Variable Types for Genericity and Abstraction*

Robert Harper

Spring, 2025

1 Introduction

In a landmark paper Reynolds (1983) develops a mathematical account of Strachey's informal concept of *parametricity* of polymorphic functions. Parametricity characterizes the "uniform" behavior of polymorphic functions using Tait computability.[1] Reynolds's work, which was motivated by the study of data abstraction in programming languages, was done around the same time as, and independently of, Girard's extension of Tait's method to second-order quantification, which was motivated by the analysis of proofs in second-order logic.[2] Whereas Girard made use of unary predicates, Reynolds used binary relations, a technically small, yet practically large, difference that gave rise to new methods for proving properties of programs knowing only their types. Reynolds observed that the type discipline of a language determines the abstraction properties enjoyed by its programs; in particular, clients of abstract types are *polymorphic*, and hence enjoy stability properties across changes of representation determined entirely by their types.

The formulation given here makes use of the aforementioned ideas from Tait, Girard, and Reynolds, but cast in an operational framework. In contrast to the presentation in PFPL the formulation given here is independent of a prior notion of equality. In compensation candidates are required to enjoy a property called *zig-zag completeness* (Krishnaswami and Dreyer, 2013), which suffices to ensure that equality is symmetric and transitive by a simple and direct argument. And, in another departure from PFPL, the development centers on the notion of a *variable type*, and only much later considers their internalization by type constructors, corresponding to universal and existential quantification over types. Such quantifiers are said to be *impredicatve* because the range of quantification includes the quantified types themselves, a kind of "quasi-circularity" that requires careful treatment to ensure consistency. In particular, it is highly relevant that the existential type is formulated as a *negative* type, from which its type component cannot be extracted.

2 Variable Types

Consider a simply typed λ-calculus with a range of type constructors including, at least, function, unit, and product types, but also void and sum types, inductive types, and coinductive types. For the present purposes it is important that the language not include any form of effect, including partiality, which

[1] See Harper (2025c) for an introduction to Tait's method.

[2] See Harper (2025a) for an introduction to Girard's method.

would greatly complicate the development and obscure the main ideas. It is also important that the language be equipped with a type $\mathtt{ans}$ of *answers* whose values are $\mathtt{yes}$ and $\mathtt{no}$, and with no elimination forms. For the time being universal and existential types, as formulated in **PFPL** are *not* to be included, but note that this choice will be reconsidered towards the end of this note.

The key concept considered here is the notion of a *variable type*, which is to say a variable that ranges over types. A *type variable context*, Δ, is a finite set of declarations $X_1\ \mathsf{type}, \ldots X_n\ \mathsf{type}$, and a *term variable context* is a finite set of declarations $x_1 : A_1, \ldots, x_n : A_n$, with no variable declared more than once. The typing judgment $\Gamma \vdash_\Delta M : A$ expresses that M is of type A relative to assumptions Γ, all of which may involve the types declared in Δ.

Exercise 1. *Give a collection of rules defining $\Delta \vdash A\ \mathsf{type}$ for a selection of types A suggested above.*

The typing rules defining the various language constructs under consideration are to be reformulated by attaching Δ to the turnstile, and introducing premises of rules requiring that types arising in the program are in fact well-formed relative to Δ.

Exercise 2. *Reformulate the statics of the function type in the suggested style. State and prove that typing is stable under substitution of types for type variables.*

The dynamics is to be understood via an implicit *erasure* of type information from terms, including omission of type labels on variable binders. The dynamics is given by a transition system with a by-name interpretation of variables for the sake of simplicity, there being no possibility of divergence or undefinedness of well-typed terms. With this understanding it is never necessary to substitute types for type variables when considering the dynamics of closed terms.

Variable types are introduced by the binding construct $\mathtt{Let}(A; X.M)$, which binds the variable type X to A for use within M:

$$\textsc{Let}\ \frac{\Delta \vdash A\ \mathsf{type} \qquad \Gamma \vdash_{\Delta, X\ \mathsf{type}} M : B \qquad \Delta \vdash B\ \mathsf{type}}{\Gamma \vdash_\Delta \mathtt{Let}(A; X.M) : B}$$

Notice that, in contrast to Reynolds's original formulation, the result type is not permitted to depend on X, which by tacit binding convention is not in Δ. It has the evident dynamics on the erasure:

$$\mathtt{Let}(\_; \_.M) \longmapsto M$$

The corresponding term-level binding construct, $\mathtt{let}(M; x.N)$, is short-hand for the application $\lambda(x.N)(M)$, with the derived typing rule

$$\textsc{Let}\ \frac{\Gamma, x : A \vdash_\Delta N : B}{\Gamma \vdash_\Delta \mathtt{let}(M; x.N) : B}$$

and dynamics given by the transition

$$\mathtt{let}(M; x.N) \longmapsto [M/x]N.$$

The constructs are introduced by a *linker* that binds the type and code of a library construct for use within a client program. Importantly, the client program is typed in ignorance of the implementation details of the library, yet at run-time there is *no overhead* in imposing this restriction.

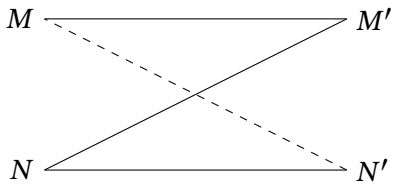

Figure 1: Zig-Zag Completeness, Pictorially

3 Correspondence and Exact Equality

A binary relation R over closed terms of types A and A', written $R \subseteq A \times A'$, is a *parametricity candidate* if it satisfies the following two closure conditions:

1. *Head expansion*: if $M\ R\ M'$, then
   (a) if $N \longmapsto M$, then $N\ R\ M'$, and
   (b) if $N' \longmapsto M'$, then $M\ R\ N'$.
2. *Zig-Zag Completeness (ZZC)*: if $M\ R\ M'$, $N\ R\ N'$, and $N\ R\ M'$, then $M\ R\ N'$.

Head expansion is a natural requirement when thinking of types as specifications of program behavior. Zig-zag completeness is depicted in Figure 1. It may be stated in terms of the converse and composition of relations as the containment $R \circ R^{\mathrm{op}} \circ R \subseteq R$. The opposite containment is always valid, so zig-zag completeness may be re-stated as the equation $R \circ R^{\mathrm{op}} \circ R = R$. As mentioned in Harper (2025d), the importance of zig-zag completeness lies in its compatibility with the type heterogeneity of the parametricity candidates, providing a sufficient condition for the symmetry and transitivity of exact equality.

A *candidate assignment*, η, for type substitutions $\delta, \delta' : \Delta$ is a function such that $\eta(X) \subseteq \delta(X) \times \delta'(X)$ is a type candidate for each type variable X such that $\Delta \vdash X$ type.

Definition 1 (Correspondence)**.** *The binary relation of* correspondence *of two closed terms relative to an open type A such that $\Delta \vdash A$ type and a candidate assignment $\eta \subseteq \delta \times \delta'$ for Δ, written $M \sim M' \in A\ [\eta \subseteq \delta \times \delta']$, is defined by induction on the structure of A as follows:*

$$M \sim M' \in \mathtt{ans}\ [\eta \subseteq \delta \times \delta'] \Leftrightarrow \textit{either } M, M' \longmapsto^* \mathtt{yes} \textit{ or } M, M' \longmapsto^* \mathtt{no}$$

$$M \sim M' \in X\ [\eta \subseteq \delta \times \delta'] \Leftrightarrow M\ \eta(X)\ M'$$

$$M \sim M' \in A_1 \to A_2\ [\eta \subseteq \delta \times \delta'] \Leftrightarrow \begin{cases} M \longmapsto^* \lambda(x.M_2),\ M' \longmapsto^* \lambda(x.M_2'), & \textit{and} \\ [M_1/x]M_2 \sim [M_1'/x]M_2' \in A_2\ [\eta \subseteq \delta \times \delta'] & \textit{if } M_1 \sim M_1' \in A_1\ [\eta \subseteq \delta \times \delta'] \end{cases}$$

It is immediate from the form of the definition that correspondence is closed under head expansion, given that candidates are required to be.

Lemma 2 (Head Expansion)**.** *If $M \sim M' \in A\ [\eta \subseteq \delta \times \delta']$, then if $N \longmapsto M$, then $N \sim M' \in A\ [\eta \subseteq \delta \times \delta']$, and if $N' \longmapsto M'$, then $M \sim N' \in A\ [\eta \subseteq \delta \times \delta']$.*

Proof. All conditions are defined in terms of evaluation to a value, and candidates are assumed to be closed under head expansion. □

Lemma 3 (Zig-Zag Completeness)**.** *For each $\Delta \vdash A$ type, and each candidate assignment η for Δ, the correspondence relation $\_ \sim \_ \in A\ [\eta \subseteq \delta \times \delta']$ is zig-zag complete.*

Exercise 3. *Prove Lemma 3.*

Exact equality of two terms in a type, written $\Gamma \gg_\Delta M \doteq M' \in A$, is defined to mean

For all candidate assignments η for $\delta, \delta' : \Delta$, and for all substitutions $\gamma \sim \gamma' \in \Gamma\ [\eta \subseteq \delta \times \delta']$,

$$\widehat{\gamma}(M) \sim \widehat{\gamma'}(M) \in A\ [\eta \subseteq \delta \times \delta'].$$

The notation $\Gamma \gg_\Delta M \in A$ is defined to mean $\Gamma \gg_\Delta M \doteq M \in A$. It expresses that M conforms to the behavior specified by exact equality by being related to itself—a non-trivial property of the type system essential to the development.

Notice that the outer quantification is over substitutions $\delta, \delta' : \Delta$ and candidate assignments for them. Although it is true that every such substitution induces, via the definition of correspondence, a parametricity candidate, there are far more parametricity candidates than may be denoted by a type expression. *This is the essential ingredient in the Girard/Reynolds account of variable types.*

Lemma 4 (Compositionality)**.** *Suppose that $\Delta \vdash B$ type and Δ, X type $\vdash A$ type. Let η be a relation assignment for $\delta, \delta' : \Delta$, and let R be the binary relation $\_ \sim \_ \in B\ [\eta \subseteq \delta \times \delta']$. Then $M \sim M' \in [B/X]A\ [\eta \subseteq \delta \times \delta']$ iff $M \sim M' \in A\ [\eta[X \longmapsto R] \subseteq \delta[X \longmapsto \widehat{\delta}(B)] \times \delta'[X \longmapsto \widehat{\delta'}B]]$.*

Exercise 4. *Prove Lemma 4.*

The *parametricity theorem* states that well-typed open terms are exactly equal to themselves, which is to say that exact equality is reflexive.

Theorem 5 (Parametricity)**.** *If $\Gamma \vdash_\Delta M : A$, then $\Gamma \gg_\Delta M \in A$.*

Proof. By induction on typing derivations, using Lemma 2 and 4. The interesting case is that for $\mathtt{Let}(A; X.M)$. Suppose that $\delta, \delta' : \Delta$ are substitutions of closed types for the type variables in Δ, that $\eta(X) \subseteq \delta(X) \times \delta'(X)$ is a parametricity candidate for each $\Delta \vdash X$ type, and that $\gamma \sim \gamma' \in \Gamma\ [\eta \subseteq \delta \times \delta']$, with the intent to show that

$$\mathtt{Let}(\widehat{\delta}(A); X.\widehat{\gamma}(M)) \sim \mathtt{Let}(\widehat{\delta'}(A); X.\widehat{\gamma'}(M)) \in B\ [\eta \subseteq \delta \times \delta'].$$

By head expansion it suffices to show

$$\widehat{\gamma}(M) \sim \widehat{\gamma'}(M) \in B\ [\eta \subseteq \delta \times \delta'].$$

By the inductive hypothesis, choosing $\widehat{\delta}(A)$, $\widehat{\delta'}(A)$, and $\_ \sim \_ \in\ [\eta \subseteq \delta \times \delta']$ as the interpretation of X, we obtain

$$\widehat{\gamma}(M) \sim \widehat{\gamma'}(M) \in B\ [\eta[X \longmapsto \_ \sim \_ \in\ [\eta \subseteq \delta \times \delta']] \subseteq \delta[X \longmapsto \widehat{\delta}(A)] \times \delta'[X \longmapsto \widehat{\delta'}(A)]].$$

The result follows immediately by compositionality. □

Judgmental, or definitional, equality of expressions of a type is defined as in Harper (2025d), albeit for equations parameterized by type variables, written $\Gamma \vdash_\Delta M \equiv M' : A$.

Exercise 5. *Prove that the extension of definitional equivalence to variable types is sound with respect to the generalized definition of exact equality given above.* Hint: *you will need to verify that exact equality is symmetric and transitive, making essential use of zig-zag closure; reflexivity is afforded by Theorem 5.*

4 Internalizing Variable Types

Variable types are the essence of data abstraction. As with any concept of variable, they are placeholders for "types unknown" that, in the formulation considered above, are provided by a linker that connects the library implementation to the client code. Parametricity guarantees that the client behavior is stable under replacement of one implementation by another, provided that they correspond under the "fiction" that the two implementation types are regarded as "equal."

The linking-based formulation of abstraction goes a long way towards explaining, and supporting, a practical data abstraction mechanism. However, it would be also be useful to internalize these mechanisms as type constructs so that client code can be broken up into reusable parts using type-level λ-abstraction and instantiation, and so that implementor code can be packaged as a unit consisting of a representation type and its associated implementation. These concepts are supported by *universal* and *existential* type quantification, as formulated in PFPL. Thus, the type $\forall(X.A)$ classifies programs that abstract over an unspecified type, X, within a body of code classifed by the type A, and the type $\exists(X.A)$ classifies packages consisting of a type, X, and an implementation of the type A for the specified choice of X.

The quantifiers are said to be *impredicative* (that is, *non-predicative*) in the sense that the universal and existential types are types, and are thereby lie within the range of the quantifiers. In particular, a universally quantified type may be instantiated by itself, disrupting attempts to define their meaning in a non-circular manner. The definability of the existential type in terms of the universal type relies on impredicativity, as does the definability of other typing constructs in terms of the universal. Although impredicativity seems natural, it is sensible only for *types-as-arguments*, and does not provide for computing *types-as-results* of computations. (The definition of the existential is chosen to rely only on types-as-inputs by requiring the usage of an abstract type to be confined to the elimination form.)

The alternative, called *predicative*[3] quantification, restricts the range of significance of the type variables to *non-quantified* types, which are sometimes said to "small" in contrast to the "large" quantified types.[4] Thus variable types range only over small types, and thus do not include quantified types. Predicativity thereby avoids the circularity inherent in the impredicative formulation, but in so doing it also reduces the expressive power of the quantified types. In particular the existential is no longer definable in terms of the universal, nor are the other encodings discussed in PFPL available in the predicative setting.

It might seem, then, that the impredicative formulation is to be preferred. However, as is explained in Harper (2025b), it also has a very significant drawback, namely that it is not possible to support modular programming using the impredicative formulation. In particular there is no way to compute a structure containing both type and value components in such a way that both the type and the value can be recovered from the structure without loss. That is, it should be possible to *compute* a structure containing a type in such a way that the computed type can be recovered from the computation. This can be achieved in a specialized form of dependent type theory that makes explicit the *phase distinction* between the static and dynamic components of a program module so that type checking need not involve comparison of code, only of types.

Exercise 6. *Extend the definition of correspondence, and hence exact equality, to account for universal and existential types under the impredicative formulation.* Hint*: Variable types, whether quantified or not, range over arbitrary parametricity candidates. Extend the proof of the parametricity theorem to account*

[3] That is, *non-non-predicative*!

[4] The "small" vs. "large" distinction may be generalized to a hierarchy of *universes*, which arise in dependent type theory.

for these constructs, following along the lines of Harper (2016), albeit in the present setting of zig-zag closure and closure under head expansion.

Exercise 7. *Extend the formulation of definitional equality and exact equality explored in Exercise 5 to account for quantification, and verify the validity of the $\beta\eta$ laws for type quantification are valid under this interpretation.*

References

Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, Cambridge, England, Second edition, 2016.

Robert Harper. How to (re)invent Girard's method. Unpublished lecture note, January 2025a. URL `https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/girard.pdf`.

Robert Harper. Program modules. (Unpublished lecture note), Spring 2025b. URL `https://www.cs.cmu.edu/~rwh/courses/atpl/notes/modules.pdf`.

Robert Harper. How to (re)invent Tait's method. Unpublished lecture note, January 2025c. URL `https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/tait.pdf`.

Robert Harper. Semantic equality for typed λ-calculus. Unpublished lecture note, January 2025d. URL `https://www.cs.cmu.edu/~rwh/courses/atpl/pdfs/semeq.pdf`.

Neelakantan R. Krishnaswami and Derek Dreyer. Internalizing Relational Parametricity in the Extensional Calculus of Constructions. pages 20 pages, 591837 bytes, 2013. ISSN 1868-8969. doi: 10.4230/LIPICS.CSL.2013.432. URL `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CSL.2013.432`. Artwork Size: 20 pages, 591837 bytes ISBN: 9783939897606 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 513–523. North-Holland/IFIP, 1983.