Parametricity and variants of Girard's J operator

Robert Harper

Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15217

John C. Mitchell

Computer Science Department, Stanford University, Stanford, CA 94305

Abstract

The Girard-Reynolds polymorphic λ-calculus is generally regarded as a calculus of parametric polymorphism in which all well-formed terms are strongly normalizing with respect to β-reductions. Girard demonstrated that the additional of a simple "non-parametric" operation, J, to the calculus allows the definition of a non-normalizing term. Since the type of J is not inhabited by any closed term, one might suspect that this may play a role in defining a non-normalizing term using it. We demonstrate that this is not the case by giving a simple variant, J', of J whose type is otherwise inhabited and which causes normalization to fail. It appears that impredicativity is essential to the argument; predicative variants of the polymorphic λ-calculus admit non-parametric operations without sacrificing normalization.

Key words: Formal semantics, functional programming, programming calculi, programming languages, theory of computation.

[1] This research was sponsored by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050, NSF grants CCR-9303099 and CCR-9629754, and ONR MURI Award N00014-97-1-0505. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

1 Introduction

The impredicative polymorphic lambda calculus, or System F [4,10], is generally recognized as a calculus of *parametric* polymorphism [11]. Intuitively, this means that every polymorphic function definable in the language must use the same algorithm at all types. There are no non-parametric functions such as a single multiplication function that computes an inner product on vectors, ordinary product on natural numbers, and similar or unrelated "products" on other types.

In his 1971 paper, Girard discusses parametricity in System F and shows that normalization fails if a non-parametric operator, J, is added to the calculus [4]. This example has been interpreted as a demonstration that System F is inherently a calculus of parametric functions. Specifically, since adding a non-parametric operator alters a fundamental property of the system, System F must be intrinsically parametric. However, this argument assumes that J is as simple a non-parametric operator as possible. Specifically, if it is possible to add a simpler non-parametric operator to System F in a manner that preserves strong normalization, then Girard's example does not convincingly demonstrate the wholly parametric nature of System F. In this short note, we give some variants of Girard's example, hoping to more clearly identify the kind of non-parametric operations that invalidate normalization. In the concluding remarks, we observe that impredicativity appears essential to the relation between parametricity and normalization.

One reason to examine Girard's J is that it combines non-parametric behavior with a change in the typing properties of System F. More specifically, the type[2] of J is $\forall s.\forall t.s \to t$, which is not the type of any pure closed term of System F since it is not a provable propositional formula. The non-normalizing term constructed using J also seems to require a term $0 : \forall t.t$. Again, there is no pure closed term with type $\forall t.t$. This led us to formulate the following question:

> Is there a simple non-parametric operator $Op : \sigma$ such that
> (i) there exist pure closed terms of type σ, and
> (ii) adding Op to System F causes normalization to fail?

We settle this question affirmatively and give a simple example, J'. Our goal is to dispel the possible misconception that in impredicative systems, Girard's example is simply a "typing trick" that could not be carried out with other non-parametric operations.

[2] Notational conventions: we use universal quantification for polymorphic types, write $\lambda t\, e$ for abstraction over a type, and $e\,[\tau]$ for type application.

One consequence of replacing J by an operator whose type already contains pure closed terms is that we can easily show that no fixed-point operator is definable. More specifically, suppose $Op : \sigma$ and there exists a pure closed term of type σ. Then using the formulas-as-types analogy [7], we can show that there exists a closed term of type τ containing Op iff there exists a pure closed term of type τ not containing Op. The argument is simply that if we add a new axiom for a formula that is already provable, we do not change the set of provable formulas. Since $\forall t.(t \to t) \to t$ is not a provable formula of intuitionistic logic, there is no pure closed term of this type. It follows that when the type of Op contains pure closed terms, no closed term of type $\forall t.(t \to t) \to t$ is definable from Op. In particular, since a polymorphic fixed-point operator Y has this type, Y is not definable from Op. In light of various studies relating logical paradoxes and fixed-point operators [8,2,3], it does not seem easy to establish that no fixed-point operator or "looping combinator" (see [8,3]) is definable from J.

2 Review of Girard's J example

Girard's example adds two constants to System F, each with associated reduction rules. The first is a constant $0 : \forall t.t$ which allows us to choose an element of each type, by application. The intent is that 0 is a form of choice function that selects elements of different types "harmoniously." Perhaps the simplest way of understanding this is to recall that in Girard's realizability-like model, HEO_2 [12,5], the natural number 0 codes the constant function 0 and therefore provides an element of all types. This explains the use of the symbol "0" and also the following associated reduction rules:

$$\begin{aligned} 0\,[\sigma \to \tau]\,M &\longrightarrow 0\,[\tau] \\ 0\,[\forall t.\sigma]\,[\tau] &\longrightarrow 0\,[\{\tau/t\}\sigma] \end{aligned}$$

While Girard's nonterminating term requires a constant with type $\forall t.t$, the reduction rules for 0 are not needed to construct the nonterminating term.

Intuitively, J produces a choice function by combining 0 with a given element $M : \sigma$ of any particular type. The resulting choice function $\lambda t.\ J\,[t]\,[\sigma]\,M$ selects element M of type σ and element $0\,[\tau]$ of any type τ different from σ. The constant $J : \forall s.\forall t.s \to t$ has reduction rules

$$J\,[\sigma]\,[\tau]\,M \to \begin{cases} M & \text{if } \sigma = \tau \\ 0\,[\tau] & \text{otherwise} \end{cases}$$

Girard does not specify whether types σ and τ in this reduction rule may contain free variables. However, the system becomes inconsistent and non-confluent if we allow arbitrary types in both forms of reduction, as illustrated below. We therefore restrict the reduction $J\,[\sigma]\,[\tau]\,M \rightarrow 0\,[\tau]$ to the case where σ and τ are distinct closed type expressions.

Intuitively, the problem with types that contain free variables is that these variables may be formal parameters of some function. If σ and τ are not syntactically identical, but have a common substitution instance, then application of the enclosing function may change the applicable J-reduction rule. For example, the term

$$(\lambda t.\ J\,[s]\,[t]\,x)\,[s]$$

with $x : s$ reduces to two normal forms, x and $0\,[s]$. The first reduction begins with the outermost β-reduction (substituting s for t):

$$(\lambda t.\ J\,[s]\,[t]\,x)\,[s] \ \rightarrow\ J\,[s]\,[s]\,x \ \rightarrow\ x$$

The second reduction applies J-reduction first, while the type arguments are different:

$$(\lambda t.\ J\,[s]\,[t]\,x)\,[s] \ \rightarrow\ (\lambda t.\ 0\,[t])\,[s] \ \rightarrow\ 0\,[s]$$

Clearly this is inconsistent, since $x = (\lambda t.\ J\,[s]\,[t]\,x)\,[s] = 0\,[s]$ implies that all elements of type s (which is arbitrary) are equal to $0\,[s]$.

Using J, we can give a form of the self-application function $\lambda x.\ xx$ an unexpected type. More specifically, let D be the term

$$D \ \stackrel{def}{=}\ \lambda x{:}\forall t.t.\ (x\,[(\forall t.t) \rightarrow (\forall t.t)]\,x)$$

of type $(\forall t.t) \rightarrow (\forall t.t)$ and note that the term

$$X \ \stackrel{def}{=}\ \lambda t.\ J\,[(\forall t.t) \rightarrow (\forall t.t)]\,[t]\,D$$

has type $\forall t.t$. It is easy to check that $Z \stackrel{def}{=} X\,[(\forall t.t) \rightarrow (\forall t.t)]\,X$ reduces to itself, proving that the extension of System F with 0 and J is non-normalizing.

Although it is difficult to characterize the "main idea" behind the term Z with much accuracy, one explanation of the construction is that the type $\forall s.\forall t.s \rightarrow t$ of J allows us to hide D inside a term of type $\forall t.t$. This unexpected type for

D allows us to apply D to itself. In this sense, one might wonder whether the unusual types of 0 and J are partly responsible for the nontermination of reduction. After all, if we add a constant $c : \forall t.t$ with reduction rule

$$c\,[(\forall t.t) \to (\forall t.t)] \quad \to \quad D$$

then it is clear that $c\,[(\forall t.t) \to (\forall t.t)]\, c$ does not normalize. While c has the essential reduction behavior of X above, c does not in any way seem to be "nonparametric."

3 Alternative non-parametric primitives

A relatively natural non-parametric operation is an equality test on types. This might take the form of a term

$$\mathit{EqTest} : \forall s.\, \forall t.\, \mathit{Bool}$$

with $\mathit{EqTest}\,[\sigma]\,[\tau] = \mathit{true}$ if σ and τ are the same type, and false otherwise. If this test is going to be useful, however, we would want to write expressions such as

$$\lambda x{:}s.\ \lambda y{:}t.\ \mathsf{if}\ \mathit{EqTest}\,[s]\,[t]\ \mathsf{then}\ x\ \mathsf{else}\ y$$

Intuitively this expression has type $s \to t \to t$. If types s and t are equal, then the result x has type $s = t$ and otherwise the result y has type t. However, we cannot obtain this typing without using the equation $s = t$ in typing the first arm of the conditional. If we focus on the required combination of equality test and conditional expression, we are led to consider a type-conditional operation *TypeCond* with the slightly unusual typing rule

$$\frac{\Gamma, \sigma = \tau \triangleright M : \rho \qquad \Gamma \triangleright N : \rho}{\Gamma \triangleright \mathit{TypeCond}\,[\sigma]\,[\tau]\, M\, N : \rho} \quad (\mathit{TypeCond})$$

Intuitively, this rule says that if M has type ρ under the assumption that $\sigma = \tau$, and N has type ρ without this assumption, then $\mathit{TypeCond}\,[\sigma]\,[\tau]\, M\, N$ has type ρ. Since we will only be interested in simple examples illustrating the main ideas, we will not need to formalize typing with equality assumptions in contexts. We show that normalization fails with *TypeCond* and then give a simpler but less intuitive primitive that does not involve type equations.

The reduction rules for $TypeCond$ are

$$\begin{array}{ll} TypeCond\,[\sigma]\,[\sigma]\,M\,N \to M & \\ TypeCond\,[\sigma]\,[\tau]\,M\,N \to N & \quad \sigma, \tau \text{ distinct, closed} \end{array}$$

One way of understanding the potential utility of $TypeCond$ is to consider writing a function that does multiplication on natural numbers and a componentwise multiplication on pairs of natural numbers. In other words, we would like to combine functions

$$\begin{array}{rcl} mult_N & \stackrel{def}{=} & \lambda x{:}nat.\ \lambda y{:}nat.\ x * y \\ & : & nat \to nat \to nat \\ mult_{N,N} & \stackrel{def}{=} & \lambda x{:}nat \times nat.\ \lambda y{:}nat \times nat.\ \langle \pi_1 x * \pi_1 y, \pi_2 x * \pi_2 y \rangle \\ & : & nat \times nat \to nat \times nat \to nat \times nat \end{array}$$

where $\pi_i : nat \times nat \to nat$ are the projection functions on natural number pairs. We can do this with $TypeCond$ by writing

$$\begin{array}{rl} mult \stackrel{def}{=} & \lambda s.\ \lambda x{:}s.\ \lambda y{:}s. \\ & \quad TypeCond\,[s]\,[nat]\,(mult_N\,x\,y) \\ & \quad\quad TypeCond\,[s]\,[nat \times nat]\,(mult_{N,N}\,x\,y) \\ & \quad\quad\quad x \\ : & \forall t.\, t \to t \to t \end{array}$$

The function $mult$ takes arguments x and y of the same type. If the type is nat, the result is $mult_N xy$, else if the type is $nat \times nat$, the result is $mult_{N,N} xy$, else the result is x. Since we obtain a result of the right type in any case, it is semantically sensible to give $mult$ type $\forall t.\, t \to t \to t$. We cannot seem to define $mult$ using J instead. The problem is that $TypeCond$ provides an “if-then-else” test for type equality, while J is only an “if-then” test, with the “else” case defaulting to 0.

Using a variant of Girard's example, we can show that $TypeCond$ destroys strong normalization. Consider the term

$$X \quad \stackrel{def}{=} \quad \lambda s.\ \lambda x{:}s.\ TypeCond\,[\sigma]\,[s]\,(x\,[\sigma]\,x)\,x$$

where $\sigma = \forall t.\, t \to t$. It should be clear from the basic properties of *TypeCond* that $X : \sigma$. Using the first reduction associated with *TypeCond*, we have

$$X\,[\sigma]\,X \;\longrightarrow\; \mathit{TypeCond}\,[\sigma]\,[\sigma]\,(X\,[\sigma]\,X)\,X \;\longrightarrow\; X\,[\sigma]\,X$$

This shows that strong normalization fails with this form of type conditional operation. A technical point is that in constructing X, we only need to vary the second type argument to *TypeCond*. Therefore, it would suffice to have a weaker primitive $\mathit{TypeCond}_\sigma : \forall t.\, \sigma \to t \to t$ for the specific type σ identified above.

Intuitively, *TypeCond* seems to give us the expressive power of J, without requiring a "universal choice function" 0. We justify this intuition by showing that using polymorphic constant $0 : \forall t.t$, we can define Girard's J operator from *TypeCond* as follows:

$$J \;\stackrel{def}{=}\; \lambda s.\ \lambda t.\ \lambda x{:}s.\ \mathit{TypeCond}\,[s]\,[t]\,x\,(0\,[t])$$

A remaining reservation that one might have about *TypeCond* is the unusual typing rule using type equations. We therefore conclude with a variant, J', of J that does not require 0 or type equations. Essentially, J' is a restriction of J to function types, using the polymorphic identity in place of 0. More concretely, let $J' : \forall s.\forall t.(s \to s) \to (t \to t)$ be a constant. We base the reduction rules for J' on the term

$$\lambda s.\ \lambda t.\ \mathit{TypeCond}\,[s]\,[t] \quad (\lambda x{:}s \to s.\ x) \quad (\lambda x{:}s \to s.\ \lambda y{:}t.\ y)$$

which could be taken as the definition of J' from *TypeCond*. We can understand the type of J' by noting that if $s = t$, then $\lambda x{:}s \to s.\ x : (s \to s) \to (t \to t)$. Since we have $\lambda x{:}s \to s.\ \lambda y{:}t.\ y : (s \to s) \to (t \to t)$ in any case, J' has the specified type.

The reduction rules for J' are

$$\begin{array}{lll} J'\,[\sigma]\,[\sigma]\,M \to M & & \\ J'\,[\sigma]\,[\tau]\,M \to \mathit{Id}\,[\tau] & \qquad & \sigma, \tau \text{ distinct, closed} \end{array}$$

where $\mathit{Id} \stackrel{def}{=} \lambda t.\ \lambda y{:}t.\ y$. We can carry out Girard's example by letting $\rho \stackrel{def}{=} \forall t.\, t \to t$ and letting $X : \rho$ be the term

$$X \;\stackrel{def}{=}\; \lambda t.\ J'\,[\rho]\,[t]\,(\lambda x{:}\rho.\ x\,[\rho]\,x)$$

Note that $\lambda x{:}\rho.\ x\,[\rho]\,x$ has type $\rho \to \rho$ and hence X has type ρ. Finally, $X\rho X$ has type ρ and reduces to $X\rho X$. Therefore J' causes normalization to fail.

4 Concluding remarks

Girard's choice function $0 : \forall t.t$ and non-parametric operator $J : \forall s.\forall t.s \to t$ cause the strongly normalizing System F of impredicative polymorphic lambda calculus to become non-normalizing. This provides one form of evidence that System F is inherently parametric. In this note, we show that "weaker" forms of non-parametricity still cause the system to be non-normalizing. A technical difference between Girard's original example and our *TypeCond* or J' is that the latter do not change the set of types that contain closed terms. A consequence of this weak form of conservativity is an easy proof that no polymorphic fixed-point or looping combinator of type $\forall t.(t \to t) \to t$ can be defined from *TypeCond* or J'. However, it is possible that a fixed-point combinator of type $(Nat \to Nat) \to Nat$, for example, where $Nat \stackrel{def}{=} \forall t.(t \to t) \to t \to t$ is the type of Church numerals, could be defined from *TypeCond* or J'.

The non-parametric operators discussed in this short note are related to two other non-parametric primitives in the literature. The first is the *typecase* construct, associated with the type *Dynamic* [1], which provides a case analysis on types. As pointed out by Abadi, *et al.* [1], a fixed point combinator of a specific type such as $((Nat \to Nat) \to (Nat \to Nat)) \to (Nat \to Nat)$ is definable in the simply-typed λ-calculus enriched with the type *Dynamic*. Since we may regard the type *Dynamic* as an abbreviation for the quantified type $\exists t.t$ (itself an abbreviation for a type involving only universal quantifiers), the existence of fixed point combinators may be traced to the combination of non-parametric polymorphism and impredicative quantified types. In particular, the definition of a fixed-point operator involves embedding functions of type $Dynamic \to T \to U$ into type *Dynamic*, which would not be allowed if *Dynamic* were considered a different class of type from the types T and U where we wish to construct a fixed point.

In a second use of non-parametricity, Harper and Morrisett [6] consider a generalization of the *typecase* construct, called *typerec*, for performing "intensional analysis" of types at run-time and compile-time. The *typerec* operator generalizes *typecase* to a form of primitive recursion on type expressions, rather than simple case analysis. However, the surrounding language is limited to predicative polymorphism, restricting intensional analysis to unquantified types. Consequently, the strong normalization property for pure terms holds and it is impossible to define a fixed-point operator from *typerec*. The restriction to predicative polymorphism was subsequently shown to be essential by Palmgren [9], who demonstrated that the combination of impredicative quantification and "universe elimination" in type theory allows for the construction of a non-normalizable term.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] T. Coquand. An analysis of Girard's paradox. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 227–236, June 1986.

[3] T. Coquand and H. Herbelin. A-translation and looping combinators in pure type systems. *J. Functional Programming*, 4(1):77–88, 1994.

[4] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, Amsterdam, 1971.

[5] J.-Y. Girard. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. These D'Etat, Universite Paris VII, 1972.

[6] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

[7] W. Howard. The formulas-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[8] A.R. Meyer and M.B. Reinhold. Type is not a type. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 287–295, January 1986.

[9] Erik Palmgren. On universes in type theory. In Giovanni Sambin and Jan Smith, editors, *Twenty-Five Years of Constructive Type Theory*, Oxford Logic Guides. Oxford University Press, Oxford, England, 1998.

[10] J.C. Reynolds. Towards a theory of type structure. In *Paris Colloq. on Programming*, pages 408–425, Berlin, 1974. Springer-Verlag LNCS 19.

[11] J.C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, Amsterdam, 1983.

[12] A.S. Troelstra. *Mathematical Investigation of Intuitionistic Arithmetic and Analysis.* Springer LNM 344, Berlin, 1973.