Notes on Takeuti's Parametricity Logic and Wadler's Girard-Reynolds Isomorphism

Kevin Watkins

April 20, 2005
Carnegie Mellon University

Takeuti's Parametricity Logic

These are notes on Takeuti's paper *An Axiomatic System of Parametricity*, Fundamenta Mathematicae 33 (1998) 397–432. A slight restriction of the same logic appears in Wadler's paper *The Girard-Reynolds Isomorphism (second edition)*. Takeuti's logic is itself a reformulation of the system presented in Plotkin and Abadi's paper *A logic for parametric polymorphism*.

It seems that for any point on the λ-cube, one can derive an associated logic in Takeuti's style. The basic idea is to duplicate the constructs of the λ-calculus at the level of the logic, then to add more abstractions so that proofs can be parameterized over the constructs of the λ-calculus. Finally, extra abstractions are introduced to allow propositions to depend on the constructs of the λ-calculus (in Takeuti's system, only the terms, but in general one might allow predicates over types, too).

I'll characterize the systems by giving the formation judgments only. Each well-formed function space has abstraction and application terms as usual, with β-reductions and so on. I'll also suppress most of the contexts; hypotheses are always global to the subderivation above the rule where they're introduced, so there's no need to mention them all the time.

On the left we have the calculus Barendregt calls $\lambda 2$; it's just the polymorphic λ-calculus expressed as a generalized type system. On the right we have the logic (initally another copy of $\lambda 2$). I'm deliberately choosing the same letters for like constructs on both sides: the charm of Takeuti's system is in the stunning number of interesting morphisms between the left- and right-hand sides, all taking advantage of this analogy.

$$\frac{}{\vdash \mathit{Type} : \mathit{Kind}} \qquad \frac{}{\vdash \mathrm{Prop} \in \mathrm{Class}}$$

$$\frac{\vdash A : \mathit{Type} \quad \vdash B : \mathit{Type}}{\vdash A \to B : \mathit{Type}} \qquad \frac{\vdash \phi \in \mathrm{Prop} \quad \vdash \psi \in \mathrm{Prop}}{\vdash \phi \supset \psi \in \mathrm{Prop}}$$

$$\frac{\vdash K : \mathit{Kind} \quad a : K \vdash B : \mathit{Type}}{\vdash \forall a : K \,.\, B : \mathit{Type}} \qquad \frac{\vdash \kappa \in \mathrm{Class} \quad \alpha \in \kappa \vdash \phi \in \mathrm{Prop}}{\vdash \forall \alpha \in \kappa \,.\, \phi \in \mathrm{Prop}}$$

The logic will be higher-order and impredicative, because it allows us to quantify over propositions; for example, we can define absurdity as $\forall \alpha \in \mathrm{Prop} \,.\, \alpha$.

However, the logic will *not* be impredicative in the same way HOL is, because HOL identifies Class and *Type* (so every proposition or predicate is also a term), while Takeuti carefully keeps them separate. In fact, HOL with polymorphic types is inconsistent, as was shown by Girard (and later, for an even weaker system, Coquand).

Next we introduce function spaces that allow proofs in the logic to abstract over terms and constructors in the λ-calculus.

$$\frac{\vdash A : \mathit{Type} \quad x : A \vdash \phi \in \mathrm{Prop}}{\vdash \forall x : A \,.\, \phi \in \mathrm{Prop}}$$

$$\frac{\vdash K : \mathit{Kind} \quad a : K \vdash \phi \in \mathrm{Prop}}{\vdash \forall a : K \,.\, \phi \in \mathrm{Prop}}$$

As it stands, the dependency in $\forall x : A \,.\, \phi$ is useless, because propositions cannot depend on terms. We want to also allow functions that map terms and constructors into propositions. By analogy with the higher-order polymorphic λ-calculus, we introduce extra terminology for elements of these function spaces, calling them *predicates*. For example, an element of $A \to B \to \mathrm{Prop}$ is a predicate but not a proposition. The following analogical glossary may help.

Term M	Proof π
Type A	Proposition ϕ
Constructor A	Predicate ϕ
Kind K	Class κ

Takeuti includes only the rule for predicates on terms. I've shown the other rule (for predicates on constructors) in brackets.

$$\frac{\vdash A : \mathit{Type} \quad \vdash \kappa \in \mathrm{Class}}{\vdash A \to \kappa \in \mathrm{Class}}$$

$$\left[\frac{\vdash K : \mathit{Kind} \quad a : K \vdash \kappa \in \mathrm{Class}}{\vdash \Pi a : K \,.\, \kappa \in \mathrm{Class}}\right]^{1}$$

Now both $\forall x : A \,.\, \phi$ and $\forall a : K \,.\, \phi$ can actually be dependent—the latter in multiple ways—even if we only adopt Takeuti's rule. Example:

$$\forall x : (\forall a : K \,.\, B) \,.\, \forall a : K \,.\, \forall y : a \,.\, \forall \alpha \in (a \to B \to \mathrm{Prop}) \,.\, \alpha\ y\ (x\ a)$$

At the moment, the only element of *Kind* is *Type*, so the only polymorphic function space is $\forall a : \mathit{Type} \,.\, B$. It's also possible to extend both the λ-calculus and the logic to ωth-order polymorphism (what Barendregt calls $\lambda\omega$). Takeuti doesn't do this.

$$\left[\frac{\vdash K : \mathit{Kind} \quad \vdash L : \mathit{Kind}}{\vdash K \to L : \mathit{Kind}}\right]^{2} \qquad \left[\frac{\vdash \kappa_1 \in \mathrm{Class} \quad \vdash \kappa_2 \in \mathrm{Class}}{\vdash \kappa_1 \to \kappa_2 \in \mathrm{Class}}\right]^{2}$$

We can present these systems as generalized type systems in Barendregt's

sense:

Sorts	*Type*, *Kind*,	Prop, Class
Axioms	*Type* : *Kind*,	Prop : Class
Rules	(*Type*, *Type*),	(Prop, Prop),
	*(*Kind*, *Type*),	*(Class, Prop),
		*(*Type*, Prop),
		*(*Kind*, Prop),
		(*Type*, Class),
		*[(*Kind*, Class)]^1,
	[(*Kind*, *Kind*)]^2,	[(Class, Class)]^2

This isn't entirely trivial. In the table, I've marked each formation rule that permits dependencies with a star. We have to consider carefully which dependencies might be possible in the g.t.s. setting. (In the logical frameworks world this is called *subordination analysis*—see the Twelf User's Guide, for example.) For each rule (s_1, s_2) we have to see whether an element of s_2 can depend on an element of an element of s_1. Each rule (s_1, s_2) introduces such a dependency of the elements of s_2 on the elements of s_1, as well as of the elements of elements of s_2 on the elements of elements of s_1.

By this reasoning, we have the following dependency structure:

Things	Depend on
Terms	Terms, Constructors (directly)
	Kinds (transitively)
Constructors	Constructors, Kinds
Kinds	Kinds
Proofs	Proofs, Predicates, Terms, Constructors (directly)
	Classes, Kinds (transitively)
Predicates	Predicates, Classes, Terms, Constructors, Kinds
Classes	Classes, Constructors, [Kinds]^1

With this table, it's easy to verify that whether or not any of the bracketed rules is included in the system, all the dependencies mentioned in the formation rules can really be non-trivial. Conversely, the formation rules (*Type*, *Type*), (Prop, Prop), (*Type*, Class), [(*Kind*, *Kind*)]^2, and [(Class, Class)]^2 can never be dependent, as the full forms given above indicate.

It should probably also be said that if we work within a signature made up of various constants, the constants all have to be elements of things that are themselves elements of sorts; we can't introduce constants c : *Kind* or $c \in$ Class, for example.

Having done all this analysis, we can now write the actual syntax of the logic:

$$
\begin{array}{ll}
K, L ::= \mathit{Type} \mid [K \to L]^2 & \kappa ::= \mathrm{Prop} \mid [\kappa_1 \to \kappa_2]^2 \mid \\
 & \quad A \to \kappa \mid [\Pi a : K \,.\, \kappa]^1 \\
A, B ::= a \mid A \to B \mid \forall a : K \,.\, B \mid & \phi, \psi ::= \alpha \mid \phi \supset \psi \mid \forall \alpha \in \kappa \,.\, \phi \mid \\
\quad [\lambda a : K \,.\, B]^2 \mid [A\ B]^2 & \quad \forall x : A \,.\, \phi \mid \forall a : K \,.\, \phi \mid \\
 & \quad [\lambda \alpha \in \kappa \,.\, \phi]^2 \mid [\phi\ \psi]^2 \mid \\
 & \quad \lambda x : A \,.\, \phi \mid \phi\ M \mid \\
 & \quad [\lambda a : K \,.\, \phi]^1 \mid [\phi\ A]^1 \mid \\
 & \quad [\phi \in \kappa]^2 \\
M, N ::= x \mid \lambda x : A \,.\, M \mid M\ N \mid & \pi ::= \xi \mid \lambda \xi \in \phi \,.\, \pi \mid \pi_1\ \pi_2 \mid \\
\quad \Lambda a : K \,.\, M \mid M\ A \mid & \quad \Lambda \alpha \in \kappa \,.\, \pi \mid \pi\ \phi \mid \\
\quad [M : A]^2 & \quad \Lambda x : A \,.\, \pi \mid \pi\ M \mid \\
 & \quad \Lambda a : K \,.\, \pi \mid \pi\ A \mid \\
 & \quad \pi \in \phi
\end{array}
$$

Variable forms $\quad a : K \mid b : K \mid x : A \mid y : A \mid \alpha \in \kappa \mid \beta \in \kappa \mid \xi \in \phi \mid \zeta \in \phi$

Now you see why I wanted to suppress all but the formation rules...

Really the most important thing to remember is the table of sorts, axioms, and rules expressing the logic as a generalized type system.

Projecting the Logic Onto the Functional Language

The first interesting morphism on this logic maps all the constructs on the right (the logical part) into the constructs on the left (the functional part), and throws away everything to do with the constructs on the left. Wadler calls this the 'Girard Projection', but since Girard treated HOL (more or less), and its variants, *not* Takeuti's system, this seems to me to be something of a misnomer.

Recalling our g.t.s. table

Sorts	*Type*, *Kind*,	Prop, Class
Axioms	*Type* : *Kind*,	Prop : Class
Rules	(*Type*, *Type*),	(Prop, Prop),
	*(*Kind*, *Type*),	*(Class, Prop),
		*(*Type*, Prop),
		*(*Kind*, Prop),
		(*Type*, Class),
		*[(*Kind*, Class)]^1,
	[(*Kind*, *Kind*)]^2,	[(Class, Class)]^2

it's easy to see what's going on: we're mapping Prop into *Type* and Class into *Kind*, the function space formed by (Prop, Prop) into (*Type*, *Type*), (Class, Prop) into (*Kind*, *Type*), and (Class, Class) into (*Kind*, *Kind*). Each thing on the right becomes its counterpart on the left.

But what about the things in the logic that don't *have* counterparts in the functional language? These just get dropped. A function space formed by (*Type*, Prop) becomes (the translation, recursively, of) its range. Similarly for the others. This is possible because of the dependency structure of the language.

Each of the function spaces that gets smashed has a domain in the functional language (which is getting dropped) and a range in the logic (which is translated recursively).

Yadda, yadda, yadda...

Embedding the Functional Language in the Logic

The next interesting morphism takes a program in the functional language on the left, and lifts it into a proof in the logic on the right. Wadler calls this (with, I believe, more justice) the 'Reynolds Embedding'.

Yadda, yadda, yadda...