A Simple Semantics for Polymorphic Recursion*

William L. Harrison

Dept. of Computer Science, University of Missouri,
Columbia, Missouri, USA

Abstract. Polymorphic recursion is a useful extension of Hindley-Milner typing and has been incorporated in the functional programming language Haskell. It allows the expression of efficient algorithms that take advantage of non-uniform data structures and provides key support for generic programming. However, polymorphic recursion is, perhaps, not as broadly understood as it could be and this, in part, motivates the denotational semantics presented here. The semantics reported here also contributes an essential building block to any semantics of Haskell: a model for first-order polymorphic recursion. Furthermore, Haskell-style type classes may be described within this semantic framework in a straightforward and intuitively appealing manner.

1 Introduction

This paper presents a denotational semantics for an extension to the Hindley-Milner type system [18] called *polymorphic recursion* [21,16,8,1]. Polymorphic recursion (sometimes called *non-uniform* recursion) allows functions in which the type of a recursive call differs from that of the function itself. The approach taken here conservatively extends the semantics for Hindley-Milner polymorphism due to Ohori [22,23]. Ohori's semantics—the *simple model of ML polymorphism*—proceeds by construction from any frame model [20] of the simply-typed λ-calculus. The principal technical result of this work is that any such Ohori model may be extended to a model of first-order polymorphic recursion in a straightforward and natural manner. A compelling prospect for this approach is as a model for ad hoc polymorphism [29,13,15]. We describe how the simple model of polymorphic recursion provides a foundation for Haskell-style type classes and how the approach relates to a previous denotational model [28].

Polymorphic recursion is part of the functional language Haskell [25] and is a building block for generic programming [11,10] as well. Its presence in Haskell allows the straightforward expression of efficient data structures and algorithms; consider this example (adapted from Okasaki [24], page 142):

$$\begin{array}{ll} \textbf{data}\ \mathit{Seq}\ a\ =\ \mathit{Nil} \mid \mathit{SCons}\ a\ (\mathit{Seq}(a,a)) & \mathit{stl}\ ::\ \mathit{Seq}\ a \rightarrow \mathit{Seq}(a,a) \\ \mathit{size}\ \ \ :: \mathit{Seq}\ a \rightarrow \mathit{Int} & \mathit{stl}\ (\mathit{SCons}\ \_\ t)\ = t \\ \mathit{size}\ s\ = \textbf{if}\ \mathit{isNil}\ s\ \textbf{then}\ 0\ \textbf{else}\ 1 + 2 * \mathit{size}\ (\mathit{stl}\ s) & \end{array}$$

* This research supported in part by subcontract GPACS0016, System Information Assurance II, through OGI/Oregon Health & Sciences University.

K. Yi (Ed.): APLAS 2005, LNCS 3780, pp. 37–51, 2005.

The *Seq* data type represents sequences compactly so as to render their counting more efficiently than in a standard list representation. The *size* function calculates the length of a sequence and runs in $O(log\ n)$ time while its list analogue, *length*, runs in $O(n)$ time. Here, *Seq* only encodes sequences of length 2^n-1 for some n, although it is simple to extend its definition to capture sequences of arbitrary length by including an "even" constructor of type $ECons :: (Seq\,(a,a)) \rightarrow Seq\ a$; see, for example, Okasaki [24], page 144, for further details. Note that the recursive call to *size* occurs at an *instance* of its declared type: $Seq\,(a,a) \rightarrow Int$. Haskell requires explicit type signatures for polymorphic recursive definitions as type inference is undecidable in the presence of polymorphic recursion [8].

It has been recognized since its inception that the Hindley-Milner type system is more restrictive than is desirable as the following example (due to Milner [18]) makes clear. Consider the following standard ML definitions:

$$\textbf{fun}\ f\ x\ =\ x;\ \textbf{fun}\ g\ y\ =\ f\ 3; \qquad\qquad \textbf{fun}\ f\ x\ =\ x\ \textbf{and}\ g\ y\ =\ f\ 3;$$

When f is defined independently of g (left), it has type $\forall a.a \rightarrow a$ as one might expect, but when defined mutually (right), it has, unexpectedly, the less general type $int \rightarrow int$. Polymorphic recursion was initially developed [21] to overcome such "anomalies" in Hindley-Milner typing. Polymorphic recursive type systems maintain the type signatures of recursive definitions in universally quantified form to avoid such unintended typings. More will be said about this in Section 2.

Consider the polymorphic (but not polymorphic recursive) function *length*:

$$\begin{array}{l} length :: [a] \rightarrow Int \\ length = \lambda\, x.\, \textbf{if}\ null\ x\ \textbf{then}\ 0\ \textbf{else}\ 1{+}(length\ (tail\ x)) \end{array}$$

An Ohori-style semantics denotes *length* by the collection of its meanings at ground instances: $\{\ \langle \tau,\ len_\tau \rangle \mid \tau\ =\ [Int] \rightarrow Int, \ldots\ \}$ where each len_τ is defined:

$$len_\tau\ = fix\,(\lambda l.\ [\![\lambda x.\ \textbf{if}\ null\ x\ \textbf{then}\ 0\ \textbf{else}\ 1{+}(length\ (tail\ x))]\!]\rho[length \mapsto l])$$

Here, *fix* is the least fixed point operator defined conventionally on a domain D_τ denoting τ. One is tempted to try a similar definition for the polymorphic recursive function *size*:

$$size_\tau\ = fix\ (\lambda s.\ [\![\lambda x.\ \textbf{if}\ isNil\ x\ \textbf{then}\ 0\ \textbf{else}\ 1 + 2 * size\ (stl\ x))]\!]\rho[size \mapsto s]) \quad (\dagger)$$

But, where does this *fix* "live"? Or, in other words, over which domain is it defined? Intuitively, here's the problem: if the "input *size*" (i.e., the "s" in "*fix* $(\lambda s \ldots$") lives in D_τ for $\tau\ =\ Seq(\tau') \rightarrow Int$, then the "output *size*" (i.e., the "*size*" in "*size*(*stl* x)") lives in the domain for $\tau\ =\ Seq(\tau', \tau') \rightarrow Int$. The above definition does not appear to make sense in any fixed D_τ as was the case with *length*. We answer this question precisely in Section 3.2 below.

The Girard-Reynolds calculus (also known as *System F* [2] and the *polymorphic λ-calculus* [26]) allows lambda abstraction and application over types as well as over values; as such, it is sometimes referred to as a second-order

λ-calculus. Denotational models of second-order λ-calculi exist (e.g., the PER model described in [2]). Such models provide one technique for specifying ML polymorphism[1]. Harper and Mitchell take this approach [5,19] for the core of Standard ML called core-ML. They translate a core-ML term (i.e., one without type abstraction or application) into a second-order core-XML term (i.e., one with type abstraction and application). A core-ML term is then modeled by the denotation of its translation in an appropriate model of core-XML.

ML polymorphism is considerably more restrictive than that of a second-order λ-calculus; type quantification and lambda abstraction occur only over base types and values. Because of its restrictiveness relative to the Girard-Reynolds calculus, it is possible to give a predicative semantics to ML polymorphism [23,22] that is a conservative extension of the frame semantics of the simply-typed λ-calculus. Ohori's model of ML polymorphism is appealing precisely because of its simplicity. It explains ML polymorphism in terms of simpler, less expressive things (such as the frame semantics of the simply-typed λ-calculus) rather than in terms of inherently richer and more expressive things (such as the semantics of the second-order λ-calculus). The model of polymorphic recursion presented here retains this simplicity.

The rest of this paper proceeds as follows. Section 2 presents the language for first-order polymorphic recursion; the semantics of this language is then formulated in Section 3. Section 3 begins with an overview of frame semantics and Section 3.1 presents the simple model of ML polymorphism. Sections 3.2 and 3.3 demonstrate how a straightforward extension of Ohori's model provides a semantic foundation for first-order polymorphic recursion. Section 4 motivates the application of this semantic framework to Haskell-style ad hoc polymorphism. Section 5 discusses the rôle of the present research in the semantics of the Haskell language and outlines future directions. Related work is discussed throughout rather than in a separate section.

2 A Language and Type System for Polymorphic Recursion

The language we consider—called PR hereafter—is shown in Figure 1; it is the first-order type system referred to as ML/1′ by Kfoury, et al. [16]. Kfoury, et al. describe several type systems of increasing expressiveness that extend Hindley-Milner with polymorphic recursion. PR is "syntax-oriented," meaning that derivations in this type system are unique (modulo the order of application of the type generalization rule GEN); this has a useful (albeit not strictly necessary) virtue w.r.t. the coherence of the semantic equations.

The type language for PR (following Kfoury, et al. [16]) is stratified into "open" and "universal" types ($\mathbf{T}_0$ and $\mathbf{T}_1$, respectively) and, following Ohori

[1] Following Ohori [23,22], we shall refer to the first-order variety of polymorphism occurring in Haskell and ML as *ML polymorphism* as both languages use varieties of Hindley-Milner polymorphism [9,18].

$$
\begin{array}{lll}
\text{simple types} & \tau \in \mathsf{Type} & \tau ::= b \mid (\tau \to \tau') \\
\text{open types} & \gamma \in \mathbf{T}_0 & \gamma ::= \alpha \mid (\gamma \to \gamma') \mid \tau \\
\text{universal types} & \sigma \in \mathbf{T}_1 & \sigma ::= \forall\, \alpha.\, \sigma \mid \gamma \\
\text{expressions} & & M ::= x \mid (M\ N) \mid (\lambda\, x.\, M) \mid \text{let } x = N \text{ in } M \mid \text{pfix } x.M
\end{array}
$$

$$
\text{GEN}\ \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall \alpha.\sigma}\ (\alpha \notin FV(\Gamma)) \qquad \text{APP}\ \frac{\Gamma \vdash M : \gamma \to \gamma' \quad \Gamma \vdash N : \gamma}{\Gamma \vdash (M\ N) : \gamma'}
$$

$$
\text{ABS}\ \frac{\Gamma, x : \gamma \vdash M : \gamma'}{\Gamma \vdash (\lambda x.M) : \gamma \to \gamma'} \qquad \text{LET}\ \frac{\Gamma \vdash N : \sigma' \quad \Gamma, x : \sigma' \vdash M : \sigma}{\Gamma \vdash (\text{let } x = N \text{ in } M) : \sigma}
$$

$$
\text{VAR}\ \frac{\Gamma(x) = \sigma, \sigma \preceq_s \gamma}{\Gamma \vdash x : \gamma} \qquad \text{PFIX}\ \frac{\Gamma, x : \forall \bar{\alpha}.\gamma' \vdash M : \gamma' \quad \forall \bar{\alpha}.\gamma' \preceq \gamma}{\Gamma \vdash \text{pfix } x.M : \gamma}\ (\bar{\alpha} \notin FV(\Gamma))
$$

Fig. 1. The type system and expression syntax of PR. The type language of PR is stratified into *simple*, *open*, and *universal* types. PR departs from Hindley-Milner in its VAR and PFIX rules and polymorphic recursion is manifested in the PFIX rule.

[23,22], includes the set of "simple" (i.e., ground) types Type. Variables τ, γ, and σ refer (whether subscripted or not) to members of Type, $\mathbf{T}_0$, and $\mathbf{T}_1$, respectively, throughout this article. The abstract syntax for PR includes an explicit fix-point operator for polymorphic recursion called "pfix".

We will sometimes write a universal type $\sigma = \forall\alpha_1 \ldots \forall\alpha_n.\gamma$ for $0 \leq n$ as $\forall\bar{\alpha}.\gamma$ where $\bar{\alpha}$ is a (possibly) empty set of type variables. The relation $\sigma \preceq \gamma$ holds when γ is an instance of σ. Formally, open type γ is an *instantiation* of $\forall\bar{\alpha}.\gamma'$ for substitution s (written $(\forall\bar{\alpha}.\gamma') \preceq_s \gamma$), if, and only if, for some open types $\gamma_1 \cdots \gamma_n$ in $\mathbf{T}_0$, $s = [\alpha_1 \mapsto \gamma_1, \ldots, \alpha_n \mapsto \gamma_n]$ and $\gamma = s\gamma'$.

The rules GEN, APP, and ABS are standard and require no further comment. The LET rule captures let-polymorphism in the usual way, although it has a slightly different form than one might expect. Both the let term in the conclusion of the rule and the M term in its hypothesis have universal type σ where one would typically find an open type γ. This is inherited from ML/1′ and its effect is merely superficial [16]; one could change the universal σ to an open γ without affecting the language defined by the type system.

Polymorphic recursion arises in PR because, in the antecedent of the PFIX rule, the type binding for the recursively defined variable, $x : \forall\bar{\alpha}.\gamma'$, contains quantifiers if $\bar{\alpha} \neq \emptyset$. Consequently, x can be applied at any instance of its type (i.e., at any γ_i such that $\forall\bar{\alpha}.\gamma' \preceq \gamma_i$). The rule allowing such instantiations is VAR. VAR performs both variable look-up and instantiation—rôles that would typically be performed by two separate rules. VAR looks up a variable binding, $x : \sigma$, in the type environment Γ and its conclusion types the variable at an instantiation of that binding, $x : \gamma$.

Figure 2 presents the outline of a type derivation for *size* in PR illustrating how PR accommodates polymorphic recursion. Assume that Γ_0 contains bindings

$$\dfrac{\dfrac{\dfrac{(\ddagger)}{\Gamma_1 \vdash \lambda s.\mathbf{if}\ \mathit{isNil}\, s\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + 2 * \mathit{size}(\mathit{stl}\ s) : \mathit{Seq}(\alpha) \to \mathit{Int}}\ \text{ABS}}{\Gamma_0 \vdash \text{pfix}\ \mathit{size}.\lambda s.\mathbf{if}\ \mathit{isNil}\, s\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + 2 * \mathit{size}(\mathit{stl}\ s) : \mathit{Seq}(\alpha) \to \mathit{Int}}\ \text{PFIX}(\alpha)}{\Gamma_0 \vdash \text{pfix}\ \mathit{size}.\lambda s.\mathbf{if}\ \mathit{isNil}\, s\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + 2 * \mathit{size}(\mathit{stl}\ s) : \forall\alpha.\mathit{Seq}(\alpha) \to \mathit{Int}}\ \text{GEN}(\alpha)$$

$$\dfrac{\dfrac{\begin{array}{l}\Gamma_2(\mathit{size}) = \forall\alpha.(\mathit{Seq}\,\alpha) \to \mathit{Int} \\ (\forall\alpha.(\mathit{Seq}\,\alpha) \to \mathit{Int}) \preceq_s \mathit{Seq}(\alpha \times \alpha) \to \mathit{Int} \\ s = [\alpha \mapsto \alpha \times \alpha]\end{array}}{\Gamma_2 \vdash \mathit{size} : \mathit{Seq}(\alpha \times \alpha) \to \mathit{Int}}\ \text{VAR} \quad \dfrac{\dfrac{}{\ldots}}{\Gamma_2 \vdash (\mathit{stl}\ s) : \mathit{Seq}\,(\alpha \times \alpha)}}{\Gamma_2 \vdash \mathit{size}\ (\mathit{stl}\ s) : \mathit{Int}}$$

Fig. 2. Type Checking *size* in PR. The type derivation for *size* in PR appears in the upper derivation. The second derivation (below) occurs within the "(‡)" above. Certain routine parts are suppressed for the sake of readability.

for arithmetic functions, *stl*, *isNil*, *SCons* and *Nil*, etc., Γ_1 extends Γ_0 with the binding $\mathit{size} : \forall\alpha.\,(\mathit{Seq}\,\alpha) \to \mathit{Int}$, and Γ_2 extends Γ_1 with $s : (\mathit{Seq}\,\alpha)$. It is at the leaves of the derivation in the bottom half of Figure 2 that this derivation becomes interesting; the application of the VAR rule instantiates *size*, not at its defined type, $\mathit{Seq}(\alpha) \to \mathit{Int}$, but rather at the instance $\mathit{Seq}(\alpha \times \alpha) \to \mathit{Int}$. This is precisely where polymorphic recursion manifests itself within *size*.

3 The Simple Model of Polymorphic Recursion

This section presents the semantics of PR. First, we briefly overview the frame semantics of the simply-typed λ-calculus and then, in Section 3.1, outline how this semantics is extended in the simple model of ML polymorphism. Section 3.2 describes the semantic setting where polymorphic recursive fixed-point equations are solved; Section 3.3 describes the semantics of PR in this setting.

Background on Type-frames Semantics. One may think of a frame model as set-theoretic version of a cartesian closed category. That is, it provides "objects" (i.e., D_τ for each simple type τ) and axioms of representability and extensionality characterizing functions from objects to objects in terms of an application operator $\bullet$. In this article, each simple type model D_τ is presumed to be built from sets with additional structure and we write $|D_\tau|$ for the underlying set of D_τ. We refer to D_τ as a *frame object* and to $|D_\tau|$ as its *frame set*. We assume that each $|D_\tau| \neq \emptyset$. A *frame* is a pair $\langle \mathcal{D}, \bullet \rangle$ where

1. $\mathcal{D} = \{ D_\tau \mid \tau \in \mathsf{Type} \}$ and $\mathcal{D} \neq \emptyset$
2. $\bullet$ is a family of operations $\bullet_{\tau_1\tau_2} \in |D_{(\tau_1 \to \tau_2)}| \to |D_{\tau_1}| \to |D_{\tau_2}|$

The set function $\phi : |D_{\tau_1}| \to |D_{\tau_2}|$ is *representable* in $|D_{(\tau_1 \to \tau_2)}|$ if

$$\exists\, f \in |D_{(\tau_1 \to \tau_2)}| \text{ s.t. } \phi(d) = f \bullet_{\tau_1\tau_2} d, \quad \forall d \in |D_{\tau_1}|$$

The frame $\langle \mathcal{D}, \bullet \rangle$ is *extensional* if, for all $f, g \in |D_{(\tau_1 \to \tau_2)}|$,

$$\text{if } f \bullet_{\tau_1 \tau_2} d = g \bullet_{\tau_1 \tau_2} d \text{ for all } d \in |D_{\tau_1}|, \text{ then } f = g$$

A value environment ρ is *compatible* with a type environment $\mathcal{A}$ when

$$\mathsf{dom}(\rho) = \mathsf{dom}(\mathcal{A}), \text{ and } \rho\, x \in |D_\tau| \text{ iff } (x : \tau) \in \mathcal{A}$$

The compatibility relation is designated by $\mathcal{A} \models \rho$. The set of value environments compatible with $\mathcal{A}$ is designated $\mathrm{Env}(\mathcal{A})$. Let $\langle \mathcal{D}, \bullet \rangle$ be any frame, $\lambda^{\to}$ be the simply-typed λ-calculus and ρ a value environment such that $\mathcal{A} \models \rho$. Then, the map $\mathcal{D}[\![-]\!] \in \lambda^{\to} \to Env \to (\bigcup |D_\tau|)$ obeys the *environment model condition* if the following equations hold:

$$\begin{array}{ll}
\mathcal{D}[\![\mathcal{A} \vdash x : \tau]\!]\rho & = \rho\ x \\
\mathcal{D}[\![\mathcal{A} \vdash (M\ N) : \tau]\!]\rho & = (\mathcal{D}[\![\mathcal{A} \vdash M : \tau' \to \tau]\!]\rho) \bullet (\mathcal{D}[\![\mathcal{A} \vdash N : \tau']\!]\rho) \\
\mathcal{D}[\![\mathcal{A} \vdash \lambda x : \tau_1 . M : \tau_1 \to \tau_2]\!]\rho = & \text{ the } f \in |D_{\tau_1 \to \tau_2}| \text{ s.t. for all } d \in |D_{\tau_1}|, \\
 & \qquad f \bullet d = \mathcal{D}[\![\mathcal{A}, x : \tau_1 \vdash M : \tau_2]\!]\rho[x \mapsto d]
\end{array}$$

For any extensional frame $\mathcal{D}$, the above equations induce a model of the simply-typed λ-calculus [3,20].

3.1 The Simple Model of ML Polymorphism

The simple model of ML polymorphism [23,22] defines the meaning of a ML-polymorphic expression in terms of type-indexed sets of denotations of its ground instances. It conservatively extends the type-frames semantics of the simply-typed λ-calculus [3,20] to accommodate polymorphism. It is a typed semantics, meaning that the denotations are given for derivable typing judgments of terms.

Ohori's model is quite intuitive. Consider the term $(\vdash \lambda x.x : \forall\alpha.\alpha \to \alpha)$. Any ground instance of this term (e.g., $\vdash \lambda x.x : Int \to Int$) has a meaning within a frame object (i.e., $D_{Int \to Int}$) of an appropriate frame $\mathcal{D}$. If the elements of $|D_{\tau \to \tau'}|$ are actually functions from $|D_\tau|$ to $|D_{\tau'}|$, then each of these ground instances is simply the identity function $id_{D_\tau} \in |D_{\tau \to \tau}|$. Accordingly, the meaning of $(\vdash \lambda x.x : \forall\alpha.\alpha \to \alpha)$ is just the set: $\{(\tau \to \tau, id_{D_\tau}) \mid \tau \in \mathsf{Type}\}$. We use product notation for the collection of these type-indexed sets: $\Pi\, \tau \in S.|D_\tau|$, each element of which is assumed to be a set function.

Defining an Ohori Model. This example illustrates the structure of Ohori's model: a semantics for the ground typings of an ML-polymorphic language may be extended conservatively to the full language; that is, a polymorphic term $(\Gamma \vdash e : \sigma)$ is defined by collecting the type-indexed denotations of its ground instances $\langle \tau, [\![\mathcal{A} \vdash e : \tau]\!] \rangle$. To accomplish this extension, one additional bit of machinery is necessary to calculate the ground instances of universal types, typing environments, and derivations. This is the subject of the remainder of this section.

We refer to a universal type σ containing no free type variables as a *closed type*. A typing environment Γ is *closed* when the type within each binding $(x : \sigma)$ is closed. A judgment $\Gamma \vdash M : \sigma$ is *closed* when Γ and σ are closed. A type

derivation Δ is *closed* when the judgment at its root is closed. N.B., Δ being closed does not imply that its subderivations are closed. Given a substitution $\eta : \{\alpha_1, \ldots, \alpha_n\} \to \mathbf{T}_0$, its canonical extension η^* to a map from $\mathbf{T}_1$ to $\mathbf{T}_1$ is:

$$\begin{array}{ll} \eta^*\tau = \tau & \eta^*\alpha = \begin{cases} \eta\alpha & \text{if } \alpha \in \mathrm{dom}(\eta) \\ \alpha & \text{otherwise} \end{cases} \\ \eta^*(\gamma \to \gamma) = (\eta^*\gamma) \to (\eta^*\gamma) & \eta^*(\forall\beta.\sigma) = \forall\beta.(\eta_0^*\sigma) \text{ where } \eta_0 = \eta \setminus \beta \end{array}$$

Furthermore, we apply η^* to type environments and derivations as well:

$$\begin{array}{ll} \eta^*(x : \sigma, \Gamma) & = (x : \eta^*\sigma), \eta^*\Gamma \\ \eta^*\{\} & = \{\} \\ \eta^*(\frac{\Delta}{\Gamma \vdash M : \forall\bar{\alpha}.\gamma}) & = \frac{\eta_0^*\Delta}{(\eta_0^*\Gamma) \vdash M : \forall\bar{\alpha}.(\eta_0^*\gamma)} \text{ where } \eta_0 = \eta \setminus \bar{\alpha} \end{array}$$

First, we define the set of ground substitutions on a universal type:

$$\mathsf{Gr}(\sigma) \triangleq \{\eta \mid \eta : FV(\sigma) \to \mathsf{Type}\}$$

For a closed universal type $\sigma = (\forall\bar{\alpha}.\gamma)$ with (possibly empty) set of quantified variables $\bar{\alpha}$, the set of its *ground instances*, σ^{Type}, is:

$$\sigma^{\mathsf{Type}} \triangleq \{\eta^*\gamma \mid \ \eta \in \mathsf{Gr}(\gamma)\}$$

3.2 Solving Polymorphic-Recursive Equations

Frame models for the simply-typed lambda calculus consist of only data necessary to model application and abstraction. This data suffices for the simple model of ML polymorphism as the language modeled there does not include recursion [22]. Being recursive, the PR language requires more structure to model with frames and we extend the notion of type frame to accommodate recursion. In particular, the notion of a type frame is extended with structure including a partial order on the elements of frame sets, pointedness of frame objects, and continuous functions that preserve order and limits. This same methodology has been employed to specify Haskell's mixture of lazy and eager evaluation [7].

We return to the question posed in Section 1 at (†): how—or rather, *where*—do we solve equations like:

$$size \ = \ \lambda x.\, \mathbf{if}\ isNil\ x\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + 2 * size\ (stl\ x)$$

As it turns out, if frame $\mathcal{D}$ is such that its frame objects, D_τ, are pointed cpos, then the denotations of polymorphic types in Ohori's model (i.e., the indexed sets $(\Pi\,\tau \in S.|D_\tau|)$) may be extended to pointed cpos[2] as well. This idea is made formal in Theorem 1 below. Within these new cpo structures, we can find appropriate solutions to polymorphic-recursive equations and define the semantics of PR (as we do below in Section 3.3).

[2] Technically, this is an ω-cpo; that is, every ascending chain possesses a lub. For a cpo, every directed set has a lub. N.B., every cpo is an ω-cpo, but not vice-versa.

Frames for Polymorphic Recursion. We introduce the following terminology for such frames with a pointed cpo structure: A *pcpo frame* is a frame $\langle \mathcal{D}, \bullet, \sqsubseteq, \sqcup, \bot \rangle$ in which each $D_\tau \in \mathcal{D}$ is a pointed, complete, partial order w.r.t. $\bot_\tau$, $\sqsubseteq_\tau$, and $\sqcup_\tau$. For a given pcpo frame $\mathcal{D}$, a type-indexed set $(\Pi\, \tau \in S.|D_\tau|)$ is uniquely determined by the set of types S, and accordingly we introduce the abbreviation $P_S \triangleq (\Pi\, \tau \in S.|D_\tau|)$. If S is the singleton set $\{\tau\}$, we write P_τ. A family of application operators $\bullet$ is defined for the P_τ in the obvious manner in terms of the $\bullet$ operators in $\mathcal{D}$. Theorem 1 demonstrates that the collection of P_S are the frame objects in a pcpo frame; we will refer to this frame hereafter as $\mathcal{P}$. The language PR is defined in the frame $\mathcal{P}$ below in Section 3.3.

Theorem 1. *Let $\langle \mathcal{D}, \bullet, \sqsubseteq, \sqcup, \bot \rangle$ be a pcpo-frame and $S \subseteq \mathsf{Type}$ be a set of ground type expressions. Then, P_S is a pointed cpo where:*

- *for any $f, g \in P_S$, $f \sqsubseteq_S g \Leftrightarrow$ for all $\tau \in S$, $f\tau \sqsubseteq_\tau g\tau$, and*
- *the bottom element is $\bot_S \triangleq \{\langle \tau, \bot_\tau \rangle \mid \tau \in S\}$*
- *Let $\sqcup_\tau$ be the lub operator within the $\mathcal{D}$ frame object D_τ. Then the least upper bound of an ascending chain $\{X_i\} \subseteq |P_S|$ is:*

$$\bigsqcup_S X_i \triangleq \{\langle \tau, u_\tau \rangle \mid \tau \in S, u_\tau = \bigsqcup_\tau (X_i \tau)\}$$

Proof. To show: $\sqsubseteq_S$ and $\bot_S$ define a pointed, complete partial order on P_S. Assume $X, X_i \in P_S$. That $\sqsubseteq_S$ is reflexive, anti-symmetric, and transitive follows directly from the fact the each $\sqsubseteq_\tau$ is so. Similarly, $\bot_S$ is the least element of P_S because, for any $\tau \in \mathsf{Type}$, so is $\bot_S\, \tau = \bot_\tau$. Let $X_0 \sqsubseteq_S X_1 \sqsubseteq_S \cdots$ be a directed chain in P_S, then it remains to show that $\{\langle \tau, u_\tau \rangle \mid \tau \in S, u_\tau = \bigsqcup_\tau (X_i \tau)\}$ is the lub of $\{X_i\}$. Define $U = \{\langle \tau, u_\tau \rangle \mid \tau \in S, u_\tau = \bigsqcup_\tau (X_i \tau)\}$. It is clear from the definition of U and $\sqsubseteq_S$ that U is in P_S and is an upper limit of $\{X_i\}$. Suppose that $V \in P_S$ is an upper bound of $\{X_i\}$. Then, $X_i \tau \sqsubseteq_\tau V\tau$ for every type $\tau \in \mathsf{Type}$. $\therefore \bigsqcup_\tau (X_i \tau) \sqsubseteq_\tau V\tau$ for every type $\tau \in \mathsf{Type}$. By the definition of U, $U\tau \sqsubseteq_\tau V\tau$ for every type $\tau \in \mathsf{Type}$ and, hence, $U \sqsubseteq_S V$. $\therefore U$ is the least such upper limit, justifying the definition of $\bigsqcup_S X_i \triangleq U$. □

Because P_S is a pointed cpo, we may define continuous functions and least fixed points over it in the standard way [3]. That is, a function $f : P_S \to P_T$ is *continuous* when $f(\bigsqcup_S X_i) = \bigsqcup_T (f\, X_i)$ for all directed chains $\{X_i\}$ in P_S. Then, $\mathit{fix}(f) = \bigsqcup_S (f^n \bot_S)$ for $n < \omega$ given a continuous endofunction $f : P_S \to P_S$. We assume that every continuous function between frame objects in frame $\mathcal{P}$ is representable in $\mathcal{P}$ and that $\mathcal{P}$ is extensional.

3.3 The Frame Semantics of PR

What remains to be seen is the definition of the PR language in terms of the pcpo frame $\mathcal{P}$. First, we consider the denotation of universal types and then the semantics of PR is given.

Denotations of Types. A closed universal type is denoted by the type-indexed set of its ground instances: $[\![\sigma]\!] = \Pi\, \tau \in (\sigma^{\mathsf{Type}}).|D_\tau|$ for closed σ.

Denotations of Terms. The denotation of PR is defined on term derivations. Recall that PR is syntax-oriented; the coherence of the semantic equations follows because the derivations are unique (modulo the order of application of the **GEN** rule). Let Δ be a derivation of $(\Gamma \vdash e : \sigma)$ and $\eta \in \mathsf{Gr}(\sigma)$, then the signature of the semantics is $[\![\Delta]\!]\eta : \mathsf{Env}(\eta^*\Gamma) \to [\![\eta^*\sigma]\!]$. N.B., $\eta^*\Gamma$ is a closed type assignment, and $\mathsf{Env}(\eta^*\Gamma)$ are the environments compatible with that closed type assignment. The semantic equations are defined by induction on derivation trees; this is the subject of the remainder of this section.

GEN. Let $\sigma = \forall\alpha_1 \ldots \forall\alpha_n.\gamma$, $\eta \in \mathsf{Gr}(\sigma)$, and $\rho \in \mathsf{Env}(\eta^*\Gamma)$. Let $\oplus$ be right-biased environment extension:

$$(\eta \oplus \eta')\alpha = \begin{cases} \eta'\alpha & \text{if } \alpha \in \mathsf{dom}(\eta') \\ \eta\alpha & \text{otherwise} \end{cases}$$

Note that $S \triangleq \sigma^{\mathsf{Type}}$ may be written in terms of extensions to η as:

$$S = \{\tau \in \mathsf{Type} \mid \eta_\tau : \{\alpha_1, \ldots, \alpha_n\} \to \mathsf{Type},\ \tau = (\eta \oplus \eta_\tau)^*\gamma\}$$

and, furthermore for any $\tau \in S$, the extension η_τ such that $\tau = (\eta \oplus \eta_\tau)^*\gamma$ is unique. We may therefore define the semantics of a GEN rule application as:

$$[\![\Gamma \vdash e : \sigma]\!]\eta\rho = \bigcup_{\tau \in S} \left([\![\Gamma \vdash e : \gamma]\!](\eta \oplus \eta_\tau)\rho \right) \tag{1}$$

VAR. An auxiliary function, $\iota_\tau\, x = \{\langle \tau, x\rangle\}$, injects D_τ into P_τ. N.B., that if $x \in |D_\tau|$, then $\iota_\tau\, x \subseteq (\Pi\, \tau \in S.|D_\tau|)$ for any $S \subseteq \mathsf{Type}$ such that $\tau \in S$. The binding of a variable x in ρ is a type-indexed set and the denotation of x is a component of this set:

$$[\![\Gamma \vdash x : \gamma]\!]\eta\rho = \iota_\tau(\rho\, x\, \tau), \text{ where } \tau = \eta^*\gamma \tag{2}$$

The definition in (2) determines the correct component of $[\![\forall\alpha.\mathit{Seq}(\alpha) \to \mathit{Int}]\!]$, τ, from the ground substitution η and the open type γ. The *size* example illustrates why this definition works. Consider the VAR application in the example derivation at the end of Section 2; recall the judgment at the root is: $\Gamma_2 \vdash \mathit{size} : \mathit{Seq}(\alpha \times \alpha) \to \mathit{Int}$. Let $\eta \in \mathsf{Gr}(\mathit{Seq}(\alpha \times \alpha) \to \mathit{Int})$ and $\rho \in \mathsf{Env}(\eta^*\Gamma_2)$, and assume $\eta\alpha = \mathit{Bool}$. Note that $(\rho\, \mathit{size}) \in (\Pi\, \tau \in \sigma^{\mathsf{Type}}.|D_\tau|)$ for $\sigma = \forall\alpha.\mathit{Seq}(\alpha) \to \mathit{Int}$. The component of $(\rho\, \mathit{size})$ denoting x is found at the ground type $\eta^*(\mathit{Seq}(\alpha \times \alpha) \to \mathit{Int}) = \mathit{Seq}(\mathit{Bool} \times \mathit{Bool}) \to \mathit{Int}$.

PFIX. Assuming $\eta \in \mathsf{Gr}(\gamma)$ and $\rho \in \mathsf{Env}(\eta^*\Gamma)$, let S and $f_S : P_S \to P_S$ be the following set and function, respectively:

$$\begin{aligned} S &= \{\tau \in \mathsf{Type} \mid \tau = \eta^*\gamma'\} \\ f_S &= d \mapsto [\![\Gamma, x : \forall\bar{\alpha}.\gamma' \vdash M : \gamma']\!]\eta(\rho[x \mapsto d]) \end{aligned}$$

By Lemma 1 (below), f_S is continuous, and we make the additional assumption that f_S is representable in $\mathcal{P}$. Then, polymorphic fixpoints are defined as:

$$[\![\Gamma \vdash \mathrm{pfix}\, x.M : \gamma]\!]\eta\rho = \bigsqcup\nolimits_S (f^n\ \bot_S) \text{ for } n < \omega \tag{3}$$

ABS. Let $\tau = \eta^*\gamma$ and $\tau' = \eta^*\gamma'$ and f be the function from $\mathcal{P}_\tau$ to $\mathcal{P}_{\tau'}$:

$$f = d \mapsto [\![\Gamma, x : \gamma \vdash M : \gamma']\!]\eta(\rho[x \mapsto d])$$

By Lemma 1 (below), f is continuous, and we assume that f is representable in $\mathcal{P}$. Then, lambda abstraction is defined as:

$$[\![\Gamma \vdash \lambda x.M : \gamma \to \gamma']\!]\eta\rho = f \tag{4}$$

APP. Let $\tau = \eta^*\gamma$, $\tau' = \eta^*\gamma'$. Then the semantics of application is simply:

$$[\![\Gamma \vdash MN : \gamma']\!]\eta\rho = ([\![\Gamma \vdash M : \gamma \to \gamma']\!]\eta\rho) \bullet_{\tau,\tau'} ([\![\Gamma \vdash N : \gamma]\!]\eta\rho) \tag{5}$$

N.B., the application operator, $\bullet_{\tau,\tau'}$, is from the objects $\mathcal{P}_{(\tau \to \tau')}$ and $\mathcal{P}_\tau$.

LET. The semantics of *let* is defined conventionally:

$$[\![\Gamma \vdash (\text{let } x = N \text{ in } M) : \sigma]\!]\eta\rho = [\![\Gamma, x : \sigma' \vdash M : \sigma]\!]\eta(\rho[x \mapsto [\![\Gamma \vdash N : \sigma']\!]\eta\rho]) \tag{6}$$

The semantic equations for PFIX and ABS rely on Lemma 1 which states that functions defined in terms of the semantic function are continuous. Lemma 1, or something very much like it, is part of any conventional cpo semantics of λ-calculi. Its proof follows along the lines of what one would find in a semantics textbook (e.g., see Gunter 1992, lemma 4.19, page 130).

Lemma 1. *For any closed term* $(\Gamma, x : \forall\bar{\alpha}.\gamma' \vdash M : \gamma)$*, the following function from* P_S *to* P_T *is continuous:* $f = (d \in |P_S|) \mapsto [\![\Gamma, x : \forall\bar{\alpha}.\gamma' \vdash M : \gamma]\!]\eta\rho[x \mapsto d])$.

Proof. By induction on M. Let $\Gamma' = \Gamma, x : \forall\bar{\alpha}.\gamma$. Each case is straightforward so we show only the case for $M = \text{pfix}y.\ M'$.

To show: f is monotonic. Let $e, e' \in |P_S|$ such that $e \sqsubseteq_S e'$. Define $d_i \in |P_S|$ for $i < \omega$ as $d_0 = \bot_S$ and $d_{i+1} = [\![\Gamma', y : \sigma \vdash M' : \gamma]\!]\eta\rho[y \mapsto d_i]$. Note that, by definition, $P_T = [\![\eta^*\gamma]\!]$. Then,

$$\begin{array}{lll}
[\![\Gamma' \vdash \text{pfix } y.M' : \gamma]\!]\eta\rho[x \mapsto e] & & \\
\quad = \bigsqcup_T([\![\Gamma', y : \sigma \vdash M' : \gamma]\!]\eta\rho[x \mapsto e][y \mapsto d_i]) & \{\text{defn.}\} \\
\quad \sqsubseteq_T \bigsqcup_T([\![\Gamma', y : \sigma \vdash M' : \gamma]\!]\eta\rho[x \mapsto e'][y \mapsto d_i]) & \{\text{ind. hyp.}\} \\
\quad = [\![\Gamma' \vdash \text{pfix } y.M' : \gamma]\!]\eta\rho[x \mapsto e'] & \{\text{defn.}\}
\end{array}$$

To show: f is preserves limits. Let $P_X = [\![\eta^*\forall\bar{\alpha}.\gamma']\!]$ and $x_i \in |P_X|$ be such that $x_i \sqsubseteq_X x_{i+1}$. Then,

$$\begin{array}{lll}
[\![\Gamma' \vdash \text{pfix } y.M' : \gamma]\!]\eta\rho[x \mapsto \sqcup_X x_i] & & \\
\quad = \bigsqcup_T([\![\Gamma', y : \sigma \vdash M' : \gamma]\!]\eta\rho[x \mapsto \sqcup_X x_i][y \mapsto d_i]) & \{\text{defn.}\} \\
\quad = \bigsqcup_T \bigsqcup_X([\![\Gamma', y : \sigma \vdash M' : \gamma]\!]\eta\rho[x \mapsto x_i][y \mapsto d_i]) & \{\text{ind. hyp.}\} \\
\quad = \bigsqcup_X \bigsqcup_T([\![\Gamma', y : \sigma \vdash M' : \gamma]\!]\eta\rho[x \mapsto x_i][y \mapsto d_i]) & \{\text{exchange}\} \\
\quad = \bigsqcup_X([\![\Gamma' \vdash \text{pfix } y.M' : \gamma]\!]\eta\rho[x \mapsto x_i]) & \{\text{defn.}\}
\end{array}$$

Here, {exchange} refers to what is frequently known as the *exchange lemma*. This states that, given a monotone function, $f : P \times Q \to D$, for directed sets P and Q and cpo D, that the ordering of the limits does not matter:

$$\bigsqcup_{x \in P} \bigsqcup_{y \in Q} f(x, y) = \bigsqcup_{y \in Q} \bigsqcup_{x \in P} f(x, y)$$

An exchange lemma holds for $\mathcal{P}$; its proof is exactly the same as that of (Gunter 1992, Lemma 4.9, page 118). □

4 Application to Haskell Type Classes

This work is part of an effort[3] to develop a formal basis for reasoning about the Haskell core language known as Haskell 98 [25]. The published work in this endeavor focuses on mixed evaluation (i.e., combined lazy and eager evaluation) in Haskell [7,6] and its semantic and logical consequences. The original interest in the simple model of ML polymorphism stems from the observation that its "type awareness" provides precisely what one needs to specify type classes in Haskell; the rest of this section presents a high-level overview of this insight. The initial presentation of polymorphic recursion [21] developed a denotational semantics based on the ideal model of recursive types [17]. Cousot [1] formulated a hierarchy of type systems—including Mycroft's—in terms of a lattice of abstract interpretations of the untyped lambda calculus. However, neither of these potential starting points—i.e., the ideal model or abstract interpretation—seem to fit quite as well to type classes as does the Ohori framework.

Type classes in Haskell are an overloading mechanism allowing variables to be interpreted differently at different types. The *Eq* class (shown in part below) defines equality (==); it is the primitive *eqInt* on instance ($Int \to Int \to Bool$). Equality on pairs of type (a, b) is inherited from equality on a and b instances; in the last instance declaration below, "x==u" and "y==v" are equality on types a and b, respectively:

```
class Eq a where
  (==) :: a → a → Bool
instance Eq Int where
  (==) = eqInt
instance (Eq a, Eq b) ⇒ Eq (a, b) where
  (x, y)==(u, v) = (x==u)&&(y==v)
```

[3] The *Programatica* project explores the use of Haskell 98 in formal methods. See `www.cse.ogi.edu/PacSoft/projects/programatica` for further details.

Polymorphic recursion first entered Haskell through its type class system when expert practitioners noticed that polymorphic recursive functions could be expressed via "kludges" such as the implementation of *size* below [12]:

$$
\begin{array}{l}
size = size'() \\
\textbf{class } Size\ d \textbf{ where} \\
\quad size' :: d \rightarrow Seq\ a \rightarrow Int \\
\textbf{instance } Size\ () \textbf{ where} \\
\quad size'\ p\ Nil \qquad\qquad = 0 \\
\quad size'\ p\ (SCons\ x\ xs) = 1 + 2 * size'\ p\ xs
\end{array}
$$

The method "$size'$" contains a dummy parameter but is otherwise identical to *size*. Because type inference is undecidable in the presence of polymorphic recursion [8], the type of $size'$ must be declared explicitly and the kludge uses the type declaration of the $size'$ method for the same purpose as the original explicit type declaration of *size*. The type of the dummy variable—d in the type signature for $size'$—is the overloaded type here. The Haskell type system considers the type variable a in the $size'$ method signature as quantified (analogously to the universally typed recursive binding of x in the PFIX rule). Creating a dummy instance of *Size* at unit type () allows the definition of the polymorphic recursive function *size* by simply applying $size'$ to the unit value ().

Without polymorphic recursion, overloading in Haskell 98 may be handled via partial evaluation [14]; that is, the finitely many class instances necessary within a Haskell program may be determined statically and inlined within the program itself. With polymorphic recursion, the number of such instances is potentially unbounded; consider the function $foo\ x\ =\ x{==}x\ \&\&\ foo\ (x, x)$ which will require an unbounded number of implementations of ==.

Within a suitable frame $\mathcal{P}$ from Section 3, however, a type class may be viewed as the set of its ground instances:

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

This denotation should appeal to the intuitions of Haskell programmers, because, according to this view, a type class is merely the type-indexed set of its instances—what would be called in Haskell terminology a *dictionary* [4]. It is important to note that, while potentially infinite dictionaries foil the use of partial evaluation as an implementation technique for type classes in general, they pose no problem for the denotational model for type classes outlined in this section.

Models and implementation techniques for type classes have been considered for many years now; a representative, albeit quite non-exhaustive, sampling of this work is [29,28,15,4,27]. All existing models of type classes have one thing in common—they are based on the translation of the source language into an intermediate form for which a model or implementation is known. Thatte [28] translates a first-order λ-calculus extended with a notation for classes and instances (called "OML") into a second-order lambda calculus with a special **typecase**

construct for examining type structure (called "OXML"). OXML is then given an interpreter and OML is defined indirectly via its translation into OXML. The *Eq* class example above would be translated as:

$$\begin{array}{ll}(==) & = \Lambda t.\, \textbf{typecase}\ t\ \textbf{of}\ \{\ \mathit{Int}\colon \mathit{eqInt}\ ;\ a{\times}b\colon \mathit{eqPair}\langle a\rangle\langle b\rangle\ \} \\ \mathit{eqPair} & = \Lambda a.\, \Lambda b.\, \lambda(x{:}a, y{:}b).\, \lambda(u{:}a, v{:}b).\, (==\ \langle a\rangle\ x\ u)\ \&\&\ (==\ \langle b\rangle\ y\ v)\end{array}$$

Here, Λ and $\langle - \rangle$ represent type abstraction and application in OXML, respectively. The translation of $(==)$ takes the type t at which it is applied and dynamically determines the appropriate method implementation. If $t = a \times b$, then this involves creating instances of $(==)$ at types a and b (as is done in the body of *eqPair*). Thatte's semantics is similar to the implementation of Haskell classes via the *dictionary-passing translation* (DPT) in which functions with overloaded types are translated to functions taking an additional dictionary parameter [29,15,4,27].

In contrast to existing models, the approach outlined in this section is direct: no intermediate translation of the source language is required. Furthermore, the "dictionary" denotations of classes in $\mathcal{P}$ require no dynamic examination of type structure. The denotation of *Eq* above, for example, contains all instances of *Eq*; no **typecase** construct or on-the-fly dictionary construction is required.

5 Conclusions and Future Directions

This article demonstrates an approach to modeling polymorphic recursion in functional languages. These models require a fundamental change to the substance of denotations rather than to their form and this shift is recorded in the construction of the $\mathcal{P}$ type frame (Theorem 1). Types may still be modeled by cpos and recursion by fixed point calculations as in traditional denotational semantics, but the underlying structure of those cpos changes to include type information. This "type awareness" accommodates the additional expressiveness required to model polymorphic recursion.

The Girard-Reynolds calculus uses abstraction over an (impredicative) universe of types. Hindley-Milner polymorphism is considerably more restrictive in that it only allows abstraction and type quantification over values and base types, respectively. Ohori's insight was that the relative restrictiveness of Hindley-Milner admits a predicative semantics; in fact, any frame model of the simply-typed lambda calculus may be conservatively extended to a model of Hindley-Milner. This article demonstrates that this extension may continue one step beyond Hindley-Milner to a predicative model of first-order polymorphic recursion and, furthermore, that this extension is straightforward (albeit non-trivial).

This work reports progress in an effort to establish a formal semantics for the entire Haskell 98 language starting from Ohori's simple model of ML polymorphism [23,22]. Extensions to the Ohori model have already been explored for characterizing Haskell's surprisingly complex mixed evaluation [7]. Obviously, any such Haskell semantics must account for polymorphic recursion for the simple reason that Haskell allows polymorphic recursive definitions. As was

described in Section 4, the type awareness of the $\mathcal{P}$ type frame suggests a natural denotational model of type classes, the precise details of which are left to a sequel.

Acknowledgements

The author wishes to thank Dick Kieburtz, Brian Dunphy, and Mark Jones for their encouragement and for numerous discussions on aspects of polymorphic recursion, its relationship to the semantics of Haskell, and feedback on this article.

References

1. P. Cousot. Types as abstract interpretations, invited paper. In *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331, Paris, France, January 1997. ACM Press, New York, NY.
2. Jean-Yves Girard. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.
3. Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.
4. 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.
5. Robert Harper and John C. Mitchell. On the type structure of standard ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):211–252, 1993.
6. William Harrison, Timothy Sheard, and James Hook. Fine control of demand in Haskell. In *6th International Conference on the Mathematics of Program Construction, Dagstuhl, Germany*, volume 2386 of *Lecture Notes in Computer Science*, pages 68–93. Springer-Verlag, 2002.
7. William L. Harrison and Richard B. Kieburtz. The logic of demand in Haskell. *Journal of Functional Programming*, 15(5), 2005.
8. Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.
9. Roger J. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.
10. Ralf Hinze. A new approach to generic functional programming. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–132, 2000.
11. Ralf Hinze and Johan Jeuring. Generic Haskell: practice and theory. In Roland Backhouse and Jeremy Gibbons, editors, *Generic Programming: Advanced Lectures*, number 2793 in Lecture Notes in Computer Science, pages 1–56. Springer-Verlag, 2003.
12. Mark Jones. Private communication.
13. Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 52–61. ACM Press, 1993.

14. 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.
15. Mark P. Jones. *Qualified Types: Theory and Practice*. Cambridge University Press, Cambridge, England, 1994.
16. 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.
17. D. B. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2), 1984.
18. Robin Milner. A theory of type polymorphism in programming languages. *Journal of Computer and System Science*, 17(3):348–375, 1978.
19. J. C. Mitchell and R. Harper. The essence of ML. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages (POPL 1988)*, pages 28–46, 1988.
20. John C. Mitchell. *Foundations for Programming Languages*. MIT Press, Cambridge, MA, Third edition, 2000.
21. 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.
22. 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.
23. Atsushi Ohori. *A Study of Semantics, Types, and Languages for Databases and Object-oriented Programming*. PhD thesis, University of Pennsylvania, 1989.
24. Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
25. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, the Revised Report*. Cambridge University Press, April 2003.
26. John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, April 1974.
27. Peter J. Stuckey and Martin Sulzmann. A theory of overloading. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2004. To appear in ACM Transactions on Programming Languages and Systems.
28. 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.
29. 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.