

UNIVERSITY OF SUSSEX
COMPUTER SCIENCE



Parametricity as Isomorphism

Edmund Robinson

Report 5/92

September 1993

Computer Science
School of Cognitive and Computing Sciences
University of Sussex
Brighton BN1 9QH

ISSN 1350-3170

Parametricity as Isomorphism

EDMUND ROBINSON

ABSTRACT. We investigate a simple form of parametricity, based on adding “abstract” copies of pre-existing types. Connections are made with the Reynolds-Ma theory of parametricity by logical relations, with the theory of parametricity via dinaturality, and with the categorical notion of equivalence.

Introduction

In his fundamental paper on the notion of parametricity in connection with type theories [Rey83], John Reynolds links the notion of parametricity firmly to the notion of data abstraction. This, unlike Strachey’s earlier characterization via algorithm re-use, is a need-driven analysis. We need things to be parametric because otherwise our data abstractions will no longer be abstract. In his subsequent paper with Ma [MR91], two further points are made. One is that the problems reside more at the level of parametrized types than at the level of the quantified polymorphic types, and the other is that the notion of parametricity is not absolute, but relative. The Ma-Reynolds work produces a notion of parametricity defined relative to some category from which logical relations are taken. The larger that category, the stronger the constraints imposed by parametricity.

This notion of relativity also makes some sense in the type abstraction setting. The stronger our mechanisms for abstraction, the stronger the form of parametricity we will require.

Our purpose in this paper is to investigate more or less thoroughly a very simple form of parametricity, and to link it to a form of data abstraction. We shall also link it to certain other forms of parametricity proposed in the literature. The form of parametricity we shall be looking at can be expressed by saying that we wish to have the freedom to declare “abstract” copies of pre-existing types. The abstraction lies in our not caring which type we use to implement our copy, so long as it has the

School of Cognitive and Computing Sciences, University of Sussex, BRIGHTON BN1 9QH, England

I would like to acknowledge support during the preparation of this paper from ESPRIT BRA programmes of the EC no. 3007 (CLICS) and no. 6811 (CLICS-II).

Email: edmundr@cogs.susx.ac.uk

Copyright ©1992, Edmund Robinson

correct number of elements in a constructive sense, i.e. it is isomorphic to the target type.

Since this is such a restricted notion, we can not, of course, claim to produce a general theory of parametricity. Rather, what we are trying to do is map out at a simple level features which such a general theory should possess. Since a general theory ought to be able to encompass differing degrees of abstraction (and hence parametricity), what we are doing is, as it were, to present a particular horizontal slice.

We shall frequently use set-theoretic language to talk about types. This apparently contradicts the non-existence of models for the second-order lambda calculus in classical set theory [Rey84]. The contradiction is resolved by working constructively, for example in the internal logic of a topos (cf. [Hyl87, HRR90b, Pit87]).

Some of the results in this paper were obtained in collaboration with Peter Freyd and Pino Rosolini. These are, essentially, those in sections 1-3, and 8. They have been reported elsewhere in our joint [FRR92b], but are included here as an essential part of this story. Peter and Pino's influence on this paper of course extends far beyond these joint results. It has, as always, been instructive, as well as a pleasure, talking to them.

I would also like to express my thanks to many other people for interesting discussions during the work that led up to this paper, but particular mention must go to Qing-Ming Ma, Peter O'Hearn, and Wesley Phoa.

1 Functoriality

The categorical analysis of notions of parametricity is deeply embedded in the concepts of functoriality and naturality (this, in a sense, is precisely what these notions, and hence the whole of category theory, were invented to embody). So when, as a category theorist, one hears that a type is supposed to behave parametrically, one immediately tries to formalise this by requiring that the type be interpreted as a functor. It is obvious what to take as the codomain of this functor: the category of (closed) types. The domain is less obvious, and in any case clearly depends on what one is parametricising over. In the case of the second-order lambda calculus, and where the type in question has one free variable, one's first attempt is most likely to be to try to use the category of closed types again. However, even at this early stage of playing with definitions, one should also be conscious that there is always a degenerate form, in which no parametricity constraints are imposed. This is modeled by taking a discrete category as domain (this has no non-identity morphisms, so functoriality and naturality constraints are satisfied trivially).

Now, at least in the case of the second-order lambda calculus, it is well

known that the first approach fails, but the second succeeds (and gives us the so-called Moggi-Hyland interpretation in which universal types are interpreted as products over `Type`). We were, however, quite surprised to realise that we had no clear picture why the first approach failed.

Let's be a little more specific. We are trying to interpret the category of `C`-parametrised types by the category $\text{Cat}(\mathbf{C}, \mathbf{Type})$ of functors and natural transformations. Now, limits in a functor category are calculated pointwise. So the functor category inherits them from `Type`. This means that if `Type` has products, then so does $\text{Cat}(\mathbf{C}, \mathbf{Type})$. (Note: this remains true for monoidal structures).

However, the analysis above breaks down for function spaces. This is not because of the natural contravariance of the function space in its first argument (at least not directly), but has more to do with the way function spaces in functor categories are calculated.

We recall that $\text{Cat}(\mathbf{C}, \mathbf{T})$ need not be a cartesian closed category simply because `T` is. However, if `C` is small, and the codomain is `Set`, then it is well known that the functor category is cartesian closed (it is a topos of pre-sheaves). The reason is that we can use the Yoneda lemma to calculate the function spaces. We shall use arrow notation, $[X \rightarrow Y]$, to represent the internal hom. If F and G are functors $\mathbf{C} \rightarrow \mathbf{T}$, then their internal hom, $[F \rightarrow G]$, is also a functor. Its value at the object C of `C` can be calculated as:

$$\begin{aligned} [F \rightarrow G](C) &\simeq \text{Set}^{\mathbf{C}}(\mathbf{C}(C, -), [F \rightarrow G]) \\ &\simeq \text{Set}^{\mathbf{C}}(\mathbf{C}(C, -) \times F, G) \end{aligned}$$

the set of natural transformations. Now, this set of natural transformations can be calculated via a limit, the equalizer of

$$\begin{aligned} \prod_{D \in \text{ob}(\mathbf{C})} [(\mathbf{C}(C, D) \times FD) \rightarrow GD] \\ \rightrightarrows \prod_{f \in \text{mor}(\mathbf{C})} [(\mathbf{C}(C, \text{dom } f) \times F(\text{dom } f) \rightarrow G(\text{cod } f))] \end{aligned}$$

where the two arrows correspond to the different ways of going round the naturality square for an archetypical f .

We have put the argument like this in order to make it obvious that it extends to the case when `C` is `T`-enriched. In the type-theoretic setting with which we are chiefly concerned, we can think of enrichment as meaning that the "hom-sets" in `C` are types.

This gives us a number of examples at the expense of a restriction on the parametrising category, albeit one we shall be seeing again later. However, if `T` is sufficiently complete, we can sometimes get function spaces in its functor categories by what amounts to brute force (viz. Freyd's adjoint

functor theorem).

We are often concerned with the case that \mathbb{T} is a small complete category (in some locally cartesian closed category \mathcal{C}). In such a case we have a simple form of the adjoint functor theorem:

if \mathbb{D} is a small category, then any functor $\mathbb{T} \rightarrow \mathbb{D}$ which preserves all small colimits, has a right adjoint.

As usual, we have to be a little careful about how we interpret the internal versions of traditional external concepts. In this case we really need a strong form of completeness. This can be variously expressed as completeness in the sense of \mathcal{C} -indexed category theory, or, using the internal logic as the possession of arbitrary families of limits (see [Rob89]), or as the existence of right Kan extensions to any functor into \mathbb{T} (recall from say Mac Lane [Mac71], that right Kan extensions are calculated pointwise as limits). Now a small category is complete in this sense if and only if it is cocomplete (in the same sense).

If \mathbb{T} is complete and cocomplete, then so is any functor category $\text{Cat}(\mathbb{D}, \mathbb{T})$ (\mathbb{D} a small category), and the limits and colimits are both calculated pointwise. Now if \mathbb{T} is cartesian closed, then all functors $T \times (-) : \mathbb{T} \rightarrow \mathbb{T}$ preserve colimits (since they are left adjoints). Hence in $\text{Cat}(\mathbb{D}, \mathbb{T})$, all functors $F \times (-) : \text{Cat}(\mathbb{D}, \mathbb{T}) \rightarrow \text{Cat}(\mathbb{D}, \mathbb{T})$ preserve colimits (since everything is calculated pointwise). It follows that they have right adjoints. Hence $\text{Cat}(\mathbb{D}, \mathbb{T})$ is cartesian closed. Full details of this are in Freyd et al. [FRR92b].

Thus, the problem is not primarily the existence of function spaces in the functor categories. The problem turns out to be the fact that these function spaces are almost never calculated pointwise. This is disastrous.

2 Pointwise function spaces

If we are modeling parametrised types functorially, then we need function spaces to be calculated pointwise. Suppose F and G are \mathcal{C} -parametrised types. In type-theoretic terms they are given to us by judgements “ $X : \mathcal{C} \vdash F : \text{Type}$ ” and “ $X : \mathcal{C} \vdash G : \text{Type}$ ”. Thus we can form “ $X : \mathcal{C} \vdash [F \rightarrow G] : \text{Type}$ ”. If F and G are modeled by functors, and values by arbitrary natural transformations, then $[F \rightarrow G]$ is modeled by the exponential in the functor category. This is forced by the same argument that for the theory of simple types tells us that the interpretation of a function type must be the exponential in the category. Instantiation at a particular object in \mathcal{C} is modeled by evaluation at that object. Thus, given that C is an object of \mathcal{C} , we can form “ $\vdash F(C) : \text{Type}$ ”. We want our type formation rules to commute with instantiation, so that $[F \rightarrow G](C) \simeq [F(C) \rightarrow G(C)]$. In other words the value of the functor $[F \rightarrow G]$ at C

must be canonically isomorphic to $[F(C) \rightarrow G(C)]$.

This certainly holds true when \mathbf{C} is a discrete category. It also holds more generally, when \mathbf{C} is a groupoid (i.e. every morphism in \mathbf{C} is an isomorphism).

Somewhat surprisingly there is a partial converse.

PROPOSITION 2.1. *If \mathbf{C} is a \mathbf{T} -enriched category, and \mathbf{T} is sufficiently complete for $\mathbf{Cat}(\mathbf{C}, \mathbf{T})$ to have exponentials then these are calculated pointwise if and only if \mathbf{C} is a groupoid.*

Here we again meet the condition that \mathbf{C} be \mathbf{T} -enriched. Note that under these circumstances a sufficient condition for $\mathbf{Cat}(\mathbf{C}, \mathbf{T})$ to have exponentials is that \mathbf{T} has (binary) equalizers, binary products, (so all non-empty finite limits), and products indexed by $\text{ob}(\mathbf{C})$ and $\text{mor}(\mathbf{C})$.

A full proof of this is once again given in our [FRR92b].

Note that the converse fails in the absence of the condition that \mathbf{C} be \mathbf{T} -enriched.

To exhibit a counterexample, note that in many toposes it is possible to find objects A and B such that the only maps from A to B are constant, i.e. the map $\Delta : B \rightarrow B^A$, mapping an element b to the constant function K_b , is an isomorphism. In such a case we say that A is *orthogonal to B* . An instance of this occurs in the effective topos, when A is any uniform object (these are the quotients of $\neg\neg$ -sheaves) with at least two distinct global sections (A could be Ω , or ∇S for any set S with two elements or more), and when B can be any \mathbf{Per} . For a detailed discussion see [HRR90b].

We can construct a one-object category out of A , by picking a global section, and taking the free monoid generated by the object A subject to the given global section being the identity. Let's call this \mathbf{A} . Let \mathbf{T} be the internal category of \mathbf{Per} 's (or, indeed, any full subcategory). A functor from \mathbf{A} to \mathbf{T} is given by the choice of an object X in \mathbf{T} , and a homomorphism of monoids $\mathbf{A} \rightarrow \mathbf{T}(X, X)$. Now, since X is a \mathbf{Per} , so is the hom-set $\mathbf{T}(X, X)$. Moreover, the map from \mathbf{A} is determined by its restriction to A (since it's a homomorphism of monoids). Since \mathbf{A} is orthogonal to $\mathbf{T}(X, X)$, this restriction must be a constant map, and since A contains the identity of the monoid, must map onto the identity of X . It follows that the whole of \mathbf{A} is mapped onto the identity of X . In other words, $\mathbf{Cat}(\mathbf{A}, \mathbf{T}) \cong \mathbf{T}$, and hence function spaces are calculated pointwise.

3 Functoriality with respect to isomorphisms

Since \mathbf{T} is always enriched over itself, Proposition 2.1 tells us that we cannot use functors $\mathbf{T} \rightarrow \mathbf{T}$ to model a theory of parametrized types, unless \mathbf{T} is a groupoid. Now, if \mathbf{T} is a groupoid, then any type with a

global element is isomorphic to the unit type (and consequently has only one element). So this is not particularly useful! However, as we remarked above, we have a degenerate form of parametricity, which corresponds to using the discrete category on the objects of \mathbb{T} (a groupoid since the only morphisms are identities). If we want to push the paradigm as far as we can, we have to find some groupoid canonically associated with \mathbb{T} .

There are two possible ways to go. One is to force every morphism in \mathbb{T} to be an isomorphism. This is useless. It amounts to taking only those functors for which the image of any morphism is an isomorphism. This does not even include the identity functor on \mathbb{T} , the interpretation of a single type variable.

The other possibility is to take the largest subcategory of \mathbb{T} , all of whose maps are isomorphisms. We shall call this \mathbb{T}_{iso} . This, in contrast, does turn out to give models of the second-order lambda calculus. The same definition has also been proposed by Phoa [Pho91], under the name of “the invariant interpretation”.

In this interpretation, types with n free type variables are interpreted as functors $\mathbb{T}_{iso}^n \rightarrow \mathbb{T}$, and values by using arbitrary natural transformations. So if

$$X_1, \dots, X_n; x_1 : A_1, \dots, x_m : A_m \vdash e : B,$$

where the X_i are type variables, and the x_j are individual variables in the context relative to which e (and B) are defined, then B and A_j are interpreted as functors

$$\mathbb{T}_{iso}^n \rightarrow \mathbb{T}$$

and e is interpreted as a natural transformation from the functor $A_1 \times \dots \times A_m$ to the functor B . Note that the objects of this category, the types, can also be regarded as functors

$$\mathbb{T}_{iso}^n \rightarrow \mathbb{T}_{iso},$$

but that this gives a different and unsound definition of the natural transformations.

Substitution at the type level is handled by composition of functors. So, for example, if

$$X, Y_1, \dots, Y_m \vdash A : \mathbf{Type}$$

is interpreted by the functor $\mathcal{A}(X, Y_1, \dots, Y_m)$, and the type

$$X_1, \dots, X_n \vdash B : \mathbf{Type}$$

is interpreted by the functor $\mathcal{B}(X_1, \dots, X_n)$, then the type

$$X_1, \dots, X_n, Y_1, \dots, Y_m \vdash A[X \mapsto B] : \mathbf{Type},$$

given by substituting B in for X in A , is interpreted by the functor

$$F(X_1, \dots, X_n, Y_1, \dots, Y_m) = \mathcal{A}(\mathcal{B}(X_1, \dots, X_n), Y_1, \dots, Y_m).$$

At the value level, substitution is handled by composition of natural transformations. So, for example if

$$X_1, \dots, X_n; x : B \vdash e : C,$$

and

$$X_1, \dots, X_n; y : A \vdash e' : B,$$

then

$$X_1, \dots, X_n; y : A \vdash e[x \mapsto e'] : C,$$

is interpreted by $e \circ e' : \mathcal{A} \rightarrow \mathcal{C}$.

The constant maps are handled by composition again, this time with projections. So if, again $X_1, \dots, X_n \vdash B : \mathbf{Type}$ then $Y, X_1, \dots, X_n \vdash B : \mathbf{Type}$ is interpreted by the functor

$$F(Y, X_1, \dots, X_n) = \mathcal{B}(X_1, \dots, X_n).$$

As always, the definition of this amount of structure determines the interpretation of the quantified types up to isomorphism. In this case they are given by limits. If, as before, $X, Y_1, \dots, Y_m \vdash A : \mathbf{Type}$, then $Y_1, \dots, Y_m \vdash \forall X.A : \mathbf{Type}$ is interpreted by the functor

$$F(Y_1, \dots, Y_m) = \lim_{\longleftarrow \mathbf{T}_{iso}} \mathcal{A}(-, Y_1, \dots, Y_m).$$

If \mathbf{T} has these limits, then the sole remaining requirement, the Beck-Chevalley conditions, certainly hold, at least up to isomorphism. If we take the limit to be the one given by a functor

$$\mathbf{Cat}(\mathbf{T}_{iso}, \mathbf{T}) \longrightarrow \mathbf{T}$$

right adjoint to the diagonal, then the Beck-Chevalley conditions hold on the nose.

4 Categorical equivalence

This definition turns out to have an interesting link with the categorical notion of equivalence.

Let's concentrate for the moment on the case that \mathbf{T} is a small category in a topos (we recall that Pitts has proved a completeness theorem for such models of the second-order lambda calculus [Pit87]).

Now, if we interpret types with free type variables in the Moggi-Hyland-Pitts style, simply as families of types indexed by $\mathbf{ob}(\mathbf{T})$, then quantification is interpreted as product over the objects of \mathbf{T} .

If we replace \mathbb{T} by an equivalent \mathbb{T}' , then a couple of things might happen. First, although, since it is equivalent to \mathbb{T} , \mathbb{T}' has $\text{ob}(\mathbb{T})$ -indexed products, it may not have $\text{ob}(\mathbb{T}')$ -indexed products. Hence \mathbb{T}' may not furnish a model for polymorphism in this sense. Second, even if \mathbb{T}' does have $\text{ob}(\mathbb{T}')$ -indexed products, they are most unlikely to be isomorphic to $\text{ob}(\mathbb{T})$ -indexed products. Thus, although \mathbb{T}' furnishes a model for polymorphism, it is not equivalent to that provided by \mathbb{T} .

Both of these possibilities seem undesirable, and it is easy to cook up *Per*-related examples in which they actually happen.

However, if we insist on functoriality of types, and naturality of values with respect to isomorphisms, then neither of these possibilities arises. If \mathbb{T} furnishes a model of this interpretation of parametric polymorphism, then so does any equivalent \mathbb{T}' , and the equivalence preserves the interpretation of the quantified types. This is essentially because if $G: \mathbb{C} \rightarrow \mathbb{D}$ is an equivalence, and $F: \mathbb{D} \rightarrow \mathbb{T}$ is a diagram in \mathbb{T} of shape \mathbb{D} , then $\varinjlim F \simeq \varinjlim F \circ G$. Hence \mathbb{T}' -*iso*-limits can be calculated as \mathbb{T} -*iso*-limits. (This fact has also been observed by Phoa [Pho91].)

However, there is also a form of converse to this. The form of parametricity we are using seems, in a certain sense, to be as weak as possible subject to the requirement that model structure is preserved by categorical equivalence.

Suppose $\mathbb{T} \xrightleftharpoons[f]{g} \mathbb{C}$ is an equivalence. Then part of the data for this equivalence is a natural isomorphism $\alpha: g \circ f \rightarrow \text{Id}_{\mathbb{T}}$. If $F(X)$ is a type expression, then α generates an endomorphism of the product $\prod X. F(X)$, by

$$\begin{array}{ccc} \prod X. F(X) & \longrightarrow & \prod X. F(X) \\ \downarrow & & \downarrow \\ F(g \circ f(A)) & \xrightarrow[\sim]{F(\alpha_A)} & F(A) \end{array}$$

If we do this for all equivalences, and take the fixed points, then we are left with $\varinjlim F$, where the limit is taken for F as a functor $\mathbb{T}_{iso} \rightarrow \mathbb{T}$.

We can, however, tell a somewhat more convincing story. We shall begin by observing that we can argue that the interpretation of $\forall X. F(X)$ should be as a subobject of the product $\prod X. F(X)$. It should consist of those tuples which are, in a suitable sense, “parametric”.

Now, Reynolds has proposed that the notion of parametricity is linked

to type abstraction. What is parametrised over, or abstracted from, is a set of allowed implementations. In his 1983 paper, [Rey83], Reynolds proposes first a form of abstract type in which one is allowed to declare an abstract type, with no constructors or destructors, or anything to link it to pre-existing types. This can be implemented by giving any (concrete) type. He then goes on to extend this to types with access functions, but no equations between these functions. These can be implemented by giving a type, together with random functions of the appropriate types.

What we are doing in this paper, is, in effect, looking at a different, and rather trivial form of type abstraction. We are allowed to introduce an abstract type together with access functions which set up an isomorphism between our new type and a given pre-existing type. In other words we can take abstract copies of types we have already got. An allowable implementation for an abstract copy of type A consists of a type B , together with inverse isomorphisms $A \rightleftharpoons B$.

We shall discuss this from a linguistic point of view later, but semantically, we shall model it by saying that we are allowed to extend our category of types \mathbb{T} , to a larger category \mathbb{C} . So we have an inclusion $i: \mathbb{T} \rightarrow \mathbb{C}$. This is full and faithful, and since everything in \mathbb{C} is supposed to be a copy of something in \mathbb{T} , it is also essentially surjective. In other words it is part of an equivalence. Now, the process of implementation involves choosing an implementation in \mathbb{T} for each type in \mathbb{C} (with types coming from \mathbb{T} being implemented by themselves). Thus we shall think of an equivalence $\mathbb{T} \xrightleftharpoons[\theta]{i} \mathbb{C}$ such that $\theta \circ i = \text{Id}_{\mathbb{T}}$ as an implementation of \mathbb{C} . Our abstraction principle is that it is impossible to distinguish between different implementations.

Now, if $e: \forall X. F(X)$, we must interpret e in \mathbb{C} . In \mathbb{T} , e is given by a tuple $(e_A)_{A \in \mathbb{T}}$. We must find a tuple $(e_B)_{B \in \mathbb{C}}$. The obvious way to do this is to use the implementation of \mathbb{C} , and define e_B to be the unique element of $F(B)$ mapped onto $e_{\theta(B)}$ by the implementation. More formally, we obtain a map $i(\prod_{A \in \mathbb{T}} F(A)) \rightarrow \prod_{B \in \mathbb{C}} F(B)$, (assuming these exist) by requiring that

$$\begin{array}{ccc}
 i(\prod_{A \in \mathbb{T}} F(A)) & \longrightarrow & \prod_{B \in \mathbb{C}} F(B) \\
 \downarrow i\pi_{\theta B} & & \downarrow \pi_B \\
 i(F(\theta B)) & \xrightarrow{\sim} & F(B)
 \end{array}$$

commute, where the lower horizontal map is that obtained canonically from the equivalence.

Our parametricity requirement is that this interpretation be independent of the choice of implementation.

LEMMA 4.1. *Suppose \mathbb{T} is a category, and that $(e_A)_{A \in \text{ob}(\mathbb{T})}$ is a cone over $(F(A))_{A \in \text{ob}(\mathbb{T})}$, with vertex I . Then the lifting of e to a cone over $(F(B))_{B \in \text{ob}(\mathbb{C})}$ with vertex $i(I)$ is independent of the choice of implementation of \mathbb{C} , for any essentially surjective inclusion $i : \mathbb{T} \rightarrow \mathbb{C}$, if and only if e is a cone over $F : \mathbb{T}_{\text{iso}} \rightarrow \mathbb{T}$.*

Proof One direction is obvious. For the other we in fact need only consider the category \mathbb{C} , whose objects are isomorphisms in \mathbb{T} , and whose morphisms are commutative squares. The inclusion $i : \mathbb{T} \rightarrow \mathbb{C}$ sends A to $(\text{Id}_A : A \rightarrow A)$. e must be stable under the two implementations $\mathbb{C} \rightarrow \mathbb{T}$ given by domain and codomain. \square

Thus, functoriality with respect to isomorphism gives the weakest form of parametricity which respects this form of abstraction.

We can view this as an “external” way of expressing parametricity. We use a family of possible extensions of our collection of types by a new “abstract” type to define our notion of parametricity. We believe that it must be possible to extend this approach to other forms of abstraction worthy of the name. Doing this, of course, involves looking at some other relation between categories than equivalence.

5 Logical relations and dinaturality

In this section we shall explore the connection between logical relations and dinaturality, and then go on to investigate how they both relate to our present framework.

The use of some form of logical relations to formalise the notion of parametricity has been proposed several times, and with varying degrees of abstraction, most notably in the work of Reynolds and his student Ma, [Rey83, MR91].

The use of dinaturality has been proposed by Bainbridge, Freyd, Girard, Scedrov, and Scott, [BFSS90, GSS91].

On an abstract level, the idea behind the use of logical relations is that given an interpretation of the second-order lambda calculus, it is possible to form a category whose objects are (say) n -ary relations, and in which the morphisms come from morphisms between the types on which the relations are defined. Under not too ferocious conditions, analogous to the kind of completeness condition we have not thought twice about using already in

this paper, it is possible to define a way of parametrising types so that this category also carries structure which turns it into a model of the second-order lambda calculus, for details see [MR91]. If the interpretation of polymorphism in this model is suitably related to that in the original one, then the model is said to be parametric. Concretely, this comes down to saying that the interpretations of parametrised terms at n different types have to satisfy certain n -ary relations.

The abstract form of the dinaturality approach is to represent a type with m free type variables as a functor

$$(\mathbb{T} \times \mathbb{T}^{\text{op}})^m \longrightarrow \mathbb{T},$$

and an element as a dinatural transformation. Thus, the contravariance of the function space in its first argument is dealt with by decoupling covariant and contravariant occurrences of the type variables. There are, however, problems with this approach in general caused by the fact that dinaturals do not compose.

These approaches are sufficiently different for it to seem very difficult to relate them in their most abstract form. We can however ask whether they are related for the types of the pure second-order lambda calculus.

Suppose $T(X)$ is a type of the pure second-order lambda calculus (with constant ground types), and that we are dealing with binary logical relations. Then from a relation $R \subseteq A \times B$, we get a relation $T(R) \subseteq T(A) \times T(B)$. This is defined by induction on the structure of T , as follows:

- $T(X) = X$ — then $T(R)$ is the relation R
- $T(X) = A$ — then $T(R)$ is the identity relation on A
- $T(X) = F(X) \times G(X)$ — then the pair $\langle x, y \rangle$ is related by $T(R)$ to the pair $\langle x', y' \rangle$ if and only if $\langle x, x' \rangle \in F(R)$ and $\langle y, y' \rangle \in G(R)$.
- $T(X) = [F(X) \rightarrow G(X)]$ — then $\langle f, g \rangle \in T(R)$ if and only if for all $\langle x, y \rangle \in F(R)$, $\langle fx, gy \rangle \in G(R)$.
- $T(X) = \forall Y. F(X, Y)$ — then $\langle x, y \rangle \in T(R)$ if and only if for all types C , $\langle x[Y \mapsto C], y[Y \mapsto C] \rangle \in F(R, C)$.

Similarly, we can define a functor $T(X, Y) : \mathbb{T}^{\text{op}} \times \mathbb{T} \rightarrow \mathbb{T}$ associated to $T(X)$.

- $T(X) = X$ — then T is projection onto the second coordinate.
- $T(X) = A$ — then T is the constant functor with value A
- $T(X) = F(X) \times G(X)$ — then T is the product of F and G : $T(X, Y) = F(X, Y) \times G(X, Y)$.

- $T(X) = [F(X) \rightarrow G(X)]$ — then $T(X, Y) = [F(Y, X) \rightarrow G(X, Y)]$.
- $T(X) = \forall Z. F(X, Z)$ — then $T(X, Y)$ is the “end”, with respect to variation in Z of the functor $(\mathbb{T}^{\text{op}} \times \mathbb{T})^2 \rightarrow \mathbb{T}$ obtained from F .

Note the contravariant “ $F(Y, X)$ ” in the definition of the interpretation of the function space.

Now suppose we are given such a T , and a morphism $f : A \rightarrow B$ in \mathbb{T} . We shall say that two elements e_A and e_B of $T(A)$ and $T(B)$ are *parametric with respect to f* if $(e_A, e_B) \in T(R)$, where the relation R is the graph of f . We shall say that they are *dinatural* (again with respect to f) if $T(A, f)(e_A) = T(f, B)(e_B)$. Note that this is equivalent to demanding that the following collapsed hexagon commute.

$$\begin{array}{ccccc}
 & & 1 & \xrightarrow{e_A} & T(A, A) \\
 & \nearrow & & & \searrow T(A, f) \\
 1 & & & & T(A, B) \\
 & \searrow & & & \nearrow T(f, B) \\
 & & 1 & \xrightarrow{e_B} & T(B, B)
 \end{array}$$

It is natural to ask whether these two notions are equivalent, and then to try to prove this by induction on the structure of the expression T . Most of the cases in this induction are fine, but it turns out that there are problems with the function space constructor.

To see this, let us suppose that the notions of parametricity and dinaturality with respect to f coincide for the types $F(X)$ and $G(X)$. Suppose we are given $e_A \in [F(A, A) \rightarrow G(A, A)]$ and $e_B \in [F(B, B) \rightarrow G(B, B)]$. We shall show first of all that if e_A and e_B are parametric with respect to f , then they are dinatural, i.e.

$$\begin{array}{ccc}
 & [F(A, A) \rightarrow G(A, A)] & \\
 e_A \nearrow & & \searrow [F(f, A) \rightarrow G(A, f)] \\
 1 & & [F(B, A) \rightarrow G(A, B)] \\
 e_B \searrow & & \nearrow [F(B, f) \rightarrow G(f, B)] \\
 & [F(B, B) \rightarrow G(B, B)] &
 \end{array}$$

commutes. This in turn is equivalent to demanding that the hexagon

$$\begin{array}{ccc}
 & F(A, A) \xrightarrow{\epsilon_A} G(A, A) & \\
 F(f, A) \nearrow & & \searrow G(A, f) \\
 F(B, A) & & G(A, B) \\
 F(B, f) \searrow & & \nearrow G(f, B) \\
 & F(B, B) \xrightarrow{\epsilon_B} G(B, B) &
 \end{array}$$

commute. However, if u is an arbitrary element of $F(B, A)$, then the elements $F(f, A)(u)$ and $F(B, f)(u)$ are automatically dinatural with respect to f , since

$$\begin{array}{ccc}
 & F(A, A) & \\
 F(f, A) \nearrow & & \searrow F(A, f) \\
 F(B, A) & & F(A, B) \\
 F(B, f) \searrow & & \nearrow F(f, B) \\
 & F(B, B) &
 \end{array}$$

commutes. By our assumption about F it follows that they are also parametric.

Since ϵ_A and ϵ_B are parametric, it follows that $\epsilon_A(F(f, A)(u))$ and $\epsilon_B(F(B, f)(u))$ are parametric too (by definition of logical relations for the function space). Hence, since dinaturality and parametricity coincide for G ,

$$G(A, f)(\epsilon_A(F(f, A)(u))) = G(f, B)(\epsilon_B(F(B, f)(u))).$$

In other words, the hexagon above commutes as required.

The problems arise when attempting the converse: showing that dinaturality implies parametricity. Suppose we are given two elements u_A and u_B of $F(A, A)$ and $F(B, B)$, respectively, which are parametric. By assumption, they are also dinatural. This tells us that $F(A, f)(u_A) = F(f, B)(u_B)$. However, in order to make use of the hexagon above, we need to know that u_A and u_B come from the same element of $F(B, A)$. Demanding this condition in general is equivalent to requiring that the square given above for F be a weak pullback. This is not in general the case. (A counterexample in Set is given by $F(X, Y) = [X \rightarrow Y]$, $A = \{*\}$, and $B = \emptyset$.)

This does not quite suffice to establish the inequivalence of the dinaturality and the logical relations form of parametricity, since, at the least, one is quantified with respect to all relations, and the other with respect to all maps. But, it is very suggestive.

Moreover, although we might hope that the analysis above would allow us to prove that parametricity implied dinaturality, that is not the case either. Our proof required that dinaturality for F implied parametricity. Thus we can get at most one or two steps up the type hierarchy.

More specifically, our proof shows that if parametricity implies dinaturality for G , and parametricity is equivalent to dinaturality for F , then parametricity implies dinaturality for $[F \rightarrow G]$. It is easy to show that parametricity is equivalent to dinaturality for types of degree 0 and 1 (such as X , $[X \rightarrow X]$, $[X \rightarrow [X \rightarrow X]]$, \dots). Therefore parametricity implies dinaturality for types of degree 2, the algebraic types. (Recall that variables have degree 0, and that $d([F \rightarrow G]) = \max\{1 + d(F), d(G)\}$, cf. Böhm & Berarducci [BB85].)

We now turn our attention to the case when f is an isomorphism. The situation here is much happier, both as regards the relationship between the different paradigms of parametricity, and to some extent in the behaviour of the paradigms themselves (transformations which are dinatural with respect to isomorphisms compose, unlike those which are dinatural with respect to arbitrary maps). Suppose $G: \mathbb{T}^{\text{op}} \times \mathbb{T} \rightarrow \mathbb{T}$ is an arbitrary functor. Then we can restrict to a functor $G': \mathbb{T}_{\text{iso}} \rightarrow \mathbb{T}$ by defining $G'(f) = G(f^{-1}, f)$ (ie composing with a “diagonal” $\mathbb{T}_{\text{iso}} \rightarrow \mathbb{T}^{\text{op}} \times \mathbb{T}$).

If F and G are two such functors then, making use of the fact that the vertical arrows are isomorphisms, and so can be reversed, we can read

$$\begin{array}{ccc}
 F(A, A) & \xrightarrow{\epsilon_A} & G(A, A) \\
 \uparrow F(f, A) & & \downarrow G(A, f) \\
 F(B, A) & & G(A, B) \\
 \downarrow F(B, f) & & \uparrow G(f, B) \\
 F(B, B) & \xrightarrow{\epsilon_B} & G(B, B)
 \end{array}$$

either as a dinaturality hexagon for F and G , or as a naturality square for F' and G' . Thus, the notions of naturality with respect to isomorphisms,

and dinaturality with respect to isomorphisms coincide, and do so in full generality. (Note that extending this equivalence to polymorphic types entails a small change in their interpretation; instead of taking the end with respect to all maps, we only take it with respect to isomorphisms).

We can not get quite this generality when it comes to logical relations, but we can follow through the inductive definition of the logical relation $T(R)$, to discover that if R is the graph of an isomorphism f , then $T(R)$ is the graph of the isomorphism $T(f)$. Thus, again, the two concepts coincide.

6 Abstract types by copying

In his 1983 paper Reynolds introduces abstract types by means of a “**lettype**” construct. Specifically he uses

$$\mathbf{lettype} \ X = A \ \mathbf{in} \ e$$

where X is a type variable, A is a type expression, and e is an expression which is well-typed in a context in which no individual variable has a type which depends on the type variable X . The type of the expression is $B[X \mapsto A]$, where B is the type of e . Reynolds goes on to observe that no additional strength is obtained by extending this to include access functions for X , since

$$\mathbf{lettype} \ X = A \ \mathbf{with} \ x : F(X) = e' \ \mathbf{in} \ e$$

can be interpreted as

$$(\mathbf{lettype} \ X = A \ \mathbf{in} \ \lambda x : F(X).e)(e').$$

We can, to some extent, think of the first form as being the declaration of an abstract type X , which can be implemented by any concrete type A . Reynolds proves that a relation between implementations extends to a relation which is satisfied by any term (his “Pure Type Definition Theorem”). However, it must be admitted that certain points of this construction are not completely clear to us. For example, it is not clear whether X is free in the **lettype** construct.

One of the ways in which pure type theories differ from programming languages is that they have no definitional mechanism to allow the binding of values (or types) to identifiers. So where, in a functional programming language a program consists of a sequence of declarations setting up an environment, followed by an expression to be evaluated in that environment, a program in a pure type theory is represented simply by an expression to be evaluated. Of course this expression must contain explicitly code that the programming language can refer to implicitly. As an example, the trivial ML program

```
val sq = 4 * 4;
3 * sq;
```

gets represented as

```
let sq = (* 4 4) in * 3 sq
```

or even as

```
(λy. * 3 y)(* 4 4)
```

Operationally, these are not necessarily exact equivalents to the original code. When the original code is compiled, `sq` is evaluated to canonical form before the commencement of the evaluation of “`3*sq`”. In the type-theoretic translation this may or may not be so, depending on the order of evaluation in use.

This is perhaps a rather picky point, but one can see some of the same issues arising in connection with the confusion over the syntax for abstract types. So in normal programming it is quite usual to declare an abstract type (stack say) and then write a program to construct one. Whether it is a useful end in itself to construct a particular abstract stack is another matter. This is an argument supporting the view that X should occur free in the `lettype` construct, i.e. that the type of e should be allowed to contain free occurrences of X .

On the other hand, “`let`” is usually used as a binder. So one expects X to be bound in the `lettype` construct.

We take the view that type declarations are intended to enlarge the range of types available, and that it is possible to return and print out an element of an abstract type, in the same way that it is possible to return and print out an element of a concrete one. The only thing that prevents this is the absence of a type-dependent print function, and there is no reason why we should not supply this. After all the compiler writers have already supplied type-dependent print functions for integers, reals, strings, and the rest. However, we should not cause confusion by using something that looks like a binder.

Moreover, linguistic constructs to support modularity have moved on from abstract types. So it seems that we should make an effort to deal more directly with something like interface specifications as in Extended ML [ST86]. What we need is a kind of abstract structure mechanism.

A full detailed solution to this problem is beyond the scope of this paper, so we shall merely make a nod in its direction. We adopt a simple one-level approach. The basic idea is to use the context to allow us to declare (and implement) structures.

First of all, as far as we are concerned a structure specification will

consist of

- a list of names for types
- a list of names for operations, with types
- a collection of axioms which the structure must satisfy.

A convenient way of setting up the second order lambda calculus is to use judgements of the form

$$X_1, \dots, X_n; x_1 : \sigma_1, \dots, x_m : \sigma_m \vdash e : \sigma$$

where “ X_1, \dots, X_n ” is a list of type variables, “ $x_1 : \sigma_1, \dots, x_m : \sigma_m$ ” is a list of bindings of individual variables to types, e is an expression, and σ is its type. We add a third component to the context: a list of structure declarations. So our judgements will now take the form

$$\Gamma; \Delta; \Sigma \vdash e : \sigma$$

where Σ is a list of structure declarations.

It is useful to assume that as well as being given sets of type and individual variables, we are given sets of type and individual identifiers to use in structures (though the structures themselves will be anonymous). This removes one source of name clashes, and avoids some problems with weakening rules.

So, if we have a valid context “ $\Gamma; \Delta; \Sigma$ ”, then we can enlarge it by adding the new structure

$$\left\{ \begin{array}{l} A(X_{a_1}, \dots, X_{a_k}) \text{ type, } \dots; \\ f_1 : \sigma_1, \dots; \\ \text{axioms } \dots \end{array} \right\}$$

where the A and f are taken from the new identifiers, excluding those already used in Σ , the X_i are type variables listed in Γ , and the σ_i are types constructible from the type variables listed in Γ , and the types listed in the other structures in Σ . We place no restriction on the possible form of the axioms that these structures are to be required to satisfy. The intention behind allowing $A(X_{a_1}, \dots, X_{a_k})$, and not just A , is that A be allowed to depend on types “ X_{a_1}, \dots, X_{a_k} ”.

We must also add rules to say that if A is a type declared in a structure, then A is a usable type

$$\Gamma; \Sigma \vdash A : \text{Type},$$

and if f is a value declared in a structure to have type σ then f is a usable value

$$\Gamma; \Delta; \Sigma \vdash f : \sigma.$$

So far, this allows us to use types and values from structures without committing ourselves to implementations of the structures. In other words, the structures are treated abstractly. However, in order to get working code we need to supply implementations of the structures used. We do this by attaching them to the declarations.

$$\Gamma; \Delta; \Sigma, S = I \vdash e : \sigma$$

where “ $S = I$ ” is of the form

$$\left\{ \begin{array}{l} \mathbf{A \ type, \dots;} \\ \mathbf{f_1 : \sigma_1, \dots;} \\ \mathbf{axioms \dots} \end{array} \right\} = \left\{ \begin{array}{l} \mathbf{A = \sigma_A,} \\ \mathbf{\dots;} \\ \mathbf{f_1 = e_1, \dots} \end{array} \right\}$$

Of course we require that the types of the expressions used to implement the values f are compatible with the types used to implement the sorts A . We also require that the implementation satisfy the axioms listed in the declaration of the structure.

So if $\Gamma; \Delta : \Sigma \vdash e : \sigma$, then the semantics of e is defined only relative to implementations of the structures in Σ . Once these implementations are given, the semantics of e is given by the obvious concretion. That is, we use

$$e[A \mapsto \sigma_A, \dots, f_i \mapsto e_i, \dots] : \sigma[A \mapsto \sigma_A].$$

In this paper, we shall only be interested in a special case, even of this grossly oversimplified view of the use of structures. This is the case in which we are only allowed to introduce structures containing one new type, and an isomorphism between that and some specified type.

$$\left\{ \begin{array}{l} \mathbf{A(\vec{X}) \ type;} \\ \mathbf{in : [B \rightarrow A], out : [A \rightarrow B];} \\ \mathbf{axioms \ \lambda x : B. out(in(x)) = \lambda x : B. x, \ \lambda y : A. in(out(y)) = \lambda y : A. y} \end{array} \right\}$$

We insist, moreover, that all type variables appearing free in B be listed in \vec{X} , in order to make sure that this does not place an implicit constraint on the interpretation of B for different values of its type variables. We shall call this extension, “second order lambda calculus with abstract copies”.

So an interpretation of this structure consists of a type isomorphic to B together with a designated pair of inverse isomorphisms.

Suppose we write S for the structure given above. If we have two different implementations of S , say

$$I = \{A = A; \text{ in} = f, \text{ out} = g\}$$

and

$$I' = \{A = A'; \text{ in} = f', \text{ out} = g'\},$$

then there is a canonical isomorphism $A \rightarrow A'$, viz. $f' \circ g$, with inverse $f \circ g'$.

Now, any model of the second order lambda calculus extends to the calculus with abstract copies. Moreover, if we have

$$\Gamma; \Delta; \Sigma \vdash e(A) : \sigma(A),$$

then σ is functorial with respect to isomorphisms in the interpretation of A . Let's write e_I and $e_{I'}$ for the interpretation of e relative to implementations I and I' . We shall say that the model satisfies the “*abstraction criterion*” if the interpretation of any expression e is natural with respect to isomorphisms in the interpretation of A , i.e. if the value e_I is functorially related to $e_{I'}$:

$$e_{I'} = \sigma(f' \circ g)(e_I).$$

Note that if e and σ do not depend on A , then this becomes

$$e_{I'} = e_I.$$

In other words, the interpretation of e is independent of the implementation of A , which is to say that as far as e is concerned, A is abstractly interpreted.

We can now state the analogue of the “abstraction theorem” proved by Reynolds in [Rey83].

PROPOSITION 6.1. *If $\Gamma; \Delta; \Sigma \vdash e(A) : \sigma(A)$ in the second order lambda calculus with abstract copies, then e is abstractly interpreted in any model.*

Proof As in Reynolds' result, the proof is by structural induction on e . Note that this result, in contrast to the full abstraction result in the next section, refers only to terms of the pure calculus. \square

7 A full abstraction theorem

Suppose we start with a model for the second order lambda calculus, which we extend to carry an interpretation of the syntax of the calculus with abstract copies. Part of the extension process involves extending the interpretation of polymorphic values to the new abstract types. If we demand that all the resulting model satisfies our abstraction criterion, then this places a constraint on the interpretation of the polymorphic types in the original model. The purpose of this section is to investigate this constraint. What we aim to do is to prove a kind of full abstraction theorem for the interpretation of polymorphic values as transformations which are natural with respect to isomorphisms.

Suppose we have a value v in our model, of type $\forall X.F[X]$, where $F[X]$ is a type in the pure second-order lambda calculus, then we will explore the constraints imposed by demanding that the interpretation of v not lead to a violation of the abstraction criterion. Note that by Proposition 6.1, this will only be interesting when v is not denotable by an expression in the calculus.

Specifically, we consider contexts $C[\]$ in the calculus with abstract copies, in which the “hole” takes type $\forall X.F[X]$ (and so is suitable for v), and where the whole context takes concrete type (i.e. one from the pure second order lambda calculus). We shall show that if for any such context $C[v]$ is interpreted abstractly, then v is natural with respect to isomorphisms.

This is a kind of full abstraction theorem. We are using facts about one interpretation to characterise the kind of element allowed in a denotational model. It is usual in this kind of result to have use contexts which produce some “observable”. In this case, we are using any element of a concrete type as an observable. We believe this makes sense because what we are trying to do is take a model of “concrete” types, load a notion of “abstract” type on top of that, demand that the “abstract” types behave abstractly, and investigate what constraints this places on the original model. We are not trying to investigate extensionality properties of the original model compared to some collection of ground types.

PROPOSITION 7.1. *Suppose v is an element of the pure second order type $\forall X.F[X]$, then the interpretation of $C[v]$ is independent of the implementation of any structures used in $C[\]$, for all contexts $C[\]$ in the calculus with abstract copies, such that $C[v]$ is well-typed, and represents an element of a pure second order type, if and only if v is natural with respect to isomorphisms.*

Proof The “if” direction is obvious. In the “only if” direction, suppose we want to show that v is natural with respect to the isomorphism $f : B \rightarrow A$. We consider the structure

$$S = \{\mathbf{A} \text{ type}; \text{in}:[B \rightarrow A], \text{out}:[A \rightarrow B]; \text{axioms}\dots\}.$$

We use **in** and **out** as inverse isomorphisms to create a context which instantiates v at **A**, to give $v[\mathbf{A}]$, and then applies the function $F[\mathbf{out}]$ to give an element of $F[\mathbf{B}]$. Now interpret this relative to the implementations

$$I = \{\mathbf{A} = B; \text{in} = \text{ld}_B, \text{out} = \text{ld}_B\}$$

and

$$I = \{\mathbf{A} = A; \text{in} = f, \text{out} = f^{-1}\}.$$

The former gives exactly $v[B]$ and the latter $F[f^{-1}](v[B])$. These are equal if and only if v is natural with respect to f . \square

Thus functoriality with respect to isomorphism is the least constraint we can impose on parametrized types which ensures that our copies are indeed treated abstractly.

8 On not using Per

In this section we investigate the pro's and con's of using **Per**-like models to investigate forms of parametricity. More is known in the context of these models than in any other, but that is still not very much.

We have known for some time that the interpretations of some of the algebraic types in the classical **Per** model of the natural numbers were the initial algebras (see [HRR90a, FRR92a]). This has been proposed as a test of the degree of parametricity of a model. However, the proofs at least seem to depend on the particular realizability algebra on which the model is based (natural numbers with Kleene application). This raises the unpleasant possibility that the degree of parametricity of these models may depend on the underlying realizability algebra.

We do not know whether uniform families of maps are always dinatural. (This is exactly equivalent to the coincidence of the dinaturality interpretation and the Moggi-Hyland interpretation). However, it is not hard to see that a family of maps which is natural with respect to isomorphisms is dinatural with respect to all maps (and thus the naturality with respect to isomorphism and dinaturality interpretations coincide).

LEMMA 8.1. *Suppose F and G are internal functors $\mathbf{Per}^{\text{op}} \times \mathbf{Per} \rightarrow \mathbf{Per}$, and $(\sigma_A : F(A, A) \rightarrow G(A, A))_{A \in \text{ob}(\mathbf{Per})}$ is an internal family of maps which is (di)natural with respect to isomorphisms, then it is dinatural with respect to all maps.*

Proof We recall that naturality and dinaturality coincide for isomorphisms. We also recall that the set of maps with respect to which a given family is dinatural is closed under composition. The proof is now immediate from Freyd's observation that maps in **Per** factor as an isomorphism, followed by a map realised by the identity, followed by a second isomorphism. \square

Note that this does not depend on the underlying realizability algebra.

A similar, but slightly more complex, proof shows that if a transformation between internal functors is natural with respect to isomorphisms, then it is natural with respect to all maps. For details see Freyd et al. [FRR92b].

Also, we should note that although we do not know whether families in Per are automatically dinatural, the main result in [FRR92a] is that there are large and well-behaved subcategories of Per for which this is so. For these categories, three of the four interpretations of polymorphism we have been considering coincide, the exception being the Reynolds logical relations form.

Finally, we note that we know next to nothing about the Reynolds or Ma-Reynolds approaches in this context.

9 Towards a more general theory

The theory we have proposed has some features which we expect will generalise to a full-blown theory, and some which we expect won't. As we stated above, we would like to see some external form of semantics for type abstraction in which the abstract types are interpreted as elements of an enlarged universe of types. We would also like to see a relationship between a linguistic theory and rules for reasoning about abstract types, and the parametric semantics. On the other hand, it is fairly clear that the extraordinary coincidence of different definitions we have seen here will not extend to a more general setting.

One point that we have tried to stress throughout this paper is that a general theory which tries to relate parametricity to a form of type abstraction must be able to cope with differing degrees of abstraction. This should include a trivial form in which everything is concrete. In our opinion this is well modeled by the Moggi-Hyland approach of taking products. We would also, of course, like to see it encompass our form of type abstraction by copying. Finally, it should encompass more general and more practical forms.

Most of the approaches to parametricity fall down by attempting to find a single correct definition. One which doesn't fall into this trap is the recent work of Reynolds and Ma [MR91]. A relative notion of parametricity emerges in this paper almost as a side-effect. The authors are interested in carrying out a logical relations approach for a general PL-category. Now, it is most unlikely that the PL-category itself will contain "enough" relations (it will only contain the ones which are themselves representable as types, but once again this includes the graphs of isomorphisms). The obvious thing to do to get round this is to embed the category of types in some larger category (of sets, say) in which it is possible to find enough relations. This is what Reynolds and Ma do. Thus they arrive at a notion of parametricity which is defined relative to a category from which relations are taken. For a strong theory, this may be much larger than the category of types. However, their formalism does

not enforce a strict embedding of the category of types in the “larger” relational homeland, but also allows for some collapse. Taken to extreme, they allow one to take relations from the terminal category. This gives a vacuous notion of parametricity, with respect to which every PL-category is parametric.

However, one way in which this approach fails to be as general as possible is to take *all* relations from a particular category. This prevents Ma and Reynolds from modeling our approach.

Quite recently, the work of Reynolds and Ma has been generalised by O’Hearn and Tennent, attempting to find a semantics for local variables. O’Hearn and Tennent suggest decoupling the operation of type constructors on relations from their functoriality. Technically this involves using a double category. Their approach can certainly be applied in our setting, where it gives the most general formalism yet.

References

- [BB85] C. Böhm and A. Berarducci. Automatic synthesis of typed λ -programs on term algebras. *Theor. Comp. Sci.*, 39:135–154, 1985.
- [BFSS90] E.S. Bainbridge, P.J. Freyd, A. Scedrov, and P.J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1990. corrigendum 71(3):431, 1990.
- [FRR92a] P.J. Freyd, E.P. Robinson, and G. Rosolini. Dinaturality for free. In M.P. Fourman, P.T. Johnstone, and A.M. Pitts, editors, *Applications of Categories in Computer Science, Proceedings of the LMS Durham Symposium on Categories in Computer Science, 20-30 July 1991*, pages 107–118, Cambridge, 1992. Cambridge University Press.
- [FRR92b] P.J. Freyd, E.P. Robinson, and G. Rosolini. Functorial parametricity. In A. Scedrov, editor, *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science, June 22-25, 1992, Santa Cruz, California*, pages 444–452, Los Alamitos, California, 1992. IEEE Computer Society Press.
- [GSS91] J-Y. Girard, A. Scedrov, and P.J. Scott. Normal forms and cut-free proofs as natural transformations. In Y.N. Moschovakis, editor, *Logic From Computer Science*, pages 217–241. Mathematical Sciences Research Institute Publications (V.21), Springer Verlag, 1991.
- [HRR90a] J.M.E. Hyland, E.P. Robinson, and G. Rosolini. Algebraic types in PER models. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics, Lecture Notes in Computer Science 442*, pages 333–350. Springer-Verlag, 1990.
- [HRR90b] J.M.E. Hyland, E.P. Robinson, and G. Rosolini. The discrete objects in the effective topos. *Proceedings of the London Mathematical Society*, 60(3):1–36, 1990.
- [Hyl87] J.M.E. Hyland. A small complete category. In *Proc. of the Conference on Church’s Thesis: Fifty Years Later*, 1987.
- [Mac71] S. Mac Lane. *Categories for the Working Mathematician*. Springer Verlag, 1971.

- [MR91] Qing-Ming Ma and J.C. Reynolds. Types, abstraction, and parametric polymorphism, part 2. In *Proceedings of the 1991 Mathematical Foundations of Programming Semantics Conference*. Springer-Verlag, 1991.
- [Pho91] W.K.S. Phoa. Two results on set-theoretic polymorphism. In D.H. Pitt, A. Pitts, A. Poigné, and D.E. Rydeheard, editors, *Category theory and Computer Science, Paris 1991*. Springer Verlag, 1991.
- [Pit87] A.M. Pitts. Polymorphism is set-theoretic, constructively. In D.H. Pitt, et al., editor, *Category Theory and Computer Science*, number 283 in Lecture Notes in Computer Science, pages 12–39. Springer-Verlag, Berlin, 1987.
- [Rey83] J.C. Reynolds. Types, abstraction, and parametric polymorphism. In R.E.A. Mason, editor, *Information Processing '83*, pages 513–523. North Holland, 1983.
- [Rey84] J.C. Reynolds. Polymorphism is not set-theoretic. In G. Kahn, et al., editor, *Semantics of Data Types*, Lecture Notes in Computer Science 173, pages 145–156. Springer-Verlag, Berlin, 1984.
- [Rob89] E.P. Robinson. How complete is PER? In A. Meyer, editor, *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 106–111. Computer Society Press of the IEEE, 1989.
- [ST86] D.T. Sannella and A. Tarlecki. Extended ML: an institution-independent framework for formal program development. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, editors, *Category Theory and Computer Programming, Proceedings 1985*, Lecture Notes in Computer Science 240, pages 364–369, 1986.