C\,\overline{\tau})$ to abbreviate $\bigcup_{x\in X} ftv(\tau_x)$, define the instantiation of a functional dependency assertion $C : X \rightsquigarrow Y$ at a predicate $\pi = C\,\overline{\tau}$, as the dependency $ftv_X(\pi) \rightsquigarrow ftv_Y(\pi)$, and write $fd(A,P)$ for the set of the instantiation of each functional dependency assertion in A at each predicate in P. We can now define the verification conditions for axioms and the new version of (CTXT), as follows.

$$\frac{\{\pi \nsim_X \pi' \mid (d : P \Rightarrow \pi), (d' : P' \Rightarrow \pi'), (class(\pi) : X \rightsquigarrow Y) \in A\}}{\vdash \textit{non-overlapping}(A)}$$

$$\frac{\{ftv(\pi_Y) \subseteq ftv(\pi_X)^+_{fd(A,P)} \mid (d : P \Rightarrow \pi), (class(\pi) : X \rightsquigarrow Y) \in A\}}{\vdash \textit{covering}(A)}$$

$$(\text{CTXT})\ \frac{\begin{array}{c}\vdash \textit{non-overlapping}(A) \quad \vdash \textit{covering}(A) \\ \{(P \mid \Gamma, \overline{x_i : \sigma_{x_i}} \vdash_A Im(y,d) : \sigma_{y,d}) \mid \langle y,d\rangle \in \mathsf{dom}(Im)\} \\ P \mid \Gamma, \overline{x_i : \sigma_{x_i}} \vdash_A M : \sigma\end{array}}{P \mid \Gamma \vdash_{\langle A,Si,Im\rangle} M : \sigma}$$

5.2 Soundness

The significant challenge in proving soundness of the extended rules is showing that when $A \vdash S$ improves P is derivable, S is an improving substitution for P. Once we have established that result, the remaining soundness results will be direct. We introduce notation for the satisfiable ground instances of predicates P:

$$\lfloor P \rfloor_A = \{S\,P \mid S \in GSubst(ftv(P)), \Vdash_A S\,P\}.$$

We can now formally describe an improving substitution.

Lemma 10. *Given a set of axioms A such that $\vdash$ non-overlapping(A) and $\vdash$ covering(A), if $A \vdash S$ improves P, then $\lfloor P \rfloor_A = \lfloor S\,P \rfloor_A$.*

Proof. By contradiction. Assume that $A \vdash S$ improves P; then we must have π_0, π_1 such that $\Vdash_A \pi_0, \Vdash_A \pi_1$ and there is a functional dependency $(class(\pi) : X \rightsquigarrow Y) \in A$ such that $\pi_0 =_X \pi_1$ but $\pi_0 \neq_Y \pi_1$. We proceed by induction on the heights of the derivations of $\Vdash_A \pi_0, \Vdash_A \pi_1$.

- There are distinct axioms $d : P \Rightarrow \pi'_0, d' : P' \Rightarrow \pi'_1 \in A$ and substitutions S_0, S_1 such that $S_0\,\pi'_0 = \pi_0$ and $S_1\,\pi'_1 = \pi'_1$. But then $S_0 \circ S_1$ is a unifier for $\pi'_0 \sim_X \pi'_1$, contradicting $\vdash$ *non-overlapping*(A).
- There is a single axiom $d : P \Rightarrow \pi'_0$ and substitutions S_0, S_1 such that $S_0\,\pi'_0 = \pi_0$ and $S_1\,\pi'_0 = \pi_1$. We identify two subcases.
  - There is some type variable in $ftv_Y(\pi'_0) \setminus ftv_X(\pi'_0)$ that is not constrained by P. This contradicts $\vdash$ *covering*(A).
  - There is some $\pi' \in P$ such that $S_0\,\pi'$ and $S_1\,\pi'$ violate a functional dependency of $class(\pi')$. The derivations of $\Vdash S_0\,\pi'$ and $\Vdash S_1\,\pi'$ must be shorter than the derivations of $\Vdash \pi_0, \Vdash \pi_1$, and so we have the desired result by induction. □

Theorem 11 (Soundness of typing)**.** *Given a class context Ψ, if Δ is a derivation of $P \mid \Gamma \vdash_\Psi M : \sigma$, S is a substitution, and η is an $(S\,\Gamma)$-environment, then $\mathcal{T}[\![\Delta]\!]S\eta \in \mathcal{T}[\![(S\,P \mid S\,\sigma)]\!]_\Psi$.*

Proof. We need only consider the (IMPR) case. From Lemma 10, we have that if T improves P, then $\mathcal{T}[\![(P \mid \sigma)]\!]_\Psi = \mathcal{T}[\![(T\,P \mid T\,\sigma)]\!]_\Psi$, and so the result follows from the induction hypothesis. □

We extend our notion of ambiguity to take account of functional dependencies: it is enough for the variables in the predicates P to be determined by the variables of τ.

Definition 12. A type scheme $\sigma = \forall \vec{t}.P \Rightarrow \tau$ is unambiguous (given class axioms A) if $ftv(P) \subseteq ftv(\tau)^+_{fd(A,P)}$.

The previous definition of ambiguity is a special case of this definition, where $fd(A,P)$ is always empty. As uses of (IMPR) do not affect the semantics of terms, its introduction does not compromise coherence.

Theorem 13. *If σ is unambiguous and Δ_1, Δ_2 are derivations of $P \mid \Gamma \vdash_\Psi M : \sigma$, then $\mathcal{T}[\![\Delta_1]\!] = \mathcal{T}[\![\Delta_2]\!]$.*

Theorem 14 (Soundness of $\equiv$)**.** *Given a class context Ψ, if σ is unambiguous, $P \mid \Gamma \vdash_\Psi M \equiv N : \sigma$, and Δ_M, Δ_N are derivations of $P \mid \Gamma \vdash_\Psi M : \sigma, P \mid \Gamma \vdash_\Psi N : \sigma$, then $\mathcal{T}[\![\Delta_M]\!]_\Psi = \mathcal{T}[\![\Delta_N]\!]_\Psi$.*

Proof. Again, we need consider only the {IMPR} case. Without loss of generality, assume Δ_M and Δ_N are by (IMPR), with subderivations Δ'_M and Δ'_N. As the interpretations of Δ_M and Δ_N are equal to the interpretations of Δ'_M and Δ'_N, the result follows from the induction hypothesis. □

6. Related Work

The semantics of polymorphism, in its various forms, has been studied extensively over the past half century; however, the particular extensions of Haskell that motivated this work are recent, and have received little formal attention.

Our approach was inspired by Ohori's semantics of Core ML [12]. While Ohori's approach describes the semantics of polymorphism, he does not represent polymorphic values directly, which leads to an unusual treatment of the typing of `let` expressions. Harrison extends Ohori's approach to treat polymorphic recursion [1]; in doing so, he provides a representation of polymorphic values. Harrison suggests that his approach could be applied to type classes as well.

Ohori's approach to the semantics of ML is somewhat unusual; more typical approaches include those of Milner [7] and Mitchell and Harper [9]. Ohori identifies reasons to prefer his approach over either that of Milner or that of Mitchell and Harper: both approaches use a semantic domain with far more values than correspond to values of ML, either because (in the untyped case) those values would not be well-typed, or (in the explicit typed case) they differ only in the type-level operations.

The semantics of type-class-based overloading has also received significant attention. Wadler and Blott [16] described the meaning of type classes using a dictionary-passing translation, in which

overloaded expressions are parameterized by type-specific implementations of class methods. Applying their approach to the full Haskell language, however, requires a target language with more complex types than their source language. For example, in translating the `Monad` class from the Haskell prelude, the dictionary for `Monad` τ must contain polymorphic values for the `return` and `(>>=)` methods.

In his system of qualified types [2], Jones generalized the treatment of evidence by translating from a language with overloading (OML) to a language with explicit evidence abstraction and application. Jones does not provide a semantics of the language with explicit evidence abstraction and application; indeed, such a semantics could not usefully be defined without choosing a particular form of predicate, and thus a particular form of evidence.

Odersky, Wadler and Wehr [11] propose an alternative formulation of overloading, including a type system and type inference algorithm, and a ideal-based semantics of qualified types. However, their approach requires a substantial restriction to the types of overloaded values which rules out many functions in the Haskell prelude as well as the examples from our previous work [10].

Jones [5] introduced functional dependencies in type classes, and discusses their use to improve type inference; his presentation of improvement is similar to ours, but he does not augment typing as does our (IMPR) rule. Sulzmann et al. [15] give an alternative approach to the interaction of functional dependencies and type inference, via a translation into constraint-handling rules; unfortunately, their presentation conflates properties of their translation, such as termination, with properties of the relations themselves. System $\mathrm{F_C}$ [14] extends System F with type-level equality constraints and corresponding coercion terms. While we are not aware of any formal presentation of functional dependencies in terms of System $\mathrm{F_C}$, we believe that a formulation of our (FUNDEP) rule in terms of equality constraints is possible. In contrast to our approach, System $\mathrm{F_C}$ requires extending the domain of the semantics, while still requiring translation of source-level features (functional dependencies or GADTs) into features of the semantics (equality constraints).

7. Conclusion

We have proposed an alternative approach to the semantics of overloading, based on interpreting polymorphic values as sets of their monomorphic interpretations, which avoids several problems with traditional translation-based approaches. We have applied this result to a simple overloaded calculus, and shown the soundness of its typing and equality judgments. Finally, we have argued that the approach is flexible enough to support extensions to the type system, such as allowing the use of improving substitutions in typing. We conclude by identifying directions for future work:

- Practical class systems are richer than the one used in this paper. We would like to extend these results to fuller systems, including our prior work on instance chains.
- Dictionary-passing provides both a semantics of overloading and an implementation technique. We would like to explore whether implementation techniques based on specialization can be used to compile practical languages.
- We claim that our approach avoids making distinctions between some observationally equivalent terms (such as in the polymorphic identity function example). We would like to explore whether adequacy and full abstraction results for the underlying frame model can be extended to similar results for our semantics.
- Our definition of equality provides η-equivalence; however, η equivalence is not sound for Haskell. We would like to explore either whether our approach can be adapted to a language without η-equivalence.

Acknowledgments. We would like to thank: Mark Jones for initially suggesting Ohori's semantics of ML polymorphism as a basis for understanding overloading; Jim Hook for proposing the polymorphic identity function example; and, Keiko Nakata for her helpful feedback on drafts of the paper.

References

[1] 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.

[2] M. P. Jones. A theory of qualified types. In B. K. Bruckner, editor, *Proceedings of the 4th European symposium on programming*, volume 582 of *ESOP'92*. Springer-Verlag, Rennes, France, 1992.

[3] M. P. Jones. Coherence for qualified types. Technical Report YALEU/DCS/RR-989, Yale University, 1993.

[4] M. P. Jones. Simplifying and improving qualified types. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 160–169, La Jolla, California, USA, 1995. ACM.

[5] M. P. Jones. Type classes with functional dependencies. In *Proceedings of the 9th European Symposium on Programming Languages and Systems*, ESOP '00, pages 230–244, Berlin, Germany, 2000. Springer-Verlag.

[6] M. P. Jones and I. S. Diatchki. Language and program design for functional dependencies. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 87–98, Victoria, BC, Canada, 2008. ACM.

[7] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, (17):348–375, 1978.

[8] J. C. Mitchell. Polymorphic type inference and containment. *Inf. Comput.*, 76(2-3):211–249, Feb. 1988.

[9] J. C. Mitchell and R. Harper. The essence of ML. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 28–46, San Diego, California, USA, 1988. ACM.

[10] J. G. Morris and M. P. Jones. Instance chains: Type-class programming without overlapping instances. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, Baltimore, MD, 2010. ACM.

[11] M. Odersky, P. Wadler, and M. Wehr. A second look at overloading. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, FPCA '95, pages 135–146, La Jolla, California, USA, 1995. ACM.

[12] 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.

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

[14] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly. System f with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in language design and implementation*, TLDI '07, pages 53–66, Nice, France, 2007. ACM.

[15] 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.

[16] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 60–76, Austin, Texas, USA, 1989. ACM.