# Making ML Polymorphism More Ad Hoc


William L. Harrison
Department of Computer Science
University of Missouri-Columbia
HarrisonWL@missouri.edu

Mark P. Jones
OGI School of Science & Technology
Oregon Health & Science University
mpj@cse.ogi.edu


January 12, 2004


**Abstract**

This report is an outline on the denotational semantics of type classes in Haskell98, which is currently being prepared for submission to a journal. The intention is to present sufficient detail so that members of the Programatica team get an up-to-date view of our approach to overloading and so that we can get useful feedback.


## 1 Introduction

Type classes have proven to be a useful and expressive feature for data and program abstraction in Haskell and they quickly gained popularity within the Haskell community. However, despite having been the subject of study since the late 1980's, type classes remain something of a mystery to the programming languages community at large. This is a shame and the purpose of this article is to rememdy this situation. While our intent is mainly to provide a suitable model of Haskell98 for Programatica[], another motivation for the current work is to render one of Haskell98's most innovative and expressive features comprehensible to language researchers.

Why might type classes be difficult to comprehend? One possible answer is that type classes are not "first-class" citizens of Haskell. While they appear as part of the surface syntax of the language, but they are treated more as syntactic sugar. Up to now, classes have been viewed as constructs that mysteriously disappear through the use of the dictionary-passing translation[]. The dictionary-passing translation (DPT), while a perfectly reasonable means of *implementing* Haskell type classes, simply does not suffice as an abstract definition of classes in Haskell. Why? Such reliance on the DPT disconnects the surface syntax of Haskell , and the confusion of the programming languages community can be reasonably attributed to this shortcoming in the presentation of Haskell.

What separates the approach advocated here from previous attempts [3, 18, 6, 19] is that we do not view type classes as second class citizens. Rather, we treat them as full members of the Haskell language. As such, we must give type classes The starting point for this work is

# 2 Foundation for Haskell Polymorphism

The meaning of the ML identity function ($\lambda x.x$) is:

$$\{\ \langle \tau \to \tau, \mathrm{id}_{D_\tau} \rangle \mid \tau \in \mathsf{Type}\ \}$$

where $\mathsf{Type}$ is the set of simple (ground) types and $D_\tau$ is the meaning of $\tau$ (a type-frame). Here, $\mathrm{id}_{D_\tau}$ is a continuous function from $D_\tau$ to $D_\tau$.

Using Ohori's model of polymorphism [11, 12], we denote the type classes as set functions from a suitable subset of $\mathsf{Type}$ to type frames:

$$[\![\mathsf{Eq}]\!] = \{\langle \mathit{Int} \to \mathit{Int} \to \mathit{Bool}, \mathit{eqInt} \rangle, \langle \mathit{Float} \to \mathit{Float} \to \mathit{Bool}, \mathit{eqFloat} \rangle, \ldots\}$$

Now consider what the meaning of the term:

$$e = (\emptyset \mid \emptyset\ \vdash\ \lambda x.\ x{=}{=}x : \forall a.\mathsf{Eq}\, a \Rightarrow a \to \mathit{Bool})$$

where "$\emptyset$" refers to empty constraints.

Just as with Core-ML, the meaning of $e$ will be defined in terms of the set of its ground instances:

$$\left\{ \begin{array}{c} \mathcal{M}[\![(\emptyset \mid \emptyset\ \vdash\ \lambda x.\ x{=}{=}x : \mathsf{Eq}\, \mathit{Int} \Rightarrow \mathit{Int} \to \mathit{Bool})]\!]^{oml}, \\ \vdots \end{array} \right\}$$

These, in turn, will be defined in terms of translations into an explicitly-typed, simply-typed language analogous to $T\Lambda$:

$$\left\{ \begin{array}{c} \mathcal{M}[\![(\emptyset \mid \emptyset\ \vdash\ \lambda(x : \mathit{Int}).\ (x : \mathit{Int})({=}{=} : \mathit{Int} \to \mathit{Int} \to \mathit{Bool})(x : \mathit{Int}) :\ \ldots)]\!], \\ \vdots \end{array} \right\}$$

Note that there is now sufficient type information at the call site of ($==$) to gives its denotation:

$$\mathcal{M}[\![({=}{=} : \mathit{Int} \to \mathit{Int} \to \mathit{Bool})]\!]\varepsilon = [\![\mathsf{Eq}\,]\!]\,(\mathit{Int} \to \mathit{Int} \to \mathit{Bool}) = \mathit{eqInt}$$

Recall that $[\![\mathsf{Eq}\,]\!]$ is a set function, so the above application makes sense. But most importantly, no dictionary-passing translation is needed!

# 3 What are Constructor Classes?

Constructor classes like $\mathsf{Eq}$, $\mathsf{Ord}$, and $\mathsf{Show}$ play two related rôles in the definition of Haskell and we explore them here in the context of OML. In the type system, a class $C$ is a predicate on $\mathsf{Type}$. A class $C$ is also a set of overloaded methods. As it turns out, Ohori's semantics of ML polymorphism is a natrual setting for describing both aspects of constructor classes, and this is the subject of this section.

Firstly, a class $C$ is a logical predicate on $\mathsf{Type}$ (see the rule ($\Rightarrow E$) in Figure **??**) and, as such, is modeled in the conventional way by a subset of $\mathsf{Type}$. This subset, defined inductively from the instance declarations, determines the types for which the methods of class $C$ must be defined.

The constructor classes also are sets of *instances*, where instances are denotations (or, more generally, n-tuples of denotations) in the sense of Definition **??**. Generally, a class $C$ contains multiple methods, If a program has class definitions $C_1, \ldots, C_n$, then the instances are products of the form:

$$(\Pi\tau \in p_1.D_\tau) \times \ldots \times (\Pi\tau \in p_n.D_\tau)$$

for $p_i \subseteq \mathsf{Type}$. The precise relationship

## 3.1 A motivating example

Consider the class $\mathsf{Eq}$ as defined below:

$$\begin{array}{ll}
\mathit{class}\ \ \mathsf{Eq}\,a\ \ \mathit{where}\ \ (==) : a \to a \to \mathit{Bool} & \\
\mathit{instance}\ \ \mathsf{Eq}\,\mathit{Int}\ \ \mathit{where}\ \ (==) = \mathit{eqInt} & \{\text{ base case }\} \\
\mathit{instance}\ \ (\mathsf{Eq}\,a, \mathsf{Eq}\,b) \Rightarrow \mathsf{Eq}\,(a \times b)\ \ \mathit{where} & \{\text{ ind. case }\} \\
\qquad (\langle x, y\rangle == \langle u, v\rangle) = (x == u)\&\&(y == v) &
\end{array}$$

We view these instance declarations as an inductive definition of $\mathsf{Eq}$ as a predicate on $\mathsf{Type}$. The instance $(\mathsf{Eq}\,\mathit{Int})$ is the base case of the definition while $(\mathsf{Eq}\,a, \mathsf{Eq}\,b \Rightarrow \mathsf{Eq}\,(a \times b))$ is the inductive case. What follows is an example of how the interpretation of $\mathsf{Eq}$ is determined:

$$\begin{array}{lcl}
\lceil\mathsf{Eq}\rceil_0 & = & \{\tau \in \mathsf{Type} \mid \exists\, s.\ (s\ \mathit{Int}) = \mathit{Int}\} \ = \ \{\mathit{Int}\} \\
\lceil\mathsf{Eq}\rceil_{(n+1)} & = & \{\tau \in \mathsf{Type} \mid \exists\, s.\ (s\,\langle a \times b\rangle) = \tau\ \wedge (s\,a \in \lceil\mathsf{Eq}\rceil_n) \wedge (s\,b \in \lceil\mathsf{Eq}\rceil_n)\} \\
\mathcal{I}(\mathsf{Eq}) & = & \bigcup_{n<\omega} \lceil\mathsf{Eq}\rceil_n
\end{array}$$

where $s$ ranges over ground substitutions. Note that $\mathcal{I}(\mathsf{Eq})$ is just the set you think it is. More importantly, $\mathcal{I}(\mathsf{Eq})$ specifies the set of types at which the overloaded variable "==" must be defined. The set $\mathcal{I}(\mathsf{Eq})$ plays the same rôle that the set $(\mathrm{TP}(\mathcal{A}, \Sigma \rhd e : \rho))$ plays in the Core-ML semantics (see Definition **??**).

The example of $\mathsf{Eq}$ may be generalised as:

$$\begin{array}{lcll}
\lceil\mathsf{C_1}, \ldots, \mathsf{C_n}\rceil_0 & = & \{\langle\tau_1^*, \ldots, \tau_n^*\rangle \in \mathsf{Type}^n \mid & (\mathsf{C_j}\,\tau_k^j) \in \mathit{Base}(\mathsf{C_j}), \\
 & & & \tau_k^* = s\,\tau_k^j, \\
 & & & s \text{ is a binding } : \{\alpha_k\} \to \mathsf{Type}\} \\
\lceil\mathsf{C_1}, \ldots, \mathsf{C_n}\rceil_{(l+1)} & = & \{\langle\tau_1^*, \ldots, \tau_n^*\rangle \in \mathsf{Type}^n \mid & (\pi_m^j \Rightarrow \mathsf{C_j}\,\tau_k^j) \in \mathit{Induct}(\mathsf{C_j}), \\
 & & & s \text{ satisfies } \lceil\pi_m^j\rceil_l, \\
 & & & \tau_k^* = s\,\tau_k^j, \\
 & & & s \text{ is a binding } : \{\alpha_1, \ldots, \alpha_n\} \to \mathsf{Type}\} \\
\mathcal{I}(\mathsf{C_1}, \ldots, \mathsf{C_n}) & = & \bigcup_{k<\omega} \lceil\mathsf{C_1}, \ldots, \mathsf{C_n}\rceil_k & \\
\mathcal{I}(\mathsf{C_i}) & = & \mathit{proj}_i \left( \bigcup_{k<\omega} \lceil\mathsf{C_1}, \ldots, \mathsf{C_n}\rceil_k \right) &
\end{array}$$

# 4 Making ML Polymorphism More Ad-Hoc

In this section, we present a precise syntax and type system for class and their instances. By doing so, we make type classes as first-class citizens of Haskell, and, as such, deserving of a denotational semantics.

## 4.1 Types for Classes and Instances

The source language is described below in Definitions 1 and 2. We will give meanings to finite sequences of class implementations:

$$\mathit{classimpl}_1 \;; \ldots ;\; \mathit{classimpl}_n$$

A *class implementation* is a class declaration for a particular $\mathsf{C}$ gathered together with each of its instance declarations. It is assumed in the above sequence that any class symbol $\mathsf{C}$ is defined by at most one class implementation. Organising the syntax as such simplifies the denotational specification of classes along the lines of Section 3.1.

**Definition 1** [Type Language] Based on Figure 1 of Hall[3]. The main difference is the inclusion of types for classes and instances. Below, it is assumed that $(n \geq 0)$.

| | |
|---|---|
| Type variable | $\alpha, \beta$ |
| Type constructor | $\mathrm{T}$ |
| Class name | $\mathsf{C}$ |
| Monotype ($n = \mathrm{arity}(\mathrm{T})$) | $\tau ::= \alpha \mid \mathrm{T}\,\tau_1 \ldots \tau_n \mid \tau' \rightarrow \tau$ |
| Predicates | $\theta ::= \mathsf{C}_1\,\alpha_1, \ldots, \mathsf{C}_n\,\alpha_n$ |
| Saturated predicates | $\phi ::= \mathsf{C}_1\,\tau_1, \ldots, \mathsf{C}_n\,\tau_n$ |
| Qualified type | $\rho ::= \phi \Rightarrow \tau$ |
| Polymorphic type | $\sigma ::= \forall \alpha_1 \ldots \alpha_n . \theta \Rightarrow \tau$ |
| Class/Instance types | $\gamma ::= \{\!\mid v_1 : \forall \vec{\alpha}_1 . \tau_1, \ldots, \forall \vec{\alpha}_n . \tau_n \mid\!\}$ |

**Definition 2** [Source Language] Based on Figure 2 of Hall[3]. Assume $(m, n, k \geq 0)$ below.

**Class implementation for $\mathsf{C}$**

$\mathit{classimpl} \;::=\; \mathit{classdecl} \;;\; \mathit{instdecl}_1 \;; \ldots ;\; \mathit{instdecl}_m$

where *classdecl* and $\mathit{instdecl}_i$ are class and instance declarations for $\mathsf{C}$.

**Class declaration for $\mathsf{C}$**

$\mathit{classdecl} \;::=\; \mathtt{class}\ (\mathsf{C}_1\alpha, \ldots, \mathsf{C}_n\alpha) \Rightarrow \mathsf{C}\,\alpha\ \mathtt{where}\ \gamma$

**Instance declaration for $\mathsf{C}$**

$\mathit{instdecl} \;::=\; \mathtt{instance}\ (\mathsf{C}_1\alpha_1, \ldots, \mathsf{C}_m\alpha_m) \Rightarrow \mathsf{C}\,(\mathrm{T}\,\beta_1 \ldots \beta_k)\ \mathtt{where}\ \mathit{binds}$

where $\{\alpha_1, \ldots, \alpha_m\} \subseteq \{\beta_1, \ldots, \beta_k\}$.

**Binding records**

$\mathit{binds} \;::=\; \{\!\mid \mathit{var}_1 = \mathit{exp}_1, \ldots, \mathit{var}_n = \mathit{exp}_n \mid\!\}$

**Expressions**

$\mathit{exp} \;::=\; \mathit{var} \mid \lambda\,\mathit{var}.\,\mathit{exp} \mid \mathit{exp}\ \mathit{exp}' \mid \mathtt{let}\ \mathit{var} = \mathit{exp}'\ \mathtt{in}\ \mathit{exp}$

| **Haskell Concrete Syntax** | **Class/Instance Type** |
| --- | --- |

## Single Methods

```
class Eq a where
   (==)   ::   a -> a -> Bool
```

$$: \forall\alpha.\ \mathsf{Eq}\,\alpha \Rightarrow \{\!|\, (\texttt{==}) : \alpha \rightarrow \alpha \rightarrow \mathit{Bool} \,|\!\}$$

```
instance Eq Int where (==) = primEqInt
```

$$: \{\} \Rightarrow \{\!|\, (\texttt{==}) : \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Bool} \,|\!\}$$

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

$$: \forall\alpha.\ \mathsf{Eq}\,\alpha \Rightarrow \{\!|\, (\texttt{==}) : [\alpha] \rightarrow [\alpha] \rightarrow \mathit{Bool} \,|\!\}$$

```
instance (Eq a, Eq b) => Eq (a,b) where
   (x,y) == (u,v)   =   (x==u) && (y==v)
```

$$: \forall\alpha,\beta.\ \mathsf{Eq}\,\alpha, \mathsf{Eq}\,\beta \Rightarrow \{\!|\, (\texttt{==}) : (\alpha \times \beta) \rightarrow (\alpha \times \beta) \rightarrow \mathit{Bool} \,|\!\}$$

## Multiple Methods

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

$$: \forall m.\ \mathsf{Monad}\,m \Rightarrow \{\!|\, \texttt{return} : \forall\alpha.\ \alpha \rightarrow m\,\alpha, (\texttt{>>=}) : \forall\alpha,\beta.\ \alpha \rightarrow (\alpha \rightarrow m\,\beta) \rightarrow m\,\beta \,|\!\}$$

```
instance Monad Maybe where
   Just x >>= k    =   k x
   Nothing >>= k   =   Nothing
   return          =   Just
```

$$: \{\} \Rightarrow \{\!|\, \texttt{return} : \forall\alpha.\ \alpha \rightarrow \mathit{Maybe}\,\alpha, (\texttt{>>=}) : \forall\alpha,\beta. \mathit{Maybe}\,\alpha \rightarrow (\alpha \rightarrow \mathit{Maybe}\,\beta) \rightarrow \mathit{Maybe}\,\beta \,|\!\}$$

```
instance Monad [ ] where
   (x:xs) >>= f   =   f x ++ (xs >>=f)
   [] >>= f       =   []
   return x       =   [x]
```

$$: \{\} \Rightarrow \{\!|\, \texttt{return} : \forall\alpha.\ \alpha \rightarrow [\alpha], (\texttt{>>=}) : \forall\alpha,\beta.\ [\alpha] \rightarrow (\alpha \rightarrow [\beta]) \rightarrow [\beta] \,|\!\}$$

## Superclasses

```
class Monad m => MonadPlus m where
   mzero   ::   m a
   mplus   ::   m a -> m a -> m a
```

$$: \forall m.\ \mathsf{Monad}\,m, \mathsf{MonadPlus}\,m \Rightarrow \{\!|\, \texttt{return} : \forall\alpha.\ \alpha \rightarrow m\,\alpha, (\texttt{>>=}) : \forall\alpha,\beta.\ \alpha \rightarrow (\alpha \rightarrow m\,\beta) \rightarrow m\,\beta \,|\!\} \oplus \{\!|\, \texttt{mzero} : \forall\alpha. m\,\alpha, \texttt{mplus} : \forall\alpha. m\alpha \rightarrow m\alpha \rightarrow m\alpha \,|\!\}$$

```
instance MonadPlus Maybe where
   mzero                 =   Nothing
   Nothing `mplus` ys    =   ys
   xs `mplus` ys         =   xs
```

$$: \{\} \Rightarrow \{\!|\, \texttt{return} : \forall\alpha.\ \alpha \rightarrow \mathit{Maybe}\,\alpha, (\texttt{>>=}) : \forall\alpha,\beta. \mathit{Maybe}\,\alpha \rightarrow (\alpha \rightarrow \mathit{Maybe}\,\beta) \rightarrow \mathit{Maybe}\,\beta \,|\!\} \oplus \{\!|\, \texttt{mzero} : \forall\alpha. \mathit{Maybe}\,\alpha, \texttt{mplus} : \forall\alpha. \mathit{Maybe}\,\alpha \rightarrow \mathit{Maybe}\,\alpha \rightarrow \mathit{Maybe}\,\alpha \,|\!\}$$

```
instance MonadPlus [ ] where
   mzero   =   []
   mplus   =   ++
```

$$: \{\} \Rightarrow \{\!|\, \texttt{return} : \forall\alpha.\ \alpha \rightarrow [\alpha], (\texttt{>>=}) : \forall\alpha,\beta. [\alpha] \rightarrow (\alpha \rightarrow [\beta]) \rightarrow [\beta] \,|\!\} \oplus \{\!|\, \texttt{mzero} : \forall\alpha. [\alpha], \texttt{mplus} : \forall\alpha. [\alpha] \rightarrow [\alpha] \rightarrow [\alpha] \,|\!\}$$

Figure 1: Giving record types to Haskell classes and instances. Here the $\oplus$ type operator combines to record types with distinct labels. The Eq class is an example of a class with one method, while the Monad class has multiple methods. The MonadPlus class illustrates how *superclass* definitions affect the type of the class.

### 4.1.1 Summary of Record Syntax from Standard ML

We need type rules for the record constructs introduced in Definitions 1 and 2, and, for these, we turn to Standard ML [9] for inspiration. This is appropriate as the record types are all known statically. We also include a record selector "`.`" in the basic syntax for simplicity and base its type rule on [14].

The following definition summarizes the treatment of records in ML. Below, $C$ is a type context (the exact structure of which is unimportant here) and $\varrho$ is a record type (i.e., a "$\{\!|\ \langle tyrow\rangle\ |\!\}$"). Note that $\varrho$ may also be viewed as a (finite) set map in $Label \stackrel{\text{fin}}{\to} MLtypes$.

**Definition 3** [Record expressions & types in Standard ML]

| | | | |
|---|---|---|---|
| Atomic expressions | $atexp$ | $::=$ | $\ldots \mid \{\!|\ \langle exprow\rangle\ |\!\} \mid \ldots$ |
| Expression rows | $exprow$ | $::=$ | $\ldots \mid lab\texttt{=}exp\ \langle \texttt{,}\, exprow\rangle \mid \ldots$ |
| Types | $ty$ | $::=$ | $\ldots \mid \{\!|\ \langle tyrow\rangle\ |\!\} \mid \ldots$ |
| Type rows | $tyrow$ | $::=$ | $\ldots \mid lab\texttt{:}\, ty\ \langle \texttt{,}\, tyrow\rangle \mid \ldots$ |

$$\frac{C \vdash exp : t}{C \vdash lab \texttt{=} exp : \{\!|\ lab : exp\ |\!\}} \qquad \frac{C \vdash exp : t \quad C \vdash exprow : \varrho}{C \vdash lab \texttt{=} exp\texttt{,}\, exprow : \{\!|\ lab : exp\ |\!\} \oplus \varrho}$$

$$\frac{}{C \vdash \{\!|\ |\!\} : \{\!|\ |\!\}} \qquad \frac{C \vdash exprow : \varrho}{C \vdash \{\!|\ exprow\ |\!\} : \varrho} \qquad \frac{C \vdash exp : \{\!|\ l_0 : t_0, \ldots, l_n : t_n\ |\!\}}{C \vdash exp\,\texttt{.}\,l_j : t_j}$$

## 4.2 A type rule for class declarations

**Definition 4 (Putative Type Rule for Class Declarations)** *Let each $classdecl_i$ below be a class declaration for $\mathsf{C}_i$. We introduce a relation ($\hookrightarrow$) between class declarations and record types which determines the precise record structure corresponding to a class.*

$$\frac{classdecl_1 \hookrightarrow \forall\alpha_1.\gamma_1 \ \ \ldots\ \ classdecl_n \hookrightarrow \forall\alpha_n.\gamma_n}{(\texttt{class}\ (\mathsf{C}_1\alpha, \ldots, \mathsf{C}_n\alpha) \Rightarrow\ \mathsf{C}\,\alpha\ \texttt{where}\ \gamma) \hookrightarrow \forall\alpha.\gamma_1[\alpha_1/\alpha] \oplus \ldots \oplus \gamma_n[\alpha_n/\alpha] \oplus \gamma}$$

## 4.3 Type inference for Classes & Instances

What follows is an informal derivation of the type of the list instance for the Eq class. We use rules for records along the lines of Standard ML (Definition 3). Also, we include a new environment for the types of instance methods in the typing judgements. The intent is to motivate general inference rules for classes and instances.

In the following, let *eqList* stand for the following term:

```
\ x -> \ y ->  case (x,y)      of
                  ([],[])      ->  True
                  (u:us,v:vs)  ->  u==v && us==vs
                  (_,_)        ->  False
```

The idea is to show how type judgements such as:

$$\texttt{instance Eq a => Eq [a] where (==)=}eqList : \forall\alpha.\,\mathsf{Eq}\,\alpha \Rightarrow \{\!|\ \texttt{(==)} : [\alpha] \to [\alpha] \to Bool\ |\!\}$$

may be plausibly derived.

Class definitions (without superclass constraints) are viewed as being little more than syntactic sugar for a particular overloaded, record type:

$$\frac{}{\texttt{class Eq a where (==)::a->a->Bool} : \forall\alpha.\,\mathsf{Eq}\,\alpha \Rightarrow \{\!|\, (\texttt{==}) : \alpha \to \alpha \to \mathit{Bool} \,|\!\}}$$

The following is a plausible derivation of the list instance of $\mathsf{Eq}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{(cont'd below)}}{\emptyset \mid \texttt{==} : \alpha \to \alpha \to \mathit{Bool}, \texttt{==} : [\alpha] \to [\alpha] \to \mathit{Bool} \mid \emptyset \;\vdash\; \mathit{eqList} : \{\!|\, (\texttt{==}) : [\alpha] \to [\alpha] \to \mathit{Bool} \,|\!\}}\,(iv)}{\mathsf{Eq}\,\alpha \mid \texttt{==} : \alpha \to \alpha \to \mathit{Bool}, \texttt{==} : [\alpha] \to [\alpha] \to \mathit{Bool} \mid \emptyset \;\vdash\; \mathit{eqList} : \mathsf{Eq}\,\alpha \Rightarrow \{\!|\, (\texttt{==}) : [\alpha] \to [\alpha] \to \mathit{Bool} \,|\!\}}\,(iii)}{\emptyset \vdash \texttt{instance Eq a => Eq [a] where (==)=}\mathit{eqList} : \mathsf{Eq}\,\alpha \Rightarrow \{\!|\, (\texttt{==}) : [\alpha] \to [\alpha] \to \mathit{Bool} \,|\!\}}\,(ii)}{\emptyset \vdash \texttt{instance Eq a => Eq [a] where (==)=}\mathit{eqList} : \forall\alpha.\,\mathsf{Eq}\,\alpha \Rightarrow \{\!|\, (\texttt{==}) : [\alpha] \to [\alpha] \to \mathit{Bool} \,|\!\}}\,(i)$$

Comments:

- $(i)$ Polymorphism rule. Nothing new here.
- $(ii)$ Here, the presence of two overloaded types, $\alpha$ and $[\alpha]$, are expanded into the *method environment* $M$. In a judgement $(P \mid M \mid \Sigma \;\vdash\; e : \sigma)$, the environments $P$ and $\Sigma$ are class constraints and variable typings, respectively, as in OML. $M$ binds method names to the particular type instances corresponding to $\mathsf{Eq}\alpha$ and $\mathsf{Eq}[\alpha]$.
- $(iii)$ This is just $(\Rightarrow E)$.

$$\dfrac{\dfrac{\dfrac{\left.\begin{array}{l}\emptyset \mid M_{\texttt{==}} \mid \texttt{x},\texttt{y},\texttt{us},\texttt{vs} : [\alpha], \texttt{u},\texttt{v} : \alpha \;\vdash\; \texttt{u==v} \;\; : \mathit{Bool} \\ \emptyset \mid M_{\texttt{==}} \mid \texttt{x},\texttt{y},\texttt{us},\texttt{vs} : [\alpha], \texttt{u},\texttt{v} : \alpha \;\vdash\; \texttt{us==vs} : \mathit{Bool}\end{array}\right\}\text{(remaining subgoals)}}{\emptyset \mid M_{\texttt{==}} \mid \texttt{x},\texttt{y},\texttt{us},\texttt{vs} : [\alpha], \texttt{u},\texttt{v} : \alpha \;\vdash\; \texttt{u==v \&\& us==vs} : \mathit{Bool}}\,(vi)}{\emptyset \mid M_{\texttt{==}} \mid \texttt{x} : [\alpha], \texttt{y} : [\alpha] \;\vdash\; \texttt{case (x,y) of } \mathit{br}_1 \;;\; \mathit{br}_2 \;;\; \mathit{br}_3 : \mathit{Bool}}\,(v)}{\emptyset \mid \texttt{==} : \alpha \to \alpha \to \mathit{Bool}, \texttt{==} : [\alpha] \to [\alpha] \to \mathit{Bool} \mid \emptyset \;\vdash\; \mathit{eqList} : \{\!|\, (\texttt{==}) : [\alpha] \to [\alpha] \to \mathit{Bool} \,|\!\}}\,(iv)$$

Comments:

- $(iv)$ This is $(\lambda I) \times 2$. Nothing new here. We abbreviate the method environment by $M_{\texttt{==}}$. Also, the record type has mysteriously disappeared.
- $(v)$ This shows the only interesting case of the $\mathit{br}_2$ case branch. The extraction of the types of pattern variables `u`, `v`,`us` and `vs` is elided.
- $(vi)$ The remaining subgoals follow by "function" application of `==`.

## 4.4 Type Rule for Instances

$$\dfrac{\begin{array}{l} FTV(\tau_i) \subseteq \{\alpha_1, \ldots, \alpha_k\} = FTV(\tau) \\ (m : \forall\beta.C\beta \Rightarrow \sigma') \in VE \\ TE \vdash EE' \cup \{C_1\tau_1, \ldots, C_n\tau_n\};\, VE \vdash e : [\tau/\beta]\sigma' \\ TE; \overbrace{EE \cup \{\forall\alpha_1, \ldots, \alpha_k.(C_1\tau_1, \ldots, C_n\tau_n) \Vdash C\tau\}}^{EE'};\, VE \vdash \mathit{prog} : \\ TE; EE \cup \{\forall\alpha.C\alpha \Vdash C_i\alpha\};\, VE \vdash \mathit{prog} : \end{array}}{A \vdash \texttt{instance } (C_1\tau_1, \ldots, C_n\tau_n) \Rightarrow C\tau \texttt{ where } \{m_1 = e_1 \ldots m_k = e_k\;\} : \forall\alpha_1 \ldots \alpha_h.\theta \Rightarrow C\tau}$$

```
data Seq a = Nil | SCons (a,Seq (a,a))

size :: (Seq a) -> Int
size s = if (isNil s) then 0 else (1 + (2 * (size (stl s))))
      where stl (Scons(x,xs)) = xs
            isNil Nil         = True
            isNil (SCons _)   = False
```

Figure 2: From Okasaki [13], p. 142: a useful poly-rec function!

# 5 Overloading with Polymorphic Recursion

**Remark 1** [Modeling Parametric Polymorphism] Consider the term polymorphic (but not polymorphic recursive) term *length*:

$$\begin{array}{lcl} \mathit{length} & : & [a] \rightarrow \mathit{Int} \\ \mathit{length} & = & \lambda l.\ \text{case}\ l\ \{[\,] \rightarrow 0\ ;\ (x : xs) \rightarrow 1 + (\mathit{length}\ xs)\} \end{array}$$

Within Ohori's framework, this should denote the following set:

$$\{\langle \tau, \mathit{len}_\tau \rangle \mid \tau = [\mathit{Int}] \rightarrow \mathit{Int}, \ldots\}$$

where each $\mathit{len}_\tau$ is defined as:

$$\mathit{len}_\tau = \mathit{fix}_\tau\,(\lambda \mathit{length}.\ \lambda l.\ \text{case}\ l\ \{[\,] \rightarrow 0\ ;\ (x : xs) \rightarrow 1 + (\mathit{length}\ xs)\})$$

Note that $\mathit{fix}_\tau$ is the least fixed point on the pointed cpo $D_\tau$.

**Remark 2** [Modeling Polymorphic Recursion] Consider now the term polymorphic recursive term *foo*:

$$\begin{array}{lcl} \mathit{foo} & : & [a] \rightarrow \mathit{Bool} \\ \mathit{foo} & = & \lambda x.\ (\mathit{null}\ x)\ \&\&\ (\mathit{foo}\ [x]) \end{array}$$

One might be tempted to view this definition as shorthand for the following:

$$\mathit{foo} \quad = \quad \mathit{fix}\,(\lambda \mathit{foo}.\lambda x.\ (\mathit{null}\ x)\ \&\&\ (\mathit{foo}\ [x])) \qquad (\dagger)$$

But, where does this *fix* live? Intuitively, here's the problem: if the "input *foo*" (i.e., "$\mathit{fix}\,(\lambda \mathit{foo} \ldots$") lives in $D_{[\tau]}$, then the "output *foo*" (i.e., "$(\mathit{foo}\ [x])$") lives in $D_{[[\tau]]}$. The above definition does not make sense in any particular type frame.

## 5.1 The Haskell Frame $\mathcal{P}$

So, what to do? Luckily, any frame in which the $D_\tau$ are pointed cpos can be extended to a pointed cpo over the indexed sets $\Pi\,\tau \in S.|D_\tau|$ used to denote polymorphic terms. This yields a least

**Theorem 1** *Let $\langle \mathcal{D}, \bullet, \sqsubseteq, \sqcup \rangle$ be a pcpo-frame and $S$ be a set of ground type expressions. Then, $\Pi\, \tau{\in} S.|D_\tau|$ is a pointed cpo where:*

- *for any $f, g \in (\Pi\, \tau{\in} S.|D_\tau|$, $f \sqsubseteq g \Leftrightarrow$ for all $\tau \in S$, $f\tau \sqsubseteq_\tau g\tau$, and*
- *the bottom element is $\perp_S \triangleq \{\langle \tau, \perp_\tau \rangle \mid \tau \in S\}$*

Proof of Theorem 1. To show that $\sqsubseteq$ and $\perp_S$ as above define a pointed, complete partial order on $\Pi\, \tau{\in} S.|D_\tau|$. Assume $X, X_i \in \Pi\, \tau{\in} S.|D_\tau|$. That $\sqsubseteq$ is reflexive, anti-symmetric, and transitive follows directly from the fact the each $\sqsubseteq_\tau$ is so. Similarly, $\perp_S$ is the least element of $\Pi\, \tau{\in} S.|D_\tau|$ because, for any $\tau \in \mathsf{Type}$, so is $\perp_S \tau = \perp_\tau$. Let $X_0 \sqsubseteq X_1 \sqsubseteq \cdots$ be a directed chain. To show: $\bigsqcup X_i$ exists. Let $U \triangleq \{\langle \tau, u_\tau \rangle \mid \tau \in S \;\&\; u_\tau = \bigsqcup_\tau (X_i \tau)\}$. Clearly, $X_i \sqsubseteq U$ and $U \in (\Pi\, \tau{\in} S.|D_\tau|)$. Say there is a $V \in (\Pi\, \tau{\in} S.|D_\tau|)$ such that $V \sqsubset U$ and $X_i \sqsubseteq V$. Then, for some $\tau \in \mathsf{Type}$, $V\tau \sqsubset U\tau$ and $X_i \tau \sqsubseteq V\tau$. $\therefore \bigcup_\tau (X_i \tau)$ is not the lub of the directed chain $X_0\tau \sqsubseteq_\tau X_1 \tau \cdots$ in $|D_\tau|$. $\rightarrow\!\leftarrow$. So, every directed chain within $(\Pi\, \tau{\in} S.|D_\tau|)$ has a least upper bound in $(\Pi\, \tau{\in} S.|D_\tau|)$.

Note that, because $(\Pi\, \tau{\in} S.|D_\tau|)$ is a pointed cpo, we may define continuous functions and fixpoints over it in the standard way [1, 2, 15]. A function $f : (\Pi\, \tau{\in} S.|D_\tau|) \rightarrow (\Pi\, \tau{\in} T.|D_\tau|)$ is ***continuous*** when $f(\bigsqcup_S X_i) = \bigsqcup_T (f\, X_i)$. Given a continuous endofunction $f$, its least fixed point of is given by:

$$\mathsf{fix}(f) = \bigsqcup\nolimits_S (f^n \perp_S) \text{ for } n < \omega \;\&\; f : (\Pi\, \tau{\in} S.|D_\tau| \rightarrow \Pi\, \tau{\in} S.|D_\tau|) \rightarrow (\Pi\, \tau{\in} S.|D_\tau|),$$

### 5.1.1 The inevitable cardinality question

How big are these $(\Pi\, \tau{\in} S.|D_\tau|)$? The answer, made precise by the theorem below, is "just as big as the $D_\tau$":

**Theorem 2** *If card $\mathsf{Type} = \aleph_0$, then card($\mathsf{Type} \rightarrow |D_\tau|) \leq$ max($\aleph_0$, card$|D_\tau|$).*

Proof. Observe first that $(\Pi\, \tau{\in} \mathsf{Type}.|D_\tau|) \subseteq \mathsf{Type} \times |D_\tau|$. Now by cardinal arithmetic (see Kaplansky [7], Theorem 16, for example), $\mathit{card}(\mathsf{Type} \times |D_\tau|) = \mathit{max}(\aleph_0, \mathit{card}|D_\tau|)$.

## 5.2 Polymorphic Recursion

### 5.2.1 ML/1′ : a "syntax-oriented" version of ML/1

**Definition 5** [Instantiations] Let $\tau, \tau' \in \mathbf{T}_0$ and $\vec{\alpha} = \alpha_1 \cdots \alpha_n$ for some $n \geq 0$. We say that $\tau'$ is an *instantiation* of $\forall \vec{\alpha}.\tau$ for substitution $s$ (written $(\forall \vec{\alpha}.\tau) \preceq_s \tau'$), if, and only if, for some $\tau_1 \cdots \tau_n$ in $\mathbf{T}_0$,

$$\tau' = \tau \underbrace{[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]}_{s}$$

**Definition 6** [Alternate Type System for Polymorphic Recursion] from Kfoury, Tiuryn, and Urzyczyn [8]. The rules GEN, APP, ABS, and LET are identical to those of ML/1.

$$M ::= x \mid (M\ N) \mid (\lambda x.M) \mid (\text{let } x = N \text{ in } M) \mid (\mathsf{pfix}\ x.M)$$

$$\begin{array}{ll} (\text{Open Types } \mathbf{T}_0) & \tau ::= \alpha \mid (\tau \to \tau') \\ (\text{Universal Types } \mathbf{T}_1) & \sigma ::= \forall \alpha.\sigma \mid \tau \end{array}$$

$$\text{VAR} \quad \frac{\Gamma(x) = \sigma, \sigma \preceq \tau}{\Gamma \vdash x : \sigma} \qquad\qquad \text{APP} \quad \frac{\Gamma \vdash M : \tau \to \tau' \quad \Gamma \vdash N : \tau}{\Gamma \vdash (M\ N) : \tau'}$$

$$\text{GEN} \quad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall \alpha.\sigma}\ (\alpha \notin FV(\Gamma), \alpha \in FV(\sigma)) \qquad\qquad \text{ABS} \quad \frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash (\lambda x.M) : \tau \to \tau'}$$

$$\text{PFIX} \quad \frac{\Gamma, x : \forall \vec{\alpha}.\tau' \vdash M : \tau'}{\Gamma \vdash \mathsf{pfix}\ x.M : \tau}\ (\forall \vec{\alpha}.\tau' \preceq \tau, \vec{\alpha} \notin FV(\Gamma)) \qquad\qquad \text{LET} \quad \frac{\Gamma \vdash N : \sigma' \quad \Gamma, x : \sigma' \vdash M : \sigma}{\Gamma \vdash (\text{let } x = N \text{ in } M) : \sigma}$$

Below is the derivation of `size` in ML/1′.

$$\cfrac{\cfrac{\cfrac{\cfrac{}{\cdots} \quad \cfrac{}{\cdots} \quad \cfrac{\cfrac{}{\cdots} \quad \cfrac{\cfrac{}{\cdots} \quad \cfrac{\cfrac{\begin{array}{l} \Gamma_2(\mathit{size}) = \forall \alpha.(\mathit{Seq}\,\alpha) \to \mathit{Int} \\ \forall \alpha.(\mathit{Seq}\,\alpha) \to \mathit{Int} \preceq_s \mathit{Seq}(\alpha \times \alpha) \to \mathit{Int} \\ s = [\alpha \mapsto \alpha \times \alpha] \end{array}}{\Gamma_2 \vdash \mathit{size} : \mathit{Seq}(\alpha \times \alpha) \to \mathit{Int}}\ \text{VAR} \quad \cfrac{\cfrac{}{\cdots}}{\Gamma_2 \vdash (\mathit{stl}\ s) : \alpha \times \alpha}}{\Gamma_2 \vdash \mathit{size}\ (\mathit{stl}\ s) : \mathit{Int}}}{\Gamma_2 \vdash 2 * (\mathit{size}\ (\mathit{stl}\ s)) : \mathit{Int}}\ \delta_*}{\Gamma_2 \vdash 1 + (2 * (\mathit{size}\ (\mathit{stl}\ s))) : \mathit{Int}}\ \delta_+}{\Gamma_1, s : (\mathit{Seq}\ \alpha) \vdash \text{if } (\mathit{isNil}\,s) \text{ then } 0 \text{ else } (1 + (2 * (\mathit{size}\ (\mathit{stl}\ s)))) : \mathit{Int}}\ \delta_{\text{if}}}{\Gamma_1 \vdash \lambda s.\text{if } (\mathit{isNil}\,s) \text{ then } 0 \text{ else } (1 + (2 * (\mathit{size}\ (\mathit{stl}\ s)))) : (\mathit{Seq}\,\alpha) \to \mathit{Int}}\ \text{ABS}}{\cfrac{\Gamma_0 \vdash \mathsf{pfix}\ \mathit{size}.\lambda s.\text{if } (\mathit{isNil}\,s) \text{ then } 0 \text{ else } (1 + (2 * (\mathit{size}\ (\mathit{stl}\ s)))) : (\mathit{Seq}\,\alpha) \to \mathit{Int}}{\Gamma_0 \vdash \mathsf{pfix}\ \mathit{size}.\lambda s.\text{if } (\mathit{isNil}\,s) \text{ then } 0 \text{ else } (1 + (2 * (\mathit{size}\ (\mathit{stl}\ s)))) : \forall \alpha.(\mathit{Seq}\,\alpha) \to \mathit{Int}}\ \text{GEN}(\alpha)}\ \text{PFIX}(\alpha)$$

where

$$\begin{array}{lcl} \Gamma_0 & = & \{\mathit{isNil} : \forall \alpha.(\mathit{Seq}\,\alpha) \to \mathit{Bool},\ \ \mathit{stl} : \forall \alpha.\mathit{Seq}\,\alpha \to \mathit{Seq}\,(\alpha \times \alpha),\ \ +, * : \mathit{Int} \to \mathit{Int} \to \mathit{Int}\} \\ \Gamma_1 & = & \Gamma_0, \mathit{size} : \forall \alpha.(\mathit{Seq}\,\alpha) \to \mathit{Int} \\ \Gamma_2 & = & \Gamma_1, s : (\mathit{Seq}\,\alpha) \end{array}$$

# 6 Related Work

Comment on [10, 8, 4, 3] and also on Schwartzbach's excellent note [16]. Furthermore, on [5, 18, 17].

# References


[1] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order: Second Edition.* Cambridge University Press, 2002.

[2] Carl A. Gunter. *Semantics of Programming Languages: Programming Techniques*. The MIT Press, Cambridge, Massachusetts, 1992.

[3] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[4] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[5] Mark P. Jones. Dictionary-free overloading by partial evaluation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Orlando, Florida, June 1994.

[6] Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, England, 1994.

[7] Irving Kaplansky. *Set Theory and Metric Spaces*. Chelsea, New York, 1977.

[8] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, April 1993.

[9] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[10] Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *Proceedings of the International Symposium on Programming*, volume 167 of *LNCS*, pages 217–228, Toulouse, France, April 1984. Springer.

[11] Atsushi Ohori. A Simple Semantics for ML Polymorphism. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture, Imperial College, London*, pages 281–292, September 1989.

[12] Atsushi Ohori. *A Study of Semantics, Types, and Languages for Databases and Object-oriented Programming*. PhD thesis, University of Pennsylvania, 1989.

[13] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[14] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

[15] David A. Schmidt. *Denotational Semantics*. Allyn and Bacon, Boston, 1986.

[16] Michael I. Schwartzbach. Polymorphic type inference. Technical Report BRICS-LS-95-3, BRICS: Basic Research in Computer Science, Aarhus, Denmark, June 1995.

[17] P. J. Stuckey and M. Sulzmann. A theory of overloading. In *Proceedings of the International Conference on Functional Programming (ICFP'02)*, pages 167–178. ACM Press, 2002.

[18] Satish R. Thatte. Semantics of type classes revisited. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 208–219. ACM Press, 1994.

[19] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, Austin, Texas, 1989.