Semantics of Type Classes Revisited

Satish R. Thatte
Department of Mathematics and Computer Science
Clarkson University, Potsdam, NY 13699-5815
satish@sun.mcs.clarkson.edu

Abstract

We present a new approach to the semantics of languages with ML-like polymorphic types and type classes. The goals of the new approach are simplicity and generality. Our typing rules are a relatively straightforward extension of the rules for translating core-ML to core-XML [11]. The new features are an encoding of classes as recursive sets of types, and class-membership constraints on types. We show that the soundness of this type system is independent of the fixedpoint operator used to interpret (recursive) classes, and thus there is room to engineer a sound notion of well-typing based on other considerations such as decidability. These ideas are applied to investigate the appropriateness of HASKELL-style type inference which uses backward chaining for inference of instance relationships. HASKELL's algorithm turns out to imply a *least* fixed point semantics for classes. We show that the HASKELL approach is correct and complete for the special case of *convergent* classes. Although this includes all classes definable in HASKELL, most proper extensions of HASKELL allow classes that are not convergent, which helps explain the negative results for decidability of type inference for many extensions [16, 17].

1 Introduction

The use of type classes is the major innovation in the functional programming language HASKELL [4]. Type classes permit a very organized form of overloading that approaches the use of abstract classes in object-oriented languages in its expressive power. Overloading resolution requires a translation of the source program guided by the typing process. Type systems of this kind are usually *prescriptive*: only programs considered well-typed are meaningful, since meaning depends upon translation. The original system of Wadler and Blott [18] (on which HASKELL's classes are based) specified this translation with a rather complex extension of the underlying Hindley-Milner type system, including a new notion of predicated types in which typing assumptions about program variables are incorporated in types, new kinds of typing assumptions, and complex new notions of instantiation of polymorphic types and valid sets of assumptions. Volpano and Smith [17] showed that the Wadler-Blott system is undecidable (even their notion of instantiation is undecidable). There have been several attempts to present a simple algorithmic account of type inference in the presence of type classes, most notably those of Nipkow and Snelting [13] and Nipkow and Prehofer [12]. Their goal has been to stay as close to the original Hindley-Milner system as possible, adding only classes (viewed as sets of types) and class-membership constraints on types. However, their systems confirm closely to HASKELL and rely on its restrictions on class and instance declarations, and moreover, they leave out semantics altogether. Like Nipkow, *et al*, we strive for simplicity and closeness to the original Hindley-Milner system, and our extensions are intuitively similar to theirs. However, we focus on semantics rather than type inference and we attempt to treat type classes in as much generality as possible. To this end, we eliminate all the restrictions on the form of classes and instances imposed by HASKELL, as well as the (need for) explicit declaration of instance contexts.

One of the motivations for studying a general semantics for type classes is that it is not clear how arbitrary HASKELL's particular choice of a notion of well-typing for type classes is, and even exactly what parameters are available to choose from. This is of interest particu-

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

larly when considering extensions since many extensions based on a HASKELL-like approach appear to lead to undecidable type reconstruction problems. For instance, it is easy to show that type reconstruction in some reasonable extensions of HASKELL with multiparameter classes is undecidable [16]. The issue is complicated by the fact that Milner's dictum for well-typing (well-typed programs do not go wrong) is meant to apply to *descriptive* type systems where program meaning is independent of any notion of well-typing. However, given that overloading resolution may occur at run-time, it is quite easy to formulate typing/translation rules that avoid rejecting programs for overloading errors if we accept the standard notion of well-typing for the underlying core-ML-like base language. We use XML [11] enhanced with a simple **typecase** construct as the target language for our translation of programs with overloading. The details are given in Section 3, but the idea is simple enough to be grasped with an example. Consider two instances of the familiar class **Eq**:

```
class Eq a where (==) :: a -> a -> bool
instance Eq int where (==) = eqInt
instance Eq a => Eq [a] where
  (==) = eqList where eqList x y = ..
```

The translation of the overloaded symbol == is, in part[1]:

```
(==) = Λτ. typecase τ of
         Int: eqInt
         [τ']: eqList τ'
           ...
```

It should be obvious that each instance declaration simply adds a new case (corresponding to the instance type) to the **typecase**(s) for the overloaded symbol(s) involved. The context in the instance declaration, which is so problematic in the decidability of typing, plays no role in the translation. How can a program translated in this way "go wrong" at run-time? The only possible run-time type error is the application of an overloaded symbol (*i.e.*, of the corresponding **typecase**) at a type for which it is not defined—the translation restricts the well-typing question precisely to the overloading aspect. Using an encoding of classes as recursive sets of types, we show that well-typing is independent of the particular fixedpoint operator used to interpret recursive classes. Therefore, a liberal notion of well-typing based on a greatest fixedpoint semantics for classes would also exclude these type errors, whereas the standard approach implies a much more restrictive least fixed point semantics. This creates options for engineering the notion of well-typing to make decidable type reconstruction possible for extensions such as multiparameter classes where the standard notion based on least-fixedpoints fails.

Of course, there are many classes for which the least and greatest fixed point semantics coincide. In fact, it is possible to define a natural notion of *convergent* classes based on treating sets of ground types as the points of a complete metric space. We show that Banach's fixed point theorem can be used to prove that the semantics of such classes is independent of the choice of fixed points since there is no choice in their case. For convergent classes, which include all classes definable in HASKELL, the established type inference approach for type classes based on Prolog-like reasoning (treating instance declarations like Horn clauses) and the associated dictionary-based implementation can be used without loss of generality.

The rest of the paper is organized as follows. In Section 2, we describe our encoding of type classes as set expressions, and their semantics. Section 3 gives a semantics for a language (OML) consisting of core ML with a class construct that eliminates *all* the restrictions of HASKELL. The semantics is based on typing rules that specify the typing and translation of OML to a language (OXML) consisting of core XML with a **typecase** construct and a polymorphic fixedpoint operator[2]. The typing rules allow a definition of well-typing which is independent of the fixedpoint operator used in the semantics of classes. In Section 4, we define the notion of convergent classes and show that HASKELL classes are convergent and their semantics is unique (independent of the choice of fixedpoints). Finally, Section 5 briefly surveys related work and Section 6 concludes with some directions for future work.

2 Types and Classes

To motivate our notation for class expressions, consider the following example which uses HASKELL-like syntax but violates various restrictions of HASKELL:

```
class Eq a where (==) :: a -> a -> bool
 instance Eq Int
          where (==) = eqInt
 instance Eq a => Eq (a,a)
          where (==) = eqMatchedPair
 instance Ord (a->Int) => Eq (Int->a)
          where (==) = eqIntFun
```

Note that both instances and contexts can have nested constructors (*e.g.*, `Eq (a->Int)` and `Ord (a->Int)`),

[1] Note that *all* polymorphic functions such as `eqList` have an extra type parameter in the usual XML style.

[2] The latter is needed because an overloaded symbol can be used at any type in any of its own definitions. This does not lead to the problems associated with polymorphic λ-abstraction in ML/2 [9] because the types concerned are explicitly declared rather than being inferred.

and instances may be nonlinear (*e.g.*, `Eq (a,a)`). Thinking of class `Eq` as a *set of ground types*, and assuming that the three instance declarations above completely describe this set, we could use the following equation to define `Eq`:

$$\texttt{Eq} = \texttt{Int} \vee \forall \texttt{a}.(\texttt{a},\texttt{a})\downarrow_{\texttt{a}\in\texttt{Eq}} \vee \forall \texttt{a}.\texttt{Int->a}\downarrow_{(\texttt{a->int})\in\texttt{Ord}}$$

where $\vee$ is essentially union, and the form $\tau\downarrow_{\tau'\in C}$ expresses a constraint on type variables in τ in an obvious way. The use of universal quantifiers permits encoding of nonlinear and constrained instances. The recursive equation can be eliminated by using the fixed point construct μ, giving us an expression

$$\mu\texttt{Eq}.\ \texttt{Int} \vee \forall \texttt{a}.(\texttt{a},\texttt{a})\downarrow_{\texttt{a}\in\texttt{Eq}} \vee \forall \texttt{a}.\texttt{Int->a}\downarrow_{(\texttt{a->int})\in\texttt{Ord}}$$

that captures the set we want (see the semantics of classes below). In the syntax for class expressions given below, we also have the intersection ($\wedge$) operator, which is useful for representing multiple constraints on a single variable (similar to the way Nipkow and Prehofer [12] use sorts to represent intersections of classes). The notation can be used to represent very general kinds of overloading, since it places no restrictions on the form of instance declarations. For example, it permits the encoding of unusual instance declarations such as:

```
instance Ord a => Eq a where
         (==) x y = (x<=y) and (y<=x)
```

which could be used as a default instance declaration to provide an instance of `Eq` corresponding to every instance of `Ord` (since `Eq` is a superclass of `Ord` in HASKELL—this default allows instances of both the `Ord` and `Eq` classes to be derived from a definition of `<=` alone).

Syntax of Types and Classes

In the following, τ ranges over monotypes, κ over type constructors (including base types, which are zero-arity constructors), σ over constrained types, α over type variables, $\overline{\alpha}$ over class variables, C over (set expressions representing) type classes. We use $\emptyset$ to denote the empty class and $\mathcal{U}$ to denote the class of all ground types. The syntax of types and classes is given in Table 1. We shall use $FV(C)$ to denote the set of free *type* variables in C, and *Tvar*, *Cvar*, *GroundType*, and *ClassExpr* to denote the sets of all type variables, class variables, ground types, and class expressions, respectively. *GTypeSet* is the powerset of *GroundType* (and the domain in which closed classes are interpreted).

Our class expressions are general enough to encode HASKELL classes as well as most extensions including multiparameter classes and instances with arbitrary

$$\begin{aligned}
\tau &::= \alpha \mid \kappa[\tau,\ldots,\tau] \\
\sigma &::= \tau \mid \sigma\downarrow_{\tau\in C} \\
C &::= \emptyset \mid \mathcal{U} \mid \overline{\alpha} \mid \kappa[C,\ldots,C] \mid C\downarrow_{\tau\in C} \\
&\quad \mid C \vee C \mid C \wedge C \mid \forall\alpha.C \mid \mu\overline{\alpha}.C
\end{aligned}$$

Table 1: **Syntax of Types and Classes**

contexts. However, note that encoded classes are necessarily *static* (all instances of a class appear in the encoding). This is both natural and advantageous: for instance in preserving principal typing in the presence of local class declarations. HASKELL classes are *dynamic* relative to modules (new instances may be added to an existing class in a new module), which means that an overloaded expression has a potentially distinct meaning relative to each module. The effect can be simulated in practice with an additional environment of class definitions. We do not consider multimodule programs and dynamic classes in this paper.

Semantics of Classes

The semantic domain for the interpretation of class expressions is the complete lattice of finite and infinite sets of ground types (based on the subset ordering). The lattice admits (among others) least and greatest fixed point operators, denoted by $\mathcal{L}$ and $\mathcal{G}$, respectively, for continuous functions. Fixed points are clearly necessary for the interpretation of the μ construct, but *which* fixed point to use is not always clear. For instance, the (pathological) class $\mu\texttt{Foo}.\,\texttt{a}\downarrow_{\texttt{a}\in\texttt{Foo}}$ will map to either $\emptyset$ or *GroundType* depending on whether the least or the greatest fixedpoint operator is used. For the moment, we defer a decision on choice of fixed point and parameterize the semantics of classes with respect to an arbitrary fixed point operator $\mathcal{F}$ (shown as a superscript for the semantic function $[\![\ ]\!]$). The semantics of class expressions is given in Table 2. The semantic equations are the obvious ones. The only point to note is that they apply to classes which may contain free *class* variables but no free *type* variables. The environment argument $\rho : Cvar \to GTypeSet$ for $[\![\ ]\!]^{\mathcal{F}}$ maps free class variables to sets of ground types.

The applicability of (at least) the least and greatest fixedpoint operators $\mathcal{L}$ and $\mathcal{G}$ in the semantics is stated in Lemma 1 (which relies on Tarski's lattice-theoretic fixedpoint theorem [15]).

Lemma 1 *Given any class expression C without free*

$$[\![\ ]\!]^{\mathcal{F}} : \mathit{ClassExpr} \to (\mathit{Cvar} \to \mathit{GTypeSet}) \to \mathit{GTypeSet}$$

$$\begin{aligned}
[\![\emptyset]\!]^{\mathcal{F}}\rho &= \emptyset \\
[\![\mathcal{U}]\!]^{\mathcal{F}}\rho &= \mathit{GroundType} \\
[\![\overline{\alpha}]\!]^{\mathcal{F}}\rho &= \rho(\overline{\alpha}) \\
[\![\kappa[C_1,\ldots,C_m]]\!]^{\mathcal{F}}\rho &= \{\kappa[\tau_1,\ldots,\tau_m] \ : \ \tau_i \in [\![C_i]\!]^{\mathcal{F}}\rho\} \\
[\![C_1 \vee C_2]\!]^{\mathcal{F}}\rho &= [\![C_1]\!]^{\mathcal{F}}\rho \cup [\![C_2]\!]^{\mathcal{F}}\rho \\
[\![C_1 \wedge C_2]\!]^{\mathcal{F}}\rho &= [\![C_1]\!]^{\mathcal{F}}\rho \cap [\![C_2]\!]^{\mathcal{F}}\rho \\
[\![\forall\alpha.C]\!]^{\mathcal{F}}\rho &= \bigcup_{\tau \in \mathit{GroundType}} [\![\,[\tau/\alpha]\,C]\!]^{\mathcal{F}}\rho \\
[\![C{\downarrow_{\tau \in C'}}]\!]^{\mathcal{F}}\rho &= \begin{cases} [\![C]\!]^{\mathcal{F}}\rho & \text{if } \tau \in [\![C']\!]^{\mathcal{F}}\rho \\ \emptyset & \text{otherwise} \end{cases} \\
[\![\mu\overline{\alpha}.C]\!]^{\mathcal{F}}\rho &= \mathcal{F}\,(\lambda x.\,([\![C]\!]^{\mathcal{F}}\,(\rho + [\overline{\alpha} \mapsto x])))
\end{aligned}$$

Table 2: **Semantics of Class Expressions**

type *variables, any class variable* $\overline{\alpha}$*, any fixedpoint operator* $\mathcal{F}$ *and any environment* ρ*, the function*

$$\lambda x.[\![C]\!]^{\mathcal{F}}(\rho + \overline{\alpha} \mapsto x)$$

is well-defined and monotonic, and hence possesses a complete lattice of fixedpoints.

3 OML: Typing and Semantics

This section describes a set of rules for typing OML programs and translating them to OXML programs in a way that incorporates the overloading resolution process in the OXML program without rejecting any program for an overloading-related type error. We also prove the soundness of this type system in an $\mathcal{F}$-independent way. In combination with an $\mathcal{F}$-independent notion of consistency of sets of overloading constraints, this leads to an $\mathcal{F}$-independent notion of well-typing.

Syntax of OML and OXML

In the following, x ranges over program variables, e over OML expressions, E over OXML expressions, O over declarations of overloaded symbols and T over **typecase** expressions. The syntax of OML and OXML is given in Table 3. OXML is just XML with **typecase** expressions

$$\begin{aligned}
O &::= x\langle\tau\rangle : \sigma \texttt{ with } x\langle\tau\rangle = e, \ldots, x\langle\tau\rangle = e \\
e &::= x \mid \lambda x.e \mid e\ e \mid \texttt{let } x = e \texttt{ in } e \mid O \texttt{ in } e \\
T &::= \texttt{typecase } \tau \texttt{ of } \tau : E, \ldots, \tau : E \\
E &::= x \mid \lambda x : \tau.E \mid E\ E \mid \texttt{let } x : \sigma = E \texttt{ in } E \\
&\quad \mid \Lambda\alpha.E \mid E\langle\tau\rangle \mid T \mid \texttt{fix } x : \sigma.E
\end{aligned}$$

Table 3: **OML and OXML Syntax**

and a polymorphic **fix** (fixedpoint) operator. The latter is needed because an overloaded symbol can be used at any type in any of its own definitions (see inference rule *OVER* in Table 4 for the sole use of this construct). For simplicity in the formalism, we assume that there are constants in the environment for the introduction and elimination of pairing. We recall that all class expressions used in type schemes are closed with respect to class variables. As an example of OML notation, consider the following HASKELL-like declarations:

```
class Eq t where eq::t->t->bool
instance Eq Int where eq = eqInt
instance Eq a, Eq b => Eq (a,b)
    where eq = eqPair
      where eqPair (x,y) (u,v)
              = (eq x u) and (eq y v)
instance Ord a => Eq a
    where eq a b = (a <= b) and (b <= a)
```

In OML these would be rendered (taking some liberties with the notation for abstractions) as:

```
eq⟨t⟩:t->t->bool with
  eq⟨Int⟩ = eqInt,
  eq⟨a×b⟩ = let eqPair (x,y) (u,v)
                      = (eq x u) and (eq y v)
            in eqPair
  eq⟨c⟩ = let eqOrd x y = (x <= y) and (y <= x)
          in eqOrd
```

In any declaration

$$x\langle\tau_0\rangle : \sigma_0 \texttt{ with } x\langle\tau_1\rangle = e_1, \ldots, x\langle\tau_k\rangle = e_k$$

of an overloaded symbol in OML, we assume the following well-formedness conditions:

1. The sets $FV(\tau_0), FV(\tau_1), \ldots, FV(\tau_k)$ of type variables are all disjoint, and moreover, all these variables are distinct from the variables used in all other declarations of overloaded symbols.

2. $\tau_i = \xi_i\tau_0$, $1 \leq i \leq k$, for some substitutions $\xi_1, \ldots, \xi_k$.

3. $FV(\tau_0) = FV(\sigma_0)$ (where σ_0 is an arbitrary type *scheme* that is closed with respect to class variables).

The language(s) we consider incorporate a number of simplifications and generalizations as compared with HASKELL. The most important one is that in the typing rules and semantics given below, we impose no restrictions at all on the form of declarations of overloaded symbols other than those given above. The question of restrictions (and the decidability of type inference) is separated and associated with a constraint resolution problem. The "parameter" in a class declaration is a type pattern rather than a single variable. This abstracts generalizations such as multiparameter classes in a simple way. A class with two parameters `a` and `b` could be coded as in the following example:

```
f⟨a×b⟩:a->b with
  f⟨Int×Bool⟩ = let eqZero x = eq x 0 in eqZero,
  f⟨Bool×Bool⟩= not
```

We allow only one overloaded symbol per class (multiple symbols are easy to code as a derived form). We omit superclass declarations since they have no role in the semantics or typing—superclass declarations in HASKELL serve only to make contexts more concise. Finally, contexts in instance declarations are omitted since they can be inferred. Note that we allow local declarations of overloaded symbols in contrast to [18], where it is claimed that this leads to lack of principal types. The problem disappears in our case since class expressions allow environment-independent types to be given to overloaded symbols. However, to avoid semantic ambiguity, we require that all instance declarations for a class occur together with the class declaration. In a language which allows multiple modules, this would have to be generalized to also allow an instance declaration to occur in the same scope as the declaration of one of the type constructors involved in the instance type.

Semantics of OML and OXML

Constructing implementations for overloaded symbols is straightforward except for ambiguity issues. The simplest way to avoid ambiguity is to ban overlapping instances, but it is possible to use ordering or "best match" schemes to resolve ambiguity when instances do overlap. In this paper we use instance ordering, reflected in the ordering of cases in the `typecase` construct used in the implementation, to resolve ambiguity—`typecase` analyzes ground types by pattern matching in exactly the same way that the `case` construct of Standard ML analyses ordinary data values. For instance, in the `Eq` example, the code for `eq` is simply

```
Λt.typecase t of Int:eqInt,
                 a×b:eqPair⟨a⟩⟨b⟩
                   c:eqOrd⟨c⟩
```

where `t` ranges over ground types, and the variables `a`, `b` and `c` used in the last two cases are local to their respective cases. The code for `eqPair` is translated to

```
Λa.Λb.λ(x:a,y:b).λ(u:a,v:b).
        (eq⟨a⟩ x u) and (eq⟨b⟩ y v)
```

and the translation for `eqOrd` is similar. In contrast to the dictionary-based approach, this approach to translation is independent of the constraints on class and instance declarations. The code produced is simple and is amenable to many standard optimizations including partial evaluation. It also avoids many of the problems of dictionary-based translations, such as repeated construction of dictionaries, which require sophisticated optimization [5].

The semantics of OML expressions cannot be given directly since it depends upon the resolution of overloading, which is a part of the typing process. The typing/translation rules are a straightforward adaptation of the rules for the translation from ML to XML given in [11]. The general form of a rule is $TE, CE \vdash e \Rightarrow E : \sigma$ where TE denotes the typing-assumptions environment for free variables, CE denotes an environment of constraints on type variables of the form $\tau \in C$, e is the (OML) expression being typed, E is its (OXML) translation and σ is the "manifest" type of E. The new rules are $\downarrow$-*INTRO*, $\downarrow$-*ELIM* and *OVER*, which are given in Table 4. The complete set of rules is given in Table 6 in Appendix A. The interesting point about the rules $\downarrow$-*INTRO* and $\downarrow$-*ELIM* is that the introduction and elimination of "predicates" (conditions of the form $\downarrow_{\tau \in C}$) in types has *no effect* on the translation. This is the technical essence of the descriptiveness of our semantics of overloading, since it is the presence or absence of these predicates that marks the difference between overloaded and ordinary types.

The last rule (*OVER*) requires a more detailed explanation. It is helpful to use class `Eq` as a running example. The elements of the overloading declaration O are

- The *overloaded symbol* x (*e.g.*, `eq`).
- The *instance template* τ_0 (*e.g.*, `t`).
- The *type template* σ_0 (*e.g.*, `t->t->bool`).
- The *instances* τ_i, $i > 0$ (*e.g.*, `Int` and `a×b`).

$$(\downarrow\text{-}INTRO) \qquad \frac{TE,\, CE \cup \{\tau \in C\} \vdash e \Rightarrow E : \sigma}{TE,\, CE \vdash e \Rightarrow E : \sigma\downarrow_{\tau \in C}}$$

$$(\downarrow\text{-}ELIM) \qquad \frac{TE,\, CE \vdash e \Rightarrow E : \sigma\downarrow_{\tau \in C}}{TE,\, CE \cup \{\tau \in C\} \vdash e \Rightarrow E : \sigma}$$

$$\left(\begin{array}{ll} O = & x\langle\tau_0\rangle : \sigma_0 \texttt{ with } x\langle\tau_1\rangle = e_1, \ldots, x\langle\tau_k\rangle = e_k \\ & \qquad \text{where } \tau_1 = \xi_1\tau_0, \ldots, \tau_k = \xi_k\tau_0 \\ C = & \forall FV(\tau_1).\, \tau_1\downarrow_{CE_1} \vee \cdots \vee \forall FV(\tau_k).\, \tau_k\downarrow_{CE_k} \\ T = & \Lambda FV(\tau_0).\, \texttt{typecase } \tau_0 \texttt{ of } \tau_1 : E_1, \ldots, \tau_k : E_k \\ \sigma = & \forall FV(\tau_0).\, \sigma_0\downarrow_{\tau_0 \in C} \end{array}\right)$$

$$(OVER) \qquad \frac{\begin{array}{c} TE + x : \sigma,\, CE \cup CE_1 \vdash e_1 \Rightarrow E_1 : \xi_1\sigma_0 \\ \vdots \\ TE + x : \sigma,\, CE \cup CE_k \vdash e_k \Rightarrow E_k : \xi_k\sigma_0 \\ TE + x : \sigma,\, CE \vdash e \Rightarrow E : \tau' \end{array}}{TE, CE \vdash O \texttt{ in } e \Rightarrow \texttt{let } x : \sigma = \texttt{fix } x : \sigma.\, T \texttt{ in } E : \tau'}$$

Table 4: **Selected Typing Rules: OML ⇒ OXML**

- The *instance substitutions* ξ_i, $i > 0$ (*e.g.*, `{t↦Int}` and `{t↦a×b}`).
- The *instance definitions* e_i, $i > 0$ (*e.g.*, `eqPair`).

Clearly, each (class) instance must be an instance of the instance template (*i.e.*, $\tau_i = \xi_i\tau_0$), and the corresponding instance definition must possess a type equal to the corresponding instance of the type template (roughly, $e_i : \xi_i\sigma_0$). For example, the definition `eqPair` for instance `a×b` must possess (after translation) the type `{t↦a×b}t->t->bool` (*i.e.*, `a×b->a×b->bool`). Now in general such a type cannot be inferred without making some assumptions about the free type variables `a` and `b` involved in the instance `a×b`—in this case the assumptions that `a∈Eq` and `b∈Eq`. These assumptions constitute the *context* for an instance declaration in HASKELL, but we do not have such contexts in the declaration of an overloaded symbol. In the rule *OVER*, these contexts are represented by the sets CE_i and they appear as (local) assumptions in the premises of the rule corresponding to each instance: these premises are of the form

$$TE + x : \sigma,\, CE \cup CE_i \vdash e_i \Rightarrow E_i : \xi_i\sigma_0$$

where E_i is the translation of the instance definition e_i and CE represents the global overloading constraints assumed in the final judgement. The typing assumption $x : \sigma$ for the overloaded symbol being declared appears in the typing of each of its own instances because the overloaded symbol may be used recursively in any instance definition. With this explanation, the definition of the class C in the rule should be clear: the class consists of the union of (explicitly quantified versions) of all instances, where each instance τ_i is constrained by the context CE_i used in establishing the type of the corresponding definition. The translation T of the declaration of the overloaded symbol x is equally straightforward: it abstracts the type variables of the instance template τ_0 and then pattern-matches the actual instance type with the instance template to select the appropriate (translated) definition for execution. Finally, the type σ of the overloaded symbol x is just an explicitly quantified version of the type template σ_0 with the constraint that the instance template constructed with the quantified variables must belong to the class C.

The rule *OVER* is admittedly complex, but note that it deals with the translation and typing of both class and instance declarations in a self-contained way. As such, it is considerably simpler and more direct than the existing treatment of the static sematics of HASKELL-style overloading due to Wadler and Blott [18].

The dynamically typed semantics of OXML expressions is given in Table 5 using the following notation:

- d in V (where $d \in D$ and D is a summand of V) is the injection of d into V. Therefore we always have $(d \text{ in } V) \in V$.

$$[\![\ ]\!]^{\mathcal{F}} : Exp \to (Pvar \to V) \to (Tvar \to GroundType) \to V$$

$$\begin{aligned}
[\![x]\!]^{\mathcal{F}}\,\rho\,\chi &= \rho(x) \\
[\![\lambda x:\sigma.\,E]\!]^{\mathcal{F}}\,\rho\,\chi &= \lambda v.\,\mathsf{if}\ v \in [\![\chi\sigma]\!]^{\mathcal{F}}\ \mathsf{then}\ ([\![E]\!]^{\mathcal{F}}\,(\rho + [x \mapsto v])\,\chi)\ \mathsf{else}\ \textit{wrong} \\
[\![E\ E']\!]^{\mathcal{F}}\,\rho\,\chi &= \mathsf{let}\ f = [\![E]\!]^{\mathcal{F}}\,\rho\,\chi;\ x = [\![E']\!]^{\mathcal{F}}\,\rho\,\chi\ \mathsf{in}\ \mathsf{if}\ f \in V \to V\ \mathsf{then}\ f(x) \\
&\qquad\qquad \mathsf{else}\ \textit{wrong} \\
[\![\mathbf{let}\ x:\sigma = E'\ \mathbf{in}\ E]\!]^{\mathcal{F}}\,\rho\,\chi &= \mathsf{let}\ v = [\![E']\!]^{\mathcal{F}}\,\rho\,\chi\ \mathsf{in}\ \mathsf{if}\ v \in [\![\chi\sigma]\!]^{\mathcal{F}}\ \mathsf{then}\ [\![E]\!]^{\mathcal{F}}\,(\rho + [x \mapsto v])\,\chi \\
&\qquad\qquad \mathsf{else}\ \textit{wrong} \\
[\![\Lambda\alpha.\,E]\!]^{\mathcal{F}}\,\rho\,\chi &= \lambda\eta \in GroundType.\,[\![E]\!]^{\mathcal{F}}\,\rho\,(\chi + [\alpha \mapsto \eta]) \\
[\![E\langle\tau\rangle]\!]^{\mathcal{F}}\,\rho\,\chi &= \mathsf{let}\ o = [\![E]\!]^{\mathcal{F}}\,\rho\,\chi\ \mathsf{in}\ \mathsf{if}\ o \in O\ \mathsf{then}\ o(\chi\tau)\ \mathsf{else}\ \textit{wrong} \\
[\![\mathtt{fix}\ x:\sigma.\,E]\!]^{\mathcal{F}}\,\rho\,\chi &= \mathbf{Y}_\sigma([\![\lambda x:\sigma.\,E]\!]^{\mathcal{F}}\,\rho\,\chi) \\
[\![\mathtt{typecase}\ \tau_0\ \mathtt{of}\ \tau_1 : E_1, \ldots, \tau_k : E_k]\!]^{\mathcal{F}}\,\rho\,\chi &= \mathsf{if\ there\ is\ a\ least}\ i\ \mathsf{such\ that}\ \zeta\tau_i = \chi\tau_0\ \mathsf{then}\ [\![E_i]\!]^{\mathcal{F}}\,\rho\,(\chi + \zeta)\ \mathsf{else}\ \textit{wrong}
\end{aligned}$$

Table 5: **Dynamically Typed Semantics for OXML**

- *wrong* is just ω in V.
- $v \in D$ (where $v \in V$ and D is a summand of V) yields $\perp_B$ if $v = \perp_V$, true if $v = d$ in V for some $d \in D$, and false otherwise.
- $v|_D$ (where $v \in V$ and D is a summand of V) yields d if $v = d$ in V for some $d \in D$, and $\perp_D$ otherwise.
- $\mathbf{Y}_\sigma$ is a type specific fixedpoint operator. Given a function $f \in V \to V$, $\mathbf{Y}_\sigma$ returns its fixedpoint iff $f([\![\sigma]\!]^{\mathcal{F}}) \subseteq [\![\sigma]\!]^{\mathcal{F}}$, and *wrong* otherwise.

where the domain V is defined by

$$\begin{aligned}
V &\cong N + B + V \to V + V \times V + O + W \\
O &= GroundType \to V \\
W &= \{\omega\}
\end{aligned}$$

Note that the semantic equation for λ-abstraction (in Table 5) permits polymorphic abstraction. The following theorem asserts the soundness of the typing rules. The most interesting aspect of the theorem is that its statement is "parameterized" with respect to the choice of $\mathcal{F}$ in the semantics of classes.

Definition A substitution $\chi : Tvar \to GroundType$ satisfies a constraint environment CE relative to a fixedpoint operator $\mathcal{F}$ (written $\chi \models^{\mathcal{F}} CE$) iff $Dom(\chi) \supseteq FV(CE)$ and $(\tau \in C) \in \chi(CE) \Rightarrow \tau \in [\![C]\!]^{\mathcal{F}}\emptyset$. A map ρ from program variables to V satisfies a closed set of typing assumptions TE relative to a fixedpoint operator $\mathcal{F}$ (written $\rho \models^{\mathcal{F}} TE$) if for every $x \in Dom(TE)$, $\rho(x) \in [\![TE(x)]\!]^{\mathcal{F}}$.

Theorem 2 *If* $TE, CE \vdash e \Rightarrow E : \sigma$, $\chi \models^{\mathcal{F}} CE$ *(with* $Dom(\chi) = FV(CE) \cup FV(TE) \cup FV(\sigma)$*) and* $\rho \models^{\mathcal{F}} \chi(TE)$ *then* $([\![E]\!]^{\mathcal{F}}\,\rho\chi) \in [\![\chi\sigma]\!]^{\mathcal{F}}$.

Well-typing

It is easy to see that the typing/translation system of Table 6 does not guarantee lack of semantic ambiguity. For instance, let TE be the set

$$\{x : \forall\alpha.\,\mathsf{int} \to \alpha\downarrow_{\alpha\in C},\ f : \forall\alpha.\,\alpha \to \mathsf{int}\downarrow_{\alpha\in C},\ 3 : \mathsf{int}\}$$

Given, say, $C = \mathtt{Int} \vee \mathtt{Bool}$, it is possible to derive both

$$TE, \emptyset \vdash f(x\ 3) \Rightarrow (f\langle\mathtt{Int}\rangle)((x\langle\mathtt{Int}\rangle)3) : \mathtt{Int}$$

and

$$TE, \emptyset \vdash f(x\ 3) \Rightarrow (f\langle\mathtt{Bool}\rangle)((x\langle\mathtt{Bool}\rangle)3) : \mathtt{Int}$$

Clearly, the two translations for the same typing are semantically distinct. This is the familiar situation of an "ambiguously overloaded" expression [4, Section 4.3.4]. The problem is flagged by the "uncoupled" type $\mathtt{Int}\downarrow_{\beta\in C}$ used during the derivation. Such types are prohibited in HASKELL and we conjecture that if derivations are constrained not to use such types, the resulting

inference system is indeed coherent. Similar results have been reported in [2, 7] for overloading systems related to HASKELL.

Definition Consider the sequent $TE, CE \vdash e \Rightarrow E : \forall\{\alpha_1, \ldots, \alpha_k\}.\tau\downarrow_{CE'}$. Assume without loss of generality that $(FV(TE) \cup FV(CE)) \cap \{\alpha_1, \ldots, \alpha_k\} = \emptyset$. This sequent is considered *uncoupled* iff $FV(CE \cup CE') - (FV(\tau) \cup FV(TE)) \neq \emptyset$. A sequent $TE, CE \vdash e \Rightarrow E : \sigma$ is considered *hygienic* if it is not uncoupled and it has a proof that does not involve uncoupled sequents.

Conjecture 3 (Coherence of hygienic Typing) *If $TE, CE \vdash e \Rightarrow E_1 : \sigma$ and $TE, CE \vdash e \Rightarrow E_2 : \sigma$ are hygienic typings then $\forall \chi \models^{\mathcal{F}} CE$, $\forall \rho \models^{\mathcal{F}} \chi TE$, $[\![E_1]\!]^{\mathcal{F}} \rho\chi = [\![E_2]\!]^{\mathcal{F}} \rho\chi$.*

The definition of well-typing in our context needs a little care. In general, the semantics $[\![\sigma]\!]^{\mathcal{F}}$ of a type scheme σ may contain ***wrong***, but the semantics of a simple type τ never does. Moreover, as in inference with subtypes, the proof of a typing $TE, CE \vdash e \Rightarrow E : \tau$ does *not* imply that e is well-typed, since the constraint set CE may be inconsistent, implying an overloading related type error. The exact notion of consistency is a bit tricky to fix. The simplest definition is that an environment CE is consistent only if there is a $\chi : Tvar \rightarrow GroundType$ such that $\chi \models^{\mathcal{F}} CE$. However, note that any particular solution of CE in a sequent $TE, CE \vdash e \Rightarrow E : \tau$ does not have any effect on the reconstructed term E. If the only reason to check for consistency is to ensure the absence of type errors, the definition suggested above seems too strong. For instance, this definition would imply that the type $\forall\alpha.\alpha\downarrow_{[\alpha]\in\emptyset}$ is "wrong" even though $[\![\forall\alpha.\alpha\downarrow_{[\alpha]\in\emptyset}]\!]^{\mathcal{F}}$ does not contain ***wrong*** and therefore does not imply any run-time type error. This suggests that only those constraints $\tau \in C$ in CE where τ is a ground type need to be satisfied. However, there are two arguments against this. First, although $[\![\forall\alpha.\alpha\downarrow_{[\alpha]\in\emptyset}]\!]^{\mathcal{F}}$ does not contain ***wrong***, the only nontrivial value it contains maps all ground types to ***wrong***. *Any* use of such a value yields a type error, and so it does not seem unreasonable to treat the corresponding expression as being ill-typed. Secondly, it is technically possible to separate the constraints with ground types only if all class and instance declarations are global, since local class declarations may yield class expressions that contain free *type* variables. For both these reasons, we use the following simple definition:

Definition Suppose TE is a a set of typing assumptions with no free variables. An OML expression e is *well-typed* under TE iff there is a hygienic typing $TE, CE \vdash e \Rightarrow E : \tau$ and a $\chi : Tvar \rightarrow GroundType$ such that $Dom(\chi) \supseteq FV(\tau)$ and $\chi \models^{\mathcal{F}} CE$.

Note that hygienic typing ensures that $FV(CE) \cup FV(E) \subseteq FV(\tau)$ in the definition above. Therefore, given any $\rho \models^{\mathcal{F}} TE$, $[\![E]\!]^{\mathcal{F}} \rho\chi$ is meaningful. The lack of type errors in the execution of well-typed expressions is then a corollary of Theorem 2.

Corollary 4 *If an OML expression e is well-typed under a closed set of assumptions TE with a hygienic typing $TE, CE \vdash e \Rightarrow E : \tau$ such that $\chi \models^{\mathcal{F}} CE$, then given any $\rho \models^{\mathcal{F}} TE$, $[\![E]\!]^{\mathcal{F}} \rho\chi \neq$ **wrong**.*

To illustrate the relationship between the choice of $\mathcal{F}$ and well-typing, consider the following typing judgement, which is easy to derive from the rules in Table 6.

$$\{\mathtt{f} : \forall \mathtt{b}.\, \mathtt{b} \rightarrow \mathtt{bool}\downarrow_{\mathtt{b}\in\mathtt{Foo}}, 3 : \mathtt{Int}\},\ \{\mathtt{Int} \in \mathtt{Foo}\}$$
$$\vdash \mathtt{f}\ 3 : \mathtt{bool}$$

where $\mathtt{Foo} = \mu\mathtt{H}.\, \mathtt{a}\downarrow_{\mathtt{a}\in\mathtt{H}}$. Is `f 3` well-typed? This depends on whether we can find a χ such that $\chi \models^{\mathcal{F}} \{\mathtt{Int} \in \mathtt{Foo}\}$, which in turn depends on the choice of $\mathcal{F}$. Since $[\![\mathtt{Foo}]\!]^{\mathcal{L}} = \emptyset$, but $[\![\mathtt{Foo}]\!]^{\mathcal{G}} = \mathcal{U}$, `f 3` is ill-typed under $\mathcal{L}$ but well-typed under $\mathcal{G}$. In general, given a hygienic typing $TE, CE \vdash e \Rightarrow E : \tau$, well-typing reduces to the consistency of CE, which in turn is dependent on the choice of $\mathcal{F}$. Since our class expressions are *positive* (the set-complement operation is absent), $\mathcal{G}$ is always the most liberal choice in defining consistency, and therefore is the best choice for $\mathcal{F}$ if we wish to reject the fewest possible programs, assuming that the resulting type reconstruction problem is tractable.

4 Convergent Classes

In this section, we show that the choice of fixedpoint operators is moot in the semantics of classes in HASKELL. The proof uses Banach's unique fixedpoint theorem for contractive maps in complete metric spaces [14]. The powerset of *GroundType* (*i.e.*, *GTypeSet*) can be interpreted as a complete metric space in a fairly standard way using a metric based on the rank of a witness element—in this case the rank is based on an inductive construction of *GroundType*. The following brief sketch of the construction follows [10]. Suppose $S_1, S_2 \subseteq GTypeSet$. A witness element (for the difference between S_1 and S_2) is a ground type η such that $\eta \in (S_1 - S_2) \cup (S_2 - S_1)$. The closeness $c(S_1, S_2)$ of S_1 and S_2 is the least possible height of a witness element, and it is ∞ if no such element exists.

Lemma 5 *The following are some elementary properties of the closeness function.*

1. $c(S_1, S_2) = \infty$ *iff* $S_1 = S_2$.

2. $c(S_1, S_2) = c(S_2, S_1)$.

3. $c(S_1, S_2) \geq \mathsf{min}(c(S_1, S_3),\ c(S_3, S_2))$.

Given such a closeness function, one can define a metric $d(S_1, S_2) = 2^{-c(S_1,S_2)}$ where, by convention, $2^{-\infty} = 0$. The resulting metric space of the set *GTypeSet* of points is complete (every Cauchy sequence converges). A detailed proof for this fact is given in [10]. By Banach's well-known theorem [14, p. 130], fixed points of contractive functions in such a space are unique, and obtained as limits of sequences starting at an arbitrary point and extended by applying the contractive function iteratively. Note that in our case points are sets of ground types, and the limit of a (converging) increasing sequence is obtained simply as the union of all points in the sequence. A class expression C can be seen as an abstraction of a class variable $\overline{\alpha}$, *i.e.*, a function in the space *GTypeSet*$\rightarrow$*GTypeSet*, given a fixed environment ρ that maps all *other* free class variables in C. If it is a *contractive* function, then it seems likely that $[\![\mu\overline{\alpha}.C]\!]^{\mathcal{F}}\rho$ will be independent of $\mathcal{F}$.

Definition A class expression C is said to be *contractive in* $\overline{\alpha}$ if, given any environment ρ : *Cvar* $\rightarrow$ *GTypeSet* which maps all free class variables in C except $\overline{\alpha}$ and any fixedpoint operator $\mathcal{F}$, there is a fixed real coefficient r, $0 \leq r < 1$, such that for any two distinct sets S_1, S_2 of ground types, it is always the case that

$$d([\![C]\!]^{\mathcal{F}}(\rho + \overline{\alpha} \mapsto S_1), [\![C]\!]^{\mathcal{F}}(\rho + \overline{\alpha} \mapsto S_2)) \leq r \cdot d(S_1, S_2)$$

C is *nonexpansive* in $\overline{\alpha}$ if the same property holds with $0 \leq r \leq 1$.

Note that all class expressions are not nonexpansive. For instance, $\kappa_1 \downarrow_{\kappa_2[\kappa_1]\in\overline{\alpha}}$ is expansive in $\overline{\alpha}$. $\mathcal{F}$-independence attaches to the notion of *convergent* classes:

Definition A class expression C is *convergent* if in every subexpression (including C itself) of the form $\mu\overline{\alpha}.C'$, C' is contractive in $\overline{\alpha}$.

Clearly, every subexpression of a convergent expression is also convergent.

Lemma 6 *If C is a* convergent *class expression with no free* type *variables, then its semantics is invariant relative to the choice of fixed point,* i.e., *given any environment ρ : Cvar $\rightarrow$ GTypeSet, $[\![C]\!]^{\mathcal{F}_1}\rho = [\![C]\!]^{\mathcal{F}_2}\rho$ for any two fixedpoint operators $\mathcal{F}_1$ and $\mathcal{F}_2$.*

The fixedpoints needed in interpreting convergent classes are always the unique fixedpoints of Banach's theorem. We shall use $\mathcal{B}$ to denote the fixedpoint operator which returns these fixedpoints. Lemma 6 proves that this operator is meaningful for convergent classes.

The following mostly syntactic sufficient conditions are useful in characterizing contractive and nonexpansive class expressions. Clearly, a contractive expression is nonexpansive, and a variable is nonexpansive in itself.

Lemma 7

1. *$\emptyset$ and $\mathcal{U}$ are contractive in every $\overline{\alpha}$.*

2. *$\overline{\alpha}'$ is contractive in $\overline{\alpha}$ iff $\overline{\alpha}' \neq \overline{\alpha}$.*

3. *$\kappa[C_1, \ldots, C_m]$ is contractive in $\overline{\alpha}$ if each C_i is nonexpansive in $\overline{\alpha}$.*

4. *$C_1 \wedge C_2$ is contractive (nonexpansive) in $\overline{\alpha}$ if C_1 and C_2 are both contractive (nonexpansive) in $\overline{\alpha}$.*

5. *$C_1 \vee C_2$ is contractive (nonexpansive) in $\overline{\alpha}$ if C_1 and C_2 are both contractive (nonexpansive) in $\overline{\alpha}$.*

6. *$\forall\beta.C$ is contractive (nonexpansive) in $\overline{\alpha}$ if $[\tau/\beta]C$ is contractive (nonexpansive) in $\overline{\alpha}$ for every $\tau \in$ GroundType.*

7. *$C_1\downarrow_{\tau\in C_2}$ (where τ is a ground type) is contractive in $\overline{\alpha}$ if C_1 is contractive in $\overline{\alpha}$, C_2 is nonexpansive in $\overline{\alpha}$, and for any ρ and $\mathcal{F}$, $[\![C_1]\!]^{\mathcal{F}}\rho$ contains no element of height less than or equal to the height of τ.*

Lemma 7 omits the most interesting case: that of recursive classes. This is addressed in the following[3]:

Lemma 8 *A* convergent *class expression $\mu\overline{\alpha}'.C$ is contractive in $\overline{\alpha}$ if C is contractive in $\overline{\alpha}$.*

It is not hard to see that all HASKELL classes are *convergent* in our sense.

Definition A class expression C is said to be a HASKELL *class* if $C = \mu\overline{\alpha}.C_1 \vee \cdots \vee C_n$, each C_i is of the form $\forall\alpha_1.\cdots\forall\alpha_{m_i}.\kappa[\tau_1, \ldots, \tau_{n_i}]\downarrow_{\alpha_1\in C_{\alpha_1}}\cdots\downarrow_{\alpha_{m_i}\in C_{\alpha_{m_i}}}$, $FV(\tau_1) \cup \cdots \cup FV(\tau_{n_i}) = \{\alpha_1, \ldots, \alpha_{m_i}\}$, and each C_{α_j} is of the form $C'_1 \wedge \cdots \wedge C'_k$ where each C'_j is either a HASKELL class or a class variable.

For instance, suppose there are two HASKELL classes **Eq** and Ord defined by the following declarations:

[3]The statement and proof of Lemma 8 are quite similar to those of Theorem 9 in [10].

```
instance Eq int
instance Eq a, Ord a => Eq (Tree a)
instance Ord bool
instance Eq a => Ord [a]
```

The class **Eq** is encoded as $\mu\overline{\epsilon}.\,\mathtt{int} \vee \forall\alpha.\eta[\alpha] \downarrow_{\alpha\in\overline{\epsilon}\wedge C}$ where $C = \mu\overline{\omega}.\,\mathtt{bool} \vee \forall\beta.\iota[\beta] \downarrow_{\delta\in\overline{\epsilon}}$. In the encoding, $\overline{\epsilon}$ and $\overline{\omega}$ represent the classes **Eq** and Ord, and η and ι represent the type constructors **tree** and **[ ]** (list), respectively. In general, the definition of HASKELL classes assumes that each class is the solution of a recursive equation $\overline{\alpha} = C$, and is therefore encoded as $\mu\overline{\alpha}.\,C$. Of course, $\overline{\alpha}$ does not always occur free in C. The definition above in fact generalizes HASKELL classes to allow nonlinearity and nested constructors in instance types, but keeps HASKELL's restriction that contexts consist of elements of the form **C a** where **a** is a variable. As we state below, this is sufficient to ensure convergence, which implies that the only problem with nonlinearity and nested constructors in instance types is semantic ambiguity, not decidability of type inference.

Theorem 9 *If H is a* HASKELL *class then*

1. *H is convergent.*
2. *H is contractive in all class variables.*

It is interesting to consider the relationship between the HASKELL-like decision procedure for class membership, and the choice of fixed point in the interpretation of classes. This can be done by considering a decision procedure **Member**$(\tau \in C)$ which attempts to establish the membership of τ in C by using instance declarations like Horn clauses. It should be obvious that **Member** does not always terminate for arbitrary classes. However, it turns out to be complete for *convergent* classes, which, as we saw above, include all classes definable in HASKELL. Most extensions of HASKELL do not yield convergent classes and for most categories of nonconvergent classes the problem of deciding whether $\tau \in [\![C]\!]^{\mathcal{L}}\emptyset$ is undecidable. This is one way to explain the undecidability results in [17, 16], since in these studies membership of a type in a class is intuitively linked with the success of a **Member**-like procedure. Detailed definition and analysis of **Member** are omitted in this paper for lack of space, but informally, **Member**$(\tau \in C)$ succeeds if and only if $\tau \in [\![C]\!]^{\mathcal{L}}\emptyset$ (assuming both τ and C are closed). This is the basis for saying that HASKELL's type inference algorithm is based on least fixedpoint semantics for classes. Partly as a result of the duality of $\mathcal{G}$ and $\mathcal{U}$ to $\mathcal{L}$ and $\emptyset$, respectively, it turns out that just as the *success* of **Member** is linked with $\mathcal{L}$, its *failure* is linked with $\mathcal{G}$: if $\tau \notin [\![SC]\!]^{\mathcal{G}}\emptyset$, then **Member**$(\tau \in C)$ terminates with failure. As a consequence, if $\mathcal{L}$ and $\mathcal{G}$ coincide, as in the case of convergent classes, then **Member** is a complete decision procedure. This means (not surprisingly) that HASKELL's overloading resolution algorithm correctly implements the unique semantics of its classes.

5 Related Work

Kaes [8] and Wadler and Blott [18] introduced typing systems for combinations of *ad hoc* and parametric polymorphism. The latter introduced type classes and a new form of types called "predicated types" to account for overloaded expressions. The type system of HASKELL incorporates these ideas in constrained form. Volpano and Smith [17] have shown that type inference with predicated types is undecidable. Nipkow and Snelting show that (a slightly constrained version of) HASKELL is decidable using order-sorted unification [13]. Nipkow and Prehofer [12] extend this to the full HASKELL system using notions of classes and constraints similar in spirit to ours but tailored to HASKELL's restrictions. It would be relatively easy to extend their system with a notion of translation using a restricted version of the system presented here. Chen, Hudak and Odersky have described an extension of HASKELL classes [3] to allow classes of type constructors (independently using many notions similar to those of Nipkow and Prehofer). They call these classes "parametric", but their "parameters" are not arguments and they do not allow classes with multiple parameters in the normal sense. Jones [6] describes an interesting general approach to "qualified" types which is inspired by the potential similarities between type systems with predicated types, and those with subtypes, among others. Our typing rules resemble the general form described by Jones—in fact it is possible to see our approach as a special case where type expressions are used as evidence.

6 Conclusions and Further Work

Type classes are a relatively new feature in the type systems of functional languages. Most descriptions of semantics, algorithms and extensions of type classes so far (*e.g.*, [12, 13, 17]) have been based on what one might call the "standard interpretation" in [18]. In this paper, we showed that there is a range of possible interpretations independent of any particular algorithm for deciding the instance relation. We obtained a natural generalization of HASKELL's type classes which we called *convergent* classes. We then showed that

- HASKELL's notion of well-typing is the only possible one for convergent classes.

- Haskell's algorithm for deciding the instance relation is complete for all convergent classes.

There are natural examples of extensions which are not convergent, and which may therefore need a different interpretation of well-typing to permit decidable type reconstruction. One obvious direction for future work is to obtain decidability results for type inference based on greatest fixedpoint semantics. If both the instance type and context types in instance declarations are constrained to be *linear*, type inference based on greatest fixedpoints can be reduced to the problem of finding a representation for the greatest solution for a set constraint resolution problem which appears to be amenable to solution using recent results of Aiken and Wimmers [1]. However, the details of this approach remain to be worked out.

It would also be attractive to obtain a fixedpoint-independent coherence result for hygienic typing judgements. The approach used in the related general coherence result of Jones for qualified types [7] is a promising starting point.

Acknowledgements

In developing the ideas of this paper, I benefited from E-mail discussions with Cordelia Hall, Kevin Hammond, Mark Lillibridge, Mark Jones, Simon Peyton Jones and Philip Wadler.

References

[1] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints. In *Proceedings of Seventh LICS Symposium*. IEEE Computer Society Press, 1992.

[2] Stephen Blott. *Type Classes*. PhD thesis, University of Glasgow, 1991.

[3] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *Proceedings of 1992 ACM Conf. on LISP and Functional Programming*. ACM Press, 1992.

[4] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.1). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, August 1991.

[5] Mark Jones. Efficient implementation of type class overloading. Manuscript, March 1992.

[6] Mark Jones. A theory of qualified types. In B. Krieg-Brückner, editor, *Proceedings of Fifth European Symposium on Programming*. Springer-Verlag, 1992. LNCS 582.

[7] Mark P. Jones. *Qualified types: Theory and practice*. D.Phil. Thesis, Programming Research Group, Oxford University Computing Laboratory, July 1992.

[8] Stefan Kaes. Parametric overloading in polymorphic programming languages. In Harald Ganzinger, editor, *Proceedings of ESOP'88*. Springer-Verlag, 1988. LNCS 300.

[9] A.J. Kfoury, J. Tiuryn, and P. Urzyczyn. Computational consequences and prtial solutions of a generalized unification problem. In *Proceedings of Fourth LICS Symposium*, pages 98–105. IEEE Computer Society Press, June 1989.

[10] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[11] John C. Mitchell and Robert Harper. The essence of ML. In *Proceedings of Fifteenth POPL Symposium*, pages 28–46. ACM Press, 1988.

[12] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proceedings of Twentieth POPL Symposium*. ACM Press, January 1993.

[13] Tobias Nipkow and Gregor Snelting. Type classes and overloading resolution via order-sorted unification. In *Proceedings of FPCA '91*, pages 1–14. ACM Press, 1991.

[14] W.A. Sutherland. *Introduction to Metric and Topological Spaces*. Oxford University Press, 1975.

[15] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[16] Satish R. Thatté. A simple notion of multiparameter classes with an undecidable type reconstruction problem. Unpublished note (available upon request), 1993.

[17] Dennis M. Volpano and Geoffrey S. Smith. On the complexity of ML typability with overloading. In *Proceedings of FPCA '91*. ACM Press, 1991.

[18] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Proceedings of Sixteenth POPL Symposium*, pages 60–76. ACM Press, 1989.

A Typing Rules

$$(VAR) \qquad \frac{x \in Dom(TE)}{TE, CE \vdash x \Rightarrow x : TE(x)}$$

$$(\rightarrow\text{-}INTRO) \qquad \frac{TE + x : \tau, CE \vdash e_{body} \Rightarrow E_{body} : \tau'}{TE, CE \vdash \lambda x.\ e_{body} \Rightarrow \lambda x : \tau.\ E_{body} : \tau \rightarrow \tau'}$$

$$(\rightarrow\text{-}ELIM) \qquad \frac{TE, CE \vdash e_{fun} \Rightarrow E_{fun} : \tau \rightarrow \tau' \quad TE, CE \vdash e_{arg} \Rightarrow E_{arg} : \tau}{TE, CE \vdash e_{fun}\ e_{arg} \Rightarrow E_{fun}\ E_{arg} : \tau'}$$

$$(\forall\text{-}INTRO) \qquad \frac{TE, CE \vdash e \Rightarrow E : \sigma \quad \alpha \in FV(\sigma) - (FV(TE) \cup FV(CE))}{TE, CE \vdash e \Rightarrow \Lambda\alpha.\ E : \forall\alpha.\sigma}$$

$$(\forall\text{-}ELIM) \qquad \frac{TE, CE \vdash e \Rightarrow E : \forall\alpha.\sigma}{TE, CE \vdash e \Rightarrow E\langle\tau\rangle : [\tau/\alpha]\sigma}$$

$$(LET) \qquad \frac{TE, CE \vdash e_{local} \Rightarrow E_{local} : \sigma \quad TE + x : \sigma, CE \vdash e_{body} \Rightarrow E_{body} : \tau}{TE, CE \vdash \texttt{let}\ x = e_{local}\ \texttt{in}\ e_{body} \Rightarrow \texttt{let}\ x : \sigma = E_{local}\ \texttt{in}\ E_{body} : \tau}$$

$$(\downarrow\text{-}INTRO) \qquad \frac{TE, CE \cup \{\tau \in C\} \vdash e \Rightarrow E : \sigma}{TE, CE \vdash e \Rightarrow E : \sigma\downarrow_{\tau \in C}}$$

$$(\downarrow\text{-}ELIM) \qquad \frac{TE, CE \vdash e \Rightarrow E : \sigma\downarrow_{\tau \in C}}{TE, CE \cup \{\tau \in C\} \vdash e \Rightarrow E : \sigma}$$

$$(OVER) \qquad \frac{\left(\begin{array}{ll} O = & x\langle\tau_0\rangle : \sigma_0\ \texttt{with}\ x\langle\tau_1\rangle = e_1, \ldots, x\langle\tau_k\rangle = e_k \\ & \qquad \text{where } \tau_1 = \xi_1\tau_0, \ldots, \tau_k = \xi_k\tau_0 \\ C = & \forall FV(\tau_1).\ \tau_1\downarrow_{CE_1} \vee \cdots \vee \forall FV(\tau_k).\ \tau_k\downarrow_{CE_k} \\ T = & \Lambda FV(\tau_0).\ \texttt{typecase}\ \tau_0\ \texttt{of}\ \tau_1 : E_1, \ldots, \tau_k : E_k \\ \sigma = & \forall FV(\tau_0).\ \sigma_0\downarrow_{\tau_0 \in C} \end{array}\right) \quad \begin{array}{c} TE + x : \sigma, CE \cup CE_1 \vdash e_1 \Rightarrow E_1 : \xi_1\sigma_0 \\ \vdots \\ TE + x : \sigma, CE \cup CE_k \vdash e_k \Rightarrow E_k : \xi_k\sigma_0 \\ TE + x : \sigma, CE \vdash e \Rightarrow E : \tau' \end{array}}{TE, CE \vdash O\ \texttt{in}\ e \Rightarrow \texttt{let}\ x : \sigma = \texttt{fix}\ x : \sigma.\ T\ \texttt{in}\ E : \tau'}$$

Table 6: **Typing Rules: OML ⇒ OXML**