2012 27th Annual ACM/IEEE Symposium on Logic in Computer Science

A Computational Interpretation of Parametricity

Jean-Philippe Bernardy Guilhem Moulin
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden
Email: {bernardy,mouling}@chalmers.se

Abstract—**Reynolds' abstraction theorem has recently been extended to lambda-calculi with dependent types. In this paper, we show how this theorem can be internalized. More precisely, we describe an extension of Pure Type Systems with a special parametricity rule (with computational content), and prove fundamental properties such as Church-Rosser's and strong normalization. All instances of the abstraction theorem can be both expressed and proved in the calculus itself. Moreover, one can apply parametricity to the parametricity rule: parametricity is itself parametric.**

Index Terms—**Type structure, Lambda Calculus.**

I. Introduction

In his seminal paper, Reynolds [26] gave formal meaning to polymorphism, by interpreting types as predicates, in such a way that all inhabitants of a given type satisfy its interpretation as a predicate. Crucially, the type of types ($*$) is interpreted by the set of predicates. A simple example is that if a function f has type $\forall a : *.\, a \to a$ — the type of the polymorphic identity — then the following proposition holds[1]

$$\forall a_0 : *.\, \forall a_1 : a_0 \to *.\, \forall x : a_0.\, a_1\, x \to a_1\, (f\, a_0\, x)$$

(which implies that f must return exactly its argument).

The above result, *abstraction*, has proved useful for reasoning about functional programs [31] . It also has deep theoretical implications: for example, the induction principle over Church numerals can be deduced from it [32].

The study of parametricity is typically semantic, including the seminal work of Reynolds. There, the concern is to construct a model that captures the polymorphic character of a λ-calculus. Mairson [17] pioneered a different angle of study, of more syntactical nature: for each concrete term, a proof term which shows that it satisfies the relational interpretation of its type is constructed. That idea has then been used by various authors, including Abadi et al. [1], Plotkin and Abadi [24] and Wadler [32]. Bernardy et al. [9] have also shown how terms, types, their relational interpretation as proofs and propositions can all be expressed in a single calculus. The calculi where this is possible must however be rich enough: in particular they must support dependent types. Systems such as the Calculus of Constructions [11] or Martin-Löf's Intuitionistic Type Theory [18] both satisfy the requirements.

Still, even though we know that all terms of the above systems satisfy the parametricity condition, and each of these conditions can be expressed in the same system as the terms they concern, the very fact that any given term satisfies the relational interpretation of its type is *itself* not provable in the system. For example, the parametricity condition arising from the type $\forall a : *.\, a \to a$, namely

$$\forall f : (\forall a : *.\, a \to a).$$
$$\forall a_0 : *.\, \forall a_1 : a_0 \to *.\, \forall x : a_0.\, a_1\, x \to a_1\, (f\, a_0\, x),$$

is not provable in the Calculus of Constructions.

One would like to be able to rely on parametricity conditions within proof assistants themselves. Indeed, the correctness of numerous functional programming techniques relies on parametricity. Examples include program transformation [13, 15], testing [8], semantic program inversion [29] and generic programming [30]. One can already describe these techniques withing proof assistants based on type theory, but sometimes not prove their correctness. In any case, it would be useful to take advantage of parametricity to mechanize the proofs.

We are not the first to recognize the need for parametricity: it is for example a recurring topic in the field of mechanized metatheory, where precise encoding of variable bindings often makes use of polymorphism. For example, Pouillard [25] describes the following representation for terms, using the AGDA [21] proof assistant:

$$\begin{array}{l} \textbf{data}\ \mathsf{Term}\,(V : \mathsf{Set}) : \mathsf{Set}\ \textbf{where} \\ \quad \mathsf{var}\ : V \to \mathsf{Term}\, V \\ \quad \mathsf{app} : \mathsf{Term}\, V \to \mathsf{Term}\, V \to \mathsf{Term}\, V \\ \quad \mathsf{abs}\ : \mathsf{Term}\,(\mathsf{Maybe}\, V) \to \mathsf{Term}\, V \end{array}$$

with the intent that the type variable V represents the context. Functions working on an arbitrary context must therefore universally quantify over V. In that case, V is abstract, and therefore such functions are guaranteed to be oblivious to renamings of free variables. Pouillard formalizes the above argument within AGDA, using logical relations as defined by Bernardy et al. [9]. Only the parametricity axiom is missing to establish the obliviousness property. Chlipala [10] and Atkey et al. [3] encounter the same type of situation.

Another application of parametricity is a new kind of *meta-programming*. Indeed, via the Curry-Howard isomorphism, proofs can be interpreted as programs, and in particular the proof that a term is parametric can be a useful program in its own right. Bernardy [5, p. 82] shows for example how to derive the vectors of length n from lists with a (modified) version of parametricity. Not only the type is derived, but operations on lists are transformed to their counterpart on vectors.

[1]By convention, the variable a_1 is the predicate corresponding to the type variable a_0.

1043-6871/12 $26.00 © 2012 IEEE
DOI 10.1109/LICS.2012.25

In this paper, we address the lack of support for parametricity in logical frameworks. Technically, we propose to extend pure type systems (PTSs) to make all the parametricity propositions provable internally. The aim is to pave the way for native support of parametricity in tools based on dependent types, such as AGDA or COQ [28]. The challenge is not to merely postulate the axiom and check its consistency with the rest of the system: because we want to retain the constructive character of these proof assistants, we must provide a *computation rule* for parametricity. This computational aspect is not only philosophically satisfying, it enables the usage of the parametricity as a meta-programming tool, as outlined above.

Furthermore, the new construction can itself be given a parametric interpretation, which preserves the abstraction theorem. That is, in our calculus, *parametricity is parametric*. As we shall explain, to support this form of self-applicability, our design uses quantification over multi-dimensional objects. Beside parametricity, the use of multiple dimensions is a characteristic feature of our calculus.

Our technical contributions are as follows:

- We describe a dependently-typed λ-calculus with internalized parametricity. The calculus is summarized in definitions 3 to 5; and motivated in Sec. II.
- We prove that it satisfies typical properties of a well-behaved λ-calculus with types. In particular, we prove the Church-Rosser property, and subject reduction. We also prove strong normalization relatively to the corresponding calculus without parametricity, by translation into that calculus.
- An implementation of a type-checker for the calculus is made available for experimentation.

II. DESIGN, STEP BY STEP

In this section we describe and motivate our design step by step, starting from pure type systems.

A. Pure type systems, and our notation

We assume familiarity with pure type systems (PTSs), but we give a brief reminder in the following paragraphs, as well as an introduction to our notation. Readers are referred to Barendregt [4] for details.

PTSs are a family of λ-calculi, parameterized by a set of sorts $\mathcal{S}$, a set of axioms $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ and set of rules $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$. The various syntactic forms of quantifications (and corresponding abstraction and application) are syntactically unified, and one needs to inspect sorts to identify which form is meant. The axioms $\mathcal{A}$ give the typing rules for sorts, and $\mathcal{R}$ determines which forms of quantification exist in the system. Many systems (*e.g.*, the Calculus of Constructions or System F) can be cast into the framework of PTSs. The syntax for PTS terms is the following:

$$\begin{array}{lrccl} \mathsf{Term} & \ni A, \ldots, U & = & s & \text{sort} \\ & & \mid & x & \text{variable} \\ & & \mid & A\,B & \text{application} \\ & & \mid & \lambda x : A.\, B & \text{abstraction} \\ & & \mid & \forall x : A.\, B & \text{product} \end{array}$$

The product $\forall x : A.\, B$ may be also written $A \to B$ when x does not occur free in B. In the rest of the paper we assume a given PTS specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, and we name the calculus arising from that specification $\mathcal{O}$.

The major part of the paper is devoted to the description and justification of another calculus, here called $\mathcal{P}$, parameterized over the same specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, which extends $\mathcal{O}$ and contains our special construction for parametricity. The syntax and typing rules for $\mathcal{P}$ are given in definitions 3 to 5.

B. Logical relations, from PTS to PTS

In this section we recall the relational interpretation of terms and types of the PTS $\mathcal{O}$ into another PTS, here called $[\![\mathcal{O}]\!]$. The material of this section is essentially the same as in Bernardy et al. [9] and Bernardy and Lasson [6].

In any PTS, types and terms can be interpreted as relations and proofs that the terms satisfy the relations. Each *type* can be interpreted as a predicate that its inhabitants satisfy; And each *term* can be turned into a proof that it satisfies the predicate of its type. Usual presentations of parametricity use binary relations, but for simplicity of notation we present here a unary version. The generalization to arbitrary arity is straightforward, and we refer the readers to [6] for details.

In the following we define what it means for a program C to satisfy the predicate generated by a type T (the proposition $C \in [\![T]\!]$); and the translation from a program C of type T to a proof $[\![C]\!]$ that C satisfies the predicate. A property of the translation is that whenever $x : A$ is free in T, there are two variables x_0 and x_1 in $[\![T]\!]$, the latter one witnessing that x_0 satisfies the parametricity condition of its type ($x_1 : x_0 \in [\![A]\!]$). This means that the translation extends contexts, as follows:

$$\begin{aligned} [\![-]\!] &= - \\ [\![\Gamma, x : A]\!] &= [\![\Gamma]\!], x_0 : \{A\}, x_1 : x_0 \in [\![A]\!] \end{aligned}$$

(Where $\{A\}$ is A where each free variable y is renamed to y_0.)

It is important to notice that this definition assumes a *global* renaming from each variable x to fresh variables x_0 and x_1. (The renaming will be made local in further sections.)

The relational interpretation of a type T to a proposition $C \in [\![T]\!]$, defined as a syntactic translation from terms to terms by structural induction on T, as follows.

- Because types in a PTS are abstract (there is no pattern matching on types), any predicate over C can be used to witness that C satisfies the relational interpretation of a sort s.

$$C \in [\![s]\!] = C \to s$$

- If the type is a product ($\forall x : A.\, B$), then C is a function, and it satisfies the relational interpretation of its type iff it maps related inputs to related outputs.

$$C \in [\![\forall x : A.\, B]\!] = \forall x_0 : \{A\}.\, \forall x_1 : x_0 \in [\![A]\!].\, (C\, x_0) \in [\![B]\!]$$

- For any other syntactic form for a type T of sort s, (which, if it is well-typed, may be a variable or an application),

T is interpreted as a predicate ($[\![T]\!] : \{T\} \to s$), and to check that C satisfies it, one can use application.

$$C \in [\![T]\!] = [\![T]\!]\, C$$

It remains to give the translation from a term C to a proof term $[\![C]\!]$, also defined by structural induction on the term C.

- The translation of a variable is done by looking up the corresponding parametric witness in the context.

$$[\![x]\!] = x_1$$

- The case for abstraction adds a witness that the input satisfies the relational interpretation of its type and returns the relational interpretation of the body, mirroring the interpretation of product:

$$[\![\lambda x : A.\, B]\!] = \lambda x_0 : \{A\}.\, \lambda x_1 : x_0 \in [\![A]\!].\, [\![B]\!]$$

- The application follows the same pattern: the function is passed a witness that the argument satisfies the interpretation of its type.

$$[\![A\, B]\!] = [\![A]\!]\, \{B\}\, [\![B]\!]$$

- If the term has another syntactic form, then it is a type (T), thus we can use λ-abstraction to create a predicate and check that the abstracted variable z satisfies the relational interpretation of the type in the body ($z \in [\![T]\!]$).

$$[\![T]\!] = \lambda z : \{T\}.\, z \in [\![T]\!]$$

At this point the definition might appear circular; but in fact the form $\cdot \in [\![T]\!]$ invokes $[\![T]\!]$ *only* when T is an application, which is processed structurally by $[\![\cdot]\!]$.

Bernardy and Lasson [6] prove the following theorem:

Theorem 1 (Abstraction). *If* $\Gamma \vdash_{\mathcal{O}} A : B : s$, *then*

$$[\![\Gamma]\!] \vdash_{[\![\mathcal{O}]\!]} [\![A]\!] : (\{A\} \in [\![B]\!]) : s$$

where $[\![\mathcal{O}]\!]$ *is a PTS computed from* $\mathcal{O}$. *Furthermore, if* $\mathcal{O}$ *is consistent, then so is* $[\![\mathcal{O}]\!]$.

Proof: By induction on the derivation. ■

A direct reading of the above result is as a typing judgement about translated terms: if A has type B, then $[\![A]\!]$ has type $\{A\} \in [\![B]\!]$. However, it can also be understood as an abstraction theorem for $\mathcal{O}$: if a program A has type B in Γ, then A satisfies the relational interpretation of its type ($\{A\} \in [\![B]\!]$). Remember that $\{A\}$ is merely the term A, but using variables in $[\![\Gamma]\!]$ instead of Γ. In particular, if A is closed then $\{A\} = A$. For arity 2, $[\![\Gamma]\!]$ contains two related environments (and witnesses that they are properly related). Thus, A has two possible interpretations $\{A\}$, each obtained by picking variables out of each copy of the environment, and $[\![A]\!]$ is a proof that the two possible interpretations of A are related.

In general, the PTS $[\![\mathcal{O}]\!]$, where parametricity conditions are expressed, is more general than the source system $\mathcal{O}$. However, for rich enough systems, such as the calculus of constructions, they are equivalent (see [9, 6] for the conditions where this occurs).

C. Aim and example

Let us assume a PTS $\mathcal{Q}$, such that $\mathcal{Q}$ is equivalent to $[\![\mathcal{Q}]\!]$. Because both types and their parametricity conditions can be expressed in $\mathcal{Q}$, one can hope that for every term A of type B, the user of the system can get a witness $[\![A]\!]$ that it is parametric ($\{A\} \in [\![B]\!]$). Even though this hope is fulfilled for closed terms, we run out of luck for open terms, because the context where $[\![A]\!]$ is meaningful is "bigger" than that where A is: for each free variable $x : A$ in Γ, we need a variable $x_1 : x \in [\![A]\!]$ in $[\![\Gamma]\!]$. In other words, from $\Gamma \vdash_{\mathcal{Q}} A : B$ we have $[\![\Gamma]\!] \vdash_{\mathcal{Q}} [\![A]\!] : \{A\} \in [\![B]\!]$, but we really need $\Gamma \vdash_{\mathcal{Q}} [\![A]\!] : A \in [\![B]\!]$. Indeed, we do not want to add an explicit assumption that every variable is bound to a parametric value. The aim of this paper is to find a system $\mathcal{P}$ such that the following proposition is verified.

Proposition 1 (Internal parametricity).

$$\Gamma \vdash_{\mathcal{P}} A : B \Rightarrow \Gamma \vdash_{\mathcal{P}} [\![A]\!] : A \in [\![B]\!]$$

In that case, for any term A, terms of $\mathcal{P}$ can invoke the fact that A is parametric, by writing $[\![A]\!]$.

Example 1. *For example, further assuming that $\mathcal{P}$ embeds the Calculus of Constructions, we can prove that any function of type $\forall a : *.\, a \to a$ is an identity, as we hinted at in the introduction. The formulation of the theorem within $\mathcal{P}$ and its proof term are as follows.*

$$\begin{aligned}&\mathit{identities} : \forall f : (\forall a : *.\, a \to a).\, \forall a : *.\, \forall x : a.\, \mathit{Eq}\, a\, (f\ a\ x)\, x\\ &\mathit{identities} = \lambda f.\lambda a.\lambda x.[\![f]\!]\, a\, (\mathit{Eq}\, a\, x)\, x\, (\mathit{refl}\, a\, x)\end{aligned}$$

where Eq stands for Leibniz equality. (The type of $[\![f]\!]$ is $\forall a : *.\, \forall a_1 : a \to *.\, \forall x : a.\, a_1\, x \to a_1\, (f\, a\, x)$.)

If identities is used on a concrete identity function, say $i = \lambda a : *.\, \lambda x : a.\, x$, then $f\, a\, x$ reduces to x, and the theorem specializes to reflexivity of equality:

$$\mathit{identities}\, i\ :\ \forall a : *.\, \forall x : a.\, \mathit{Eq}\, a\, x\, x$$

after reduction the proof no longer mentions $[\![\cdot]\!]$:

$$\begin{aligned}&\mathit{identities}\, i\\ &\quad \to_\beta \lambda a.\lambda x.[\![f]\!][f \mapsto \lambda a : *.\, \lambda x : a.\, x]\, a\, (\mathit{Eq}\, a\, x)\, x\, (\mathit{refl}\, a\, x)\\ &\quad = \lambda a.\lambda x.[\![\lambda a : *.\, \lambda x : a.\, x]\!]\, a\, (\mathit{Eq}\, a\, x)\, x\, (\mathit{refl}\, a\, x)\\ &\quad = \lambda a.\lambda x.(\lambda a.\lambda a_1.\lambda x.\lambda x_1.x_1)\, a\, (\mathit{Eq}\, a\, x)\, x\, (\mathit{refl}\, a\, x)\\ &\quad \to_\beta \lambda a.\lambda x.\mathit{refl}\, a\, x\end{aligned}$$

D. Internalization

We have seen that the abstraction theorem (Th. 1) for PTSs gives us something very close to Prop. 1, except that for each free variable $x : A$ in Γ, we need an explicit witness that x is parametric ($x_1 : x \in [\![A]\!]$) in the environment.

However, we know that every closed term is parametric. Therefore, ultimately, we know that for each possible *concrete* term a that can be substituted for a free variable x, it is possible to construct a concrete term $[\![a]\!]$ to substitute for x_1. This means that the witness of parametricity for x does not need to be given explicitly (if x is bound). Therefore we allow to access such a witness via the new syntactic form $\lceil\!\lceil x \rceil\!\rceil$. This intuition justifies the addition of the substitution rule

$$\lceil\!\lceil x \rceil\!\rceil [x \mapsto a] = \lceil\!\lceil a \rceil\!\rceil$$

as well as the following typing rule, expressing that if x is found in the context, then it is valid to use $\lceil\!\lceil x \rceil\!\rceil$, which is the witness that x satisfies the parametricity condition of its type.

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash \lceil\!\lceil x \rceil\!\rceil : x \in \llbracket A \rrbracket}$$

At the same time, we must amend the parametric interpretation to remember which variables are assigned an explicit witness, and which variables must wait for a concrete term. We write the list of assignments as an index to $\llbracket \cdot \rrbracket$. (From here on, we may omit the index $\llbracket A \rrbracket$ to mean $\llbracket A \rrbracket_\varnothing$.) For example, abstraction is translated as follows:

$$\llbracket \lambda x : A.\, B \rrbracket_\xi = \lambda x_0 : \{A\}_\xi .\, \lambda x_1 : x_0 \in \llbracket A \rrbracket_\xi .\, \llbracket B \rrbracket_{\xi, x \mapsto (x_0, x_1)}$$

and other cases are modified accordingly. In particular, the interpretation of variables becomes the following[2] (and $\{x\}_\xi$ is modified symmetrically):

$$\begin{aligned} \llbracket x \rrbracket_\xi &= x_1 && \text{if } x \mapsto (x_0, x_1) \in \xi \\ \llbracket x \rrbracket_\xi &= \lceil\!\lceil x \rceil\!\rceil && \text{if } x \notin \xi \end{aligned}$$

The above construction solves the issue of context extension. That is, every term A of a PTS $\mathcal{Q}$ can be proven parametric by using $\llbracket A \rrbracket$ without extending the context where A is typeable. Another aspect of the result is that, assuming parametricity on variables, we have parametricity for all terms. This means that, in a practical language featuring parametricity, the parametric construction can be used on any term, but in normal forms $\lceil\!\lceil \cdot \rceil\!\rceil$ only appears on variables, maybe in a nested way.

Unfortunately, Prop. 1 does not quite hold at this stage, because there is an issue in applying abstraction to the new $\lceil\!\lceil \cdot \rceil\!\rceil$ construct, as we show on an example in the next section.

E. Parametricity of parametricity

The fact that all values are parametric is also captured by the following theorem (inside the calculus):

$$\begin{aligned} &\mathit{parametricity} : \forall A : *.\, \forall (a : A).\, a \in \llbracket A \rrbracket \\ &\mathit{parametricity} = \lambda A.\, \lambda a : A.\, \lceil\!\lceil a \rceil\!\rceil \end{aligned}$$

Since all terms are assumed parametric, in particular it should be possible to apply $\llbracket \cdot \rrbracket$ to the above term. For some closed type A, consider the term $\llbracket \mathit{parametricity}\ A \rrbracket$. It is convertible to $\lambda x_0 : A.\, \lambda x_1 : x_0 \in \llbracket A \rrbracket .\, \llbracket \lceil\!\lceil x \rceil\!\rceil \rrbracket_{\{x \mapsto (x_0, x_1)\}}$.

So far, we have not specified how to reduce the subterm $\llbracket \lceil\!\lceil x \rceil\!\rceil \rrbracket_{\{x \mapsto (x_0, x_1)\}}$ (where x is a free variable). Indeed, it is actually not possible to substitute x for a value in it, because x also appears as an index of $\llbracket \cdot \rrbracket$, and only variables can appear there (not arbitrary terms). This means that $\llbracket \lceil\!\lceil x \rceil\!\rceil \rrbracket_{\{x \mapsto (x_0, x_1)\}}$ is not an acceptable normal form: it must reduce to something else. A perhaps natural idea is to modify the reduction rules in such a way as to allow the following reduction, which exchanges the two occurrences of the parametric interpretation:

$$\llbracket \lceil\!\lceil x \rceil\!\rceil \rrbracket_{\{x \mapsto (x_0, x_1)\}} \longrightarrow \lceil\!\lceil \llbracket x \rrbracket_{\{x \mapsto (x_0, x_1)\}} \rceil\!\rceil .$$

In that case the expression further reduces to $\lceil\!\lceil x_1 \rceil\!\rceil$, which is a proper normal form. Unfortunately, this reduction *does not preserve types*. This can be checked by assuming $x : A$, and by computing the types of the expression before and after reduction. By Prop. 1 we have $\lceil\!\lceil x \rceil\!\rceil : \llbracket A \rrbracket\, x$. By Abstraction (giving an explicit parametric witness for x), we get

$$\begin{aligned} \llbracket \lceil\!\lceil x \rceil\!\rceil \rrbracket_{\{x \mapsto (x_0, x_1)\}} &: \llbracket \llbracket A \rrbracket\, x \rrbracket_{\{x \mapsto (x_0, x_1)\}}\, \{\lceil\!\lceil x \rceil\!\rceil\}_{\{x \mapsto (x_0, x_1)\}} \\ &: \llbracket \llbracket A \rrbracket \rrbracket_{\{x \mapsto (x_0, x_1)\}}\, x_0\, \llbracket x \rrbracket_{\{x \mapsto (x_0, x_1)\}}\, \lceil\!\lceil x_0 \rceil\!\rceil \\ &: \llbracket \llbracket A \rrbracket \rrbracket x_0\, \llbracket x \rrbracket_{\{x \mapsto (x_0, x_1)\}}\, \lceil\!\lceil x_0 \rceil\!\rceil \\ &: \llbracket \llbracket A \rrbracket \rrbracket x_0\, x_1\, \lceil\!\lceil x_0 \rceil\!\rceil \end{aligned}$$

On the other hand, by Abstraction we have $\llbracket x \rrbracket_{\{x \mapsto (x_0, x_1)\}} : \llbracket A \rrbracket\, x_0$, and by application of Prop. 1, we get

$$\begin{aligned} \lceil\!\lceil \llbracket x \rrbracket_{\{x \mapsto (x_0, x_1)\}} \rceil\!\rceil &: \llbracket \llbracket A \rrbracket\, x_0 \rrbracket\, \llbracket x \rrbracket_{\{x \mapsto (x_0, x_1)\}} \\ &: \llbracket \llbracket A \rrbracket \rrbracket\, x_0\, \lceil\!\lceil x_0 \rceil\!\rceil\, \llbracket x \rrbracket_{\{x \mapsto (x_0, x_1)\}} \\ &: \llbracket \llbracket A \rrbracket \rrbracket\, x_0\, \lceil\!\lceil x_0 \rceil\!\rceil\, x_1 \end{aligned}$$

That is, in the above example, the reduction rule suggested above has the effect to swap the second and third arguments to $\llbracket \llbracket A \rrbracket \rrbracket$ in the type, which means that subject reduction would not hold if we were to have the above, naive rule.

However, one observes that, for a closed type A, the relation $\llbracket \llbracket A \rrbracket \rrbracket\, x$ is symmetric: a proof of $\llbracket \llbracket A \rrbracket \rrbracket\, x\, B\, C$ *is logically equivalent* to a proof of $\llbracket \llbracket A \rrbracket \rrbracket\, x\, C\, B$. Thus the swapping observed above is harmless, and it is sufficient to deal with it in a technical fashion.

Example 2. *The relation $\llbracket \llbracket (a : *) \to a \to a \rrbracket \rrbracket\, f$ is symmetric for any f.*

In the light of this observation, we introduce a special-purpose operator (pronounced exchange) $\cdot \ddagger^{\pi}$, which applies a permutation to the arguments of relations, and which permutes their types in the same way.

$$\frac{\Gamma \vdash A : B}{\Gamma \vdash A \ddagger^{\pi} : B \ddagger^{\pi}}$$

Thanks to this operation we can now have a reduction relation that preserves types in the above situation:

$$\llbracket \lceil\!\lceil x \rceil\!\rceil \rrbracket_{\{x \mapsto (x_0, x_1)\}} \longrightarrow \lceil\!\lceil \llbracket x \rrbracket_{\{x \mapsto (x_0, x_1)\}} \rceil\!\rceil \ddagger^{(1,2)} .$$

Supporting this operation requires deep changes in the syntax, exposed in the next section.

F. A syntax for hypercubes

In order to support the swapping operation, we need to indicate the role of each of the arguments to the relations explicitly, in the syntax. That is, the type of $\llbracket \lceil\!\lceil x \rceil\!\rceil \rrbracket_{\{x \mapsto (x_0, x_1)\}}$ should be written

$$\llbracket \llbracket A \rrbracket \rrbracket \bullet \begin{pmatrix} x_0 & x_1 \\ \lceil\!\lceil x_0 \rceil\!\rceil & \cdot \end{pmatrix} .$$

[2]Careful readers might worry that we discard the index in the second case. An informal justification is that if x has no explicit witness, then the free variables of its type do not either; thus types are preserved by this equation.

Then, we can arrange to have the following:

$$\left(\llbracket\llbracket A\rrbracket\rrbracket\bullet\begin{pmatrix} x_0 & x_1 \\ \llbracket x_0\rrbracket & \cdot \end{pmatrix}\right)\ddagger^{(1,2)} =_\beta \llbracket\llbracket A\rrbracket\rrbracket\bullet\begin{pmatrix} x_0 & \llbracket x_0\rrbracket \\ x_1 & \cdot \end{pmatrix}$$

In general, we need to remember the grouping of arguments when applying the relational interpretation. Essentially, one iteration of the relational interpretation transforms an application of an argument into application of two arguments. After a second iteration, there will be four arguments, and 2^n after n iterations. We must change the syntax of application to make these 2^n arguments appear grouped together. Abstraction and product follow the same pattern as application. Hence, we can arrange our bindings as oriented n-cubes in general. Using overbar to denote cube meta-variables, the syntax becomes the following:

$$\begin{array}{rcl} \mathsf{Term} & = & A\,\bar{B} \\ & \mid & \lambda\bar{x} : \bar{A}.\, B \\ & \mid & \forall\bar{x} : \bar{A}.\, B \\ & \dots & \end{array}$$

A binding $\bar{x} : \bar{B}$ introduces 2^n variables x_i, where i is any bit-vector of size n, and n is the dimension of $\bar{B}$. Consider the binding $\bar{x} : \bar{B}$. If $\bar{B}$ has dimension zero, it stands for a single binding $x : B$. If it has dimension 1, it contains a type B_0, and a predicate B_1 over B_0. Abusing matrix notation, one could write

$$\bar{x} : \begin{pmatrix} B_0 \\ B_1 \end{pmatrix} \simeq \begin{pmatrix} x_0 : B_0 \\ x_1 : B_1\,x_0 \end{pmatrix}$$

At dimension two, the cube $\bar{B}$ contains a type B_{00}, two predicates B_{01} and B_{10} over B_{00}, and a relation B_{11}, between B_{00}, $B_{10}\,x_{00}$, and $B_{01}\,x_{00}$.

$$\bar{x} : \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix} \simeq \begin{pmatrix} x_{00} : B_{00} & x_{01} : B_{01}\,x_{00} \\ x_{10} : B_{10}\,x_{00} & x_{11} : B_{11}\,x_{00}\,x_{01}\,x_{10} \end{pmatrix}$$

We furthermore need a special syntax for the introduction, elimination and formation of relations, which correspond to application, abstraction and quantification over incomplete cubes (those that lack an element at index 1...1). Such a cube is found for example in the type of x_{11} above. Using a check to denote incomplete cube meta-variables:

$$\begin{array}{rcl} \mathsf{Term} & = & A\bullet\check{B} \\ & \mid & \lambda^{\bullet}\check{x} : \check{A}.\, B \\ & \mid & \check{A} \overset{\bullet}{\to} B \\ & \dots & \end{array}$$

Using this syntax, we can finally write the type of x_{11} in the form we need: $B_{11}\bullet\begin{pmatrix} x_{00} & x_{01} \\ x_{10} & \cdot \end{pmatrix}$. The type of B_{11} is $\begin{pmatrix} B_{00} & B_{01} \\ B_{10} & \cdot \end{pmatrix} \overset{\bullet}{\to} s$. For a cube $\bar{B}$ of arbitrary dimension, $x_{1...1} : B_{1...1}\bullet(\bar{x}/\!\!/1...1)$ and $B_i : (\bar{B}/\!\!/i) \overset{\bullet}{\to} s$, where $\bar{B}/\!\!/1...1$ denotes the cube $\bar{B}$ with the top vertex removed. Further generalizing, x_i is a witness that the sub-cube found by removing all the dimensions d such that $i_d = 0$ satisfies the relation B_i:

$$x_i : B_i\bullet(\bar{x}/\!\!/i)$$

where $\bar{B}/\!\!/i$ is the cube obtained by discarding the elements of the cube $\bar{B}$ for each dimension d where $i_d = 0$, and then removing the top vertex.

$$\bar{B}/\!\!/i = \left[\ j \mapsto B_{j\&i}\ \right]^{\|i\|}_{j\in\mathbf{2}^{\|i\|}-1}$$

$$\text{where}\quad \|i\| = \textstyle\sum_d i_d \quad\text{and}\quad \begin{array}{rl} j\&(0i) & = 0(j\&i) \\ (bj)\&(1i) & = b(j\&i) \end{array}$$

B_i is then a relation over corresponding sub-cube of $\bar{B}$, which is written formally:

$$B_i : (\bar{B}/\!\!/i) \overset{\bullet}{\to} s$$

G. The interpretation of hypercubes

Having given the new syntax of terms, we can express the relational interpretation using this new syntax. The interpretation of a cube increases its dimension; to each element is associated its interpretation:

$$\llbracket\bar{A}\rrbracket_\xi = \begin{bmatrix} 0i \mapsto \{A_i\}_\xi \\ 1i \mapsto \llbracket A_i\rrbracket_\xi \end{bmatrix}^{\dim\bar{A}+1}_{i\in\mathbf{2}^{\dim\bar{A}}}$$

If a binding $\bar{x}$ has been extended by the interpretation, a variable x_i is then interpreted as x_{1i}.

$$\llbracket x_i\rrbracket_{\xi,x} = x_{1i}$$

The interpretation of terms mentioning full cubes (of size 2^n for some n) is the following:

$$\begin{aligned} \llbracket A\,\bar{B}\rrbracket_\xi &= \llbracket A\rrbracket_\xi\,\llbracket\bar{B}\rrbracket_\xi \\ \llbracket\lambda\bar{x} : \bar{A}.\,B\rrbracket_\xi &= \lambda\bar{x} : \llbracket\bar{A}\rrbracket_\xi.\,\llbracket B\rrbracket_{\xi,x\mapsto(x_0,x_1)} \\ C \in \llbracket\forall\bar{x} : \bar{A}.\,B\rrbracket_\xi &= \forall\bar{x} : \llbracket\bar{A}\rrbracket_\xi.\,(C\,(\bar{x}/01...1)) \in \llbracket B\rrbracket_{\xi,x\mapsto(x_0,x_1)} \end{aligned}$$

The interpretation of the cubes of size $2^n - 1$ used for relations requires some care. Because the index 1...1 is missing in such a cube, applying the same method as for full cubes leaves two elements missing, at indices 1...1 and 01...1. The former is supposed to be missing (because the resulting cube is also incomplete), but the latter is dependent on the context. Hence we introduce the following notation for interpretation of incomplete cubes where the "missing element" is explicitly specified to be B:

$$(\llbracket\check{A}\rrbracket_\xi \oplus B) = \begin{bmatrix} 0i \mapsto \{A_i\}_\xi \\ 1i \mapsto \llbracket A_i\rrbracket_\xi \\ 01...1 \mapsto B \end{bmatrix}^{\dim\check{A}+1}_{i\in\mathbf{2}^{\dim\check{A}}-1}$$

The parametric interpretation of the special forms for relation formation, membership and product are as follows.

$$\begin{aligned} C \in \llbracket\check{A} \overset{\bullet}{\to} s\rrbracket_\xi &= (\llbracket\check{A}\rrbracket_\xi \oplus C) \overset{\bullet}{\to} s \\ C \in \llbracket A\bullet\check{B}\rrbracket_\xi &= \llbracket A\rrbracket_\xi\bullet(\llbracket\check{B}\rrbracket_\xi \oplus C) \\ \llbracket\lambda^{\bullet}\check{x} : \check{A}.\,B\rrbracket_\xi &= \lambda^{\bullet}\check{x} : (\llbracket\check{A}\rrbracket_\xi \oplus (\lambda^{\bullet}\check{x} : \check{A}.\,B)). \\ & \qquad x_{01...1} \in \llbracket B\rrbracket_{\xi,x\mapsto(x_0,x_1)} \end{aligned}$$

They are a straightforward consequence of the usual parametric interpretation and our choice of grouping arguments in cubes. Readers familiar with realizability interpretations (in the style for example of [22]) will notice a similarity here: the interpretation of a function space adds a quantification; and the other forms behave accordingly. Note that the form $A{\bullet}\check{B}$ is always a type, and therefore we interpret it as such.

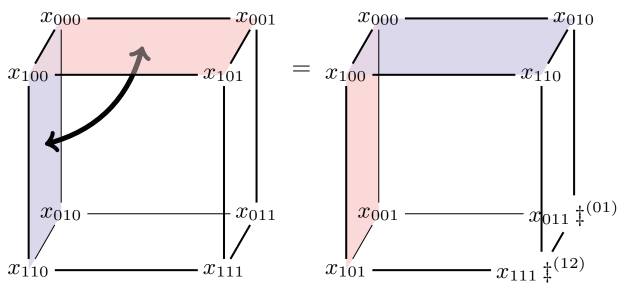
Example 3 (Repeated application of $[\![\cdot]\!]$)**.**

$$[\![\mathit{parametricity}\ A]\!] = \lambda\bar{a} : [\![(A)]\!].\ \lceil\!\lceil a_1 \rceil\!\rceil \ddagger^{(01)}$$

where $\bar{a} : [\![(A)]\!]$ *can be understood as* $\begin{pmatrix} a_0 \\ a_1 \end{pmatrix} : \begin{pmatrix} A \\ a_0 \in [\![A]\!] \end{pmatrix}$.

H. Exchanging dimensions

Given the above definition of cubes, it is straightforward to define an operation that applies an arbitrary permutation of its dimensions. For dimension $n = 0$ or $n = 1$, there is no non-trivial permutation. In the case of a square ($n = 2$), there is only one permutation, which is a simple swapping of the elements at indices 01 and 10. For higher dimensions ($n \geq 3$), the elements of the cube are multidimensional themselves (the dimension of an element at index i is $\|i\|$). Thus, one must take care to perform the exchange properly for each element. For instance, performing an exchange of dimensions 1 and 2 in a cube $\bar{x}$ for $n = 3$ involves exchanging dimensions 0 and 1 of the element x_{011}. Indeed, exchanging the dimensions 1 and 2 in the cube has the effect to exchange dimensions in the square occupied by x_{011}; so an exchange has to be performed on x_{011} to restore the cube structure. Geometrically, exchanging the dimensions as above corresponds to twisting the cube: two faces are swapped, and another is twisted. The situation is shown graphically in the following picture.



In general, applying a permutation π on the dimensions of a cube $\bar{C}$ is done as follows:

Definition 1 (Cube exchange)**.**

$$\bar{C} \ddagger^{\pi} = \left[\ i \mapsto C_{\pi(i)} \ddagger^{\pi/i}\ \right]_{i \in \mathbf{2}^{\dim \bar{C}}}^{\dim \bar{C}}$$

Where π/i *stands for the permutation* π *restricted to the dimensions* d *where* $i_d = 1$.
Incomplete cubes are permuted in the same way (simply omitting the top vertex).

Definition 2. *If* π *is a permutation* $\{d \mapsto x_d\}$,
$\pi/i = \mathrm{canon}\{d \mapsto x_d \mid i_d = 1\}$, *where* canon *maps the domain and co-domain of the function* $\{d \mapsto x_d \mid i_d = 1\}$ *to the set* $\{0..\|i\| - 1\}$, *preserving the order.*

Example 4. *If* $\pi = \{0 \mapsto 0, 1 \mapsto 2, 2 \mapsto 1\}$ *swaps dimensions 1 and 2, we have*

i	$\{d \mapsto \pi(d) \mid i_d = 1\}$	π/i
001	$\{2 \mapsto 1\}$	$\{0 \mapsto 0\}$
010	$\{1 \mapsto 2\}$	$\{0 \mapsto 0\}$
100	$\{0 \mapsto 0\}$	$\{0 \mapsto 0\}$
011	$\{1 \mapsto 2, 2 \mapsto 1\}$	$\{0 \mapsto 1, 1 \mapsto 0\}$
101	$\{0 \mapsto 0, 2 \mapsto 1\}$	$\{0 \mapsto 0, 1 \mapsto 1\}$
110	$\{0 \mapsto 0, 1 \mapsto 2\}$	$\{0 \mapsto 0, 1 \mapsto 1\}$

Applying a permutation to terms is then a matter of permuting all the cubes encountered:

$$\begin{aligned}(A\,\bar{B}) \ddagger^{\pi}_{\xi} &= A \ddagger^{\pi}_{\xi}\ \bar{B} \ddagger^{\pi}_{\xi} \\ (\lambda\bar{x} : \bar{A}.\, B) \ddagger^{\pi}_{\xi} &= \lambda\bar{x} : \bar{A} \ddagger^{\pi}_{\xi} .\, B[\bar{x} \mapsto \bar{x} \ddagger^{\pi}] \ddagger^{\pi}_{\xi,x} \\ (\forall\bar{x} : \bar{A}.\, B) \ddagger^{\pi}_{\xi} &= \forall\bar{x} : \bar{A} \ddagger^{\pi}_{\xi} .\, B[\bar{x} \mapsto \bar{x} \ddagger^{\pi}] \ddagger^{\pi}_{\xi,x}\end{aligned}$$

(and similarly for the incomplete cubes). It remains to explain the interaction with the special constructs, $[\![\cdot]\!]$ and $\cdot \ddagger^{\cdot}$ itself. We do so by listing four laws which hold in our calculus.

The first law is not surprising: the composition of exchanges is the exchange of the composition.

$$A \ddagger^{\rho} \ddagger^{\pi} =_{\beta} A \ddagger^{\rho \circ \pi} \tag{1}$$

Regarding the interactions between $[\![\cdot]\!]$ and $\cdot \ddagger^{\pi}$, recall first that the relational interpretation adds one dimension to cubes. By convention, the dimension added by $[\![\cdot]\!]$ is at index 0, and all other dimensions are shifted by one. Therefore, the relational interpretation of an exchange merely lifts the exchange out, and shifts indices by ones in its permutation, leaving dimension 0 intact.

$$[\![A \ddagger^{(x_1\ x_2 \cdots x_n)}]\!] =_{\beta} [\![A]\!] \ddagger^{(x_1+1\ x_2+1 \cdots x_n+1)} \tag{2}$$

The law that motivates the introduction of exchanges is the following:

$$[\![[\![A]\!]]\!]_{\xi} =_{\beta} [\![[\![A]\!]_{\xi}]\!] \ddagger^{(01)} \tag{3}$$

This law can also be explained by the convention that $[\![\cdot]\!]$ inserts always dimension 0. By commuting the uses of parametricity, dimensions are be swapped, and the exchange operator restores the order.

Lastly, one can also simplify exchanges in the presence of symmetric terms. We know that a term $[\![A]\!]^n$ is symmetric in its n first dimensions. Thus, applying a permutation that touches only dimensions $0..n-1$ to such a term has no effect. Formally, we have:

$$[\![A]\!]^n \ddagger^{(x_1\ x_2 \cdots x_m)} =_{\beta} [\![A]\!]^n \qquad \text{if } \forall i \in 1..m, x_i < n \tag{4}$$

We have seen before that it suffices to provide parametricity only for variables, and that the construct $[\![\cdot]\!]$ essentially acts as a "macro" on other constructs. The situation is not changed in the presence of dimension exchanges: (2) explains how to compute the parametricity witness of an exchange. For the $\cdot \ddagger^{\pi}$ construct, the situation is analogous: it suffices to provide the construct for variables, perhaps themselves enclosed by $\lceil\!\lceil\cdot\rceil\!\rceil$. The reason is that the above laws give a way to compute the exchange for any term which is not a parametricity witness (The result is given in Def. 4). When we want to be explicit about exchange being the syntactic construct, we write simply $\mathbf{x}\dagger^{\pi}$. The syntax fragment for parametricity and exchanges is as follows.

$$\begin{array}{llcll}
\mathsf{Var} & \ni x, y, z & & & \\
\mathsf{Param} & \ni \mathbf{x} & ::= & x & \text{variable} \\
 & & \mid & \llbracket \mathbf{x} \rrbracket & \text{parametric witness} \\
\mathsf{Term} & \ni a, \ldots, u & ::= & \mathbf{x}\dagger^{\pi} & \text{permutation of dimensions} \\
 & & \mid & \ldots &
\end{array}$$

I. Dimension checks

If a permutation acts on dimensions 0 to $n-1$, every cube where it is applied to must exhibit at least n dimensions. So far we have not discussed this restriction, which is the final feature of the system to present. To implement it we choose to annotate sorts with the dimension of the type which inhabits it. The sort s at dimension n is written s^n. Hence, we can capture the restriction in the exchange rule.

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash A : s^n}{\Gamma \vdash A\ddagger^{\pi} : B\ddagger^{\pi}} \dim(\pi) \leq n$$

If a type inhabits a sort of dimension n, all the quantifications found inside the type must at least be over cubes of dimension n. This is realized in the product rule as follows:

$$\frac{\Gamma \vdash \bar{A} : s_1^n \qquad \Gamma, \bar{x} : \bar{A} \vdash B : s_2^m}{\Gamma \vdash (\forall \bar{x} : \bar{A}.\, B) : s_3^{m \sqcap n}}$$
$$\textsc{Product}\ (s_1, s_2, s_3) \in \mathcal{R}$$

Similarly, relations found in the type must be over cubes of dimension n.

J. Our calculus

The full definition of system $\mathcal{P}$, parameterized on a PTS specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$, is given in figures 1 and 2. For our system, the proof of parametricity (Prop. 1) goes through even for the case of the parametricity rule itself. The proof follows the structure of the abstraction theorem, with the difference that the START rule is translated to PARAM when an explicit witness is not available, and PARAM is translated to PARAM + EXCHANGE. We have proved confluence, subject reduction, and strong normalization for our calculus. Since parametricity acts as a typing rule for $\llbracket\cdot\rrbracket$, subject reduction for our calculus stems directly from it. Strong normalization is proved by modeling the system in the PTS without parametricity. This model is done by introducing explicit witnesses of parametricity for all variables. Details of the proofs of these theorems are found in the extended version of this paper [7].

The syntactic changes made to the system require theorems to be adapted accordingly. In the case of Example 1, (proving that any function of type $\forall a : *.\, a \to a$ is an identity), the definition of Equality must be amended to make it inhabit $*^1$. This mostly involves augmenting the dimension of cubes by adding unit types as indices:

Definition 3 (Relational interpretation)**.**

$$\begin{aligned}
\llbracket \llbracket x \rrbracket^n \rrbracket_\xi &= \llbracket x \rrbracket^{n+1} && \textit{if } x \notin \xi \\
\llbracket \llbracket x_i \rrbracket^n \rrbracket_\xi &= \llbracket x_{1i} \rrbracket^n \dagger^{(0..n)} && \textit{if } x \in \xi \\
\llbracket \mathbf{x}\dagger^{\pi} \rrbracket_\xi &= \llbracket \mathbf{x} \rrbracket_\xi \dagger^{\pi+1} && \\
\llbracket \lambda\bar{x} : \bar{A}.\, B \rrbracket_\xi &= \lambda\bar{x} : \llbracket \bar{A} \rrbracket_\xi .\, \llbracket B \rrbracket_{\xi, x \mapsto (x_0, x_1)} && \\
\llbracket \lambda^{\bullet}\check{x} : \check{A}.\, B \rrbracket_\xi &= \lambda^{\bullet}\check{x} : (\llbracket \check{A} \rrbracket_\xi \oplus (\lambda^{\bullet}\check{x} : \check{A}.\, B)). && \\
&\qquad\qquad x_{01\ldots1} \in \llbracket B \rrbracket_{\xi, x \mapsto (x_0, x_1)} && \\
\llbracket A\, \bar{B} \rrbracket_\xi &= \llbracket A \rrbracket_\xi\, \llbracket \bar{B} \rrbracket_\xi && \\
\llbracket T \rrbracket_\xi &= \lambda\check{z} : \begin{pmatrix} T \\ \cdot \end{pmatrix} .\, z_0 \in \llbracket T \rrbracket_\xi && \textit{if } T \textit{ is } \forall, \bullet \textit{ or } s^n
\end{aligned}$$

$$\begin{aligned}
C \in \llbracket s^n \rrbracket_\xi &= \begin{pmatrix} C \\ \cdot \end{pmatrix} \overset{\bullet}{\to} s^{n+1} && \\
C \in \llbracket \forall\bar{x} : \bar{A}.\, B \rrbracket_\xi &= \forall\bar{x} : \llbracket \bar{A} \rrbracket_\xi .\, (C\, (\bar{x}/01\ldots1)) \in \llbracket B \rrbracket_{\xi, x \mapsto (x_0, x_1)} && \\
C \in \llbracket \check{A} \overset{\bullet}{\to} s^n \rrbracket_\xi &= (\llbracket \check{A} \rrbracket_\xi \oplus C) \overset{\bullet}{\to} s^{n+1} && \\
C \in \llbracket A \bullet \check{B} \rrbracket_\xi &= \llbracket A \rrbracket_\xi \bullet (\llbracket \check{B} \rrbracket_\xi \oplus C) && \\
C \in \llbracket T \rrbracket_\xi &= \llbracket T \rrbracket_\xi \bullet \begin{pmatrix} C \\ \cdot \end{pmatrix} && \textit{if } T \textit{ is not } \forall, \bullet \textit{ nor } s^n
\end{aligned}$$

$$\begin{aligned}
\llbracket - \rrbracket_\xi &= - && \\
\llbracket \Gamma, x : A \rrbracket_{\xi, x \mapsto (x_0, x_1)} &= \llbracket \Gamma \rrbracket_\xi, x_0 : A, x_1 : x_0 \in \llbracket A \rrbracket_\xi && \textit{if } x \in \xi \\
\llbracket \Gamma, x : A \rrbracket_\xi &= \llbracket \Gamma \rrbracket_\xi, x : A && \textit{if } x \notin \xi
\end{aligned}$$

$$\llbracket \bar{A} \rrbracket_\xi = \begin{bmatrix} 0i \mapsto \{A_i\}_\xi \\ 1i \mapsto \llbracket A_i \rrbracket_\xi \end{bmatrix}^{\operatorname{dims} \bar{A}+1}_{i \in \mathbf{2}^{\operatorname{dims} \bar{A}}}$$

$$(\llbracket \check{A} \rrbracket_\xi \oplus B) = \begin{bmatrix} 0i \mapsto \{A_i\}_\xi \\ 1i \mapsto \llbracket A_i \rrbracket_\xi \\ 01\ldots1 \mapsto B \end{bmatrix}^{\operatorname{dims} \check{A}+1}_{i \in \mathbf{2}^{\operatorname{dims} \check{A}-1}}$$

Definition 4 (Term exchange)**.**

$$\begin{aligned}
\llbracket x \rrbracket^n \dagger^{\rho} \ddagger^{\pi}_{\xi} &= \llbracket x \rrbracket^n \dagger^{\rho} && \textit{if } x \in \xi \\
\llbracket x \rrbracket^n \dagger^{\rho} \ddagger^{\pi}_{\xi} &= \llbracket x \rrbracket^n \dagger^{\mathrm{normal}_n(\pi \circ \rho)} && \textit{if } x \notin \xi \\
(A\, \bar{B}) \ddagger^{\pi}_{\xi} &= A \ddagger^{\pi}_{\xi}\ \bar{B} \ddagger^{\pi}_{\xi} && \\
(\lambda\bar{x} : \bar{A}.\, B) \ddagger^{\pi}_{\xi} &= \lambda\bar{x} : \bar{A} \ddagger^{\pi}_{\xi} .\, B[\bar{x} \mapsto \bar{x} \ddagger^{\pi}] \ddagger^{\pi}_{\xi, x} && \\
(\forall\bar{x} : \bar{A}.\, B) \ddagger^{\pi}_{\xi} &= \forall\bar{x} : \bar{A} \ddagger^{\pi}_{\xi} .\, B[\bar{x} \mapsto \bar{x} \ddagger^{\pi}] \ddagger^{\pi}_{\xi, x} && \\
(A \bullet \check{B}) \ddagger^{\pi}_{\xi} &= A \ddagger^{\pi}_{\xi} \bullet \bar{B} \ddagger^{\pi}_{\xi} && \\
(\lambda^{\bullet}\check{x} : \check{A}.\, B) \ddagger^{\pi}_{\xi} &= \lambda^{\bullet}\check{x} : \check{A} \ddagger^{\pi}_{\xi} .\, B[\bar{x} \mapsto \bar{x} \ddagger^{\pi}] \ddagger^{\pi}_{\xi, x} && \\
(\check{A} \overset{\bullet}{\to} s^n) \ddagger^{\pi}_{\xi} &= \check{A} \ddagger^{\pi}_{\xi} \overset{\bullet}{\to} s^n && \\
s^n \ddagger^{\pi}_{\xi} &= s^n &&
\end{aligned}$$

where $\mathrm{normal}_n(\pi)$ *removes all cycles of* π *entirely contained in* $0..n-1$.

Fig. 1. Relational interpretation of terms

Definition 5 (Syntax)**.**

$$
\begin{array}{llccc}
\textsf{Sort} & \ni s, s_1, s_2, s_3 & ::= & \mathcal{S} & \\
\textsf{Var} & \ni x, y, z & & & \\
\textsf{Param} & \ni \mathbf{x} & ::= & x & \textit{variable} \\
& & | & [\![\mathbf{x}]\!] & \textit{parametric witness} \\
\textsf{Term} & \ni a, \ldots, u & ::= & \mathbf{x}\,\dagger^{\pi} & \textit{permutation of dimensions} \\
& \quad A, \ldots, U & | & s^n & \textit{sort at dimension } n \\
& & | & A\,\bar{B} & \textit{application (of hypercubes)} \\
& & | & \lambda\bar{x} : \bar{A}.\, B & \textit{abstraction (of hypercubes)} \\
& & | & \forall\bar{x} : \bar{A}.\, B & \textit{function space} \\
& & | & A\bullet\breve{B} & \textit{relation membership} \\
& & | & \lambda^{\bullet}\breve{x} : \breve{A}.\, B & \textit{relation formation} \\
& & | & \breve{A} \overset{\bullet}{\to} s^n & \textit{relation space} \\
\textsf{Cube} & \ni \bar{A} & ::= & \left[\ i \mapsto A_i\ \right]^{n}_{i \in \mathbf{2}^n} & \textit{cube of size } 2^n \\
\textsf{Cube}' & \ni \breve{A} & ::= & \left[\ i \mapsto A_i\ \right]^{n}_{i \in \mathbf{2}^n - 1} & \textit{cube of size } 2^n - 1 \\
\textsf{Context} & \ni \Gamma, \Delta & ::= & - & \textit{empty context} \\
& & | & \Gamma, x : A & \textit{context extension}
\end{array}
$$

Definition 6 (Typing rules)**.**

$$\dfrac{}{\vdash s_1^n : s_2^n}\ (s_1, s_2) \in \mathcal{A}$$
AXIOM

$$\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s^n}{\Gamma, x : C \vdash A : B}$$
WEAKENING

$$\dfrac{\Gamma \vdash F : (\breve{A} \overset{\bullet}{\to} s^n) \qquad \Gamma \vdash \breve{a} : \breve{A}}{\Gamma \vdash F\bullet\breve{a} : s^n}$$
REL-ELIM

$$\dfrac{\Gamma, \breve{x} : \breve{A} \vdash B : s^n \qquad \Gamma \vdash \breve{A} : s^n}{\Gamma \vdash (\lambda^{\bullet}\breve{x} : \breve{A}.\, B) : (\breve{A} \overset{\bullet}{\to} s^n)}$$
REL-INTRO

$$\dfrac{\Gamma \vdash \breve{A} : s_1^n}{\Gamma \vdash (\breve{A} \overset{\bullet}{\to} s_1^n) : s_2^n}$$
REL-FORM $(s_1, s_2) \in \mathcal{A}$

$$\dfrac{\Gamma \vdash F : (\forall\bar{x} : \bar{A}.\, B) \qquad \Gamma \vdash \bar{a} : \bar{A}}{\Gamma \vdash F\,\bar{a} : B[\bar{x} \mapsto \bar{a}]}$$
APPLICATION

$$\dfrac{\Gamma, \bar{x} : \bar{A} \vdash b : B \qquad \Gamma \vdash (\forall\bar{x} : \bar{A}.\, B) : s^n}{\Gamma \vdash (\lambda\bar{x} : \bar{A}.\, b) : (\forall\bar{x} : \bar{A}.\, B)}$$
ABSTRACTION

$$\dfrac{\Gamma \vdash \bar{A} : s_1^n \qquad \Gamma, \bar{x} : \bar{A} \vdash B : s_2^m}{\Gamma \vdash (\forall\bar{x} : \bar{A}.\, B) : s_3^{m \sqcap n}}$$
PRODUCT $(s_1, s_2, s_3) \in \mathcal{R}$

$$\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : s^n \qquad B =_\beta B'}{\Gamma \vdash A : B'}$$
CONVERSION

$$\dfrac{\Gamma \vdash A : s^n}{\Gamma, x : A \vdash x : A}$$
START

$$\dfrac{\Gamma \vdash \mathbf{x} : A}{\Gamma \vdash [\![\mathbf{x}]\!] : \mathbf{x} \in [\![A]\!]_{\varnothing}}$$
PARAM

$$\dfrac{\Gamma \vdash \mathbf{x} : A \qquad \Gamma \vdash A : s^n}{\Gamma \vdash \mathbf{x}\,\dagger^{\pi} : A\,\ddagger^{\pi}}\ \dim(\pi) \leq n$$
EXCHANGE

- $\mathbf{2}^n$ *stands for all bit-vectors of size* n*; and* $\mathbf{2}^n - 1$ *stands for all bit-vectors of size* n*, except* 1...1.
- $\operatorname{ind}(\bar{A})$ *stands for* $\mathbf{2}^{\operatorname{dims}\bar{A}}$*; and* $\operatorname{ind}(\breve{A})$ *stands for* $\mathbf{2}^{\operatorname{dims}\breve{A}} - 1$.
- $\bar{x} : \bar{A}$ *stands for the bindings* $x_i : A_i\bullet(\bar{x}/\!/i)$ *where* $i \in \operatorname{ind}(\bar{A})$*; and* $\breve{x} : \breve{A}$ *stands for the bindings* $x_i : A_i\bullet(\breve{x}/\!/i)$ *where* $i \in \operatorname{ind}(\breve{A})$.
- *The typing judgement* $\Gamma \vdash \bar{a} : \bar{A}$ *stands for the conjunction of the judgements* $\Gamma \vdash a_i : A_i\bullet(\bar{a}/\!/i)$*, where* $i \in \operatorname{ind}(\bar{A})$*;*
  $\Gamma \vdash \breve{a} : \breve{A}$ *stands for the conjunction of the judgements* $\Gamma \vdash a_i : A_i\bullet(\breve{a}/\!/i)$*, where* $i \in \operatorname{ind}(\breve{A})$
- *Similarly,* $\bar{A} : s^n$ *stands for* $A_i : \bar{A}/\!/i \overset{\bullet}{\to} s^{\|i\|}$ *and* $\breve{A} : s^n$ *stands for* $A_i : \breve{A}/\!/i \overset{\bullet}{\to} s^{\|i\|}$.
- *The substitution* $\bar{x} \mapsto \bar{a}$ *stands for the parallel substitution* $x_i \mapsto a_i$ *for* $i \in \operatorname{ind}(\bar{a})$.

Fig. 2. Syntax and typing rules of $\mathcal{P}$, parameterized on a PTS specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$.

$$
\begin{aligned}
&Eq : \forall a : *.\, a \to \begin{pmatrix} a \\ \cdot \end{pmatrix} \overset{\bullet}{\to} *^1 \\
&Eq = \lambda a : *.\, \lambda x : A.\, \lambda^{\bullet}\begin{pmatrix} y \\ \cdot \end{pmatrix} : \begin{pmatrix} a \\ \cdot \end{pmatrix}.\, \forall \begin{pmatrix} - \\ P \end{pmatrix} : \begin{pmatrix} \top \\ \begin{pmatrix} a \\ \cdot \end{pmatrix} \overset{\bullet}{\to} *^1 \end{pmatrix}. \\
&\qquad \begin{pmatrix} \top \\ P\bullet\begin{pmatrix} x \\ \cdot \end{pmatrix} \end{pmatrix} \to P\bullet\begin{pmatrix} y \\ \cdot \end{pmatrix}
\end{aligned}
$$

The proof term needs fewer amendments:

$$
\begin{aligned}
\mathit{identities} &: \forall f : (\forall a : *.\, a \to a). \\
&\qquad \forall a : *.\, \forall x : a.\, Eq\,a\,(f\,a\,x)\bullet\begin{pmatrix} x \\ \cdot \end{pmatrix} \\
\mathit{identities} &= \lambda f.\lambda a.\lambda x.[\![f]\!] \begin{pmatrix} a \\ Eq\,a\,x \end{pmatrix} \begin{pmatrix} x \\ \mathit{refl}\,a\,x \end{pmatrix}
\end{aligned}
$$

$$[\![f]\!] : \forall \begin{pmatrix} a_0 \\ a_1 \end{pmatrix} : \begin{pmatrix} * \\ a_0 \overset{\bullet}{\to} *^1 \end{pmatrix}.\, \forall \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} : \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}.\, a_1\bullet(f\,a_0\,x_0)$$

III. DISCUSSION

A. Alternative design: named dimensions

One may wonder if the introduction of the permutation operation on cubes is strictly necessary. In fact, it is a technical consequence of the convention that $[\![\cdot]\!]$ always adds dimension 0. Therefore, if we were to *name* dimensions, the need for permuting dimensions would disappear. However, we would instead have to deal with renaming of dimensions, which is conceptually as complex as permutations.

The situation is analogous to the issue of the representation of variables in lambda-calculi. One can either use explicit names or De Brujin indices (and we have chosen indices here). Where abstraction introduces new variables, parametricity introduces new dimensions; where application substitutes a variable by its value, permutation substitutes dimensions by others.

B. Extension: n-ary parametricity

Here we have presented only unary parametricity, whereas most of the literature deals with binary parametricity. The generalization to arbitrary arity can be done in a straightforward way by mere syntactic duplication of relation indices, as detailed by Bernardy et al. [9] (Instead of x being interpreted as and index x_0 and a witness of parametricity x_1, we'd have many indices $x_0 ... x_{n-1}$ and a witness x_n that they are related by the parametric interpretation of the type.)

However, we have shown that by n iterations of unary parametricity, one gets n-ary parametricity between the faces of an n-cube. If one were to *erase* the indices of these n faces (which are themselves cubes of dimension $n-1$), one would recover n-ary parametricity. This insight is developed by Bernardy and Lasson [6], with the difference that, in that work, sorts are used to determine which parts of terms must be erased. In future work we want to investigate the addition of an erasure operation based on the shape of cubes, which would make explicit the relation between parametricity at different arities within the calculus.

C. Extension: inductive types

Even though we considered only PTSs, it is straightforward to support the addition of inductive constructions. This can be done in the same way as Bernardy et al. [9]. Namely, the interpretation of an inductive definition is another inductive definition obtained by applying relational interpretation to every component of the original definition (type and constructors).

Example 5 (Inductive naturals and their interpretation)**.**

$$\begin{aligned} &\mathbf{data}\ \mathsf{N} : \mathsf{Type}\ \mathbf{where} \\ &\quad \mathsf{zer} : \mathsf{N} \\ &\quad \mathsf{suc} : \mathsf{N} \to \mathsf{N} \end{aligned}$$

$$\begin{aligned} &\mathbf{data}\ [\![\mathsf{N}]\!] : \begin{pmatrix} \mathsf{N} \\ . \end{pmatrix} \overset{\bullet}{\to} \mathsf{Type}\ \mathbf{where} \\ &\quad [\![\mathsf{zer}]\!] : [\![\mathsf{N}]\!] \bullet \begin{pmatrix} \mathsf{zer} \\ . \end{pmatrix} \\ &\quad [\![\mathsf{suc}]\!] : \begin{pmatrix} n_0 \\ n_1 \end{pmatrix} : \begin{pmatrix} \mathsf{N} \\ [\![\mathsf{N}]\!] \end{pmatrix} \to [\![\mathsf{N}]\!] \bullet \begin{pmatrix} \mathsf{suc}\, n \\ . \end{pmatrix} \end{aligned}$$

In our case we need also to verify that repetition of the interpretation yields symmetric relations, which is straightforward.

Usually, inductive definitions may inhabit any sort. In our case, if one wishes to introduce a definition at a dimension greater than zero, it must respect symmetry. That is, an inductive definition $I : s^n$ is legal only if, for any permutation π of the dimensions 0 to $n-1$, $I \ddagger^{\pi} = I$.

D. Implementation

An implementation of the calculus, realized in the Haskell programming language, is available for experimentation (http://hackage.haskell.org/package/uAgda).

E. Related Work

The relationship between logic and programming languages goes both ways. On the one hand, logical systems can be given meaning by assigning a computational interpretation to them. On the other hand, one would like to prove programs correct within formal logical systems. This means that one needs logical systems with enough power to support reasoning about programs. Seminal work featuring both sides of the connection includes [20] and [18].

The work described in this paper falls right into this tradition: we not only attempt to improve the support for reasoning about parametric reasoning in a logical framework, but also give parametricity a computational meaning. Furthermore, our parametricity construct is itself parametric. To our knowledge this has not been achieved before. Some logical frameworks with computational meaning have features which are related to parametricity, without subsuming it. Some logical systems have featured parametricity, but not in a computationally meaningful way.

a) Inductive families: In the first category, we must first mention inductive families, studied in different contexts by Pfenning and Paulin-Mohring [23], and Dybjer [12].

The computational realization of the induction principle over an inductive definition is a recursive function over the data, but which also carries information about the data being recursed over.

One can draw a parallel with our interpretation of terms: computational constructs such as application and abstraction are interpreted as application and abstraction, but with additional information about the original term carried as an index. In fact, by taking advantage of extensionality, it is possible to see induction principles over data as a special case of parametricity, as shown for example by Wadler [32].

b) Extensionality: The ability to reason about functions can be supported not just by parametricity, but also by the principle of extensionality, which says that two functions are equal if they map equal inputs to equal outputs.

Altenkirch et al. [2] show how to integrate extensionality in a computationally meaningful way. Parallels can be drawn between the work of Altenkirch et al. and ours. Mainly, their equality relation depends on the structure of the types that it relates, in the same way as the interpretation of types we propose.

c) Higher-dimensional type-theories: Recent work on the interpretation of the equality-type in intensional type theory suggests that it should be modeled using higher-dimensional structures [16]. In this work we have found the need to introduce higher-dimensional objects as well, but to interpret parametricity instead of equality. Furthermore it has been known since [26] that for System F, the parametric interpretation of closed types coincides with equality. We hope that this work will help extending the connection to intensional type theory.

d) Meta-level reasoning: Miller and Tiu [19] propose a logical framework where one can reason precisely about terms and types of an object language. Their domain of application overlaps with ours. For example, one can prove in their framework that the object type $\forall a : \star.a$ is uninhabited. A characteristic of [19] is the total separation between object and host language. Here, we are able to unify both languages.

e) Parametricity: Plotkin and Abadi [24] have formulated a logic extended with axioms of parametricity. However, they are content with the consistency of parametricity, and do not give any computational interpretation for it.

As mentioned previously, it has been shown before *e.g.*, by Hasegawa [14], that parametricity is consistent with some models of System F. Consistency of parametricity with dependently-typed theories does not appear to have been proved previously.

In unpublished work, Takeuti [27] attempted to extend CC with parametricity, but Takeuti does not attempt to assign a computational content to his parametricity axioms, and only conjectures consistency of his system.

Bernardy et al. [9] have described logical relations from PTSs to PTSs, but in an *external* fashion: the abstraction theorem itself remains outside the system, as we have exposed in detail in Sec. II.

F. Future Work

A natural application of this work would be to integrate it in a real proof assistant (*e.g.*, COQ or AGDA). One could then experiment and analyze how much power parametricity gives on practical examples.

We also plan to analyze the relationship parametricity with other extensions of type-theory, such as for example (co)inductive constructions or extensionality. Because parametricity is a powerful reasoning principle, it might be possible to understand other aspects of type-theory in terms of it. In particular, understanding data in terms of their Church-Encodings is a tempting application.

Acknowledgments We are grateful to Thierry Coquand, Peter Dybjer and Simon Huber for helpful comments and discussions.

REFERENCES

[1] M. Abadi, L. Cardelli, and P. Curien. Formal parametric polymorphism. In *Proc. of POPL'93*, pages 157–170. ACM, 1993.
[2] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *PLPV 2007*, pages 57–68. ACM, 2007.
[3] R. Atkey, S. Lindley, and J. Yallop. Unembedding domain-specific languages. In *Proc. of Haskell '09*, pages 37–48. ACM, 2009.
[4] H. P. Barendregt. Lambda calculi with types. *Handbook of logic in computer science*, 2:117–309, 1992.
[5] J.-P. Bernardy. *A theory of parametric polymorphism and an application*. Phd thesis, Chalmers Tekniska Högskola, 2011.
[6] J.-P. Bernardy and M. Lasson. Realizability and parametricity in pure type systems. In M. Hofmann, editor, *FoSSaCS*, volume 6604 of *LNCS*, pages 108–122. Springer, 2011.
[7] J.-P. Bernardy and G. Moulin. A computational interpretation of parametricity. http://publications.lib.chalmers.se/cpl/record/index.xsql?pubid=153094, 2012.
[8] J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In A. Gordon, editor, *European Symposium on Programming*, volume 6012 of *LNCS*, pages 125–144. Springer, 2010.
[9] J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *Proc. of ICFP 2010*, pages 345–356. ACM, 2010.
[10] A. Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceeding of ICFP 2008*, pages 143–156. ACM, 2008.
[11] T. Coquand and G. Huet. The calculus of constructions. Technical report, INRIA, 1986.
[12] P. Dybjer. Inductive families. *Formal Aspects of Comp.*, 6(4): 440–465, 1994.
[13] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proc. of FPCA*, pages 223–232. ACM, 1993.
[14] R. Hasegawa. Categorical data types in parametric polymorphism. *Mathematical Structures in Comp. Sci.*, 4(01):71–109, 1994.
[15] P. Johann. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbol. Comp.*, 15(4):273–300, 2002.
[16] D. Licata and R. Harper. Canonicity for 2-dimensional type theory. In *Proc. of POPL 2012*. ACM, 2012.
[17] H. Mairson. Outline of a proof theory of parametricity. In *Proc. of FPCA 1991*, volume 523 of *LNCS*, pages 313–327. Springer-Verlag, 1991.
[18] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, 1984.
[19] D. A. Miller and A. F. Tiu. A proof theory for generic judgments: An extended abstract. In *LICS 2003*, pages 118–127. IEEE, 2003.
[20] R. Milner. Logic for Computable Functions: description of a machine implementation. *Artificial Intelligence*, 1972.
[21] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers Tekniska Högskola, 2007.
[22] C. Paulin-Mohring. Extracting Fω's programs from proofs in the calculus of constructions. In *POPL'89*, pages 89–104. ACM, 1989.
[23] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the calculus of constructions. In *MFPS*, volume 442 of *LNCS*, pages 209–228. Springer, 1990.
[24] G. Plotkin and M. Abadi. A logic for parametric polymorphism. In *Proc. of TLCA*, volume 664 of *LNCS*, page 361–375. Springer, 1993.
[25] N. Pouillard. Nameless, painless. In *Proc. of the 16th ACM SIGPLAN international conference on Funct. programming*, ICFP '11. ACM, 2011. to appear.
[26] J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information processing*, 83(1):513–523, 1983.
[27] I. Takeuti. The theory of parametricity in lambda cube. Manuscript, 2004.
[28] The Coq development team. The Coq proof assistant, 2011.
[29] J. Voigtländer. Bidirectionalization for free! (Pearl). In *Proc. of POPL 2009*, pages 165–176. ACM, 2009.
[30] D. Vytiniotis and S. Weirich. Parametricity, type equality, and higher-order polymorphism. *J. Funct. Program.*, 20(02):175–210, 2010.
[31] P. Wadler. Theorems for free! In *Proc. of FPCA 1989*, pages 347–359. ACM, 1989.
[32] P. Wadler. The Girard–Reynolds isomorphism. *Theor. Comp. Sci.*, 375(1–3):201–226, 2007.