

Semantics of Parametric Polymorphism in Imperative Programming Languages

Brian P. Dunphy¹ and Uday S. Reddy²

¹ Department of Mathematics, University of Illinois at Urbana-Champaign

² School of Computer Science, University of Birmingham

Abstract. Programming languages such as CLU, Ada and Modula-3 have included facilities for parametric polymorphism and, more recently, C++ and Java have also added similar facilities. In this paper, we examine the issues of defining denotational semantics for imperative programming languages with polymorphism. We use the framework of reflexive graphs of categories previously developed for a general axiomatization of relational parametricity constraints implicit in polymorphic functions. We specialize it to the context of imperative programming languages, which in turn involve parametricity constraints implicit in local variables. The two levels of parametricity inherent in such languages can be captured in a pleasing way in "higher-order" reflexive graphs.

1 Introduction

Imperative programming languages, dating back to CLU in 1976 [15], have incorporated parametrized types and polymorphic functions. Functional programming languages such as ML [10], Hope [5] and Haskell [13] also use parametric polymorphism in an essential way. But note that ML includes imperative features as well, as do some variants of Haskell. Other imperative languages incorporating polymorphic features include Modula-2 to a limited extent, Ada and Modula-3. More recently, C++ and Java have added parametric polymorphism via "templates" and "generics".

Parametric polymorphism provides information hiding. When a procedure is given an argument whose type is expressed in terms of a type variable α , all the α -typed information in the argument is hidden from the procedure. The idea can be made explicit by using abstract data types (also provided in CLU and its successors). An abstract data type is given a concrete representation inside its implementation module but, outside the module, it is treated as an unknown type or "opaque" type, in essence as a type variable α . The remainder of the program is parametrically polymorphic in α . It cannot access any of the internal structure of the α -typed information.

The semantics of parametric polymorphism has been studied in detail in the context of functional programming [1, 23, 17, 27, 31]. However, its semantics in the imperative programming context has received scant attention. The problem is nontrivial. Imperative programming languages already have elements of parametricity and information hiding implicit in local variables [16, 20, 18, 25]. That

is how object-oriented programming is able to provide information hiding even without any explicit type parameters [24]. Parametric polymorphism provides a second layer of information hiding *in addition* to that of local variables. This signifies a rich semantic structure in such programming languages. In this paper, we provide what seems to be the first account of this semantic structure.

The general scheme is as follows. To account for the information hiding implicit in local variables, imperative languages are given a form of possible world semantics using functor categories enriched with relational parametricity [26, 22, 20, 25]. To account for type-based polymorphism and information hiding, we need to construct again what one might call “second-order functors”, enriched with relational parametricity, over the structures obtained in the previous layer. To carry out such sophisticated constructions, one needs a suitably general mathematical theory of functors enriched with relational parametricity. Such a theory was defined in our previous work [8, 9] based on O’Hearn and Tennent’s reflexive graph categories [20]. We make essential use of this framework.

After defining the constructions, we were surprised to discover that the traditional Reynolds-Oles models of imperative programming languages do not provide the ideal structure for this semantics. By treating states abstractly, their models admit too many morphisms which have no counterparts in the programming language. We show that, by paring down the models to location-based models, which might at first sight appear to be low-level or “non-uniform”, we actually obtain a better account of what is representable in the programming language.

2 Motivation

Consider defining a generic sorting procedure, which can work for arrays of all kinds of elements. The sorting procedure will know nothing about the elements. So, we also need to provide it a comparison operation for the elements. The type of such a procedure may be given as follows:

```
sort : ∀t. (t * t → bool) * (array t) * int → comm
```

The prefix $\forall t$ signifies that `sort` is parametrized by a type (here referred to as `t`). The first argument is a comparison operation for `t`-typed objects. The remaining arguments are an array of `t`-typed objects and its size. The procedure does not return any result; `sort` applied to these arguments has the status of a command, which when executed, rearranges the elements in the array.

Suppose that we have an abstract data type of points. A type called “point” is defined in a module, along with various operations on points. The signature of operations on points might be defined as:

```
Pointsig(point) =
  {newpoint : (point → comm) → comm,
   xcoord, ycoord, distance, angle : point → real,
   translate : point * real * real → comm,
   scale, rotate : point * real → comm,
   comp : point * point → bool }
```

The module implementing the point type itself has a type of the form

```
∃point. Pointsig(point)
```

The prefix `∃point` signifies that the module defines a type (here referred to as `point`) and provides a record of operations on this type.

A main program can use the module to create a collection of points, populate an array with these and call the sorting procedure by instantiating `t` \mapsto `point` and passing the `comp` function provided by the module as the first argument. All of this can be done without any knowledge of how `point` is represented inside the module. Likewise, `sort` can sort the array of points without such knowledge.

It is important to keep in mind that a “point” is not just a pair of data values. The points are mutable, whose state can be changed by operations like `translate` and `rotate`. A typical representation of point would be `var [real] * var [real]`, i.e., as a pair of storage variables that store the attributes of points, may they be cartesian coordinates or polar coordinates.

It is a fundamental property of parametrically polymorphic functions like `sort` that they are insensitive to the actual types that the type variables are instantiated to. This is captured mathematically as the parametricity condition [27]: If there is any relation between two potential instantiations of the type variable `t`, which is respected by all the arguments of the procedure, then the action of the procedure respects them too. So, for example, if `t` is instantiated to a cartesian representation of `point` or a polar representation of `point`, as long as the comparison operations provided respect the mathematical relationship between them, the `sort` procedure produces exactly the same sort.

However, it is not immediately apparent how to formulate the relation between two representations of `point`. Since the representations are types defined on the basis of store, rather than just data values, we cannot directly use the mathematical relationship between cartesian and polar coordinates. A point object is a piece of the store. So, to say that two *point types* are related, one must say that, for all possible stores and all point objects potentially stored in them, the corresponding point objects bear the mathematical relationship. How to formulate such relations between types is the main concern of the semantics defined in this paper.

3 Reflexive graphs of categories

Reynolds’s relational parametricity, formulated in a set-theoretic setting, involves relations of the form $R \rightarrow S \subseteq (A \rightarrow B) \times (A' \rightarrow B')$, defined by

$$f [R \rightarrow S] f' \iff \forall x \in A, x' \in A'. x [R] x' \implies fx [S] f'x'$$

We depict a fact $f [R \rightarrow S] f'$ diagrammatically as a square of the form:

$$\begin{array}{ccc}
 A & \xrightarrow{f} & B \\
 R \uparrow & \phi & \downarrow S \\
 A' & \xrightarrow{f'} & B'
 \end{array} \tag{1}$$

(where the top-to-bottom order is significant, unlike in categorical diagrams). Relation-preservation properties of this form are involved in formulating the uniformity of parametric families of functions, which are used in addition to, or in place of, the uniformity involved in the naturality conditions. Intuitively, this involves two degrees of freedom:

- functions, represented by the horizontal arrows in (1), represent the “operations that may be performed”, and
- relations, represented by the vertical arrows in (1), represent the “properties that need to be preserved”.

Traditional category theory incorporates only a single degree of freedom, that of morphisms, and is unable to capture relational parametricity conditions. O’Hearn and Tennent [20] proposed the framework of *reflexive graphs* (of categories) to address this issue. (In [11], fibrations are proposed to achieve a similar effect. Birkedal and Møgelberg have also used reflexive graph-based models for Plotkin-Abadi logic.)

The reflexive graph **Set** has the category of sets and functions as vertex category \mathbf{Set}_v and the category of binary relations and relation-preserving pairs of functions as the edge category \mathbf{Set}_e . The identity edge I_A is the equality relation of A . This reflexive graph is relational.

The following definitions are from our previous work [8, 9]. Let $\Delta : \mathbf{H} \rightarrow \mathbf{H}^{\mathbf{G}}$ be the evident diagonal RG-functor. The *parametric limit* and *parametric colimit* functors are defined by:

$$\begin{array}{lll}
 \text{parametric limits} & \forall : \mathbf{H}^{\mathbf{G}} \rightarrow \mathbf{H} & \text{right adjoint to } \Delta \\
 \text{parametric colimits} & \exists : \mathbf{H}^{\mathbf{G}} \rightarrow \mathbf{H} & \text{left adjoint to } \Delta
 \end{array}$$

So, $\forall_X F(X)$ is a vertex of \mathbf{H} together with a parametric natural transformation $\omega : \forall_X F(X) \rightarrow F$ that is universal in the sense that any other parametric natural transformation $A \rightarrow F$ uniquely factors through ω . $\exists_X F(X)$ is the evident dual concept. The reflexive graph **Set** has all small parametric limits and colimits [8, 9] and these correspond to the notions used in [27, 17].

In our application to the semantics of polymorphic imperative languages, these parametric limit and colimit functors get used at two levels. First, the semantics of the basic imperative language is given in a functor reflexive graph $\mathbf{W} \rightarrow \mathbf{S}$ where the local variable declarations and function type interpretations involve parametric limits $\forall_{\mathbf{W}}$ and class constructions involve parametric colimits

\exists_W , as in [24, 25]. Here W ranges over the vertices of \mathbf{W} . Second, the semantics of polymorphic functions is given in functor graphs of the form $(\mathbf{S}^{\mathbf{W}})^{(\mathbf{S}^{\mathbf{W}})}$. The interpretation of polymorphic functions involves parametric limits \forall_T and the interpretation of abstract types involves parametric colimits \exists_T . In this case, T ranges over the vertices of $\mathbf{S}^{\mathbf{W}}$.

While reflexive graphs identify the structure necessary for interpreting relational parametricity constraints, they do not provide any existential conditions to ensure that parametricity is actually ensured. In [8, 9], three axioms were identified to obtain a satisfactory interpretation of parametric polymorphism and the resulting structures were termed parametricity graphs. A *parametricity graph* is a relational reflexive graph that is fibred with a chosen cleavage and satisfies the identity condition of [?]. These structures form a 2-subcategory \mathbf{PG} of the 2-category of reflexive graphs, with 1-cells called PG-functors and 2-cells being parametric transformations (which are automatically natural). All the initial algebra and final coalgebra results normally expected of parametricity hold in such structures.

3.1 Interpreting predicative polymorphism

A predicative polymorphic lambda calculus has a type structure of the form:

$$\begin{array}{ll} \text{(simple types)} & \tau ::= \alpha \mid \beta \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \\ \text{(polymorphic types)} & \sigma ::= \tau \mid \forall \alpha. \sigma \mid \exists \alpha. \sigma \end{array}$$

where α stands for type variables and β stands for primitive types (which are left unspecified). The label “predicative” signifies the fact that types are organized into two layers. Type variables are presumed to range over simple types, not polymorphic types. The polymorphic type systems of most practical programming languages, including that of ML, fit this pattern.

The type syntax of the language is described by judgement of the form:

$$\Sigma; \Gamma \vdash M : \tau \quad \Sigma; \Gamma \vdash M : \sigma$$

where Σ is a sequence of type variables α_i , called the *type context*; Γ is a sequence of typed variables $x_i : \sigma_i$, called the *typing context*; M is a term. All the type variables used in a judgment must be included in its type context Σ . See Fig. 1 for the type syntax of terms. The syntax is parametrized by a set of constants chosen for the language. We cover them in the next section for Polymorphic Idealized Algol. The function FTV gives the free type variables in type terms.

A *parametric categorical model* for the predicative calculus consists of:

- a parametricity graph \mathbf{G} , and
- a small cartesian closed parametricity subgraph \mathbf{K} of \mathbf{G} (with an inclusion functor $J : \mathbf{K} \rightarrow \mathbf{G}$)

such that there are parametric limit and parametric colimit functors $\forall, \exists : \mathbf{G}^{|\mathbf{K}|} \rightarrow \mathbf{G}$, where $|\mathbf{K}|$ denotes the discrete reflexive graph of \mathbf{K} (obtained by

$\overline{\Sigma; \Gamma \vdash x : \tau}$ if $(x : \tau)$ is in Γ	$\overline{\Sigma; \Gamma \vdash k : \tau}$ if $k : \tau$ is a constant
$\overline{\Sigma; \Gamma, x : \tau \vdash M : \tau'}$	$\overline{\Sigma; \Gamma \vdash M : \tau \rightarrow \tau' \quad \Sigma; \Gamma \vdash N : \tau}$
$\overline{\Sigma; \Gamma \vdash \lambda x. M : \tau \rightarrow \tau'}$	$\overline{\Sigma; \Gamma \vdash MN : \tau'}$
$\overline{\Sigma, \alpha; \Gamma \vdash M : \sigma}$	$\overline{\Sigma; \Gamma \vdash M : \forall \alpha. \sigma}$
$\overline{\Sigma; \Gamma \vdash \Lambda \alpha. M : \forall \alpha. \sigma}$ if $\alpha \notin FTV(\Gamma)$	$\overline{\Sigma; \Gamma \vdash M[\tau] : \sigma[\tau/\alpha]}$
$\overline{\Sigma; \Gamma \vdash M : \sigma[\tau/\alpha]}$	
$\overline{\Sigma; \Gamma \vdash \mathbf{abstype} \alpha = \tau \mathbf{with} M : \exists \alpha. \sigma}$	
$\overline{\Sigma; \Gamma \vdash M : \exists \alpha. \sigma \quad \Sigma, \alpha; \Gamma, x : \sigma \vdash N : \sigma'}$	if $\alpha \notin FTV(\Gamma) \cup FTV(\sigma')$
$\overline{\Sigma; \Gamma \vdash \mathbf{open} M \mathbf{as} \alpha \mathbf{with} x : \sigma \mathbf{in} N : \sigma'}$	

Table 1. Type syntax of terms

dropping all non-identity morphisms). The reasons for dropping all non-identity morphisms are that, first, parametricity has all the uniformity that we need, and, second, the type expressions are not functorial in the type variables.

A simple type expression with n type variables is interpreted as a PG-functor $|\mathbf{K}|^n \rightarrow \mathbf{K}$:

$$\begin{aligned} \llbracket \alpha_i \rrbracket &= \Pi_i && \text{(the } i\text{th projection)} \\ \llbracket \tau_1 \times \tau_2 \rrbracket &= \times \circ \langle \llbracket \tau_1 \rrbracket, \llbracket \tau_2 \rrbracket \rangle \\ \llbracket \tau_1 \rightarrow \tau_2 \rrbracket &= \Rightarrow \circ \langle \llbracket \tau_1 \rrbracket, \llbracket \tau_2 \rrbracket \rangle \end{aligned}$$

A polytype expression with n type variables is interpreted as a PG-functor $|\mathbf{K}|^n \rightarrow \mathbf{G}$:

$$\begin{aligned} \llbracket \tau \rrbracket &= J \circ \llbracket \tau \rrbracket \\ \llbracket \forall \alpha_{n+1}. \sigma \rrbracket &= \forall^{|\mathbf{K}|^n} (\llbracket \sigma \rrbracket) \\ \llbracket \exists \alpha_{n+1}. \sigma \rrbracket &= \exists^{|\mathbf{K}|^n} (\llbracket \sigma \rrbracket) \end{aligned}$$

where $\forall^{|\mathbf{K}|^n}, \exists^{|\mathbf{K}|^n} : (\mathbf{G}^{|\mathbf{K}|^n})^{|\mathbf{K}|} \rightarrow \mathbf{G}^{|\mathbf{K}|^n}$ are the parametric limit and parametric colimit functors. The interpretation of terms can be given in the evident fashion using the combinators associated with parametric limits and colimits.

4 Polymorphic Idealized Algol

To demonstrate the semantics of polymorphic imperative programs, we instantiate the predicate polymorphic calculus of Section 3.1 with the language Idealized Algol defined by Reynolds [26]. Reynolds's design, based on a type-theoretic analysis of the Algol 60 programming language, identifies two layers of types:

- Data types (δ), denoting types of values storable in program variables.
- Phrase types (τ), denoting types of program phrases, including expressions, commands, procedures/functions and modules.

The syntax of types is given by:

$$\begin{aligned} \text{(data types)} \quad \delta &::= \text{bool} \mid \text{int} \mid \text{real} \mid \dots \\ \text{(phrase types)} \quad \tau &::= \mathbf{exp}[\delta] \mid \mathbf{var}[\delta] \mid \mathbf{comm} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \end{aligned}$$

To construct a polymorphic language we add type variables (α) to phrase types, and supplement the type system with the layer of polymorphic types as in Sec. 3.1.

We use the following sample of constants for representing imperative computations:

$$\begin{aligned} 0 &: \mathbf{exp}[\text{int}] \\ + &: \mathbf{exp}[\text{int}] \times \mathbf{exp}[\text{int}] \rightarrow \mathbf{exp}[\text{int}] \\ \mathbf{skip} &: \mathbf{comm} \\ \mathbf{seq} &: \mathbf{comm} \times \mathbf{comm} \rightarrow \mathbf{comm} \\ \mathbf{assign}_\delta &: \mathbf{var}[\delta] \times \mathbf{exp}[\delta] \rightarrow \mathbf{comm} \\ \mathbf{deref}_\delta &: \mathbf{var}[\delta] \rightarrow \mathbf{exp}[\delta] \\ \mathbf{newvar}_\delta &: (\mathbf{var}[\delta] \rightarrow \mathbf{comm}) \rightarrow \mathbf{comm} \end{aligned}$$

We use the usual syntactic sugar in writing examples: $C_1; C_2$ for $\mathbf{seq}(C_1, C_2)$ and $V := E$ for $\mathbf{assign}_\delta(V, E)$. We also omit writing \mathbf{deref}_δ on the right hand sides of assignment commands.

It is significant that we have not included a recursion or looping construct in the language. This is because we omit the details of handling divergence for simplicity of presentation. It is possible to treat divergence in a way similar to the traditional approaches [26, 30]. A more detailed treatment may be found in [8, Ch. 6].

The semantics of Idealized Algol was defined by Reynolds [26] using a categorical version of a possible world semantics. Every phrase type τ is interpreted as a functor $\mathbf{W} \rightarrow \mathbf{C}$, where \mathbf{W} is a category of objects (“worlds”) appropriate for describing store shapes and \mathbf{C} is a category appropriate for semantic values. This approach is now something of a standard for interpreting imperative programming languages. See the various papers in the collection [21] as well as the more recent work [2–4, 7, 14, 25] that extends the framework for call-by-value languages and heap-allocated storage. O’Hearn and Tennent [20] initiated the addition of relational parametricity to the framework by replacing categories \mathbf{W} and \mathbf{C} by reflexive graphs. In the present paper, we use parametricity graphs in place of reflexive graphs. So, the semantic categories will be functor graphs of the form $\mathbf{S}^{\mathbf{W}}$ where \mathbf{S} is a parametricity graph of sets and \mathbf{W} is a parametricity graph of store shapes.

4.1 Store shapes

We will consider two parametricity graphs of store shapes for interpreting Idealized Algol, for reasons that will become apparent in the sequel. The first model closely follows Reynolds’s original treatment. The second model represents the structure of the store more concretely in terms of sets of locations. In both the cases, if W is a store shape, we write $S(W)$ for the set of states for store shape

W and $T(W)$ for the set of state transformations for W . $T(W)$ is general a monoid, with the monoid operation $(a; b)$ representing sequential composition of transformations and the monoid unit the identity transformation. In addition, there is a “diagonal” operation $D_W : [S(W) \rightarrow T(W)] \rightarrow T(W)$ which mimics the effect of the assignment operation.

The small parametricity graph \mathbf{R} is given by the following data:

- Vertices are sets, meant to represent the set of states taken by a particular store shape. So, $S(W) = W$ and $T(W) = [W \rightarrow W]$. The operation D_W is defined by $D_W(p) = \lambda s. p s s$.
- Vertex morphisms $j: W \rightarrow X$ are pairs (ϕ_j, τ_j) where $\phi_j : S(X) \rightarrow S(W)$ is a function and $\tau_j : T(W) \rightarrow T(X)$ is a monoid homomorphism such that:
 1. $\tau_j(a) \circ \phi_j = \phi_j \circ a$ for all $a \in T(W)$, and
 2. $\tau_j(D_W(p)) = D_X(\tau_j \circ p \circ \phi_j)$ for all $p \in [S(W) \rightarrow T(W)]$.
- Edges $R : W \leftrightarrow W'$ are pairs (R_s, R_t) where $R_s : S(W) \leftrightarrow S(W')$ is a relation and $R_t : T(W) \leftrightarrow T(W')$ is a monoid relation such that
 1. $a [R_t] a' \implies a [R_s \rightarrow R_s] a'$, and
 2. $D_W [R_s \rightarrow R_t] D_X$.
- An edge morphism $j [R \rightarrow S] j'$ exists iff $\phi_j [S_s \rightarrow R_s] \phi_{j'}$ and $\tau_j [R_t \rightarrow S_t] \tau_{j'}$.
- The weakest pre-edge operation $[j, j']R$ is given by:

$$\begin{aligned} ([j, j']R)_s &= R_s[\phi_j, \phi_{j'}] \\ ([j, j']R)_t &= [\tau_j, \tau_{j'}]R_t \end{aligned}$$

It can be verified that this data constitutes a parametricity graph. The intuition behind the categorical structure is that if $j: W \rightarrow X$ is a vertex morphism then X represents a larger store than W . The map ϕ_j projects the small store from the larger one and the monoid morphism τ_j reinterprets a transformation of the small store as a transformation of the larger store. Despite the somewhat abstract nature of the definitions, it can be shown that a morphism $j: W \rightarrow X$ exists iff $X \cong W \times V$ for some set V .

In the parametricity graph \mathbf{L} we represent these intuitions concretely. Vertices are finite maps from sets of location names to data types, written in the form $W = \{l_1: \delta_1, \dots, l_n: \delta_n\}$. Such store shapes can also be thought of as “record types” [6, 12, 25]. The set of states for a world as shown is $S(W) = \prod_{l_i} [\delta_i]$. The set of transformations is $T(W) = [S(W) \rightarrow S(W)]$ and D_W is defined by $D_W(p) = \lambda s. p s s$. A vertex morphism $W \rightarrow X$ exists iff $W \subseteq X$, i.e., X is a longer record type than W , and it is unique when it exists. Calling this morphism j , we can define a function $\phi_j: S(X) \rightarrow S(W)$ and a monoid morphism $\tau_j: T(W) \rightarrow T(X)$ as follows:

$$\begin{aligned} \phi_j(s) &= s \upharpoonright \text{dom}(W) \\ \tau_j(a) &= \lambda s. s \triangleleft (s \upharpoonright \text{dom}(W)) \end{aligned}$$

where \triangleleft is the record update operation. The edges and edge morphisms are exactly as for \mathbf{R} . It can be verified that this data makes \mathbf{L} a parametricity sub-graph of \mathbf{R} .

Both the parametricity graphs \mathbf{R} and \mathbf{L} represent the same intuitions about store shapes. The graph \mathbf{R} at first sight appears to be more abstract because it finesses the low-level details of locations and location naming. However, this abstraction is illusory. Since both the parametricity graphs have the same collection of edges for the common vertices, the parametricity conditions are able to abstract away from the low-level details. The main technical difference is that the graph \mathbf{R} has many non-trivial endomorphisms. Consider a world \mathbb{Z} representing a single integer location. Every bijection $r: \mathbb{Z} \leftrightarrow \mathbb{Z}$ gives rise to a vertex morphism $(r, [r^{-1} \rightarrow r]) : \mathbb{Z} \rightarrow \mathbb{Z}$. In contrast, the graph \mathbf{L} has no nontrivial endomorphisms. It turns out that the presence of nontrivial endomorphisms has consequences for the inhabitants of the semantic types.

4.2 Interpretation of Algol types

The interpretation of phrase types is given in the functor graph $\mathbf{S}^{\mathbf{W}}$ where \mathbf{W} is the parametricity graph of store shapes (either \mathbf{R} or \mathbf{L} in our setting) and \mathbf{S} is a small parametricity subgraph of \mathbf{Set} that is cartesian closed and complete. The meaning of polymorphic types is given in $\mathbf{Set}^{\mathbf{W}}$.

We first show the interpretations of the base types of Idealized Algol in order to provide intuition for the structure:

- The PG-functor $\llbracket \mathbf{exp}[\delta] \rrbracket$ evaluates an expression in a state.
 - $\llbracket \mathbf{exp}[\delta] \rrbracket W = [S(W) \rightarrow \llbracket \delta \rrbracket]$.
 - $\llbracket \mathbf{exp}[\delta] \rrbracket f$, for $f: W \rightarrow X$, is the mapping $e \mapsto e \circ \phi_f$.
 - $\llbracket \mathbf{exp}[\delta] \rrbracket R$, for $R: W \leftrightarrow W'$, is the relation $R_s \rightarrow I_{\llbracket \delta \rrbracket}$.
- The PG-functor $\llbracket \mathbf{comm} \rrbracket$ provides state transformation.
 - $\llbracket \mathbf{comm} \rrbracket W = T(W)$.
 - $\llbracket \mathbf{comm} \rrbracket f = \tau_f$.
 - $\llbracket \mathbf{comm} \rrbracket R = R_t$.
- The PG-functor $\llbracket \mathbf{var}[\delta] \rrbracket$ provides an indexed state transformation for the left hand side of assignment commands, and an expression valuation for the right hand side of assignment commands.
 - $\llbracket \mathbf{var}[\delta] \rrbracket W = (\llbracket \delta \rrbracket \rightarrow \llbracket \mathbf{comm} \rrbracket W) \times \llbracket \mathbf{exp}[\delta] \rrbracket W$.
 - $\llbracket \mathbf{var}[\delta] \rrbracket f = (\text{id}_{\llbracket \delta \rrbracket} \rightarrow \llbracket \mathbf{comm} \rrbracket f) \times \llbracket \mathbf{exp}[\delta] \rrbracket f$.
 - $\llbracket \mathbf{var}[\delta] \rrbracket R = (I_{\llbracket \delta \rrbracket} \rightarrow \llbracket \mathbf{comm} \rrbracket R) \times \llbracket \mathbf{exp}[\delta] \rrbracket R$.

We note that the required edge morphisms exist, e.g., if $f [R \rightarrow S] f'$ then $\phi_f [S_s \rightarrow R_s] \phi_{f'}$ and, so, $e [R_s \rightarrow I_{\llbracket \delta \rrbracket}] e' \implies e \circ \phi_f [S_s \rightarrow I_{\llbracket \delta \rrbracket}] e' \circ \phi_{f'}$.

4.3 Cartesian closed structure

Theorem 1. *If \mathbf{W} is a small parametricity graph and \mathbf{S} is a parametricity graph of sets that is cartesian closed and complete, then the functor graph $\mathbf{S}^{\mathbf{W}}$ is cartesian closed.*

The products in $\mathbf{S}^{\mathbf{W}}$ are given pointwise, using the products in \mathbf{S} .

- For functors $F, G \in \mathbf{W} \rightarrow \mathbf{S}$:
 - $(F \times G)W = FW \times GW$.
 - $(F \times G)f = Ff \times Gf$, for $f: W \rightarrow W'$.
 - $(F \times G)R = FR \times GR$, for $R: W \rightarrow W'$.
 - If $f [R \rightarrow S] f'$ then $(F \times G)f [(F \times G)R \rightarrow (F \times G)S] (F \times G)f'$ because $Ff [FR \rightarrow FS] Ff'$ and $Gf [GR \rightarrow GS] Gf'$.
- For parametric (natural) transformations $\eta : F \rightarrow F'$, $\theta : G \rightarrow G'$ the product action is given by $(\eta \times \theta)_W = \eta_W \times \theta_W$.
- For edges $\rho : F \leftrightarrow F'$ and $\sigma : G \leftrightarrow G'$, the product action is given by $(\rho \times \sigma)_R = \rho_R \times \sigma_R$.

The exponents in $\mathbf{S}^{\mathbf{W}}$ are similar to those in functor categories.

- For functors $F, G \in \mathbf{W} \rightarrow \mathbf{S}$:
 - $(F \Rightarrow G)W$ is the set of families $\langle p_j : FX \rightarrow GX \rangle_{j:W \rightarrow X}$ indexed by morphisms $j: W \rightarrow X$ such that, for all edges $S: X \leftrightarrow X'$,

$$j [I_W \rightarrow S] j' \Longrightarrow p_j [FS \rightarrow GS] p_{j'}$$

(Note that naturality follows by considering edges S of the form E_f .)

- $(F \Rightarrow G)f$, for $f: W \rightarrow W'$ is the mapping $\langle p_j \rangle_{j:W \rightarrow X} \mapsto \langle p_{j \circ f} \rangle_{j':W' \rightarrow X}$.
- $(F \Rightarrow G)R$, for $R: W \leftrightarrow W'$, is a binary relation that relates families $\langle p_j \rangle_{j:W \rightarrow X}$ and $\langle p_{j'} \rangle_{j':W' \rightarrow X'}$ iff, for all $S: X \leftrightarrow X'$

$$j [R \rightarrow S] j' \Longrightarrow p_j [FS \rightarrow GS] p_{j'}$$

- If $f [R \rightarrow S] f'$ then $(F \Rightarrow G)f [(F \Rightarrow G)R \rightarrow (F \Rightarrow G)S] (F \Rightarrow G)f'$
- For parametric (natural) transformations $\eta : F' \rightarrow F$, $\theta : G \rightarrow G'$ the exponent action is given by $(\eta \Rightarrow \theta)_W : \langle p_j \rangle_{j:W \rightarrow X} \mapsto \langle \theta_X \circ p_j \circ \eta_X \rangle_{j:W \rightarrow X}$.
- For edges $\rho : F \rightarrow F'$ and $\sigma : G \rightarrow G'$, the exponent action is $(\rho \Rightarrow \sigma)_R = \rho_R \rightarrow \sigma_R$.

The terminal object 1 in $\mathbf{S}^{\mathbf{W}}$ maps all vertices to the singleton set 1 , all morphisms to id_1 and all edges to I_1 .

Recall from Section 3.1 that the procedure types in the programming language are interpreted as exponents in the semantic category, $\mathbf{S}^{\mathbf{W}}$ in our case. A procedure of type $\tau_1 \rightarrow \tau_2$, defined in a store W , may be called in an expanded store X . Given an argument in $\llbracket \tau_1 \rrbracket X$, the procedure maps it to a computation in $\llbracket \tau_2 \rrbracket X$. However, the procedure does not have direct access to the new memory locations of X . This constraint is enforced by the parametricity condition in the definition of $F \Rightarrow G$. The meaning of the procedure should preserve all relations for the expanded store X as long as they keep the W part of the store fixed or, in other words, all relations for the new parts of the store in X . These insights were explained in the previous work [20, 25]. However, the previous work only treated $\mathbf{S}^{\mathbf{W}}$ as a functor *category*. Here, we have added the parametricity *graph* structure of $\mathbf{S}^{\mathbf{W}}$ as it is needed for the interpretation of polymorphic types.

4.4 Parametric limits and colimits

Theorem 2. *If \mathbf{W} is a small parametricity graph then $\mathbf{Set}^{\mathbf{W}}$ has all small parametric limits and parametric colimits.*

In the following, we abbreviate $\mathbf{Set}^{\mathbf{W}}$ to \mathbf{G} . Let \mathbf{K} be a small parametricity graph and $T: \mathbf{K} \rightarrow \mathbf{G}$ be a PG-functor. The parametric limit $\forall_K T(K)$ in $\mathbf{Set}^{\mathbf{W}}$ can be constructed as follows. $(\forall_K T K)W$ is expected to be a subset of $(\prod_K T K)W = \prod_K (T K)W$. So, it consists of families of the form $\langle p_K \in (T K)W \rangle_{K \in \mathbf{K}_v}$ that are parametric in the following sense: for all $P: K \leftrightarrow K'$, $p_K [(TP)_{I_W}] p_{K'}$.

For $f: W \rightarrow W'$, $(\forall_K T K)f$ is the mapping $\langle p_K \rangle_{K \in \mathbf{K}_v} \mapsto \langle (TKf)(p_K) \rangle_{K \in \mathbf{K}_v}$.

For $R: W \leftrightarrow W'$, $(\forall_K T K)R$ is the relation that relates families $\langle p_K \rangle_{K \in \mathbf{K}_v}$ and $\langle q_K \rangle_{K \in \mathbf{K}_v}$, iff, for all $P: K \leftrightarrow K'$, $P_K [(TP)_R] P_{K'}$.

Recall that $\forall: \mathbf{G}^{\mathbf{K}} \rightarrow \mathbf{G}$ is a PG-functor that is right adjoint to the diagonal functor. So, it has an action on the edges in $\mathbf{G}^{\mathbf{K}}$ as well. If $\xi: T \leftrightarrow T'$ is an edge, then it is mapped by the \forall functor to an edge that we denote as $\forall_P \xi_P: (\forall_K T K) \leftrightarrow (\forall_K T' K)$. This edge is in turn a family of edges in \mathbf{Set} , indexed by edges $R: W \leftrightarrow W'$, and the component $(\forall_P \xi_P)_R$ is defined by:

$$\langle p_K \rangle_K [(\forall_P \xi_P)_R] \langle q_K \rangle_K \iff \text{for all } P: K \leftrightarrow K', p_K [(\xi_P)_R] q_{K'}$$

Parametric limits make use of edges between PG-functors to enforce uniformity. This uniformity is at a distinctly different level than the uniformity criteria in exponents (preserving edges between store shapes), although the same mathematical theory is used for both levels of uniformity. One can see the preservation of edges at both levels when one considers polymorphic types involving the exponent. For instance the interpretation of the type $\forall \alpha. \alpha \rightarrow \alpha$, which is interpreted as the parametric limit $\forall_F F \Rightarrow F$ in $\mathbf{Set}^{\mathbf{W}}$ with F ranging over $\mathbf{K} = \mathbf{S}^{\mathbf{W}}$, illustrates the two levels of uniformity.

$$\begin{aligned} (\forall_F F \Rightarrow F)W &= \{ \langle p_F \in (F \Rightarrow F)W \rangle_{F \in \mathbf{K}_v} \mid \\ &\quad \forall \rho: F \leftrightarrow F'. p_F [(\rho \Rightarrow \rho)_{I_W}] p_{F'} \} \\ &= \{ \langle t_{F,j} \in F X \rightarrow F X \rangle_{F \in \mathbf{K}_v, j: W \rightarrow X} \mid \\ &\quad \forall \rho: F \leftrightarrow F', \forall S: X \leftrightarrow X', \\ &\quad j [I_W \rightarrow S] j' \implies t_{F,j} [\rho_S \rightarrow \rho_S] t_{F',j'} \} \end{aligned}$$

The preservation of the edges ρ represents the type-level parametric polymorphism whereas the preservation of the edges S represents the store-level parametricity implicit in the use of local variables.

4.5 Interpretation of constants

Finally, we are in a position to give the interpretation of the constants for Polymorphic Idealized Algol. Each constant $k: \tau$ must be given an interpretation $\llbracket k \rrbracket: 1 \rightarrow \llbracket \tau \rrbracket$. We finesse some of the inessential detail by giving directly the components $\llbracket k \rrbracket_W \in \llbracket \tau \rrbracket W$. Recall that we are actually giving two models, one

using the Reynolds-style store shapes \mathbf{R} and the other using a local-based store shapes \mathbf{L} . The following basic constants have the same interpretation in both the models. The interpretation of **newvar** is the only one that differs in the two models.

$$\begin{aligned}
\llbracket 0 \rrbracket_W &= \lambda s. 0 \\
\llbracket + \rrbracket_W &: (j: W \rightarrow X) \mapsto \lambda(e_1, e_2). \lambda s. e_1(s) + e_2(s) \\
\llbracket \text{skip} \rrbracket_W &= \lambda s. s \\
\llbracket \text{seq} \rrbracket_W &: (j: W \rightarrow X) \mapsto \lambda(a_1, a_2). a_1; a_2 \\
\llbracket \text{assign}_\delta \rrbracket_W &: (j: W \rightarrow X) \mapsto \lambda(v, e). D_X (\lambda s. \pi_1(v)(e(s))) \\
\llbracket \text{deref}_\delta \rrbracket_W &: (j: W \rightarrow X) \mapsto \lambda v. \pi_2(v)
\end{aligned}$$

To interpret **newvar** in the Reynolds-style model $\mathbf{S}^{\mathbf{R}}$, when starting in a world X , we move to the larger world $X \times \llbracket \delta \rrbracket$ containing an extra component for a δ -typed storage variable. We have a morphism $f: X \rightarrow X \times \llbracket \delta \rrbracket$ given by:

$$\begin{aligned}
\phi_f(s, d) &= s \\
\tau_f(a) &= a \times \text{id}_{\llbracket \delta \rrbracket}
\end{aligned}$$

We construct a denotation of type $\llbracket \text{var}[\delta] \rrbracket(X \times \llbracket \delta \rrbracket)$ as a pair $(\text{upd}_\delta^X, \text{read}_\delta^X)$:

$$\begin{aligned}
\text{upd}_\delta^X(k) &= \lambda(s, d). (s, k) \\
\text{read}_\delta^X &= \lambda(s, d). d
\end{aligned}$$

Assume that init_δ denotes the initial value that should be stored in a new δ -typed variable. Then the interpretation of **newvar** is as follows:

$$\llbracket \text{newvar}_\delta \rrbracket_W : (j: W \rightarrow X) \mapsto \lambda p. \lambda s. \phi_f(p_f(\text{upd}_\delta^X, \text{read}_\delta^X)(s, \text{init}_\delta))$$

The argument p , in $\llbracket \text{var}[\delta] \rightarrow \text{comm} \rrbracket X$, is a family of the form $\langle p_f \rangle_{f: X \rightarrow Y}$. We instantiate it with the particular f defined above.

To interpret **newvar** $_\delta$ in the location-based model $\mathbf{S}^{\mathbf{L}}$, when starting in a world X , we move to a larger world $X \uplus \{l: \delta\}$ choosing some $l \notin \text{dom}(X)$. The morphism $f: X \rightarrow X \uplus \{l: \delta\}$ is unique and comes equipped with derived operations ϕ_f and τ_f . The denotation of type $\llbracket \text{var}[\delta] \rrbracket(X \uplus \{l: \delta\})$ is given by:

$$\begin{aligned}
\text{upd}_\delta^X(k) &= \lambda s. s \triangleleft \{l \mapsto k\} \\
\text{read}_\delta^X &= \lambda s. s(l)
\end{aligned}$$

Then the interpretation of **newvar** $_\delta$ is as follows:

$$\llbracket \text{newvar}_\delta \rrbracket_W : (j: W \rightarrow X) \mapsto \lambda p. \lambda s. \phi_f(p_f(\text{upd}_\delta^X, \text{read}_\delta^X)(s \triangleleft \{l \mapsto \text{init}_\delta\}))$$

Note that the choice of the location used for the new variable is immaterial. If some other location l' could to be used, we can define a relation $R: X \uplus \{l: \delta\} \leftrightarrow X \uplus \{l': \delta\}$ and show that upd and read preserve the relation. Hence, the meaning of **newvar** is independent of the choice of the location. The sample equivalence:

$$\text{newvar}_\delta \lambda x. \text{newvar}_\delta \lambda y. C \cong \text{newvar}_\delta \lambda y. \text{newvar}_\delta \lambda x. C$$

holds in this model, unlike the situation in the functor category semantics [19].

5 Analysis

Relational parametricity conditions are meant to ensure that polymorphic functions are defined uniformly at all types. If they do, then it becomes possible to prove a variety of program equivalences such as “free” theorems [31] and correctness of data representations [29]. One way to verify if the parametricity conditions capture enough uniformity is to examine a number of type isomorphisms that are dubbed “initial algebra” and “final coalgebra” results [28]. We examine a few such results that are expressible in the predicative polymorphic calculus.

A basic result is to show that the type $\forall\alpha. \alpha \rightarrow \alpha$ is isomorphic to the terminal object. This should intuitively be the case because we expect that the only way to uniformly return a result of an unknown type α using only an argument of that type, is to return that argument (or any other values uniformly available at all types). It was shown in [9] that this is always the case if the semantic category is well-pointed. However, functor categories (and functor graphs) are not well-pointed. So, that result does not apply.

Examining the case for the functor graph $\mathbf{Set}^{\mathbf{R}}$, which we have dubbed the Reynolds-like model, we find that $\forall_F(F \Rightarrow F)$ is *not* isomorphic to the terminal object. Here is an explanation. As noted in Sec. 4.1, \mathbf{R} has non-identity endomorphisms. Suppose $m:W \rightarrow W$ is such an endomorphism. Then it is possible to uniformly replicate the effect of m at every future world X with a morphism $j:W \rightarrow X$. An easy way to see this is to note that, in \mathbf{R} , $X \cong W \times V$. Since $m \times \text{id}_V$ is an endomorphism on $W \times V$, it can be simulated by an endomorphism on X . Denote such endomorphisms by $j\{m\}:X \rightarrow X$. Now, the family $\hat{m}_F = \langle F(j\{m\}) \rangle_{j:W \rightarrow X}$ is an element of $(F \Rightarrow F)W$. It is possible to show that the family $\langle \hat{m}_F \rangle_F$ is parametric and belongs to $(\forall_F F \Rightarrow F)W$. Since the terminal object has a single element in $1(W)$, $\forall_F F \Rightarrow F$ is not isomorphic to 1.

Thinking about the issue a bit more abstractly, we find that this problem confirms our intuition about the division of labour between morphisms and edges. Morphisms represent “operations that may be performed” and edges represent “properties that should be preserved.” The nontrivial endomorphisms in \mathbf{R} are more in the second category than the first. So, they are better treated as edges. Our modified parametricity graph of worlds \mathbf{L} does not have this problem. All the morphisms in the graph do correspond to operations that may be performed and there are no non-identity endomorphisms. Indeed, we are able to show that all the expected isomorphisms hold in $\mathbf{Set}^{\mathbf{L}}$ [8, Sec. 6.3]:

$$\begin{aligned} \forall_F(F \Rightarrow F) &\cong 1 \\ \forall_F((A \Rightarrow F) \Rightarrow F) &\cong A \\ \forall_F((A \Rightarrow B \Rightarrow F) \Rightarrow F) &\cong A \times B \\ \forall_F F &\cong 0 \end{aligned}$$

We have provided a semantic account of polymorphic imperative languages, and demonstrated that the parametricity of local variables and polymorphic

functions can be modelled in a unified framework. Further work is needed to understand the limits of this approach and how it compares with the other approaches such as that of [11].

References

1. Abadi, M., Cardelli, L., Curien, P.L.: Formal parametric polymorphism. *Theoretical Comput. Sci.* 121(1-2), 9–58 (Dec 1993)
2. Ahmed, A., Dryer, D., Rossberg, A.: State-dependent representation independence. In: *Thirty Sixth Ann. ACM Symp. on Princ. of Program. Lang.* ACM (2009)
3. Appel, A.W., McAllester, D.: An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23(5), 657–683 (2001)
4. Benton, N., Leperchey, B.: Relational reasoning in a nominal semantics for storage. In: *TLCA* (2005)
5. Burstall, R.M., MacQueen, D.B., Sanella, D.T.: Hope: an experimental applicative language. In: *ACM LISP Conference*. pp. 136–143 (1980)
6. Cardelli, L., Wegner, P.: On understanding types, data abstraction, and polymorphism. *Computing Surveys* 17(4), 471–522 (1986)
7. Dreyer, D., Neis, G., Rossberg, A., Birkedal, L.: A relational modal logic for higher-order stateful ADTs. In: *Thirty Seventh Ann. ACM Symp. on Princ. of Program. Lang.* ACM (2010)
8. Dunphy, B.P.: Parametricity as a Notion of Uniformity in Reflexive Graphs. Ph.D. thesis, University of Illinois, Dep. of Mathematics (2002), available electronically from <http://www.cs.bham.ac.uk/~udr>
9. Dunphy, B.P., Reddy, U.S.: Parametric limits. In: *Proceedings, Nineteenth Annual IEEE Symposium on Logic in Computer Science*. pp. 242–253. IEEE Computer Society Press (Jul 2004)
10. Gordon, M., Milner, R., Morris, L., Newey, N., Wadsworth, C.: A metalanguage for interactive proof in LCF. In: *ACM Symp. on Princ. of Program. Lang.* pp. 119–130 (1978)
11. Hermida, C., Tennent, R.D.: A fibrational framework for possible-world semantics of Algol-like languages. *Theoretical Comput. Sci.* 375, 3–19 (2007)
12. Hoffman, M., Pierce, B.C.: Positive subtyping. *Information and Computation* 126, 11–33 (1996)
13. Hudak, P., Jones, S.P., Wadler (eds), P.: Report on the programming language Haskell: A non-strict purely functional language (Version 1.2). *SIGPLAN Notices* 27(5), Section R (May 1992)
14. Levy, P.B.: Possible world semantics for general storage in call-by-value. In: *CSL 2002*. pp. 232–246 (2002)
15. Liskov, B.: An introduction to CLU. In: Schuman, S.A. (ed.) *New Directions in Algorithmic Languages 1975*, pp. 139–156. IRIA, Rocquencourt (1975)
16. Meyer, A.R., Sieber, K.: Towards fully abstract semantics for local variables. In: *Fifteenth Ann. ACM Symp. on Princ. of Program. Lang.* pp. 191–203. ACM (1988), (Reprinted as Chapter 7 of [21])
17. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential types. *ACM Trans. Program. Lang. Syst.* 10(3), 470–502 (1988)
18. O’Hearn, P.W., Reynolds, J.C.: From Algol to polymorphic linear lambda-calculus. *J. ACM* 47(1), 167–223 (Jan 2000)

19. O’Hearn, P.W., Tennent, R.D.: Semantics of local variables. In: Fourman, M.P., Johnstone, P.T., Pitts, A.M. (eds.) Applications of Categories in Computer Science, pp. 217–238. Cambridge Univ. Press (1992)
20. O’Hearn, P.W., Tennent, R.D.: Parametricity and local variables. J. ACM 42(3), 658–709 (1995), (Reprinted as Chapter 16 of [21])
21. O’Hearn, P.W., Tennent, R.D.: Algol-like Languages (Two volumes). Birkhäuser, Boston (1997)
22. Oles, F.J.: Type algebras, functor categories and block structure. In: Nivat, M., Reynolds, J.C. (eds.) Algebraic Methods in Semantics, pp. 543–573. Cambridge Univ. Press (1985)
23. Plotkin, G., Abadi, M.: A logic for parametric polymorphism. In: Typed Lambda Calculi and Applications - TLCA ’93. pp. 361–375. LNCS, Springer-Verlag (1993)
24. Reddy, U.S.: Objects and classes in Algol-like languages. Information and Computation 172, 63–97 (2002)
25. Reddy, U.S., Yang, H.: Correctness of data representations involving heap data structures. Science of Computer Programming 50(1-3), 129–160 (Mar 2004)
26. Reynolds, J.C.: The essence of Algol. In: de Bakker, J.W., van Vliet, J.C. (eds.) Algorithmic Languages, pp. 345–372. North-Holland (1981), (Reprinted as Chapter 3 of [21])
27. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: Mason, R.E.A. (ed.) Information Processing ’83, pp. 513–523. North-Holland, Amsterdam (1983)
28. Reynolds, J.C., Plotkin, G.D.: On functors expressible in the polymorphic typed lambda calculus. Information and Computation 105(1), 1–29 (July 1993)
29. Tennent, R.D.: Correctness of data representations in Algol-like languages. In: Roscoe, A.W. (ed.) A Classical Mind: Essays in Honor of C. A. R. Hoare, pp. 405–417. Prentice-Hall International (1994)
30. Tennent, R.D.: Denotational semantics. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science, vol. 3, pp. 169–322. Oxford University Press (1994)
31. Wadler, P.: Theorems for free! In: Fourth Intern. Conf. Functional Program. Lang. and Computer Architecture. pp. 347–359. ACM (1989)

A Reflexive graphs of categories

Let \mathbf{B} be the category with two objects v and e , and five non-identity arrows as follows:

$$\begin{array}{ccccc}
 & \xleftarrow{\partial_0} & & \xrightarrow{\partial_0; I} & \\
 & I & \longrightarrow & \partial_1; I & \\
 v & \xrightarrow{\quad} & e & \xrightarrow{\quad} & e \\
 & \xleftarrow{\partial_1} & & &
 \end{array}$$

All other composite arrows are identities, in particular $I; \partial_i = \text{id}_e$. Then a *reflexive graph* is a functor $\mathbf{B} \rightarrow \mathbf{CAT}$, i.e., a \mathbf{B}^{op} -indexed category.

Unpacking the definition, here is what this data means. If $\mathbf{G} : \mathbf{B} \rightarrow \mathbf{CAT}$ is a reflexive graph, it involves two categories \mathbf{G}_v and \mathbf{G}_e , where \mathbf{G}_v is thought of as a category of “vertices” and “vertex morphisms”, and \mathbf{G}_e is thought of as a category of “edges” and “edge morphisms”. Referring to the diagram (1), A, A', B, B', f and f' are in \mathbf{G}_v , whereas R, S , and the square ϕ itself are in \mathbf{G}_e . The

images of ∂_0 and ∂_1 are functors, which we also write as ∂_0 and ∂_1 , are functors that send each edge (e.g., R) and edge morphism (e.g., ϕ) to their underlying vertices and vertex morphisms. The image of I is a functor that assigns to each vertex A , a distinguished edge $I_A: A \leftrightarrow A$ and, to each vertex morphism f , a distinguished edge morphism $I_f: I_A \rightarrow I_{A'}$. In most examples of interest to us, there is *at most one* edge morphism ϕ with a given shape. If such is the case we call the reflexive graph *relational*.

As a basic example, for any category \mathbf{C} , its arrow category \mathbf{C}^{\rightarrow} can be used as the edge category of a relational reflexive graph. The edges are just the arrows of \mathbf{C} and an edge morphism $f [g \rightarrow h] f'$ exists iff the following square commutes:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ g \downarrow & & \downarrow h \\ A' & \xrightarrow{f'} & B' \end{array}$$

The identity edges are the identity arrows of \mathbf{C} . This reflexive graph is not interesting from our point of view because it does not utilise the two degrees of freedom provided by reflexive graphs. As another example, any category \mathbf{C} can be given a reflexive graph structure by taking as the edge category, the category of spans in \mathbf{C} of the form $A \xleftarrow{p} R \xrightarrow{p'} A'$. This category is not relational, as there can be multiple arrows $R \rightarrow S$ witnessing a particular preservation property.

Indexed categories $\mathbf{CAT}^{\mathbf{B}}$ form a 2-category. The 1-cells, called “RG-functors” are just indexed functors, and the 2-cells, called “parametric natural transformations” are just indexed natural transformations. We denote this 2-category by \mathbf{RG} . Viewed as a category, \mathbf{RG} is cartesian closed [8, 9]. The product reflexive graphs $\mathbf{G} \times \mathbf{H}$ are defined pointwise: $(\mathbf{G} \times \mathbf{H})_v = \mathbf{G}_v \times \mathbf{H}_v$ and $(\mathbf{G} \times \mathbf{H})_e = \mathbf{G}_e \times \mathbf{H}_e$. Quite remarkably, the exponents $\mathbf{H}^{\mathbf{G}}$ are also given pointwise:

- $(\mathbf{H}^{\mathbf{G}})_v$ is the category of RG-functors $\mathbf{G} \rightarrow \mathbf{H}$ and parametric natural transformations between them.
- $(\mathbf{H}^{\mathbf{G}})_e$ is the category of RG-functors $\mathbf{E} \times \mathbf{G} \rightarrow \mathbf{H}$ and parametric natural transformations between them, where \mathbf{E} is the formal reflexive graph with two vertices 0 and 1 with one non-identity edge $e: 0 \leftrightarrow 1$.

Unpacking the definition, it may be seen that an edge $\rho: F \leftrightarrow F'$ in $\mathbf{H}^{\mathbf{G}}$ is a family of the form $\langle \rho_R: FX \leftrightarrow F'X' \rangle_{R: X \leftrightarrow X'}$, indexed by edges R of \mathbf{G} . An edge morphism $\phi: \rho \rightarrow \sigma$ in $\mathbf{H}^{\mathbf{G}}$ is again an indexed family $\langle \phi_R: \rho_R \rightarrow \sigma_R \rangle_{R: X \leftrightarrow X'}$. (Such a simple structure for the exponential is quite unique in the domain of indexed categories, and gives reflexive graphs their special place in the formulation.)

A.1 Parametricity graphs

In [8, 9], three axioms were identified to obtain a satisfactory interpretation of parametric polymorphism.

The first condition is that the reflexive graphs involved should be *relational*. The existence of multiple edge morphisms with the same shape implies that polymorphic types may have duplicate elements which differ only in the edge morphisms used to witness the parametricity constraints. Note that, in a relational reflexive graph, the edge morphisms are relations $\text{Hom}(R, S) \subseteq \text{Hom}(\partial_0 R, \partial_0 S) \times \text{Hom}(\partial_1 R, \partial_1 S)$. So, we can use the notation $f [R \rightarrow S] f'$ to mean that there is an edge morphism $R \rightarrow S$ above f and f' .

The second condition is that the reflexive graphs should be *fibred* with a chosen cleavage. This amounts to the condition that, whenever there are vertex morphisms $f: A \rightarrow B$ and $f': A' \rightarrow B'$ leading to an edge $R: B \leftrightarrow B'$, there is a distinguished *weakest pre-edge* or “reindexing”, denoted $[f, f']R: A \leftrightarrow A'$ such that we have an edge morphism:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \uparrow [f, f']R & \phi & \downarrow R \\ A' & \xrightarrow{f'} & B' \end{array}$$

Moreover, any other edge morphism of the form:

$$\begin{array}{ccc} X & \xrightarrow{f \circ g} & B \\ \uparrow T & \psi & \downarrow R \\ X' & \xrightarrow{f' \circ g'} & B' \end{array}$$

uniquely factors through ϕ . Note that **Set** always has such weakest pre-edges:

$$[f, f']R = \{ (x, x') \mid fx [R] f'x' \}$$

The weakest pre-edges in **Set**^{op} are the strongest post-edges of **Set**, defined by:

$$R[f, f'] = \{ (fx, f'x') \mid x \in A, x' \in A', x [R] x' \}$$

We also require that the RG-functors we use preserve the chosen weakest pre-edges, i.e.,

$$F([f, f']R) = [Ff, Ff'](FR)$$

Note that Hermida and Tennent [11] have also proposed the use of fibrations for modeling parametricity, even though they do not use reflexive graphs centrally in their treatment.

The third condition, taken from [?], is that the reflexive graphs should satisfy the *identity condition*: if $f [I_A \rightarrow I_B] f'$ for $f, f' \in A \rightarrow B$ then $f = f'$.

A reflexive graph satisfying these three conditions is called a *parametricity graph*. RG-functors between parametricity graphs that preserve the chosen

weakest pre-edges will be called *PG-functors*. Parametricity graphs, PG-functors and parametric (natural) transformations form a 2-category \mathbf{PG} (in fact, a 2-subcategory of \mathbf{RG}). \mathbf{PG} is cartesian closed. The product reflexive graphs $\mathbf{G} \times \mathbf{H}$ have pointwise weakest pre-edges:

$$[(f, g), (f', g')](R, S) = ([f, f']R, [g, g']S)$$

The exponent graphs $\mathbf{H}^{\mathbf{G}}$ have weakest pre-edges given by:

$$[\eta, \eta']\rho = \langle [\eta_X, \eta'_{X'}]\rho_R \rangle_{R: X \leftrightarrow X'}$$

It needs to be verified that \mathbf{PG} is cartesian closed with these constructions [8, Sec. 4.5].

An important point about \mathbf{PG} is that naturality is subsumed by parametricity. This is because every vertex morphism $f: A \rightarrow A'$ has a representative among edges $E_f: A \leftrightarrow A'$ given by $E_f = [f, \text{id}_{A'}]I_{A'}$. Since PG-functors preserve weakest pre-edges, they also preserve these representatives: $F(E_f) = E_{Ff}$. Moreover, commutative squares correspond to edge morphism squares:

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ g \downarrow & & \downarrow h \\ A' & \xrightarrow{f'} & B' \end{array} \iff \begin{array}{ccc} A & \xrightarrow{f} & B \\ E_g \downarrow & & \downarrow E_h \\ A' & \xrightarrow{f'} & B' \end{array}$$

Hence, the 2-cells in \mathbf{PG} involve just the parametricity condition, i.e., a 2-cell $\eta: F \rightarrow G: \mathbf{G} \rightarrow \mathbf{H}$ is a family $\eta_X: FX \rightarrow GX$ of vertex morphisms such that, for all edges $R: X \leftrightarrow X'$ in \mathbf{G} , we have $\eta_X [FR \rightarrow GR] \eta_{X'}$.