Bernardy, J-P., Jansson, P. & Paterson, R. A. (2010). Parametricity and Dependent Types. Paper presented at the International Conference on Functional Programming, 27-09-2010 - 29-09-2010, Baltimore, USA.

Parametricity and Dependent Types

Jean-Philippe Bernardy Patrik Jansson
Chalmers University of Technology and
University of Gothenburg
{bernardy,patrikj}@chalmers.se

Ross Paterson
City University London
ross@soi.city.ac.uk

Abstract

Reynolds' abstraction theorem shows how a typing judgement in System F can be translated into a relational statement (in second order predicate logic) about inhabitants of the type. We obtain a similar result for a single lambda calculus (a pure type system), in which terms, types and their relations are expressed. Working within a single system dispenses with the need for an interpretation layer, allowing for an unusually simple presentation. While the unification puts some constraints on the type system (which we spell out), the result applies to many interesting cases, including dependently-typed ones.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type Structure

General Terms Languages, Theory

Keywords Pure type system, Abstraction theorem, Free theorems

1. Introduction

Reynolds [1983] defined a relational interpretation of System F types, and showed that interpretations of a term of that type in related contexts yield related results. He was thus able to constrain interpretations of polymorphic types.

Wadler [1989] observed that if a type has no free variables, the relational interpretation can thus be viewed as a *parametricity* property satisfied by all terms of that type. Such properties have been used in a variety of situations. A few examples include:

program transformation The $fold/build$ rule can be used to remove intermediate lists [Gill et al. 1993]. Its correctness can be proved using the parametricity condition derived from the type of the function $build$ [Johann 2002].

testing The testing of a polymorphic function can often be reduced to testing a single monomorphic instance. Bernardy et al. [2010a] present a scheme for constructing such a monomorphic instance for which the correctness proof relies on parametricity.

automatic program inversion It is possible to write a function that inverts a polymorphic function given as input. The inversion process essentially relies on the parametric behaviour of the input function, and therefore its correctness relies on the corresponding parametricity condition [Voigtländer 2009a].

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

generic programming In a certain style of generic programming, functions can be type-indexed. However, in some cases it is useful to show that functions behave uniformly for all types. Vytiniotis and Weirich [2009] use parametricity to show that certain casting functions are equivalent to the identity.

encoding of inductive types Via Church-encodings, inductive types can be encoded in pure System F. The proof of isomorphism relies on the parametricity condition. Hinze [2005] gives an illuminating example.

Parametricity in System F is useful enough that there has been much research to transport it to related calculi. Johann and Voigtländer [2005] have applied it to a system with explicit strictness; Vytiniotis and Weirich [2010] to F_ω extended with representation types; Takeuti [2004] sketches how it can be applied to the λ-cube, Neis et al. [2009] to a system with dynamic casting. In this paper, we apply Reynolds' idea to *dependently-typed* systems. In fact, we go one step further and generalize to a large class of *pure type systems* [Barendregt 1992].

By targeting pure type systems (PTSs), we aim to provide a framework which unifies previous descriptions of parametricity and forms a basis for future studies of parametricity in specific type systems. As a by-product, we get parametricity for dependently-typed languages.

Our specific contributions are:

- A concise definition of the translation of types to relations (Definition 4), which yields parametricity propositions for PTSs.
- A formulation (and a proof) of the abstraction theorem for a useful class of PTSs (Theorem 1). A remarkable feature of the theorem is that the translation from types to relations and the translations from terms to proofs are unified.
- An extension of the translation to inductive definitions (Section 4). Our examples use a notation close to that of Agda [Norell 2007], for greater familiarity for users of dependently-typed functional programming languages.
- A demonstration by example of how to derive free theorems for (and as) dependently-typed functions (sections 3.1 and 5). Two examples of functions that we tackle are:

  generic catamorphism $fold \;:\; ((F, map_F) : Functor) \;\to\; (A : \star) \to (F\,A \to A) \to \mu\,F \to A$, which is a generic catamorphism function defined within a dependently-typed language (see Section 5.2).

  generic cast $gcast \;:\; (F : \star \to \star) \to (u\,t : U) \to Maybe\,(F\,(El\,u) \to F\,(El\,t))$, which comes from a modelling of representation types with universes (see Section 5.3).

  In both cases, the derived parametricity condition yields useful properties to reason about the correctness of the function.

2. Pure type systems

In this section we briefly review the notion of PTS as described by Barendregt [1992, sec. 5.2], and the basic intuitions behind it. We introduce our notation along the way, as well as our running example type system.

Definition 1 (Syntax of terms)**.** A PTS is a type system over a λ-calculus with the following syntax:

$$\begin{array}{rcll} \mathcal{T} & = & \mathcal{C} & \text{constant} \\ & | & \mathcal{V} & \text{variable} \\ & | & \mathcal{T}\,\mathcal{T} & \text{application} \\ & | & \lambda \mathcal{V}{:}\mathcal{T}.\,\mathcal{T} & \text{abstraction} \\ & | & \forall \mathcal{V}{:}\mathcal{T}.\,\mathcal{T} & \text{dependent function space} \end{array}$$

We often write $(x : \mathsf{A}) \to \mathsf{B}$ for $\forall x{:}\mathsf{A}.\ \mathsf{B}$, and sometimes just $\mathsf{A} \to \mathsf{B}$ when x does not occur free in B.

The typing judgement of a PTS is parametrized over a *specification* $S = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, where $\mathcal{S} \subseteq \mathcal{C}$, $\mathcal{A} \subseteq \mathcal{C} \times \mathcal{S}$ and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$. The set $\mathcal{S}$ specifies the sorts, $\mathcal{A}$ the axioms (an axiom $(c, s) \in \mathcal{A}$ is often written $c : s$), and $\mathcal{R}$ specifies the typing rules of the function space. A rule (s_1, s_2, s_2) is often written $s_1 \rightsquigarrow s_2$. The rules for typing judgements in a PTS are given in Figure 1.

An attractive feature of PTSs is that the syntax for types and values is unified. It is the type of a term that tells how to interpret it (as a value, type, kind, etc.).

the λ-cube Barendregt [1992] defined a family of calculi each with $\mathcal{S} = \{\star, \Box\}$, $\mathcal{A} = \{\star : \Box\}$ and $\mathcal{R}$ a selection of rules of the form $s_1 \rightsquigarrow s_2$, for example:

- The (monomorphic) λ-calculus has $\mathcal{R}_\lambda = \{\star \rightsquigarrow \star\}$, corresponding to ordinary functions.
- System F has $\mathcal{R}_\mathrm{F} = \mathcal{R}_\lambda \cup \{\Box \rightsquigarrow \star\}$, adding (impredicative) universal quantification over types.
- System Fω has $\mathcal{R}_{\mathrm{F}\omega} = \mathcal{R}_\mathrm{F} \cup \{\Box \rightsquigarrow \Box\}$, adding type-level functions.
- The Calculus of Constructions (CC) has $\mathcal{R}_\mathrm{CC} = \mathcal{R}_{\mathrm{F}\omega} \cup \{\star \rightsquigarrow \Box\}$, adding dependent types.

Here $\star$ and $\Box$ are conventionally called the sorts of *types* and *kinds* respectively.

Notice that F is a subsystem of Fω, which is itself a subsystem of CC. (We say that $S_1 = (\mathcal{S}_1, \mathcal{A}_1, \mathcal{R}_1)$ is a subsystem of $S_2 = (\mathcal{S}_2, \mathcal{A}_2, \mathcal{R}_2)$ when $\mathcal{S}_1 \subseteq \mathcal{S}_2$, $\mathcal{A}_1 \subseteq \mathcal{A}_2$ and $\mathcal{R}_1 \subseteq \mathcal{R}_2$.) In fact, the λ-cube is so named because the lattice of the subsystem relation between all the systems forms a cube, with CC at the top.

sort hierarchies Difficulties with impredicativity[1] have led to the development of type systems with an infinite hierarchy of sorts. The "pure" part of such a system can be captured in the following PTS, which we name I_ω.

Definition 2 (I_ω)**.** I_ω is a PTS with this specification:

- $\mathcal{S} = \{\star_i \mid i \in \mathbb{N}\}$
- $\mathcal{A} = \{\star_i : \star_{i+1} \mid i \in \mathbb{N}\}$
- $\mathcal{R} = \{(\star_i, \star_j, \star_{\max(i,j)}) \mid i, j \in \mathbb{N}\}$

Compared to the monomorphic λ-calculus, $\star$ has been expanded into the infinite hierarchy $\star_0, \star_1, \ldots$ In I_ω, the sort $\star_0$ (abbreviated $\star$) is called the sort of types. Type constructors, or type-level functions have type $\star \to \star$. The set of types ($\star$), the set of type constructors ($\star \to \star$) and similar have type $\star_1$ (the sort of kinds). Terms like $\star_1$ and $\star \to \star_1$ have type $\star_2$, and so on.

Impredicativity can in fact coexist with an infinite hierarchy of sorts, as Coquand [1986] has shown. For example, in the Generalized Calculus of Constructions (CC_ω) of Miquel [2001], impredicativity exists for the sort $\star$ (conventionally called the sort of *propositions*), which lies at the bottom of the hierarchy.

[1] It is inconsistent with strong sums [Coquand 1986].

$$\text{axiom}\ \frac{}{\vdash c : s}\ c : s \in \mathcal{A}$$

$$\text{start}\ \frac{\Gamma \vdash \mathsf{A} : s}{\Gamma, x : \mathsf{A} \vdash x : \mathsf{A}}$$

$$\text{weakening}\ \frac{\Gamma \vdash \mathsf{A} : \mathsf{B} \qquad \Gamma \vdash \mathsf{C} : s}{\Gamma, x : \mathsf{C} \vdash \mathsf{A} : \mathsf{B}}$$

$$\text{product}\ \frac{\Gamma \vdash \mathsf{A} : s_1 \qquad \Gamma, x : \mathsf{A} \vdash \mathsf{B} : s_2}{\Gamma \vdash (\forall x{:}\mathsf{A}.\ \mathsf{B}) : s_3}\ (s_1, s_2, s_3) \in \mathcal{R}$$

$$\text{application}\ \frac{\Gamma \vdash \mathsf{F} : (\forall x{:}\mathsf{A}.\ \mathsf{B}) \qquad \Gamma \vdash \mathsf{a} : \mathsf{A}}{\Gamma \vdash \mathsf{F}\,\mathsf{a} : \mathsf{B}[x \mapsto \mathsf{a}]}$$

$$\text{abstraction}\ \frac{\Gamma, x : \mathsf{A} \vdash \mathsf{b} : \mathsf{B} \qquad \Gamma \vdash (\forall x{:}\mathsf{A}.\ \mathsf{B}) : s}{\Gamma \vdash (\lambda x{:}\mathsf{A}.\ \mathsf{b}) : (\forall x{:}\mathsf{A}.\ \mathsf{B})}$$

$$\text{conversion}\ \frac{\Gamma \vdash \mathsf{A} : \mathsf{B} \qquad \Gamma \vdash \mathsf{B}' : s \qquad \mathsf{B} =_\beta \mathsf{B}'}{\Gamma \vdash \mathsf{A} : \mathsf{B}'}$$

Figure 1. Typing judgements of the PTS $(\mathcal{S}, \mathcal{A}, \mathcal{R})$

Definition 3 (CC_ω)**.** CC_ω is a PTS with this specification:

- $\mathcal{S} = \{\star\} \cup \{\Box_i \mid i \in \mathbb{N}\}$
- $\mathcal{A} = \{\star : \Box_0\} \cup \{\Box_i : \Box_{i+1} \mid i \in \mathbb{N}\}$
- $\mathcal{R} = \{\star \rightsquigarrow \star, \star \rightsquigarrow \Box_i, \Box_i \rightsquigarrow \star \mid i \in \mathbb{N}\} \cup \{(\Box_i, \Box_j, \Box_{\max(i,j)}) \mid i, j \in \mathbb{N}\}$

Both CC and I_ω are subsystems of CC_ω, with $\star_i$ in I_ω corresponding to $\Box_i$ in CC_ω. Because $\Box$ in CC corresponds to $\Box_0$ in CC_ω, we often abbreviate $\Box_0$ as $\Box$.

Many dependently-typed programming languages and proof assistants are based on variants of I_ω or CC_ω, often with the addition of inductive definitions [Dybjer 1994; Paulin-Mohring 1993]. Such tools include Agda [Norell 2007], Coq [The Coq development team 2010] and Epigram [McBride and McKinna 2004].

2.1 PTS as logical system

Another use for PTSs is as logical systems: types correspond to propositions and terms to proofs. This correspondence extends to all aspects of the systems and is widely known as the Curry-Howard isomorphism. The judgement $\vdash \mathsf{p} : \mathsf{P}$ means that p is a *witness*, or *proof* of the proposition P.

In the logical system reading, an inhabited type corresponds to a tautology and dependent function types correspond to universal quantification. Predicates over a type A have type $\mathsf{A} \to s$, for some sort s: a value satisfies the predicate whenever the returned type is inhabited. Similarly, binary relations between values of types A_1 and A_2 have type $\mathsf{A}_1 \to \mathsf{A}_2 \to s$.

For this approach to be safe, it is important that the system be *consistent*: some types must be uninhabited, or equivalently each witness p must reduce to a normal form. This is the case for the systems used here.

In fact, in I_ω and similarly rich type systems, one may both represent programs and logical formulae about them. In the following sections, we make full use of this property: we encode programs and parametricity statements about them in the same type system.

3. Types to relations

We start by defining the relational interpretation of a term, as a syntactic translation from terms to terms. As we see in Section 3.1, it is a generalization of the classical rules given by Reynolds [1983], extended to application and abstraction.

In this section, we assume that the only constants are sorts. We also assume for each sort s another sort $\widetilde{s}$ of parametricity propositions about terms of type s. In our examples, we simply choose $\widetilde{s} = s$. We shall return to the general case in Section 6.2.

Definition 4 ($[\![\_]\!]$, translation from types to relations)**.** Given a natural number n (the arity of relations), we assume for each variable x, fresh variables $x_1, \ldots, x_n$ and x_R. We write $\overline{\mathsf{A}}$ for the n terms A_i, each obtained by replacing each *free* variable x in A with x_i. Correspondingly, $\overline{x : \mathsf{A}}$ stands for n bindings $(x_i : \mathsf{A}_i)$. We define a mapping $[\![\_]\!]$ from $\mathcal{T}$ to $\mathcal{T}$ as follows:

$$
\begin{aligned}
[\![s]\!] &= \lambda \overline{x : s}.\, \overline{x} \to \widetilde{s} \\
[\![x]\!] &= x_R \\
[\![\forall x : \mathsf{A}.\ \mathsf{B}]\!] &= \lambda \overline{f : (\forall x : \mathsf{A}.\ \mathsf{B})}.\, \forall \overline{x : \mathsf{A}}.\ \forall x_R : [\![\mathsf{A}]\!]\, \overline{x}.\ [\![\mathsf{B}]\!]\, (\overline{f\, x}) \\
[\![\mathsf{F}\ \mathsf{a}]\!] &= [\![\mathsf{F}]\!]\, \overline{\mathsf{a}}\, [\![\mathsf{a}]\!] \\
[\![\lambda x : \mathsf{A}.\ \mathsf{b}]\!] &= \lambda \overline{x : \mathsf{A}}.\, \lambda x_R : [\![\mathsf{A}]\!]\, \overline{x}.\, [\![\mathsf{b}]\!]
\end{aligned}
$$

Note that for each variable x free in A, the translation $[\![\mathsf{A}]\!]$ has free variables $x_1, \ldots, x_n$ and x_R. There is a corresponding replication of variables bound in contexts, which is made explicit in the following definition.

Definition 5 (translation of contexts)**.**

$$[\![\Gamma, x : \mathsf{A}]\!] = [\![\Gamma]\!], \overline{x : \mathsf{A}}, x_R : [\![\mathsf{A}]\!]\, \overline{x}$$

Note that each tuple $\overline{x} : \overline{\mathsf{A}}$ in the translated context must satisfy the relation $[\![\mathsf{A}]\!]$, as witnessed by x_R. Thus, one may interpret $[\![\Gamma]\!]$ as n related environments.

In order for a PTS to be able to express both programs and parametricity propositions about them, it must satisfy certain closure conditions, for which we coin the term *reflective*:

Definition 6 (reflective)**.** A PTS $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ is *reflective* if

- for each sort $s \in \mathcal{S}$
  - $\exists \widetilde{s} \in \mathcal{S}$
  - $\exists s' \in \mathcal{S}$ such that $s : s' \in \mathcal{A}$
- for each axiom $s : s' \in \mathcal{A}$
  - $\widetilde{s} : \widetilde{s'} \in \mathcal{A}$
  - $s \rightsquigarrow \widetilde{s'} \in \mathcal{R}$
- for each rule $(s_1, s_2, s_3) \in \mathcal{R}$
  - $(\widetilde{s_1}, \widetilde{s_2}, \widetilde{s_3}) \in \mathcal{R}$
  - $s_1 \rightsquigarrow \widetilde{s_3} \in \mathcal{R}$

We can then state our main result:

Theorem 1 (abstraction)**.** *Given a reflective PTS* $(\mathcal{S}, \mathcal{A}, \mathcal{R})$*,* $\Gamma \vdash \mathsf{A} : \mathsf{B} \implies [\![\Gamma]\!] \vdash [\![\mathsf{A}]\!] : [\![\mathsf{B}]\!]\, \overline{\mathsf{A}}$

Proof. By induction on the derivation. A brief sketch of the proof is given in appendix A. □

The above theorem can be read in two ways. A direct reading is as a typing judgement about translated terms: if A has type B, then $[\![\mathsf{A}]\!]$ has type $[\![\mathsf{B}]\!]\, \overline{\mathsf{A}}$. The more fruitful reading is as an abstraction theorem for pure type systems: if A has type B in environment Γ, then n interpretations $\overline{\mathsf{A}}$ in related environments $[\![\Gamma]\!]$ are related by $[\![\mathsf{B}]\!]$. Further, $[\![\mathsf{A}]\!]$ is a witness of this proposition *within the type system*. In particular, closed terms are related to themselves:

Corollary 2 (parametricity)**.** $\vdash \mathsf{A} : \mathsf{B} \implies \vdash [\![\mathsf{A}]\!] : [\![\mathsf{B}]\!]\, \overline{\mathsf{A}}$

example systems Note that both I_ω and CC_ω are reflective, with $\widetilde{s} = s$. Therefore we can write programs in these systems and derive valid statements about them, using $[\![\_]\!]$, within the same PTS. We proceed to do so in the rest of the paper.

3.1 Examples: the λ-cube

In this section, we show that $[\![\_]\!]$ specializes to the rules given by Reynolds [1983] to read a System F type as a relation. Having shown that our framework can explain parametricity theorems for System-F-style types, we move on to progressively higher-order constructs. In these examples, the binary version of parametricity is used (arity $n = 2$). For examples using the unary version (arity $n = 1$) see Section 5.3.

While the systems of the λ-cube are not reflective, they are embedded in CC_ω, which is. This means that our translation rules take System F types to terms in CC_ω (instead of second order propositional logic). The possibility of using a different PTS for the logic is discussed in Section 6.3.

types to relations Note that, by definition,

$$[\![\star]\!]\, T_1\, T_2 \;=\; T_1 \to T_2 \to \star$$

Assuming that types inhabit the sort $\star$, this means that types are translated to relations (as expected). Here we also use $\star$ on the right side as the sort of propositions ($\widetilde{\star} = \star$), but other choices are possible, as we shall discuss in Section 6.2.

function types Applying our translation to non-dependent function types, we get:

$$
\begin{aligned}
&[\![\mathsf{A} \to \mathsf{B}]\!] : [\![\star]\!]\, (\mathsf{A} \to \mathsf{B})\, (\mathsf{A} \to \mathsf{B}) \\
&[\![\mathsf{A} \to \mathsf{B}]\!]\, f_1\, f_2 \;=\; \forall a_1 : \mathsf{A}.\ \forall a_2 : \mathsf{A}. \\
&\qquad\qquad [\![\mathsf{A}]\!]\, a_1\, a_2 \to [\![\mathsf{B}]\!]\, (f_1\, a_1)\, (f_2\, a_2)
\end{aligned}
$$

That is, functions are related iff they take related arguments into related outputs.

type schemes System F includes universal quantification of the form $\forall A : \star.\ \mathsf{B}$. Applying $[\![\_]\!]$ to this type expression yields:

$$
\begin{aligned}
&[\![\forall A : \star.\ \mathsf{B}]\!] : [\![\star]\!]\, (\forall A : \star.\ \mathsf{B})\, (\forall A : \star.\ \mathsf{B}) \\
&[\![\forall A : \star.\ \mathsf{B}]\!]\, g_1\, g_2 \;=\; \forall A_1 : \star.\ \forall A_2 : \star.\ \forall A_R : [\![\star]\!]\, A_1\, A_2. \\
&\qquad\qquad [\![\mathsf{B}]\!]\, (g_1\, A_1)\, (g_2\, A_2)
\end{aligned}
$$

In words, polymorphic values are related iff instances at related types are related. Note that as A may occur free in B, the variables A_1, A_2 and A_R may occur free in $[\![\mathsf{B}]\!]$.

type constructors With the addition of the rule $\Box \rightsquigarrow \Box$, one can construct terms of type $\star \to \star$, which are sometimes known as type constructors, type formers or type-level functions. As Voigtländer [2009b] remarks, extending Reynolds-style parametricity to support type constructors appears to be folklore. Such folklore can be precisely justified by our framework by applying $[\![\_]\!]$ to obtain the relational counterpart of type constructors:

$$
\begin{aligned}
&[\![\star \to \star]\!] : [\![\Box]\!]\, (\star \to \star)\, (\star \to \star) \\
&[\![\star \to \star]\!]\, F_1\, F_2 \;=\; \forall A_1 : \star.\ \forall A_2 : \star. \\
&\qquad\qquad [\![\star]\!]\, A_1\, A_2 \to [\![\star]\!]\, (F_1\, A_1)\, (F_2\, A_2)
\end{aligned}
$$

That is, a term of type $[\![\star \to \star]\!]\, F_1\, F_2$ is a (polymorphic) function converting a relation between any types A_1 and A_2 to a relation between $F_1\, A_1$ and $F_2\, A_2$, a *relational action*.

dependent functions In a system with the rule $\star \rightsquigarrow \Box$, value variables may occur in dependent function types like $\forall x : \mathsf{A}.\ \mathsf{B}$, which we translate as follows:

$$
\begin{aligned}
&[\![\forall x : \mathsf{A}.\ \mathsf{B}]\!] : [\![\star]\!]\, (\forall x : \mathsf{A}.\ \mathsf{B})\, (\forall x : \mathsf{A}.\ \mathsf{B}) \\
&[\![\forall x : \mathsf{A}.\ \mathsf{B}]\!]\, f_1\, f_2 \;=\; \forall x_1 : \mathsf{A}.\ \forall x_2 : \mathsf{A}.\ \forall x_R : [\![\mathsf{A}]\!]\, x_1\, x_2. \\
&\qquad\qquad [\![\mathsf{B}]\!]\, (f_1\, x_1)\, (f_2\, x_2)
\end{aligned}
$$

proof terms We have used $[\![\_]\!]$ to turn types into relations, but we can also use it to turn terms into proofs of abstraction properties. As a simple example, the relation corresponding to the type $T = \forall A{:}\star.\ A \to A$, namely

$$\begin{aligned}[\![T]\!]\, f_1\, f_2 \ = \ & \forall A_1 : \star.\ \forall A_2 : \star.\ \forall A_R : [\![\star]\!]\, A_1\, A_2. \\ & \forall x_1 : A_1.\ \forall x_2 : A_2. \\ & A_R\, x_1\, x_2 \to A_R\, (f_1\, A_1\, x_1)\, (f_2\, A_2\, x_2)\end{aligned}$$

states that functions of this type map related inputs to related outputs. From a term $id = \lambda A{:}\star.\ \lambda x{:}A.\ x$ of this type, by Theorem 2 we obtain a term $[\![id]\!] : [\![T]\!]\, id\, id$, that is, a proof of the abstraction property:

$$[\![id]\!]\, A_1\, A_2\, A_R\, x_1\, x_2\, x_R \ = \ x_R$$

4. Constants and data types

While the above development assumes pure type systems with $\mathcal{C} = \mathcal{S}$, it is possible to add constants to the system and retain parametricity, as long as each constant is parametric. That is, for each new axiom $\vdash k : \mathsf{A}$ (where k is an arbitrary constant and A an arbitrary term such that $\vdash \mathsf{A} : s$, not a mere sort) we require a term $[\![k]\!]$ such that the judgement $\vdash [\![k]\!] : [\![\mathsf{A}]\!]\, \overline{k}$ holds. (Additionally, β-conversion rules involving those constants must preserve types.)

One source of constants in many languages is data type definitions. In the rest of the this section we detail how to handle such definitions (in a system extending I_ω).

4.1 Inductive families

Many languages permit data type declarations like those in Figure 2. Dependently typed languages typically allow the return types of constructors to have different arguments, yielding *inductive families* [Dybjer 1994; Paulin-Mohring 1993] such as the family *Vec*, in which the type is indexed by the number of elements.

Data family declarations of sort s ($\star$ in the examples) have the typical form:[2]

$$\begin{array}{l}\mathbf{data}\ T\ (a{:}\mathsf{A}) : \forall n{:}\mathsf{N}.\ s\ \mathbf{where} \\ \quad c : \forall b{:}\mathsf{B}.\ (\forall x{:}\mathsf{X}.\ T\, a\, \mathsf{i}) \to T\, a\, \mathsf{v}\end{array}$$

Arguments of the type constructor T may be either parameters a, which scope over the constructors and are repeated at each recursive use of T, or indices n, which may vary between uses. Data constructors c have non-recursive arguments b, whose types are otherwise unrestricted, and recursive arguments with types of a constrained form, which cannot be referred to in the other terms.

Such a declaration can be interpreted as a simultaneous declaration of formation and introduction constants

$$\begin{array}{l}T : \forall a{:}\mathsf{A}.\ \forall n{:}\mathsf{N}.\ s \\ c\ : \forall a{:}\mathsf{A}.\ \forall b{:}\mathsf{B}.\ (\forall x{:}\mathsf{X}.\ T\, a\, \mathsf{i}) \to T\, a\, \mathsf{v}\end{array}$$

and also an eliminator to analyse values of that type:

$$\begin{aligned}T\text{-}elim : \forall a{:}\mathsf{A}. \\ & \forall P{:}(\forall n{:}\mathsf{N}.\ T\, a\, n \to s). \\ & \mathsf{Case}_c \to \forall n{:}\mathsf{N}.\ \forall t{:}T\, a\, n.\ P\, n\, t\end{aligned}$$

where the type Case_c of the case for each constructor c is

$$\forall b{:}\mathsf{B}.\ \forall u{:}(\forall x{:}\mathsf{X}.\ T\, a\, \mathsf{i}).\ (\forall x{:}\mathsf{X}.\ P\, \mathsf{i}\, (u\, x)) \to P\, \mathsf{v}\, (c\, a\, b\, u)$$

with beta-equivalences (one for each constructor c):

$$T\text{-}elim\, a\, P\, e\, \mathsf{v}\, (c\, a\, b\, u) = e\, b\, u\, (\lambda x{:}\mathsf{X}.\, T\text{-}elim\, a\, P\, e\, \mathsf{i}\, (u\, x))$$

We shall often use corresponding pattern matching definitions instead of these eliminators [Coquand 1992].

[2] We show only one of each element here, but the generalization to arbitrary numbers is straightforward.

$\mathbf{data}\ Bool : \star\ \mathbf{where}$
  $true : Bool$
  $false : Bool$

$\mathbf{data}\ Nat : \star\ \mathbf{where}$
  $zero : Nat$
  $succ : Nat \to Nat$

$\mathbf{data}\ \bot : \star\ \mathbf{where}$ -- no constructors

$\mathbf{data}\ \top : \star\ \mathbf{where}$
  $tt : \top$

$\mathbf{data}\ List\ (A : \star) : \star\ \mathbf{where}$
  $nil : List\ A$
  $cons : A \to List\ A \to List\ A$

$\mathbf{data}\ Vec\ (A : \star) : Nat \to \star\ \mathbf{where}$
  $nilV : Vec\ A\ zero$
  $consV : A \to (n : Nat) \to Vec\ A\ n \to Vec\ A\ (succ\ n)$

$\mathbf{data}\ \Sigma\ (A : \star)\ (B : A \to \star) : \star\ \mathbf{where}$
  $\_,\_ : (a : A) \to B\ a \to \Sigma\ A\ B$

$\mathbf{data} \equiv\ (A : \star)\ (a : A) : A \to \star\ \mathbf{where}$
  $refl : \equiv A\ a\ a$

Figure 2. Example inductive families

For example, the definition of *List* in Figure 2 gives rise to the following constants:

$$\begin{array}{ll}List & : (A : \star) \to \star \\ nil & : (A : \star) \to List\ A \\ cons & : (A : \star) \to A \to List\ A \to List\ A \\ List\text{-}elim & : (A : \star) \to (P : List\ A \to \star) \to \\ & \quad P\, (nil\ A) \to \\ & \quad ((x : A) \to (xs : List\ A) \to P\, xs \to \\ & \qquad P\, (cons\ A\, x\, xs)) \to \\ & \quad (l : List\ A) \to P\, l\end{array}$$

In the following sections, we consider two ways to define an abstraction proof $[\![k]\!] : [\![\tau]\!]\, \overline{k}$ for each constant $k : \tau$ introduced by the data definition.

4.2 Deductive-style translation

First, we define each proof as a term (using pattern matching to simplify the presentation). We begin with the translation of the equation for each constructor:

$$\begin{aligned}[\![T\text{-}elim\, a\, P\, e\, \mathsf{v}]\!]\ \overline{(c\, a\, b\, u)}\ ([\![c]\!]\, \overline{a}\, a_R\, \overline{b}\, b_R\, \overline{u}\, u_R) = \\ [\![e\, b\, u\, (\lambda x{:}\mathsf{X}.\, T\text{-}elim\, a\, P\, e\, \mathsf{i}\, (u\, x))]\!]\end{aligned}$$

To turn this into a pattern matching definition of *T-elim*, we need a suitable definition of $[\![c]\!]$, and similarly for the constructors in v. The only arguments of $[\![c]\!]$ not already in scope are b_R and u_R, so we package them as a dependent pair, because the type of u_R may depend on that of b_R. Writing $(x{:}\mathsf{A}) \times \mathsf{B}$ for $\Sigma\, \mathsf{A}\, (\lambda x{:}\mathsf{A}.\, \mathsf{B})$, and elements of this type as (a, b), omitting the arguments A and $\lambda x{:}\mathsf{A}.\, \mathsf{B}$, we define[3]

$$\begin{array}{l}[\![T]\!] : [\![\forall a{:}\mathsf{A}.\ \forall n{:}\mathsf{N}.\ s]\!]\, \overline{T} \\ [\![T]\!]\, \overline{a}\, a_R\, \overline{\mathsf{v}}\, [\![\mathsf{v}]\!]\ \overline{(c\, a\, b\, u)} = (b_R{:}[\![\mathsf{B}]\!]\, \overline{b}) \times [\![\forall x{:}\mathsf{X}.\ T\, a\, \mathsf{i}]\!]\, \overline{u} \\ [\![T]\!]\, \overline{a}\, a_R\, \overline{u}\, u_R\, \overline{t} = \bot \\ [\![c]\!] : [\![\forall a{:}\mathsf{A}.\ \forall b{:}\mathsf{B}.\ (\forall x{:}\mathsf{X}.\ T\, a\, \mathsf{i}) \to T\, a\, \mathsf{v}]\!]\, \overline{c} \\ [\![c]\!]\, \overline{a}\, a_R\, \overline{b}\, b_R\, \overline{u}\, u_R = (b_R, u_R)\end{array}$$

[3] The definition of $[\![T]\!]$ relies on the weak elimination constant to sort $\widetilde{s}'$.

and the translation of *T-elim* becomes

$$[\![T\text{-}elim\ a\ P\ e\ \mathsf{v}]\!]\ \overline{(c\ a\ b\ u)}\ (b_R, u_R) = \\ [\![e\ b\ u\ (\lambda x : \mathsf{X}.\ T\text{-}elim\ a\ P\ e\ \mathsf{i}\ (u\ x))]\!]$$

Because $[\![T]\!]$ yields $\bot$ unless the constructors match, these clauses provide complete coverage.

The reader may have noted by now that the argument lists of the translated constants tend to be quite long. The use of the translated constants can be substantially simplified using implicit arguments (arguments which can be inferred from contextual knowledge). We avoid using them in this paper to explicitly show the underlying machinery, but the Agda library implementing the translation makes heavy use of implicit arguments for convenience.

Booleans To get an intuition of the meaning of the above translation scheme we proceed to apply it to a number of examples, starting with the data type for Booleans. We obtain:

$$[\![Bool]\!] : [\![\star]\!]\ Bool\ Bool$$
$$[\![Bool]\!]\ true\ true\ = \top$$
$$[\![Bool]\!]\ false\ false\ = \top$$
$$[\![Bool]\!]\ \_\ \ \_\ \ = \bot$$
$$[\![true]\!] : [\![Bool]\!]\ true\ true$$
$$[\![true]\!] = tt$$
$$[\![false]\!] : [\![Bool]\!]\ false\ false$$
$$[\![false]\!] = tt$$

(We use $\top$ for nullary constructors as it is the identity of $\times$.)

parametricity and elimination Reynolds [1983] and Wadler [1989] assume that each type constant $K : \star$ is translated to the identity relation, as we have done for $Bool$ above. This definition is certainly compatible with the condition required by Theorem 1 for such constants: $[\![K]\!] : [\![\star]\!]\ K\ K$, but so are many other relations. Are we missing some restriction for constants? This question might be answered by resorting to a translation to pure terms via Church encodings [Böhm and Berarducci 1985], as Wadler [2007] does. However, in the hope to shed a different light on the issue, we give another explanation, using our machinery.

Consider a base type, such as $Bool : \star$, equipped with constructors $true : Bool$ and $false : Bool$. In order to derive parametricity theorems in a system containing such a constant $Bool$, we must define $[\![Bool]\!]$, satisfying $\vdash\ [\![Bool]\!] : [\![\star]\!]\ \overline{Bool}$. What are the restrictions put on the term $[\![Bool]\!]$? First, we must be able to define $[\![true]\!] : [\![Bool]\!]\ \overline{true}$. Therefore, $[\![Bool]\!]\ \overline{true}$ must be inhabited. The same reasoning holds for the $false$ case.

Second, to write any useful program using Booleans, a way to test their value is needed. This may be done by adding a constant $if : Bool \to (A : \star) \to A \to A \to A$, such that $if\ true\ A\ x\ y \longrightarrow_\beta x$ and $if\ false\ A\ x\ y \longrightarrow_\beta y$. (This special case of ***Bool-elim*** is sufficient for the present example.)

Now, if a program uses if, we must also define $[\![if]\!]$ of type $[\![Bool \to (A : \star) \to A \to A \to A]\!]\ \overline{if}$ for parametricity to work. Let us expand the type of $[\![if]\!]$ and attempt to give a definition case by case:

$$[\![if]\!] : (b_1\ b_2 : Bool) \to (b_R : [\![Bool]\!]\ b_1\ b_2) \to \\ (A_1\ A_2 : \star) \to (A_R : [\![\star]\!]\ A_1\ A_2) \to \\ (x_1 : A_1) \to (x_2 : A_2) \to (x_R : A_R\ x_1\ x_2) \to \\ (y_1 : A_1) \to (y_2 : A_2) \to (y_R : A_R\ y_1\ y_2) \to \\ A_R\ (if\ b_1\ A_1\ x_1\ y_1)\ (if\ b_2\ A_2\ x_2\ y_2)$$
$$[\![if]\!]\ true\ true\ b_R\ \_\ \_\ \_\ x_1\ x_2\ x_R\ y_1\ y_2\ y_R\ = x_R$$
$$[\![if]\!]\ true\ false\ b_R\ \_\ \_\ \_\ x_1\ x_2\ x_R\ y_1\ y_2\ y_R\ = ?$$
$$[\![if]\!]\ false\ true\ \ b_R\ \_\ \_\ \_\ x_1\ x_2\ x_R\ y_1\ y_2\ y_R\ = ?$$
$$[\![if]\!]\ false\ false\ b_R\ \_\ \_\ \_\ x_1\ x_2\ x_R\ y_1\ y_2\ y_R\ = y_R$$

(From this example onwards, we use a layout convention to ease the reading of translated types: each triple of arguments, corresponding to one argument in the original function, is written on its own line if space permits.)

In order to complete the above definition, we must provide a type-correct expression for each question mark. In the case of the second equation, this means that we must construct an expression of type $A_R\ x_1\ y_2$. Neither $x_R : A_R\ x_1\ x_2$ nor $y_R : A_R\ y_1\ y_2$ can help us here. The only liberty left is in $b_R : [\![Bool]\!]\ true\ false$. If we let $[\![Bool]\!]\ true\ false$ be $\bot$, then this case can never be reached and we need not give an equation for it. This reasoning holds symmetrically for the third equation. Therefore, we have the restrictions:

$$[\![Bool]\!]\ x\ x\ = \text{some inhabited type}$$
$$[\![Bool]\!]\ x\ y\ = \bot \quad \text{if } x \neq y$$

We have some freedom regarding picking "some inhabited type", so we choose $[\![Bool]\!]\ x\ x\ = \top$, yielding an encoding of the identity relation. In general, for any base type, the identity is the most permissive relation which allows for a definition of the translation of the eliminator.

An intuition behind parametricity is that, the more programs "know" about a type, the more restricted parametricity theorems are. Through the $Bool$ example, we have seen how our framework captures this intuition, in a fine grained manner. We revisit this idea in Section 5.4.

lists and vectors From the definition of $List$ in Figure 2, we have the constant $List : \star \to \star$, so $List$ is an example of a type constructor, and thus $[\![List]\!]$ is a relation transformer. The relation transformer we get by applying our scheme is exactly that given by Wadler [1989]: lists are related iff their lengths are equal and their elements are related point-wise.

$$[\![List]\!] : [\![\star \to \star]\!]\ List\ List$$
$$[\![List]\!]\ A_1\ A_2\ A_R\ nil\ \ \ nil\ \ \ = \top$$
$$[\![List]\!]\ A_1\ A_2\ A_R\ (cons\ x_1\ xs_1)\ (cons\ x_2\ xs_2)\ = \\ A_R\ x_1\ x_2\ \times\ [\![List]\!]\ A_1\ A_2\ A_R\ xs_1\ xs_2$$
$$[\![List]\!]\ A_1\ A_2\ A_R\ \_\ \ \ \_\ \ \ = \bot$$
$$[\![nil]\!] : [\![(A : \star) \to List\ A]\!]\ nil\ nil$$
$$[\![nil]\!]\ A_1\ A_2\ A_R\ = tt$$
$$[\![cons]\!] : [\![(A : \star) \to A \to List\ A \to List\ A]\!]\ cons\ cons$$
$$[\![cons]\!]\ A_1\ A_2\ A_R\ x_1\ x_2\ x_R\ xs_1\ xs_2\ xs_R\ = (x_R, xs_R)$$

The translations of the constants of Vec are given in Figure 3.

list rearrangements The first example of parametric type examined by Wadler [1989] is the type of list rearrangements: $R = (A : \star) \to List\ A \to List\ A$. Intuitively, functions of type R know nothing about the actual argument type A, and therefore they can only produce the output list by taking elements from the input list. In this section we recover that result as an instance of Theorem 1.

Applying the translation to R yields:

$$[\![R]\!] : R \to R \to \star$$
$$[\![R]\!]\ r_1\ r_2\ = (A_1\ A_2 : \star) \to (A_R : [\![\star]\!]\ A_1\ A_2) \to \\ (l_1 : List\ A_1) \to (l_2 : List\ A_2) \to \\ (l_R : [\![List]\!]\ A_1\ A_2\ A_R\ l_1\ l_2) \to \\ [\![List]\!]\ A_1\ A_2\ A_R\ (r_1\ A_1\ l_1)\ (r_2\ A_2\ l_2)$$

In words: two list rearrangements r_1 and r_2 are related iff for all types A_1 and A_2 with relation A_R, and for all lists l_1 and l_2 point-wise related by A_R, the resulting lists $r_1\ A_1\ l_1$ and $r_2\ A_2\ l_2$ are also point-wise related by A_R. By corollary 2 (parametricity), we have, for any r:

$$\vdash r : R \implies \vdash [\![r]\!] : [\![R]\!]\ r\ r$$

$$
\begin{array}{l}
[\![\mathit{Vec}]\!] : [\![(A : \star) \to \mathit{Nat} \to \star]\!]\ \overline{\mathit{Vec}} \\
[\![\mathit{Vec}]\!]\ A_1\ A_1\ A_R\ \mathit{zero}\ \mathit{zero} \qquad\quad \_ \quad \mathit{nilV} \qquad\quad \mathit{nilV} \ = \ \top \\
[\![\mathit{Vec}]\!]\ A_1\ A_1\ A_R\ (\mathit{succ}\ n_1)\ (\mathit{succ}\ n_2) \quad n_R\ (\mathit{consV}\ n_1\ x_1\ \mathit{xs}_1)\ (\mathit{consV}\ n_2\ x_2\ \mathit{xs}_2) \ = \\
\quad A_R\ x_1\ x_2 \times (n_R : [\![\mathit{Nat}]\!]\ n_1\ n_2) \times [\![\mathit{Vec}]\!]\ A_1\ A_1\ A_R\ n_1\ n_2\ n_R\ \mathit{xs}_1\ \mathit{xs}_2\ \mathit{xs}_R \\
[\![\mathit{Vec}]\!]\ A_1\ A_1\ A_R\ n_1 \quad n_2 \qquad\quad n_R\ \mathit{xs}_1 \qquad\quad \mathit{xs}_2 \quad = \ \bot \\
\\
[\![\mathit{nilV}]\!] : [\![(A : \star) \to \mathit{Vec}\ A\ \mathit{zero}]\!]\ \overline{\mathit{nilV}} \\
[\![\mathit{nilV}]\!]\ A_1\ A_1\ A_R \ = \ \mathit{tt} \\
\\
[\![\mathit{consV}]\!] : [\![(A : \star) \to A \to (n : \mathit{Nat}) \to \mathit{Vec}\ A\ n \to \mathit{Vec}\ A\ (\mathit{succ}\ n)]\!]\ \overline{\mathit{consV}} \\
[\![\mathit{consV}]\!]\ A_1\ A_1\ A_R\ x_1\ x_2\ x_R\ n_1\ n_2\ n_R\ \mathit{xs}_1\ \mathit{xs}_2\ \mathit{xs}_R \ = \ (x_R, (n_R, \mathit{xs}_R)) \\
\\
[\![\mathit{Vec\text{-}elim}]\!] : [\![\ (A : \star) \to \\
\qquad\qquad (P : (n : \mathit{Nat}) \to \mathit{Vec}\ n\ A \to \star) \to \\
\qquad\qquad (\mathit{en} : P\ \mathit{zero}\ (\mathit{nilV}\ A)) \to \\
\qquad\qquad (\mathit{ec} : (x : A) \to (n : \mathit{Nat}) \to (\mathit{xs} : \mathit{Vec}\ n\ A) \to P\ n\ \mathit{xs} \to P\ (\mathit{succ}\ n)\ (\mathit{consV}\ A\ x\ n\ \mathit{xs})) \to \\
\qquad\qquad (n : \mathit{Nat}) \to (v : \mathit{Vec}\ n\ A) \to P\ n\ v]\!]\ \overline{\mathit{Vec\text{-}elim}} \\
[\![\mathit{Vec\text{-}elim}]\!]\ A_1\ A_2\ A_R\ P_1\ P_2\ P_R\ \mathit{en}_1\ \mathit{en}_2\ \mathit{en}_R\ \mathit{ec}_1\ \mathit{ec}_2\ \mathit{ec}_R\ \mathit{zero}\ \mathit{zero}\ \_\ \mathit{nilV}\ \mathit{nilV}\ \_ \ = \ \mathit{en}_R \\
[\![\mathit{Vec\text{-}elim}]\!]\ A_1\ A_2\ A_R\ P_1\ P_2\ P_R\ \mathit{en}_1\ \mathit{en}_2\ \mathit{en}_R\ \mathit{ec}_1\ \mathit{ec}_2\ \mathit{ec}_R\ (\mathit{succ}\ n_1)\ (\mathit{succ}\ n_2)\ n_R \\
\qquad\qquad (\mathit{consV}\ x_1\ n_1\ \mathit{xs}_1)\ (\mathit{consV}\ x_2\ n_2\ \mathit{xs}_2)\ (x_R, (n_R, \mathit{xs}_R)) \ = \\
\quad \mathit{ec}_R\ x_1\ x_2\ x_R\ n_1\ n_2\ n_R\ \mathit{xs}_1\ \mathit{xs}_2\ \mathit{xs}_R\ (\mathit{Vec\text{-}elim}\ A_1\ P_1\ \mathit{en}_1\ \mathit{ec}_1\ n_1\ \mathit{xs}_1)\ (\mathit{Vec\text{-}elim}\ A_2\ P_2\ \mathit{en}_2\ \mathit{ec}_2\ n_2\ \mathit{xs}_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad (\![\![\mathit{Vec\text{-}elim}]\!]\ A_1\ A_2\ A_R\ P_1\ P_2\ P_R\ \mathit{en}_1\ \mathit{en}_2\ \mathit{en}_R\ \mathit{ec}_1\ \mathit{ec}_2\ \mathit{ec}_R\ n_1\ n_2\ n_R\ \mathit{xs}_1\ \mathit{xs}_2\ \mathit{xs}_R)
\end{array}
$$

Figure 3. Deductive translation of Vec constants. ($[\![\mathit{Nat}]\!]$ is the identity relation.)

In words: applying r preserves (point-wise) any relation existing between input lists. By specializing A_R to a function ($A_R\ a_1\ a_2 \ = \ f\ a_1 \equiv a_2$) we obtain the well-known result:

$$
\begin{array}{l}
\vdash \ r : R \implies \\
(A_1\ A_2 : \star) \to (f : A_1 \to A_2) \to \\
(l : \mathit{List}\ A_1) \to \\
\quad \mathit{map}\ f\ (r\ A_1\ l) \equiv r\ A_2\ (\mathit{map}\ f\ l)
\end{array}
$$

(This form relies on the facts that $[\![\mathit{List}]\!]$ preserves identities and composes with map.)

proof terms We have seen that applying $[\![\_]\!]$ to a type yields a parametricity property for terms of that type. However, by Theorem 1 we can also apply $[\![\_]\!]$ to a term of that type to obtain a proof of the property.

Consider a list rearrangement function odds that returns every second element from a list.

$$
\begin{array}{l}
\mathit{odds} : (A : \star) \to \mathit{List}\ A \to \mathit{List}\ A \\
\mathit{odds}\ A\ \mathit{nil} \ = \ \mathit{nil}\ A \\
\mathit{odds}\ A\ (\mathit{cons}\ x\ \mathit{nil}) \ = \ \mathit{cons}\ A\ x\ \mathit{nil} \\
\mathit{odds}\ A\ (\mathit{cons}\ x\ (\mathit{cons}\ \_\ \mathit{xs})) \ = \ \mathit{cons}\ A\ x\ (\mathit{odds}\ A\ \mathit{xs})
\end{array}
$$

Any list rearrangement function must satisfy the parametricity condition seen above. We know by Theorem 1 that $[\![\mathit{odds}]\!]$ is a proof that odds satisfies parametricity. Expanding it yields:

$$
\begin{array}{l}
[\![\mathit{odds}]\!] : [\![(A : \star) \to \mathit{List}\ A \to \mathit{List}\ A]\!]\ \mathit{odds}\ \mathit{odds} \\
[\![\mathit{odds}]\!]\ A_1\ A_2\ A_R\ \mathit{nil}\ \mathit{nil}\ \_ \ = \ \mathit{tt} \\
[\![\mathit{odds}]\!]\ A_1\ A_2\ A_R\ (\mathit{cons}\ x_1\ \mathit{nil})\ (\mathit{cons}\ x_2\ \mathit{nil})\ (x_R, \_) \ = \\
\quad (x_R, \mathit{tt}) \\
[\![\mathit{odds}]\!]\ A_1\ A_2\ A_R\ (\mathit{cons}\ x_1\ (\mathit{cons}\ \_\ \mathit{xs}_1)) \\
\qquad\qquad\qquad (\mathit{cons}\ x_2\ (\mathit{cons}\ \_\ \mathit{xs}_2))\ (x_R, (\_, \mathit{xs}_R)) \ = \\
\quad (x_R, [\![\mathit{odds}]\!]\ A_1\ A_2\ A_R\ \mathit{xs}_1\ \mathit{xs}_2\ \mathit{xs}_R)
\end{array}
$$

We see that $[\![\mathit{odds}]\!]$ performs essentially the same computation as odds, on two lists in parallel. However, instead of building a new list, it keeps track of the relations (in the R-subscripted variables). This behaviour stems from the last two cases in the definition of $[\![\mathit{odds}]\!]$. Performing such a computation is enough to prove the parametricity condition.

4.3 Inductive-style translation

Inductive definitions offer another way of defining the translations $[\![c]\!]$ of the constants associated with a data type, an *inductive* definition in contrast to the *deductive* definitions of the previous section. Given an inductive family

$$
\begin{array}{l}
\mathbf{data}\ T\ (a{:}\mathsf{A}) : \mathsf{K}\ \mathbf{where} \\
\quad c : \mathsf{C}
\end{array}
$$

by applying our translation to the components of the **data**-declaration, we obtain an inductive family that defines the relational counterparts of the original type T and its constructors c at the same time:

$$
\begin{array}{l}
\mathbf{data}\ [\![T]\!]\ ([\![a{:}\mathsf{A}]\!]) : [\![\mathsf{K}]\!]\ \overline{(T\ a)}\ \mathbf{where} \\
\quad c : [\![\mathsf{C}]\!]\ \overline{(c\ a)}
\end{array}
$$

It remains to supply a proof term for the parametricity of the elimination constant *T-elim*. If the inductive family has the form

$$
\begin{array}{l}
\mathbf{data}\ T\ (a{:}\mathsf{A}) : \forall n{:}\mathsf{N}.\ s\ \mathbf{where} \\
\quad c : \forall b{:}\mathsf{B}.\ (\forall x{:}\mathsf{X}.\ T\ a\ \mathsf{i}) \to T\ a\ \mathsf{v}
\end{array}
$$

then the proof $[\![\textit{T-elim}]\!]$ can be defined using $[\![T]\!]$-*elim* and *T-elim* as follows:

$$
\begin{array}{l}
[\![\textit{T-elim}]\!] : [\![\forall a{:}\mathsf{A}.\ \forall P{:}(\forall n{:}\mathsf{N}.\ T\ a\ n \to s).\ \forall e{:}\mathsf{Case}_c. \\
\qquad\qquad\qquad \forall n{:}\mathsf{N}.\ \forall t{:}T\ a\ n.\ P\ n\ t]\!]\ \overline{\textit{T-elim}} \\
[\![\textit{T-elim}\ a\ P\ e]\!] = [\![T]\!]\text{-}\mathit{elim}\ \overline{a}\ a_R \\
\qquad\qquad\qquad\qquad (\lambda [\![n{:}\mathsf{N}, t{:}T\ a\ n]\!]. \\
\qquad\qquad\qquad\qquad\ [\![P\ n\ t]\!]\ \overline{(\textit{T-elim}\ a\ P\ e\ n\ t)}) \\
\qquad\qquad\qquad\qquad (\lambda [\![b{:}\mathsf{B}, u{:}(\forall x{:}\mathsf{X}.\ T\ a\ \mathsf{i})]\!]. \\
\qquad\qquad\qquad\qquad\ [\![e\ b\ u]\!]\ \overline{(\lambda x{:}\mathsf{X}.\ \textit{T-elim}\ a\ P\ e\ \mathsf{i}\ (u\ x))})
\end{array}
$$

Deductive and inductive-style translations define the same relation, but the objects witnessing the instances of the inductively defined-relation record additional information, namely which rules are used to prove membership of the relation. However, since the same constructor never appears in more than one case of the inductive definition, that additional content can be recovered from a witness of the deductive-style; therefore the two styles are truly isomorphic.

Booleans Applying the above scheme to the **data**-declaration of $Bool$ (from Figure 2), we obtain:

$$\begin{array}{l}\mathbf{data}\ [\![Bool]\!] : [\![\star]\!]\ \overline{Bool}\ \mathbf{where}\\ \quad [\![true]\!] : [\![Bool]\!]\ \overline{true}\\ \quad [\![false]\!] : [\![Bool]\!]\ \overline{false}\end{array}$$

The main difference from the deductive-style definition is that it is possible, by analysis of a value of type $[\![Bool]\!]$, to recover the arguments of the relation (either all $true$, or all $false$).

The elimination constant for $Bool$ is

$$\begin{array}{l}Bool\text{-}elim : (P : Bool \to \star) \to P\ true \to P\ false \to\\ \qquad (b : Bool) \to P\ b\end{array}$$

Similarly, our new type $[\![Bool]\!]$ (with $n = 2$) has an elimination constant with the following type:

$$\begin{array}{l}[\![Bool]\!]\text{-}elim :\\ \quad (C : (a_1\ a_2 : Bool) \to [\![Bool]\!]\ a_1\ a_2 \to \star) \to\\ \quad C\ true\ true\ [\![true]\!] \to C\ false\ false\ [\![false]\!] \to\\ \quad (b_1\ b_2 : Bool) \to (b_R : [\![Bool]\!]\ b_1\ b_2) \to C\ b_1\ b_2\ b_R\end{array}$$

As an instance of the above scheme, we can define $[\![Bool\text{-}elim]\!]$ using the elimination constants $[\![Bool]\!]$ and $[\![Bool]\!]\text{-}elim$ as follows (where $t = true$ and $f = false$):

$$\begin{array}{l}[\![Bool\text{-}elim]\!] :\\ \quad (P_1\ P_2 : Bool \to \star) \to (P_R : [\![Bool \to \star]\!]\ P_1\ P_2) \to\\ \quad (x_1 : P_1\ t) \to (x_2 : P_2\ t) \to (P_R\ t\ t\ [\![t]\!]\ x_1\ x_2) \to\\ \quad (y_1 : P_1\ f) \to (y_2 : P_2\ f) \to (P_R\ f\ f\ [\![f]\!]\ y_1\ y_2) \to\\ \quad (b_1\ b_2 : Bool) \to (b_R : [\![Bool]\!]\ b_1\ b_2) \to\\ \quad P_R\ b_1\ b_2\ b_R\ (Bool\text{-}elim\ P_1\ x_1\ y_1\ b_1)\\ \qquad\qquad (Bool\text{-}elim\ P_2\ x_2\ y_2\ b_2)\end{array}$$

$$\begin{array}{l}[\![Bool\text{-}elim]\!]\ P_1\ P_2\ P_R\ x_1\ x_2\ x_R\ y_1\ y_2\ y_R\\ \quad = [\![Bool]\!]\text{-}elim\\ \qquad (\lambda\, b_1\ b_2\ b_R \to P_R\ b_1\ b_2\ b_R\ (Bool\text{-}elim\ P_1\ x_1\ y_1\ b_1)\\ \qquad\qquad\qquad\qquad (Bool\text{-}elim\ P_2\ x_2\ y_2\ b_2))\\ \qquad x_R\ y_R\end{array}$$

lists For $List$, as introduced in Figure 2, we have the following translation:

$$\begin{array}{l}\mathbf{data}\ [\![List]\!]\ ([\![A : \star]\!]) : [\![\star]\!]\ \overline{(List\ A)}\ \mathbf{where}\\ \quad [\![nil]\!] : [\![List\ A]\!]\ \overline{(nil\ A)}\\ \quad [\![cons]\!] : [\![A \to List\ A \to List\ A]\!]\ \overline{(cons\ A)}\end{array}$$

or after expansion (for $n = 2$):

$$\begin{array}{l}\mathbf{data}\ [\![List]\!]\ \ (A_1\ A_2 : \star)\ (A_R : [\![\star]\!]\ A_1\ A_2) :\\ \qquad\qquad List\ A_1 \to List\ A_2 \to \star\ \mathbf{where}\\ \quad [\![nil]\!] : [\![List]\!]\ A_1\ A_2\ A_R\ (nil\ A_1)\ (nil\ A_2)\\ \quad [\![cons]\!] : (x_1 : A_1) \to (x_2 : A_2) \to (x_R : A_R\ x_1\ x_2) \to\\ \qquad\qquad (xs_1 : List\ A_1) \to (xs_2 : List\ A_2) \to\\ \qquad\qquad\quad (xs_R : [\![List]\!]\ A_1\ A_2\ A_R\ xs_1\ xs_2) \to\\ \qquad\qquad [\![List]\!]\ A_1\ A_2\ A_R\ (cons\ A_1\ x_1\ xs_1)\\ \qquad\qquad\qquad\qquad (cons\ A_2\ x_2\ xs_2)\end{array}$$

The above definition encodes the same relational action as that given in Section 4.2. Again, the difference is that the *derivation* of a relation between lists l_1 and l_2 is available as an object of type $[\![List]\!]\ A_1\ A_2\ A_R\ l_1\ l_2$.

proof terms The proof term for the list-rearrangement example can be constructed in a similar way to the inductive one. The main difference is that the target lists are also built and recorded in the $[\![List]\!]$ structure. In short, this version has more of a computational flavour than the inductive version.

$$\begin{array}{l}[\![odds]\!] : [\![(A : \star) \to List\ A \to List\ A]\!]\ odds\ odds\\ [\![odds]\!]\ A_1\ A_2\ A_R\ nil\ nil\ [\![nil]\!] = [\![nil]\!]\ A_1\ A_2\ A_R\end{array}$$

$$\begin{array}{l}[\![odds]\!]\ A_1\ A_2\ A_R\ (cons\ \_\ nil)\ (cons\ \_\ nil)\\ \qquad\qquad ([\![cons]\!]\ x_1\ x_2\ x_R\ nil\ nil\ [\![nil]\!]) =\\ \quad [\![cons]\!]\ A_1\ A_2\ A_R\ x_1\ x_2\ x_R\\ \qquad (nil\ A_1)\ (nil\ A_2)\ ([\![nil]\!]\ A_1\ A_2\ A_R)\\ [\![odds]\!]\ A_1\ A_2\ A_R\\ \qquad (cons\ \_\ (cons\ \_\ \_))\ (cons\ \_\ (cons\ \_\ \_))\\ \qquad ([\![cons]\!]\ x_1\ x_2\ x_R\ xs_1\ xs_2\\ \qquad\qquad ([\![cons]\!]\ \_\ \_\ \_\ xs_1\ xs_2\ xs_R)) =\\ \quad [\![cons]\!]\ A_1\ A_2\ A_R\ x_1\ x_2\ x_R\\ \qquad (odds\ A_1\ xs_1)\ (odds\ A_2\ xs_2)\\ \qquad ([\![odds]\!]\ A_1\ A_2\ A_R\ xs_1\ xs_2\ xs_R)\end{array}$$

vectors We can apply the same translation method to inductive families. For example, Figures 4 and 5 give the translation of the family Vec, corresponding to lists indexed by their length. The relation obtained by applying $[\![\_]\!]$ encodes that vectors are related if their lengths are the same and if their elements are related pointwise. The difference with the $List$ version is that the equality of lengths is encoded in $[\![consV]\!]$ as a Nat (identity) relation.

5. Applications

In this section we shall see how examples going beyond Wadler [1989] can be expressed in our setting. All examples fit within the system I_ω augmented with inductive definitions.

5.1 Type classes

What if a function is not parametrized over all types, but only types equipped with decidable equality? One way to model this difference in a pure type system is to add an extra parameter to capture the extra constraint. For example, a function $nub : Nub$ removing duplicates from a list may be given the following type:

$$Nub = (A : \star) \to Eq\ A \to List\ A \to List\ A$$

The equality requirement itself may be modelled as a mere comparison function: $Eq\ A = A \to A \to Bool$. In that case, the parametricity statement is amended with an extra requirement on the relation between types, which expresses that eq_1 and eq_2 must respect the A_R relation. Formally:

$$\begin{array}{l}[\![Eq\ A]\!]\ eq_1\ eq_2 = (a_1 : A_1) \to (a_2 : A_2) \to A_R\ a_1\ a_2 \to\\ \qquad\qquad\qquad\qquad (b_1 : A_1) \to (b_2 : A_2) \to A_R\ b_1\ b_2 \to\\ \qquad\qquad\qquad\qquad [\![Bool]\!]\ (eq_1\ a_1\ b_1)\ (eq_2\ a_2\ b_2)\\ [\![Nub]\!]\ n_1\ n_2 =\\ \quad (A_1\ A_2 : \star) \to (A_R : [\![\star]\!]\ A_1\ A_2) \to\\ \quad (eq_1 : Eq\ A_1) \to (eq_2 : Eq\ A_2) \to [\![Eq\ A]\!]\ eq_1\ eq_2 \to\\ \quad (l_1 : List\ A_1) \to (l_2 : List\ A_2) \to [\![List\ A]\!]\ l_1\ l_2 \to\\ \quad [\![List\ A]\!]\ (n_1\ A_1\ eq_1\ l_1)\ (n_2\ A_2\ eq_2\ l_2)\end{array}$$

So far, this is just confirming the informal description in Wadler [1989]. But with access to full dependent types, one might wonder: what if we model equality more precisely, for example by requiring eq to be reflexive?

$$\begin{array}{l}Eq'\ A = (eq : A \to A \to Bool) \times Refl\ eq\\ Refl\ eq = (x : A) \to eq\ x\ x \equiv true\end{array}$$

In the case of Eq', the parametricity condition does not become more exciting. It merely requires the proofs of reflexivity at A_1, A_2 to be related. This extra condition adds nothing new: since there is at most one element in (and thus proof of) $x \equiv y$, one already expects proofs to be related.

The observations drawn from this simple example can be generalized in two ways. First, proof arguments do not strengthen parametricity conditions in useful ways. One often does not care about the *actual* proof of a proposition, but merely that it exists, so knowing that two proofs are related adds nothing. Secondly, type-classes

$$
\begin{aligned}
&\mathbf{data}\,[\![\mathit{Vec}]\!]\,([\![A:\star]\!]) : [\![\mathit{Nat}\to\star]\!]\,\overline{(\mathit{Vec}\;A)}\;\mathbf{where}\\
&\quad [\![\mathit{nilV}]\!] \;\; : [\![\mathit{Vec}\;A\;\mathit{zero}]\!]\,\overline{(\mathit{nilV}\;A)}\\
&\quad [\![\mathit{consV}]\!] : [\![(x:A)\to(n:\mathit{Nat})\to\mathit{Vec}\;A\;n\to\mathit{Vec}\;A\,(\mathit{succ}\;n)]\!]\,\overline{(\mathit{consV}\;A)}\\
&\mathbf{data}\,[\![\mathit{Vec}]\!]\,(A_1\;A_2:\star)\,(A_R:A_1\to A_2\to\star) : (n_1\;n_2:\mathit{Nat})\to(n_R:[\![\mathit{Nat}]\!]\;n_1\;n_2)\to\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Vec}\;A_1\;n_1\to\mathit{Vec}\;A_2\;n_2\to\star\;\mathbf{where}\\
&\quad [\![\mathit{nilV}]\!] \;\; : [\![\mathit{Vec}]\!]\;A_1\;A_2\;A_R\;\mathit{zero}\;\mathit{zero}\;[\![\mathit{zero}]\!]\;(\mathit{nilV}\;A_1)\;(\mathit{nilV}\;A_2)\\
&\quad [\![\mathit{consV}]\!] : (x_1:A_1)\to(x_2:A_2)\to(x_R:A_R\;x_1\;x_2)\to\\
&\qquad\qquad (n_1:\mathit{Nat})\to(n_2:\mathit{Nat})\to(n_R:[\![\mathit{Nat}]\!]\;n_1\;n_2)\to\\
&\qquad\qquad (\mathit{xs}_1:\mathit{Vec}\;A_1\;n_1)\to(\mathit{xs}_2:\mathit{Vec}\;A_2\;n_2)\to(\mathit{xs}_R:[\![\mathit{Vec}]\!]\;A_1\;A_2\;A_R\;n_1\;n_2\;n_R\;\mathit{xs}_1\;\mathit{xs}_2)\to\\
&\qquad\qquad [\![\mathit{Vec}]\!]\;A_1\;A_2\;A_R\,(\mathit{succ}\;n_1)\,(\mathit{succ}\;n_2)\,([\![\mathit{succ}]\!]\;n_1\;n_2\;n_R)\,(\mathit{consV}\;A_1\;x_1\;n_1\;\mathit{xs}_1)\,(\mathit{consV}\;A_2\;x_2\;n_2\;\mathit{xs}_2)
\end{aligned}
$$

Figure 4. Inductive translation of Vec, both before and after expansion.

$$
\begin{aligned}
&[\![\;\mathit{Vec\text{-}elim}\;]\!] : [\![\,(A:\star)\to\\
&\qquad\qquad (P:(n:\mathit{Nat})\to\mathit{Vec}\;n\;A\to\star)\to\\
&\qquad\qquad (\mathit{en}:P\;\mathit{zero}\;(\mathit{nilV}\;A))\to\\
&\qquad\qquad (\mathit{ec}:(x:A)\to(n:\mathit{Nat})\to(\mathit{xs}:\mathit{Vec}\;n\;A)\to P\;n\;\mathit{xs}\to P\,(\mathit{succ}\;n)\,(\mathit{consV}\;A\;x\;n\;\mathit{xs}))\to\\
&\qquad\qquad (n:\mathit{Nat})\to(v:\mathit{Vec}\;n\;A)\to P\;n\;v]\!]\;\overline{\mathit{Vec\text{-}elim}}\\
&[\![\;\mathit{Vec\text{-}elim}\;A\;P\;\mathit{en}\;\mathit{ec}]\!] \;=\; [\![\mathit{Vec}]\!]\text{-}\mathit{elim}\;\overline{A}\;A_R\\
&\qquad\qquad\qquad\qquad (\lambda\,[\![n:\mathit{Nat},v:\mathit{Vec}\;n\;A]\!]\to[\![P\;n\;v]\!]\,\overline{(\mathit{Vec\text{-}elim}\;A\;P\;\mathit{en}\;\mathit{ec}\;v)})\\
&\qquad\qquad\qquad\qquad \mathit{en}_R\\
&\qquad\qquad\qquad\qquad (\lambda\,[\![x:A,n:\mathit{Nat},\mathit{xs}:\mathit{Vec}\;n\;A]\!]\to[\![\mathit{ec}\;x\;n\;\mathit{xs}]\!]\,\overline{(\mathit{Vec\text{-}elim}\;A\;P\;\mathit{en}\;\mathit{ec}\;\mathit{xs})})
\end{aligned}
$$

Figure 5. Proof term for $\mathit{Vec\text{-}elim}$ using the inductive-style definitions.

may be encoded as their dictionary of methods [Wadler and Blott 1989]. Indeed, even if a type class has associated laws, they have little impact on the parametricity results.

5.2 Constructor classes

Having seen how to apply our framework both to type constructors and type classes, we now apply it to types quantified over a type constructor, with constraints.

Voigtländer [2009b] provides many such examples, using the Monad constructor class. They fit well in our framework. For the sake of brevity, we do not detail them more here. We can however detail the definition of the simpler $\mathit{Functor}$ class, which can be modelled as follows:

$$
\begin{aligned}
\mathit{Functor} \;=\; &(F:\star\to\star)\times\\
&((X\;Y:\star)\to(X\to Y)\to F\;X\to F\;Y)
\end{aligned}
$$

Our translation readily applies to the above definition, and yields the following relation between functors:

$$
\begin{aligned}
&[\![\mathit{Functor}]\!]\,(F_1,\mathit{map}_1)\,(F_2,\mathit{map}_2)\\
&\quad=\;(F_R:(A_1\;A_2:\star)\to(A_R:A_1\to A_2\to\star)\to\\
&\qquad\quad (F_1\;A_1\to F_2\;A_2\to\star))\times\\
&\qquad((X_1\;X_2:\star)\to(X_R:X_1\to X_2\to\star)\to\\
&\qquad\;(Y_1\;\;Y_2\;\;:\star)\to(Y_R\;\;:Y_1\to Y_2\to\star)\to\\
&\qquad\;(f_1:X_1\to Y_1)\to(f_2:X_2\to Y_2)\to\\
&\qquad\;((x_1:X_1)\to(x_2:X_2)\to(x_R:X_R\;x_1\;x_2)\to\\
&\qquad\;\;Y_R\,(f_1\;x_1)\,(f_2\;x_2))\to\\
&\qquad\;(y_1:F_1\;X_1)\to(y_2:F_2\;X_2)\to\\
&\qquad\;(y_R:F_R\;X_R\;y_1\;y_2)\to\\
&\qquad\;F_R\;Y_R\,(\mathit{map}_1\;f_1\;y_1)\,(\mathit{map}_2\;f_2\;y_2))
\end{aligned}
$$

In words, the translation of a functor is the product of a relation transformer (F_R) between functors F_1 and F_2, and a witness (map_R) that map_1 and map_2 preserve relations.

Such $\mathit{Functor}$s can be used to define a generic fold operation, which typically takes the following form:

$$
\begin{aligned}
&\mathbf{data}\,\mu\,((F,\mathit{map}):\mathit{Functor}):\star\,\mathbf{where}\\
&\quad \mathit{In}:F\,(\mu\,(F,\mathit{map}))\to\mu\,(F,\mathit{map})\\
&\mathit{fold}:((F,\mathit{map}):\mathit{Functor})\to(A:\star)\to\\
&\qquad (F\;A\to A)\to\mu\,(F,\mathit{map})\to A\\
&\mathit{fold}\,(F,\mathit{map})\;A\;\phi\,(\mathit{In}\;d)\;=\\
&\quad \phi\,(\mathit{map}\,(\mu\,(F,\mathit{map}))\;A\,(\mathit{fold}\,(F,\mathit{map})\;A\;\phi)\;d)
\end{aligned}
$$

Note that the μ datatype is not strictly positive, so its use would be prohibited in many dependently-typed languages to avoid inconsistency. However, if one restricts oneself to well-behaved functors (yielding strictly positive types), then consistency is restored both in the source and target systems, and the parametricity condition derived for fold is valid.

One can see from the type of fold that it behaves uniformly over (F,map) as well as A. By applying $[\![\_]\!]$ to fold and its type, this observation can be expressed (and justified) formally and used to reason about fold. Further, every function defined using fold, and in general any function parametrized over any functor enjoys the same kind of property.

Gibbons and Paterson [2009] previously made a similar observation in a categorical setting, showing that fold is a natural transformation between higher-order functors. Their argument heavily relies on categorical semantics and the universal property of fold, while our type-theoretical argument uses the type of fold as a starting point and directly obtains a parametricity property. However some additional work is required to obtain the equivalent property using natural transformations and horizontal compositions from the parametricity property.

5.3 Generic cast

Continuing to apply our framework to terms of increasingly rich types, the next candidate is dependently typed.

An important application of dependent types is that of generic programming with universes, as in the work of Altenkirch and McBride [2003]; Benke et al. [2003]. The basic idea is to represent the "universe" of types as data, and provide an interpretation function from values of this data type to types (in $\star$). Generic functions can then be written by pattern matching on the type representation. While universes usually capture large classes of types, we use as an example a very simple universe of types for Booleans and natural numbers, as follows.[4]

$$\begin{aligned}
&\mathbf{data}\ U : \star_1\ \mathbf{where}\\
&\quad bool : U\\
&\quad nat : U\\
&El : U \to \star\\
&El\ bool\ =\ Bool\\
&El\ nat\ =\ Nat
\end{aligned}$$

An example of a dependently-typed, generic function is $gcast$, which for any type context F and any two (codes for) types u and t, returns a casting function between $F\,(El\,u)$ and $F\,(El\,t)$, if u and t are the same (and $nothing$ otherwise).

$$\begin{aligned}
&gcast : (F : \star \to \star) \to (u\,t : U) \to\\
&\qquad Maybe\,(F\,(El\,u) \to F\,(El\,t))\\
&gcast\ F\ bool\ bool\ =\ just\,(\lambda\,x \to x)\\
&gcast\ F\ nat\ \ nat\ \ =\ just\,(\lambda\,x \to x)\\
&gcast\ F\ _\ \ \ _\ \ \ =\ nothing\\
&\mathbf{data}\ Maybe\,(A : \star) : \star\,\mathbf{where}\\
&\quad nothing : Maybe\ A\\
&\quad just : A \to Maybe\ A
\end{aligned}$$

The function $gcast$ is deemed *safe* if it returns the identity function whenever it returns something. Vytiniotis and Weirich [2009] show that this theorem can be deduced from the type of $gcast$ alone, by parametricity. While the result can be re-derived in a simple way by reasoning directly on the definition of $gcast$, there is a good reason for using parametricity: as the universe is extended to a realistic definition, the definition of $gcast$ gets more complex, but its type remains the same, and therefore the argument relying on parametricity is unchanged.

The rest of this section is devoted to rederiving the result using our framework. The first step is to encode the theorem. We can encode that an arbitrary function $f : A \to B$ is the identity as the formula $(x : A) \to f\,x \cong x$. Note that because the input and output types of the cast are not definitionally equal, we must use a heterogeneous equality ($\cong$), defined as follows:

$$\begin{aligned}
&\mathbf{data} \cong (A : \star)\,(a : A) : (B : \star) \to B \to \star\,\mathbf{where}\\
&\quad refl' :\ \cong A\ a\ A\ a
\end{aligned}$$

Now, $gcast$ is not a direct conversion function: sometimes it returns no result; its result is wrapped in $Maybe$. Hence we use a helper function to lift the identity predicate to a $Maybe$ type:

$$\begin{aligned}
&onMaybe : (A : \star) \to (A \to \star) \to Maybe\ A \to \star\\
&onMaybe\ A\ P\ nothing\ =\ \top\\
&onMaybe\ A\ P\,(just\ a)\ =\ P\ a
\end{aligned}$$

The theorem can then be expressed as follows:

Theorem 3 (gcast is safe)**.**

$$\begin{aligned}
&(F : \star \to \star) \to (u\,t : U) \to (x : F\,(El\,u)) \to\\
&\quad onMaybe\,(F\,(El\,u) \to F\,(El\,t))\\
&\qquad\qquad (\lambda\,cast \to cast\,x \cong x)\\
&\qquad\qquad (gcast\ F\ u\ t)
\end{aligned}$$

[4] For the present section, $U : \star$ would be sufficient, but we define $U : \star_1$ to permit a different definition of $[\![U]\!]$ in the next section.

We remark that $onMaybe$ is in fact the deductive version of $[\![Maybe]\!]$, for the *unary* version of $[\![\_]\!]$. We take this as a hint to use the unary version of $[\![\_]\!]$, and derive relations of the following types:

$$\begin{aligned}
&[\![U]\!] : U \to \star_1\\
&[\![El]\!] : (u : U) \to (u_R : [\![U]\!]\,u) \to [\![\star]\!]\,(El\,u)\\
&[\![gcast]\!] : (F : \star \to \star) \to (F_R : [\![\star \to \star]\!]\,F) \to\\
&\quad (u : U) \to (u_R : [\![U]\!]\,u) \to\\
&\quad (t : U) \to (t_R : [\![U]\!]\,t) \to\\
&\quad [\![Maybe]\!]\,(F\,(El\,u) \to F\,(El\,t))\\
&\qquad\qquad (\lambda\,cast \to (x : F\,(El\,u)) \to\\
&\qquad\qquad\qquad\qquad F_R\,(El\,u)\,([\![El]\!]\,u\,u_R)\,x \to\\
&\qquad\qquad\qquad\qquad F_R\,(El\,t)\ ([\![El]\!]\,t\,t_R)\ \ (cast\,x))\\
&\qquad\qquad (gcast\ F\ u\ t)
\end{aligned}$$

Additionally, we can define $param_U$:

$$\begin{aligned}
&param_U : (u : U) \to [\![U]\!]\,u\\
&param_U\ bool\ =\ [\![bool]\!]\\
&param_U\ nat\ \ =\ [\![nat]\!]
\end{aligned}$$

We can then use $[\![gcast]\!]$ to prove the theorem. The idea is to specialize it to the types and relations of interest:

$$\begin{aligned}
&lemma1 : (F : \star \to \star) \to (u\,t : U) \to (x : F\,(El\,u)) \to\\
&\quad [\![Maybe]\!]\,(F\,(El\,u) \to F\,(El\,t))\\
&\qquad\qquad (\lambda\,cast \to (x' : F\,(El\,u)) \to x' \cong x \to\\
&\qquad\qquad\qquad\qquad\qquad\qquad cast\,x' \cong x)\\
&\qquad\qquad (gcast\ F\ u\ t)\\
&lemma1\ F\ u\ t\ x\ =\ [\![gcast]\!]\ F\,(\lambda\,t_R\ y \to y \cong x)\\
&\qquad\qquad\qquad\qquad\qquad\qquad u\ \ (param_U\ u)\\
&\qquad\qquad\qquad\qquad\qquad\qquad t\ \ (param_U\ t)
\end{aligned}$$

By fixing x' to x in the argument to $[\![Maybe]\!]$, the condition $x' \cong x$ is fulfilled, and the proof is complete.

The remarkable feature of this proof is that it is essentially independent of the definitions of U and El: only their types matter. Adding constructors in U would not change anything in the proof: $[\![gcast]\!]$ isolates Theorem 3 from the actual definitions of U, El and $gcast$; it can be generated automatically from $gcast$.

In summary, we have proved the correctness of $gcast$ in three steps:

1. Model representation types within our dependently-typed language;
2. use $[\![\_]\!]$ to obtain parametricity properties of any function of interest;
3. prove correctness by using the properties.

We think that the above process is an economical way to work with parametricity for extended type systems. Indeed, step one of the above process is becoming an increasingly popular way to develop languages with exotic type systems as an embedding in a dependently-typed language [Oury and Swierstra 2008]. By providing (an automatic) step two, we hope to spare language designers the effort to adapt Reynolds' abstraction theorem for new type systems in an ad-hoc way.

5.4 A partially constrained universe

So far we have only seen universes which are either completely unconstrained (like $\star$) and translate to arbitrary relations, or universes which are completely constrained (like $Bool$ or U in the previous section) and translate to the identity relation. In this section we show that a middle ground is also possible.

Suppose that we want the same universe as in the above section, but with only limited capabilities to dispatch on the type. That is, we allow users to define functions that have special behaviour for

Booleans, but are otherwise oblivious to the actual type at which they are used. This particular functionality may be encoded by only providing an eliminator for U with restricted capabilities:

$$\begin{aligned}
&\mathit{typeTest} : (u : U) \to (F : \star \to \star) \to \\
&\qquad F\ \mathit{Bool} \to ((A : \star) \to F\ A) \to F\ (\mathit{El}\ u) \\
&\mathit{typeTest}\ \mathit{bool}\ F\ A_B\ A_{Gen} = A_B \\
&\mathit{typeTest}\ t \quad F\ A_B\ A_{Gen} = A_{Gen}\ (\mathit{El}\ t)
\end{aligned}$$

This restriction of elimination allows us to "relax" the definitions of $[\![U]\!]$ and $[\![El]\!]$, by translating the cases that do not involve bool to an arbitrary relation (for $n = 2$):

$$\begin{aligned}
&[\![U]\!] : U \to U \to \star_1 \\
&[\![U]\!]\ \mathit{bool}\ \mathit{bool} = \top \\
&[\![U]\!]\ \mathit{bool}\ _ \quad = \bot \\
&[\![U]\!]\ _ \quad \mathit{bool} = \bot \\
&[\![U]\!]\ u_1 \quad u_2 \quad = [\![\star]\!]\ (\mathit{El}\ u_1)\ (\mathit{El}\ u_2) \\
&[\![El]\!] : (u_1\ u_2 : U) \to (u_R : [\![U]\!]\ u_1\ u_2) \to \\
&\qquad [\![\star]\!]\ (\mathit{El}\ u_1)\ (\mathit{El}\ u_2) \\
&[\![El]\!]\ \mathit{bool}\ \mathit{bool}\ r = [\![\mathit{Bool}]\!] \\
&[\![El]\!]\ u_1 \quad u_2 \quad r = r
\end{aligned}$$

Given the above definitions, free theorems involving U reduce to the constrained case if presented with Booleans, and to the unconstrained case otherwise.

While the above is a toy example, it points the way towards more sophisticated representations of universes. An example would be an encoding of fresh abstract type variables, as in Neis et al. [2009].

6. Discussion

6.1 Proof

A detailed sketch of the proof of Theorem 1 is available online [Bernardy et al. 2010b]. Beyond the pen-and-paper version, we also have a machine-checked proof, for the unary case, as an Agda program [Bernardy 2010]. A few improvements are necessary before it can be considered a fully-machine-checked proof:

- some substitution lemmas need to be proved;
- the top-level structure needs some superficial restructuring to pass the termination-check of the Agda system;
- proofs of some lemmas given by Barendregt [1992] should be formalized.

6.2 Different source and target sorts

Even though the sort-mapping function $\widetilde{\ }$ used in all our examples has been the identity, there are other possible choices. For example, I_ω is reflective with $\widetilde{\star_i} = \star_{i+k}$, for any natural k. Other examples can be constructed by mapping $\widetilde{\ }$ to "fresh" sorts. The following system ($I_\omega{}^+$) is reflective with $\widetilde{\star_i} = \triangle_i$ and $\widetilde{\triangle_i} = \triangle_i$.

Definition 7 ($I_\omega{}^+$)**.** $I_\omega{}^+$ is a PTS with this specification:

- $\mathcal{S} = \{\star_i \mid i \in \mathbb{N}\} \cup \{\triangle_i \mid i \in \mathbb{N}\}$
- $\mathcal{A} = \{\triangle_i : \triangle_{i+1} \mid i \in \mathbb{N}\} \cup \{\star_i : \star_{i+1} \mid i \in \mathbb{N}\}$
- $\mathcal{R} = \{(\star_i, \star_j, \star_{\max(i,j)}) \mid i, j \in \mathbb{N}\}$
  $\{(\triangle_i, \triangle_j, \triangle_{\max(i,j)}) \mid i, j \in \mathbb{N}\}$
  $\{\star_i \rightsquigarrow \triangle_j \mid i \leqslant j \in \mathbb{N}\}$

6.3 Different source and target systems

For simplicity, we have chosen to use the same source and target PTS in Theorem 1. However, the theorem may be generalized to the case where source and target are different. One way to relax the hypothesis is to allow any source PTS which is a subsystem of the target one, keeping the same conditions for the target PTS.

For example, using this generalization, we see that all the parametricity statements about terms in the λ-cube are expressible and provable in the generalized calculus of constructions (CC_ω). Indeed, we observe that

- CC_ω is reflective with $\widetilde{s} = s$ and,
- All eight systems of the λ-cube are embedded in CC_ω

While extending our abstraction to subsystems is useful, further generalization is possible. For example, parametricity theorems (and proofs) generated from terms in the λ-cube will never use the higher sorts of CC_ω. Specifying necessary and sufficient conditions for the two-system case is left as future work.

6.4 Internalizing the meta-theorem

Theorem 1 and Corollary 2 ($\vdash \mathsf{A} : \mathsf{B} \implies \vdash [\![\mathsf{A}]\!] : [\![\mathsf{B}]\!]\ \overline{\mathsf{A}}$) are meta-theorems. One can instantiate the corollary by choosing specific terms A and B; then $[\![\mathsf{A}]\!]$ is a proof of $[\![\mathsf{B}]\!]\ \overline{\mathsf{A}}$ in the system, derived from the structure of $\vdash \mathsf{A} : \mathsf{B}$. Our examples consist of many such instantiations.

However, one would like to go further and make a general statement about about all values of type B within the system. That is, for a type B, to define $\mathit{param}_\mathsf{B} : (\forall x : \mathsf{B}.\ [\![\mathsf{B}]\!]\ x\ \ldots\ x)$, as we did with param_U in Section 5.3, essentially making the semantics of the type available for reasoning within the system. In particular, for any constant $k : \mathsf{B}$, we could define $[\![k]\!] = \mathit{param}_\mathsf{B}\ k$.

One way to proceed is to assert parametricity at all types, with a constant $\mathit{param}_\mathsf{B}$ for each B. This approach was applied to CC by Takeuti [2004], extending similar axiom schemes for System F by Plotkin and Abadi [1993]. For each $\alpha : \Box$ and $\mathsf{P} : \alpha$, Takeuti defined a relational interpretation $\langle\mathsf{P}\rangle$ and a kind $(\!|\mathsf{P} : \alpha|\!)$ such that $\langle\mathsf{P}\rangle : (\!|\mathsf{P} : \alpha|\!)$. Then for each type $\mathsf{T} : \star$, he postulated an axiom $\mathit{param}_\mathsf{T} : (\forall x : \mathsf{T}.\ \langle\mathsf{T}\rangle\ x\ x)$, conjecturing that such axioms did not make the system inconsistent. For closed terms P, Takeuti's translations $\langle\mathsf{P}\rangle$ and $(\!|\mathsf{P} : \alpha|\!)$ resemble our $[\![\mathsf{P}]\!]$ and $[\![\alpha]\!]\ \overline{\mathsf{P}}$ respectively (with $n = 2$), but the pattern is obscured by an error in the translation rule for the product $\Box \rightsquigarrow \star$, and the omission of a witness x_R for the relationship between values x_1 and x_2 in the rules corresponding to the product $\star \rightsquigarrow \Box$.

Another approach would be to provide access to the terms via some form of *reflection*.

6.5 Related work

Some of the many studies of parametricity have already been mentioned and analysed in the rest of the paper. In this section we compare our work to only a couple of the most relevant pieces of work.

One direction of research is concerned with parametricity in extensions of System F. Our work is directly inspired by Vytiniotis and Weirich [2010], which extend parametricity to (an extension of) Fω: indeed, Fω can be seen as a PTS with one more product rule than System F.

Besides supporting more sorts and function spaces, an orthogonal extension of parametricity theory is to support impure features in the system. For example, [Johann and Voigtländer 2005] studied how explicit strictness modifies parametricity results. It is not obvious how to support such extensions in our framework.

Another direction of research is concerned with better understanding of parametricity. Here we shall mention only [Wadler 2007], which gives a particularly lucid presentation of the abstraction theorem, as the inverse of Girard's Representation theorem [Girard 1972]. Our version of the abstraction theorem differs in the following aspects compared to that of Wadler (and to our knowledge all others):

1. Instead of targeting a logic, we target its *propositions-as-types* interpretation, expressed in a PTS.

2. We abstract from the details of the systems, generalizing to a class of PTS's.
3. We add that the translation function used to interpret types as relations can also be used to interpret terms as witnesses of those relations. In short, the $[\![A]\!]$ part of $\Gamma \vdash A : B \implies [\![\Gamma]\!] \vdash [\![A]\!] : [\![B]\!]\ \overline{A}$ is new. This additional insight depends heavily on using the propositions-as-types interpretation.

It also appears that the function $[\![\_]\!]$ (for the unary case) has been discovered independently by Monnier and Haguenauer [2010], for a very different purpose. They use $[\![\_]\!]$ as a compilation function from CC to a language with singleton types only, in order to enforce phase-distinction. Type preservation of the translation scheme is the main formal property presented by Monnier and Haguenauer. We remark that this property corresponds to the abstraction theorem for CC.

6.6 Future work

Our explanation of parametricity for dependent types has opened a whole range of interesting topics for future work.

We should investigate whether our framework can be applied (and extended if need be) to more exotic systems, for example those incorporating strictness annotations (seq) or non-termination.

We should extend our translation to support non-informative function spaces, as found for example in Coq. In Coq, the sort $\star$ of CC is split into two separate sorts, one for types (Set) and one for propositions ($Prop$). Inhabitants of Set can depend on inhabitants of $Prop$: for example, a program may depend on a certain property to terminate. However, computational content can never "leak" from $Prop$ to Set: programs may only depend on the existence of a proof; it is forbidden to inspect their structure. In such a situation, our translation scheme appears to generate parametricity results that are too weak, as we have briefly alluded to in Section 5.1. The reason is that we always assume that computational content may be transferred from the argument of a function to its result. We could modify the translation to omit the superfluous relation parameter in such cases.

Reynolds' abstraction theorem can be understood as an embedding of polymorphic lambda calculus into second order propositional logic. Wadler [2007] showed that Girard's representation theorem [Girard 1972] can be understood as the corresponding projection. In this work we have shown that the embedding can be generalized for more complex type systems. The question of how the projection generalizes naturally arises, and should also be addressed.

It is straightforward to derive translated types using our schema, but tedious. Providing $[\![\_]\!]$ as a meta-function would greatly ease experimentation with our technique. Another direction worth exploring is to provide the parametricity axiom ($param\_$) as a meta-function in a logical framework.

We presented only simple examples. Applying the results to more substantial applications should be done as well.

7. Conclusion

We have shown that it is not only possible, but easy to derive parametricity conditions in a dependently-typed language.

Further, it is possible to analyse parametricity properties of custom languages, via their embedding in a dependently-typed host language.

Acknowledgments

Thanks to Andreas Abel, Thierry Coquand, Peter Dybjer, Marc Lasson, Guilhem Moulin, Ulf Norell, Nicolas Pouillard and anonymous reviewers for providing us with very valuable feedback.

References

T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *Proc. of the IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 1–20. Kluwer, B.V., 2003.

H. P. Barendregt. Lambda calculi with types. *Handbook of logic in computer science*, 2:117–309, 1992.

M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic J. of Computing*, 10(4):265–289, 2003.

J.-P. Bernardy. A proof of the abstraction theorem for pure type systems (unary case). http://www.cse.chalmers.se/~bernardy/ParDep/html/Theorem.html, 2010.

J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In *Proc. of ESOP 2010*, volume 6012 of *LNCS*. Springer, 2010a.

J.-P. Bernardy, P. Jansson, and R. Paterson. An abstraction theorem for pure type systems. Available from http://www.cse.chalmers.se/~bernardy/ParDep/abstraction-pts.pdf, 2010b.

C. Böhm and A. Berarducci. Automatic synthesis of typed lambda-programs on term algebras. *Theor. Comp. Sci.*, 39(2-3):135–154, 1985.

T. Coquand. An analysis of Girard's paradox. In *Proc. of LICS 1986*, pages 227–236. IEEE Comp. Society Press, 1986.

T. Coquand. Pattern matching with dependent types. In *Proc. of the Workshop on Types for Proofs and Programs*, pages 66–79, 1992.

P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

J. Gibbons and R. Paterson. Parametric datatype-genericity. In *Proc. of WGP 2009*, pages 85–93, Edinburgh, Scotland, 2009. ACM.

A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *Proc. of FPCA*, pages 223–232, Copenhagen, Denmark, 1993. ACM.

J. Y. Girard. *Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, Université de Paris 7, 1972.

R. Hinze. Church numerals, twice! *J. Funct. Program.*, 15(1):1–13, 2005.

P. Johann. A generalization of short-cut fusion and its correctness proof. *Higher-Order and Symbol. Comput.*, 15(4):273–300, 2002.

P. Johann and J. Voigtländer. The impact of seq on free theorems-based program transformations. *Fundam. Inf.*, 69(1-2):63–102, 2005.

C. McBride and J. McKinna. The view from the left. *J. Funct. Program.*, 14(01):69–111, 2004.

A. Miquel. *Le Calcul des Constructions implicite: syntaxe et sémantique*. Thèse de doctorat, Université Paris 7, 2001.

S. Monnier and D. Haguenauer. Singleton types here, singleton types there, singleton types everywhere. In *Proc. of PLPV 2010*, pages 1–8, Madrid, Spain, 2010. ACM.

G. Neis, D. Dreyer, and A. Rossberg. Non-parametric parametricity. In *Proc. of ICFP 2009*, pages 135–148, Edinburgh, Scotland, 2009. ACM.

U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers Tekniska Högskola, 2007.

N. Oury and W. Swierstra. The power of Pi. In *Proc. of ICFP 2008*, pages 39–50, Victoria, BC, Canada, 2008. ACM.

C. Paulin-Mohring. Inductive definitions in the system Coq – rules and properties. In *Typed Lambda Calculi and Applications*, pages 328–345. Springer, 1993.

G. Plotkin and M. Abadi. A logic for parametric polymorphism. In *LNCS*, volume 664, page 361–375. Springer-Verlag, 1993.

J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information processing*, 83(1):513–523, 1983.

I. Takeuti. The theory of parametricity in lambda cube. Manuscript, 2004.

The Coq development team. The Coq proof assistant, 2010.

J. Voigtländer. Bidirectionalization for free! (Pearl). In *Proc. of POPL 2009*, pages 165–176, Savannah, GA, USA, 2009a. ACM.

J. Voigtländer. Free theorems involving type constructor classes: Funct. pearl. *SIGPLAN Not.*, 44(9):173–184, 2009b.

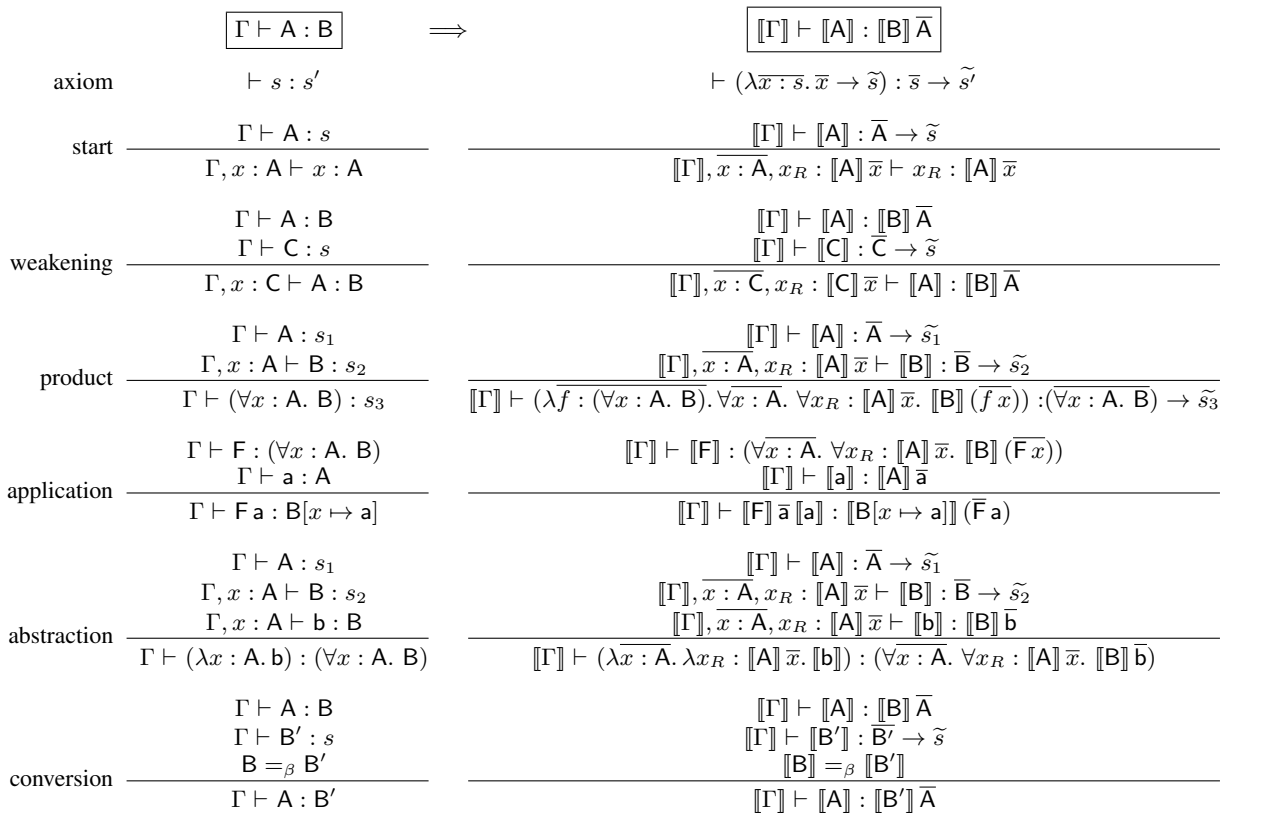

Figure 6. Outline of a proof of Theorem 1 by induction over the derivation of $\Gamma \vdash \mathsf{A} : \mathsf{B}$.

D. Vytiniotis and S. Weirich. Type-safe cast does no harm: Syntactic parametricity for Fω and beyond. Preliminary version of "Parametricity, Type Equality and Higher-order Polymorphism", 2009.

D. Vytiniotis and S. Weirich. Parametricity, type equality, and higher-order polymorphism. *J. Funct. Program.*, 20(2):175–210, 2010.

P. Wadler. Theorems for free! In *Proc. of FPCA 1989*, pages 347–359, Imperial College, London, United Kingdom, 1989. ACM.

P. Wadler. The Girard–Reynolds isomorphism. *Theor. Comp. Sci.*, 375(1–3):201–226, 2007.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL'89*, pages 60–76. ACM, 1989.

A. Proof of the abstraction theorem

In this appendix we sketch the proof of our main theorem, using the following lemma:

Lemma 4 (translation preserves β-reduction)**.**

$$\mathsf{A} \longrightarrow_\beta^* \mathsf{A}' \implies [\![\mathsf{A}]\!] \longrightarrow_\beta^* [\![\mathsf{A}']\!]$$

Proof sketch. The proof proceeds by induction on the derivation of $\mathsf{A} \longrightarrow_\beta^* \mathsf{A}'$. The interesting case is where the term A is a β-redex $(\lambda x : \mathsf{B}.\, \mathsf{C})\,\mathsf{b}$. That case relies on the way $[\![\_]\!]$ interacts with substitution:

$$[\![\mathsf{b}[x \mapsto \mathsf{C}]]\!] = [\![\mathsf{b}]\!][\overline{x} \mapsto \overline{\mathsf{C}}][x_R \mapsto [\![\mathsf{C}]\!]]$$

The remaining cases are congruences. □

Theorem (abstraction)**.** *In a reflective PTS,*

$$\Gamma \vdash \mathsf{A} : \mathsf{B} \implies [\![\Gamma]\!] \vdash [\![\mathsf{A}]\!] : [\![\mathsf{B}]\!]\,\overline{\mathsf{A}}$$

Proof sketch. A derivation of $[\![\Gamma]\!] \vdash [\![\mathsf{A}]\!] : [\![\mathsf{B}]\!]\,\overline{\mathsf{A}}$ is constructed by induction on the derivation of $\Gamma \vdash \mathsf{A} : \mathsf{B}$, using the syntactic properties of PTSs. We have one case for each typing rule: each type rule translates to a portion of a corresponding relational typing judgement, as shown in Figure 6.

For convenience, the proof uses a variant form of the abstraction rule; equivalence of the two systems follows from Barendregt [1992, Lemma 5.2.13]. The conversion case uses Lemma 4. □