# Realizability and Parametricity in Pure Type Systems

Jean-Philippe Bernardy[1] and Marc Lasson[2]

[1] Chalmers University of Technology and University of Gothenburg
[2] ENS Lyon, Université de Lyon, LIP (UMR 5668 CNRS ENS Lyon UCBL INRIA)

**Abstract.** We describe a systematic method to build a logic from any programming language described as a Pure Type System (PTS). The formulas of this logic express properties about programs. We define a parametricity theory about programs and a realizability theory for the logic. The logic is expressive enough to internalize both theories. Thanks to the PTS setting, we abstract most idiosyncrasies specific to particular type theories. This confers generality to the results, and reveals parallels between parametricity and realizability.

## 1 Introduction

During the past decades, a recurring goal among logicians was to give a computational interpretation of the reasoning behind mathematical proofs. In this paper we adopt the converse approach: we give a systematic way to build a logic from a programming language. The structure of the programming language is replicated at the level of the logic: the expressive power of the logic (e.g. the ability of expressing conjunctions) is directly conditioned by the constructions available in the programming language (e.g. presence of products).

We use the framework of Pure Type Systems (PTS) to represent both the starting programming language and the logic obtained by our construction. A PTS [2, 3] is a generalized $\lambda$-calculus where the syntax for terms and types are unified. Many systems can be expressed as PTSs, including the simply typed $\lambda$-calculus, Girard and Reynolds polymorphic $\lambda$-calculus (System F) and its extension System F$\omega$, Coquand's Calculus of Constructions, as well as some exotic, and even inconsistent systems such as $\lambda$U [8]. PTSs can model the functional core of many modern programming languages (Haskell, Objective Caml) and proof assistants (Coq [25], Agda [19], Epigram [17]). This unified framework provides meta-theoretical such as substitution lemmas, subject reduction and uniqueness of types.

In Sec. 3, we describe a transformation which maps any PTS $P$ to a PTS $P^2$. The starting PTS $P$ will be viewed as a programming language in which live *types* and *programs* and $P^2$ will be viewed as a proof system in which live *proofs* and *formulas*. The logic $P^2$ is expressive enough to state properties about the programs. It is therefore a setting of choice to develop a parametricity and a realizability theory.

*Parametricity.* Reynolds [23] originally developed the theory of parametricity to capture the meaning of types of his polymorphic $\lambda$-calculus (equivalent to Girard's System F). Each closed type can be interpreted as a predicate that all its inhabitants satisfy. Reynolds' approach to parametricity has proven to be a successful tool: applications range from program transformations to speeding up program testing [28, 7, 4].

Parametricity theory can be adapted to other $\lambda$-calculi, and for each calculus, parametricity predicates are expressed in a corresponding logic. For example, Abadi et al. [1] remark that the simply-typed lambda calculus corresponds to LCF [18]. For System F, predicates can be expressed in second order predicate logic, in one or another variant [1, 16, 29]. More recently, Bernardy et al. [5] have shown that parametricity conditions for a reflective PTS can be expressed in the PTS itself.

*Realizability.* The notion of realizability was first introduced by Kleene [10] in his seminal paper. The idea of relating programs and formulas, in order to study their constructive content, was then widely used in proof theory. For example, it provides tools for proving that an axiom is not derivable in a system (excluded middle in [11, 26]) or that intuitionistic systems satisfy the *existence property*[3] [9, 26]; see Van Oosten [27] for an historical account of realizability.

Originally, Kleene represented programs as integers in a theory of recursive functions. Later, this technique has been extended to other notions of programs like combinator algebra [24, 26] or terms of Gödel's System T [12, 26] in Kreisel's modified realizability. In this article, we generalize the latter approach by using an arbitrary pure type system as the language of programs.

Krivine [13] and Leivant [15] have used realizability to prove Girard's representation theorem[4] [8] and to build a general framework for extracting programs from proofs in second-order logic [14]. In this paper, we extend Krivine's methodology to languages with dependent types, like Paulin-Mohring [20, 21] did with the realizability theory behind the program extraction in the Coq proof assistant [25].

*Contributions.* Viewed as syntactical notions, realizability and parametricity bear a lot of similarities. Our aim was to understand through the generality of PTSs how they are related. Our main contributions are:

- The general construction of a logic from the programming language of its realizers with syntactic definitions of parametricity and realizability (Sec. 3).
- The proof that this construction is strongly normalizing if the starting programming language is (Thm. 2).
- A characterization of both realizability in terms of parametricity (Thm. 6) and parametricity in terms of realizability (Thm. 5).

[3] If $\forall x \exists y, \varphi(x, y)$ is a theorem, then there exists a program $f$ such that $\forall x, \varphi(x, f(x))$.
[4] Functions definable in System F are exactly those provably total in second-order arithmetic.

## 2 The First Level

In this section, we recall basic definitions and theorems about pure types systems (PTSs). We refer the reader to [2] for a comprehensive introduction to PTSs. A PTS is defined by a specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ where $\mathcal{S}$ is a set of *sorts*, $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ a set of *axioms* and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ a set of *rules*, which determines the typing of product types. The typing judgement is written $\Gamma \vdash A : B$. The notation $\Gamma \vdash A : B : C$ is a shorthand for having both $\Gamma \vdash A : B$ and $\Gamma \vdash B : C$ simultaneously.

*Example 1 (System* F*).* The PTS F has the following specification:

$$\mathcal{S}_{\mathrm{F}} = \{\star, \Box\} \qquad \mathcal{A}_{\mathrm{F}} = \{(\star, \Box)\} \qquad \mathcal{R}_{\mathrm{F}} = \{(\star, \star, \star), (\Box, \star, \star)\}.$$

It defines the $\lambda$-calculus with polymorphic types known as system F [8]. The rule $(\star, \star, \star)$ corresponds to the formation of arrow types (usually written $\sigma \to \tau$) and the rule $(\Box, \star, \star)$ corresponds to quantification over types $(\forall \alpha, \tau)$.

Even though we use F as a running example throughout the article to illustrate our general definitions our results apply to any PTS.

*Sort annotations.* We sometimes decorate terms with *sort annotations*. They function as a syntactic reminder of the first component of the rule used to type a product. We divide the set of variables into disjoint infinite subsets $\mathcal{V} = \bigsqcup\{\mathcal{V}_s | s \in \mathcal{S}\}$ and we write $x^s$ to indicate that a variable $x$ belongs to $\mathcal{V}_s$. We also annotate applications $F\,a$ with the sort of the variable of the product type of $F$. Using this notation, the product rule and the application rule are written

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x^{s_1} : A \vdash B : s_2}{\Gamma \vdash (\Pi x^{s_1} : A.\, B) : s_3}\ \textsc{Product}\ (s_1, s_2, s_3) \in \mathcal{R} \qquad \frac{\Gamma \vdash F : (\Pi x^s : A.\, B) \qquad \Gamma \vdash a : A}{\Gamma \vdash (F\,a)_s : B[x \mapsto a]}\ \textsc{Application}.$$

Since sort annotations can always be recovered by using the type derivation, we do not write them in our examples.

*Example 2 (System* F *terms).* In System F, we adopt the following convention: the letters $x$, $y$, $z$, ... range over $\mathcal{V}_\star$, and $\alpha$, $\beta$, $\gamma$, ... over $\mathcal{V}_\Box$. For instance, the identity program $\mathrm{Id} \equiv \lambda(\alpha : \star)(x : \alpha).x$ is of type $\mathrm{Unit} \equiv \Pi\alpha : \star.\alpha \to \alpha$. The Church numeral $0 \equiv \lambda(\alpha : \star)(f : \alpha \to \alpha)(x : \alpha).x$ has type $\mathrm{Nat} \equiv \Pi\alpha : \star.(\alpha \to \alpha) \to (\alpha \to \alpha)$ and the successor function on Church numerals $\mathrm{Succ} \equiv \lambda(n : \mathrm{Nat})(\alpha : \star)(f : \alpha \to \alpha)(x : \alpha).f\,(n\,\alpha\,f\,x)$ is a program of type $\mathrm{Nat} \to \mathrm{Nat}$.

## 3 The Second Level

In this section we describe the logic to reason about the programs and types written in an arbitrary PTS $P$, as well as basic results concerning the consistency

of the logic. This logic is also a PTS, which we name $P^2$. Because we carry out most of our development in $P^2$, judgments refer to that system unless the symbol $\vdash$ is subscripted with the name of a specific system.

**Definition 1 (second-level system).** *Given a PTS $P = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, we define $P^2 = (\mathcal{S}^2, \mathcal{A}^2, \mathcal{R}^2)$ by*

$$\begin{aligned}
\mathcal{S}^2 &= \mathcal{S} \cup \{\lceil s \rceil \mid s \in \mathcal{S}\} \\
\mathcal{A}^2 &= \mathcal{A} \cup \{(\lceil s_1 \rceil, \lceil s_2 \rceil) \mid (s_1, s_2) \in \mathcal{A}\} \\
\mathcal{R}^2 &= \mathcal{R} \cup \{(\lceil s_1 \rceil, \lceil s_2 \rceil, \lceil s_3 \rceil), (s_1, \lceil s_3 \rceil, \lceil s_3 \rceil) \mid (s_1, s_2, s_3) \in \mathcal{R}\} \\
&\quad \cup \{(s_1, \lceil s_2 \rceil, \lceil s_2 \rceil) \mid (s_1, s_2) \in \mathcal{A}\}
\end{aligned}$$

Because we see $P$ as a programming language and $P^2$ as a logic for reasoning about programs in $P$, we adopt the following terminology and conventions. We use the metasyntactic variables $s, s_1, s_2, \ldots$ to range over sorts in $\mathcal{S}$ and $t, t_1, t_2, \ldots$ to range over sorts in $\mathcal{S}^2$. We call *type* a term inhabiting a first-level sort in some context (we write $\Gamma \vdash A : s$ for a type $A$), *programs* are inhabitants of types ($\Gamma \vdash A : B : s$ for a program $A$ of type $B$), *formulas* denote inhabitants of a lifted sort (written $\Gamma \vdash A : \lceil s \rceil$) and *proofs* are inhabitants of formulas ($\Gamma \vdash A : B : \lceil s \rceil$). We also say that types and programs are *first-level* terms, and formulas and proofs are *second-level* terms.

If $s$ is a sort of $P$, then $\lceil s \rceil$ is the sort of formulas expressing properties of types of sort $s$. For each rule $(s_1, s_2, s_3)$ in $\mathcal{R}$, $(\lceil s_1 \rceil, \lceil s_2 \rceil, \lceil s_3 \rceil)$ maps constructs of the programming language at the level of the logic, and $(s_1, \lceil s_3 \rceil, \lceil s_3 \rceil)$ allows to build the quantification of programs of sort $s_1$ in formulas of sort $\lceil s_3 \rceil$.

For each axiom $(s_1, s_2)$ in $\mathcal{A}$, we add the rule $(s_1, \lceil s_2 \rceil, \lceil s_2 \rceil)$ in order to build the type of predicates of sort $\lceil s_2 \rceil$ parameterized by programs of sort $s_1$.

*Example 3.* The PTS $\mathrm{F}^2$ has the following specification:

$$\begin{aligned}
\mathcal{S}^2_{\mathrm{F}} &= \{ \quad \star, \Box, \lceil\star\rceil, \lceil\Box\rceil \quad \} \\
\mathcal{A}^2_{\mathrm{F}} &= \{ \quad (\star, \Box), (\lceil\star\rceil, \lceil\Box\rceil) \quad \} \\
\mathcal{R}^2_{\mathrm{F}} &= \{ \ (\star,\star,\star), (\Box,\star,\star), (\lceil\star\rceil,\lceil\star\rceil,\lceil\star\rceil), (\lceil\Box\rceil,\lceil\star\rceil,\lceil\star\rceil) \\
&\qquad (\star,\lceil\Box\rceil,\lceil\Box\rceil), (\star,\lceil\star\rceil,\lceil\star\rceil), (\Box,\lceil\star\rceil,\lceil\star\rceil) \quad \}.
\end{aligned}$$

We extend our variable-naming convention to $\mathcal{V}_{\lceil\star\rceil}$ and $\mathcal{V}_{\lceil\Box\rceil}$ as follows: the variables $h$, $h_1$, $h_2$, ... range over $\mathcal{V}_{\lceil\star\rceil}$, and the variables $X$, $Y$, $Z$, ... range over $\mathcal{V}_{\lceil\Box\rceil}$. The logic $\mathrm{F}^2$ is a second-order logic with typed individuals (Wadler [29] gives another presentation of the same system). The sort $\star$ is the type of types and the only inhabitant of $\Box$, while $\lceil\star\rceil$ is the sort of propositions. $\lceil\Box\rceil$ is inhabited by the type of propositions ($\lceil\star\rceil$), the type of predicates ($\tau \to \lceil\star\rceil$), and in general the type of relations ($\tau_1 \to \cdots \to \tau_n \to \lceil\star\rceil$). The rules correspond to various type of quantifications as follows:

- $(\lceil\star\rceil, \lceil\star\rceil, \lceil\star\rceil)$ allows to build implication between formulas, written $P \to Q$.
- $(\star, \lceil\star\rceil, \lceil\star\rceil)$ allows to quantify over individuals (as in $\Pi x : \tau.P$).
- $(\Box, \lceil\star\rceil, \lceil\star\rceil)$ allows to quantify over types (as in $\Pi\alpha : \star.P$).
- $(\star, \lceil\Box\rceil, \lceil\Box\rceil)$ is used to build types of predicates depending on programs.

– $(\lceil\Box\rceil, \lceil\star\rceil, \lceil\star\rceil)$ allows to quantify over predicates (as in $\Pi X : \tau_1 \to \cdots \to \tau_n \to \lceil\star\rceil.P$).

In $F^2$, truth can be encoded by $\top \equiv \Pi X : \lceil\star\rceil.X \to X$ and is proved by Obvious $\equiv \lambda(X : \lceil\star\rceil)(h : X).h$. The formula $x =_\tau y \equiv \Pi X : \tau \to \lceil\star\rceil.X\,x \to X\,y$ define the Leibniz equality at type $\tau$. The term Refl $\equiv \lambda(\alpha : \star)(x : \alpha)(X : \alpha \to \lceil\star\rceil)(h : X\,x).h$ is a proof of the reflexivity of equality $\Pi(\alpha : \star)(x : \alpha).x =_\alpha x$. And the induction principle over Church numerals is a formula $N \equiv \lambda x : \mathrm{Nat}\,.\Pi X : \mathrm{Nat} \to \lceil\star\rceil.(\Pi y : \mathrm{Nat}\,.X\,y \to X\,(\mathrm{Succ}\ y)) \to X\,0 \to X\,x$.

### 3.1 Structure of $P^2$

Programs (or types) can never refer to proofs (nor formulas). In other words, a first-level term never contains a second-level term: it is typable in $P$. Formally:

**Theorem 1 (separation).** *For $s \in \mathcal{S}$, if $\Gamma \vdash A : B : s$ (resp. $\Gamma \vdash B : s$), then there exists a sub-context $\Gamma'$ of $\Gamma$ such that $\Gamma' \vdash_P A : B : s$ (resp. $\Gamma' \vdash_P B : s$).*

*Proof.* By induction on the structure of terms, and relying on the generation lemma [2, 5.2.13] and on the form of the rules in $\mathcal{R}^2$: assuming $(t_1, t_2, t_3) \in \mathcal{R}^2$ then $t_3 \in \mathcal{S} \Rightarrow (t_1 \in \mathcal{S} \wedge t_2 \in \mathcal{S})$ and $t_2 \in \mathcal{S} \Rightarrow (t_1 \in \mathcal{S} \wedge t_3 \in \mathcal{S})$.

*Lifting.* The major part of the paper is about transformations and relations between the first and the second level. The first and simplest transformation lifts terms from the first level to the second level, by substituting occurrences of a sort $s$ by $\lceil s\rceil$ everywhere (see Fig. 1). The function is defined only on first-level terms, and is extended to contexts in the obvious way. In addition to substituting sorts, lifting performs renaming of a variable $x$ in $\mathcal{V}_s$ to $\mathring{x}$ in $\mathcal{V}_{\lceil s\rceil}$.

$$
\begin{array}{ll}
\lceil x\rceil & = \mathring{x} \\
\lceil s\rceil & = \lceil s\rceil \\
\lceil \Pi x : A.\,B\rceil & = \Pi\mathring{x} : \lceil A\rceil.\,\lceil B\rceil \\
\lceil \lambda x : A.\,b\rceil & = \lambda\mathring{x} : \lceil A\rceil.\,\lceil b\rceil \\
\lceil A\,B\rceil & = \lceil A\rceil\,\lceil B\rceil \\
\hline
\lceil <>\rceil & = <> \\
\lceil \Gamma, x : A\rceil & = \lceil\Gamma\rceil, \mathring{x} : \lceil A\rceil
\end{array}
\qquad
\begin{array}{ll}
\lfloor x^{\lceil s\rceil}\rfloor & = \dot{x}^s \\
\lfloor\lceil s\rceil\rfloor & = s \\
\lfloor \Pi x^s : A.B\rfloor & = \lfloor B\rfloor \\
\lfloor \Pi x^{\lceil s\rceil} : A.B\rfloor & = \Pi\dot{x}^s : \lfloor A\rfloor.\lfloor B\rfloor \\
\lfloor \lambda x^s : A.B\rfloor & = \lfloor B\rfloor \\
\lfloor \lambda x^{\lceil s\rceil} : A.B\rfloor & = \lambda\dot{x}^s : \lfloor A\rfloor.\lfloor B\rfloor \\
\lfloor (A\,B)_s\rfloor & = \lfloor A\rfloor \\
\lfloor (A\,B)_{\lceil s\rceil}\rfloor & = \lfloor A\rfloor\,\lfloor B\rfloor \\
\hline
\lfloor <>\rfloor & = <> \\
\lfloor \Gamma, x^s : A\rfloor & = \lfloor\Gamma\rfloor \\
\lfloor \Gamma, x^{\lceil s\rceil} : A\rfloor & = \lfloor\Gamma\rfloor, \dot{x}^s : \lfloor A\rfloor.
\end{array}
$$

**Fig. 1.** lifting (left) and projection (right)

*Example 4.* In $F^2$, the lifting of inhabited types gives rise to logical tautologies. For instance, $\lceil \text{Unit} \rceil = \lceil \Pi\alpha : \star.\alpha \to \alpha \rceil = \Pi X : \lceil\star\rceil.X \to X = \top$, and $\lceil \text{Nat} \rceil = \Pi X : \lceil\star\rceil.(X \to X) \to (X \to X)$.

**Lemma 1 (lifting preserves typing).**
$\Gamma \vdash A : B : s \Rightarrow \lceil\Gamma\rceil \vdash \lceil A\rceil : \lceil B\rceil : \lceil s\rceil$

*Proof.* A consequence of $P^2$ containing a copy of $P$ with $s$ mapped to $\lceil s\rceil$.

**Lemma 2 (lifting preserves $\beta$-reduction).**
$A \longrightarrow_\beta B \Rightarrow \lceil A\rceil \longrightarrow_\beta \lceil B\rceil$

*Proof.* By induction on the structure of $A$.

*Projection.* We define a projection from second-level terms into first-level terms, which maps second-level constructs into first-level constructs. The first-level subterms are removed, as well as the interactions between the first and second levels. The reader may worry that some variable bindings are removed, potentially leaving some occurrences unbound in the body of the transformed term. However, these variables are first level, and hence their occurrences are removed too (by the application case).

The function is defined only on second-level terms, and behaves differently when facing pure second level or interaction terms. In order to distinguish these cases, the projection takes sort-annotated terms as input. Like the lifting, the projection performs renaming of each variable $x$ in $\mathcal{V}_{\lceil s\rceil}$ to $\dot{x}$ in $\mathcal{V}_s$. We postulate that this renaming cancels that of the lifting: we have $\dot{\dot{x}} = x$.

*Example 5 (projections in $F^2$).*

$$\lfloor\top\rfloor = \text{Unit} \quad \lfloor\text{Obvious}\rfloor = \text{Id} \quad \lfloor\Pi(\alpha : \star)(x : \alpha).x =_\alpha x\rfloor = \text{Unit} \quad \lfloor N\,t\rfloor = \text{Nat}$$

**Lemma 3 (projection is the left inverse of lifting).** $\lfloor\lceil A\rceil\rfloor = A$

*Proof.* By induction on the structure of $A$.

**Lemma 4 (projection preserves typing).**
$\Gamma \vdash A : B : \lceil s\rceil \Rightarrow \lfloor\Gamma\rfloor \vdash \lfloor A\rfloor : \lfloor B\rfloor : s$

*Proof.* By induction on the derivation $\Gamma \vdash A : B$.

In contrast to lifting, which keeps a term intact, projection may remove parts of a term, in particular abstractions at the interaction level. Therefore, $\beta$-reduction steps may be removed by projection.

**Lemma 5 (projection preserves or removes $\beta$-reduction).**
*If $A \longrightarrow_\beta B$, then either $\lfloor A\rfloor \longrightarrow_\beta \lfloor B\rfloor$ or $\lfloor A\rfloor = \lfloor B\rfloor$.*

### 3.2 Strong normalization

**Theorem 2 (normalization).** *If $P$ is strongly normalizing, so is $P^2$.*

*Proof.* The proof is based on the observation that, if a term $A$ is typable in $P^2$ and not normalizable, then at least either:

- one of the first-level subterms of $A$ is not normalizable, or
- the first-level term $\lfloor A \rfloor$ is not normalizable.

And yet $\lfloor A \rfloor$ and the first-level subterms are typable in $P$ (Thm. 1) which would contradict the strong normalization of $P$.

### 3.3 Parametricity

In this section we develop Reynolds-style [23] parametricity for $P$, in $P^2$. While parametricity theory is often defined for binary relations, we abstract from the arity and develop the theory for an arbitrary arity $n$, though we omit the index $n$ when the arity of relations plays no role or is obvious from the context.

The definition of parametricity is done in two parts: first we define what it means for a $n$-tuple of programs $\overline{z}$ to satisfy the relation generated by a type $T$ ($\overline{z} \in [\![T]\!]_n$); then we define the translation from a program $z$ of type $T$ to a proof $[\![z]\!]_n$ that a tuple $\overline{z}$ satisfies the relation.

The definition below uses $n+1$ renamings: one of them ($\mathring{\cdot}$) coincides with that of lifting, and the others map $x$ respectively to $x_1, \ldots, x_n$. The tuple $\overline{A}$ denotes $n$ terms $A_i$, where $A_i$ is the term $A$ where each free variable $x$ is replaced by a fresh variable $x_i$.

**Definition 2 (parametricity).**

$$
\begin{array}{ll}
\overline{C} \in [\![s]\!] & = \overline{C} \to \lceil s \rceil \\
\overline{C} \in [\![\Pi x : A.\, B]\!] & = \Pi \overline{x : A}.\, \Pi \mathring{x} : \overline{x} \in [\![A]\!].\, \overline{C\, x} \in [\![B]\!] \\
\overline{C} \in [\![T]\!] & = [\![T]\!]\, \overline{C} \text{ otherwise} \\
\hline
[\![x]\!] & = \mathring{x} \\
[\![\lambda x : A.\, B]\!] & = \lambda \overline{x : A}.\, \lambda \mathring{x} : \overline{x} \in [\![A]\!].\, [\![B]\!] \\
[\![A\, B]\!] & = [\![A]\!]\, \overline{B}\, [\![B]\!] \\
[\![T]\!] & = \lambda \overline{z : T}.\, \overline{z} \in [\![T]\!] \text{ otherwise} \\
\hline
[\![<>]\!] & = <> \\
[\![\Gamma, x : A]\!] & = [\![\Gamma]\!], \overline{x : A}, \mathring{x} : \overline{x} \in [\![A]\!]
\end{array}
$$

Because the syntax of values and types are unified in a PTS, each of the definitions $\cdot \in [\![\cdot]\!]$ and $[\![\cdot]\!]$ must handle all constructions. In both cases, this is done by using a catch-all case (the last line) that refers to the other part of the definition.

**Remark 1** *For arity 0, parametricity specializes to lifting* ($[\![A]\!]_0 = \lceil A \rceil$).

*Example 6.* For instance, in $\mathrm{F}^2$, we have

$$(f,g) \in [\![\Pi(\alpha:\star).\alpha \to \Pi(\beta:\star).\beta \to \alpha]\!] \equiv \Pi(\alpha_1\,\alpha_2:\star)(X:\alpha_1 \to \alpha_2 \to \lceil\star\rceil)$$
$$(\beta_1\,\beta_2:\star)(Y:\beta_1 \to \beta_2 \to \lceil\star\rceil)(x_1:\alpha_1)(x_2:\alpha_2).X\,x_1\,x_2 \to$$
$$\Pi(y_1:\beta_1)(y_2:\beta_2).Y\,y_1\,y_2 \to X\,(f\,\alpha_1\,\beta_1\,x_1\,y_1)\,(g\,\alpha_2\,\beta_2\,x_2\,y_2).$$

**Theorem 3 (abstraction).** *If* $\Gamma \vdash A : B : s$, *then* $[\![\Gamma]\!] \vdash [\![A]\!] : (\overline{A} \in [\![B]\!]) : \lceil s\rceil$

*Proof.* The result is a consequence of the following lemmas which are proved by simultaneous induction on the typing derivation:

- $A \longrightarrow_\beta B \Rightarrow [\![A]\!] \longrightarrow_\beta^* [\![B]\!]$
- $\Gamma \vdash A : B \Rightarrow [\![\Gamma]\!] \vdash \overline{A : B}$
- $\Gamma \vdash B : s \Rightarrow [\![\Gamma]\!], \overline{z : B} \vdash \overline{z} \in [\![B]\!] : \lceil s\rceil$
- $\Gamma \vdash A : B : s \Rightarrow [\![\Gamma]\!] \vdash [\![A]\!] : \overline{A} \in [\![B]\!]$

A direct reading of the above result is as a typing judgement about translated terms (as for lemmas 1 and 4): if $A$ has type $B$, then $[\![A]\!]$ has type $\overline{A} \in [\![B]\!]$. However, it can also be understood as an abstraction theorem for system $P$: if a program $A$ has type $B$ in $\Gamma$, then various interpretations of $A$ ($\overline{A}$) in related environments ($[\![\Gamma]\!]$) are related, by the formula $\overline{A} \in [\![B]\!]$.

The system $P^2$ is a natural setting to express parametricity conditions for $P$. Indeed, the interaction rules of the form $(s, \lceil s'\rceil, \lceil s'\rceil)$ coming from axioms in $P$ are needed to make the sort case valid; and the interaction rules $(s_1, \lceil s_3\rceil, \lceil s_3\rceil)$ are needed for the quantification over individuals in the product case.

### 3.4 Realizability

We develop here a Krivine-style [13] internalized realizability theory. Realizability bears similarities both to the projection and the parametricity transformations defined above.

**Definition 3 (realizability).**

$$\begin{array}{ll}
C \Vdash \lceil s\rceil & = C \to \lceil s\rceil \\
C \Vdash \Pi x^s : A.B & = \Pi x^s : A.C \Vdash B \\
C \Vdash \Pi x^{\lceil s\rceil} : A.B & = \Pi(\dot{x}^s : \lfloor A\rfloor)(x^{\lceil s\rceil} : \dot{x} \Vdash A).(C\,\dot{x}) \Vdash B \\
C \Vdash F & = \langle F\rangle\, C \text{ otherwise} \\
\hline
\langle x^{\lceil s\rceil}\rangle & = x^{\lceil s\rceil} \\
\langle \lambda x^s : A.B\rangle & = \lambda x^s : A.\langle B\rangle \\
\langle \lambda x^{\lceil s\rceil} : A.B\rangle & = \lambda(\dot{x}^s : \lfloor A\rfloor)(x^{\lceil s\rceil} : \dot{x} \Vdash A).\langle B\rangle \\
\langle (A\,B)_s\rangle & = (\langle A\rangle\, B)_s \\
\langle (A\,B)_{\lceil s\rceil}\rangle & = ((\langle A\rangle\, \lfloor B\rfloor)_s\, \langle B\rangle)_{\lceil s\rceil} \\
\langle T\rangle & = \lambda z^s : \lfloor T\rfloor.\, z \Vdash T \text{ otherwise} \\
\hline
\langle \Gamma, x^s : A\rangle & = \langle \Gamma\rangle, x^s : A \\
\langle \Gamma, x^{\lceil s\rceil} : A\rangle & = \langle \Gamma\rangle, \dot{x}^s : \lfloor A\rfloor, x^{\lceil s\rceil} : \dot{x} \Vdash A
\end{array}$$

Like the projection, the realizability transformation is applied on second-level constructs, and behaves differently depending on whether it treats interaction constructs or pure second-level ones. It is also similar to parametricity, as it is defined in two parts. In the first part we define what it means for a program $C$ to realize a formula $F$ ($C \Vdash F$); then we define the translation from a proof $p$ to a proof $\langle p \rangle$ that the program $\lfloor p \rfloor$ satisfies the realizability predicate.

**Theorem 4 (adequacy).** *If* $\Gamma \vdash A : B : \lceil s \rceil$*, then* $\langle \Gamma \rangle \vdash \langle A \rangle : \lfloor A \rfloor \Vdash B : \lceil s \rceil$

*Proof (idea).* Similar in structure to the proof of the abstraction theorem.

*Example 7.* In $F^2$, the formula $y \Vdash N\, x$ unfolds to

$$\Pi(\alpha : \star)(X : \mathrm{Nat} \to \alpha \to \star)(f : \alpha \to \alpha).$$
$$(\Pi(n : \mathrm{Nat})(y : \alpha).X\, n\, y \to X\, (\mathrm{Succ}\ n)\, (f\, y)) \to \Pi(z : \alpha).X\, 0\, y \to X\, x\, (y\, \alpha\, f\, z)$$

In $F^2$ this formula may be used to prove a representation theorem. We can prove that $\Sigma \vdash \Pi x\, y : \mathrm{Nat}\,.y \Vdash N\, x \Leftrightarrow x =_{\mathrm{Nat}} y \wedge N\, x$ where $\Sigma$ is a set of extensionality axioms ($\wedge$ and $\Leftrightarrow$ are defined by usual second-order encodings). Let $\pi$ be a proof of $\Pi x : \mathrm{Nat}\,.N\, x \to N\, (f\, x)$ then $\vdash \lfloor \pi \rfloor : \mathrm{Nat} \to \mathrm{Nat}$ and $\vdash \langle \pi \rangle : \lfloor \pi \rfloor \Vdash \Pi x : \mathrm{Nat}\,.N\, x \to N\, (f\, x)$ which unfold to $\vdash \langle \pi \rangle : \Pi x\, y : \mathrm{Nat}\,.y \Vdash Nx \to \lfloor \pi \rfloor y \Vdash N(fx)$. Let $m$ be a term in closed normal form such that $\vdash m : \mathrm{Nat}$, we can prove $N\, m$ and therefore $m \Vdash N\, m$. We now have a proof (under $\Sigma$) that $\lfloor \pi \rfloor\, m \Vdash N\, (f\, m)$ and we conclude that $\lfloor \pi \rfloor\, m =_{\mathrm{Nat}} f\, m$. We have proved that the projection of any proof of $\Pi x : \mathrm{Nat}\,.N\, x \to N\, (f\, x)$ can be proved extensionally equal to $f$. See [29, 13, 15] for more details.

## 4 The Third Level

By casting both parametricity and realizability in the mold of PTSs, we are able to discern the connections between them. The connections already surface in the previous sections: the definitions of parametricity and realizability bear some resemblance, and the adequacy and abstraction theorems appear suspiciously similar. In this section we precisely spell out the connection: realizability and parametricity can be defined in terms of each other.

**Theorem 5 (realizability increases arity of parametricity).** *For any tuple terms* $(B, \overline{C})$*,*
$\big(B, \overline{C}\big) \in [\![A]\!]_{n+1} = B \Vdash \big(\overline{C} \in [\![A]\!]_n\big)$ *and* $[\![A]\!]_{n+1} = \langle [\![A]\!]_n \rangle$

*Proof.* By induction on the structure of $A$.

As a corollary, $n$-ary parametricity is the composition of lifting and $n$ realizability steps:

**Corollary 1 (from realizability to parametricity).**
$\overline{C} \in [\![A]\!]_n = C_1 \Vdash C_2 \Vdash \cdots \Vdash C_n \Vdash \lceil A \rceil$ *and* $[\![A]\!]_n = \langle \cdots \langle \lceil A \rceil \rangle \cdots \rangle$
*(assuming right-associativity of* $\Vdash$*)*

*Proof.* By induction on $n$. The base case uses $[\![A]\!]_0 = \lceil A \rceil$.

One may also wonder about the converse: is it possible to define realizability in terms of parametricity? We can answer by the affirmative, but we need a bigger system to do so. Indeed, we need to extend $[\![\cdot]\!]$ to work on second-level terms, and that is possible only if a third level is present in the system. To do so, we can iterate the construction used in Sec. 3 to build a logic for an arbitrary PTS.

**Definition 4 (third-level system).** *Given a PTS $P = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, we define $P^3 = (P^2)^2$, where the sort-lifting $\lceil\cdot\rceil$ used by both instances of the $\cdot^2$ transformation are the same.*

**Remark 2** *Because the sort-lifting used by both instances of the $\cdot^2$ transformation are the same, $P^3$ contains only three copies of $P$ (not four). In fact $P^3 = (\mathcal{S}^3, \mathcal{A}^3, \mathcal{R}^3)$, where*

$$\begin{aligned}
\mathcal{S}^3 &= \mathcal{S} \ \cup \lceil \mathcal{S} \rceil \cup \lceil \lceil \mathcal{S} \rceil \rceil \\
\mathcal{A}^3 &= \mathcal{A} \ \cup \lceil \mathcal{A} \rceil \cup \lceil \lceil \mathcal{A} \rceil \rceil \\
\mathcal{R}^3 &= \mathcal{R} \ \cup \lceil \mathcal{R} \rceil \cup \lceil \lceil \mathcal{R} \rceil \rceil \\
&\qquad \cup \{(s_1, \lceil s_3 \rceil, \lceil s_3 \rceil), (\lceil s_1 \rceil, \lceil \lceil s_3 \rceil \rceil, \lceil \lceil s_3 \rceil \rceil) \mid (s_1, s_2, s_3) \in \mathcal{R}\} \\
&\qquad \cup \{(s_1, \lceil s_2 \rceil, \lceil s_2 \rceil), (\lceil s_1 \rceil, \lceil \lceil s_2 \rceil \rceil, \lceil \lceil s_2 \rceil \rceil) \mid (s_1, s_2) \in \mathcal{A}\}
\end{aligned}$$

The $[\![\cdot]\!]$ transformation is extended second-level constructs in $P^2$, mapping them to third-level ones in $P^3$. The $\lfloor\cdot\rfloor$ transformation is be similarly extended, to map the third level constructs to the second level, in addition of mapping the second to the first one (only the first level is removed).

Given these extensions, we obtain that realizability is the composition of parametricity and projection.

**Lemma 6.** *If $A$ is a first-level term, then*
$A = \lfloor C \in [\![A]\!]_1 \rfloor$ *and* $A = \lfloor [\![A]\!]_1 \rfloor$

*Proof.* By induction on the structure of $A$, using separation (Thm. 1).

**Theorem 6 (from parametricity to realizability).** *If $A$ is a second-level term, then*
$C \Vdash A = \lfloor \lceil C \rceil \in [\![A]\!]_1 \rfloor$ *and* $\langle A \rangle = \lfloor [\![A]\!]_1 \rfloor$

*Proof.* By induction on the structure of $A$, using the above lemma.

## 5 Extensions

### 5.1 Inductive definitions

Even though our development assumes pure type systems, with only axioms of the form $(s_1, s_2)$, the theory easily accommodates the addition of inductive definitions.

For parametricity, the way to extend the theory is exposed by Bernardy et al. [5]. In brief: if for every inductive definition in the programming language there is a corresponding inductive definition in the logic, then the abstraction theorem holds. For instance, to the indexed inductive definition $I$ corresponds $[\![I]\!]$, as defined below. (We write only one constructor $c_p$ for concision, but the result applies to any number of constructors.)

$$\begin{array}{l}\textbf{data}\ I : \Pi(x_1 : A_1) \cdots (x_n : A_n).s\ \textbf{where}\\ \quad c_p : \Pi(x_1 : B_{p,1}) \cdots (x_{n_1} : B_{p,n_1}).I\, a_{p,1} \cdots a_{p,n}\end{array}$$

$$\begin{array}{l}\textbf{data}\ [\![I]\!] : \overline{I} \in [\![\Pi(x_1 : A_1) \cdots (x_n : A_n).s]\!]\ \textbf{where}\\ \quad [\![c_p]\!] : \overline{c_p} \in [\![\Pi(x_1 : B_{p,1}) \cdots (x_{n_1} : B_{p,n_1}).I\, a_{p,1} \cdots a_{p,n}]\!]\end{array}$$

The result can be transported to realizability by following the correspondence developed in the previous section. By taking the composition of $[\![\cdot]\!]$ and $\lfloor\cdot\rfloor$ for the definition of realizability, and knowing how to extend $[\![\cdot]\!]$ to inductive types, it suffices to extend $\lfloor\cdot\rfloor$ as well (respecting typing: Lem. 4). The corresponding extension to realizability is compatible with the definition for a pure system (by Thm. 6). Adequacy is proved by the composition of abstraction and Lem. 4. The definition of $\lfloor\cdot\rfloor$ is straightforward: each component of the definition must be transformed by $\lfloor\cdot\rfloor$. That is, for any inductive definition in the logic, there must be another inductive definition in the programming language that realizes it. For instance, given the definition $I$ given below, one must also have $\lfloor I\rfloor$. $\langle I\rangle$ is then given by $\langle I\rangle = \lfloor[\![I]\!]\rfloor$, but can also be expanded as below.

$$\begin{array}{l}\textbf{data}\ I : \Pi(x_1 : A_1) \cdots (x_n : A_n).\lceil s\rceil\ \textbf{where}\\ \quad c_p : \Pi(x_1 : B_{p,1}) \cdots (x_{n_1} : B_{p,n_1}).I\, a_{p,1} \cdots a_{p,n}\end{array}$$

$$\begin{array}{l}\textbf{data}\ \lfloor I\rfloor : \lfloor\Pi(x_1 : A_1) \cdots (x_n : A_n).\lceil s\rceil\rfloor\ \textbf{where}\\ \quad \lfloor c_p\rfloor : \lfloor\Pi(x_1 : B_{p,1}) \cdots (x_{n_1} : B_{p,n_1}).I\, a_{p,1} \cdots a_{p,n}\rfloor\end{array}$$

$$\begin{array}{l}\textbf{data}\ \langle I\rangle : \lfloor I\rfloor \Vdash (\Pi(x_1 : A_1) \cdots (x_n : A_n).\lceil s\rceil)\ \textbf{where}\\ \quad \langle c_p\rangle : \lfloor c_p\rfloor \Vdash (\Pi(x_1 : B_{p,1}) \cdots (x_{n_1} : B_{p,n_1}).I\, a_{p,1} \cdots a_{p,n})\end{array}$$

We can use inductive types to encode usual logical connectives, and derive realizability for them.

*Example 8 (conjunction).* The encoding of conjunction in a sort $\lceil s\rceil$ is as follows:

$$\begin{array}{l}\textbf{data}\ \_\wedge\_ : \lceil s\rceil \to \lceil s\rceil \to \lceil s\rceil\ \textbf{where}\\ \quad \mathsf{conj} : \Pi\, P\, Q : \lceil s\rceil.P \to Q \to P \wedge Q\end{array}$$

If we apply the projection operator to the conjunction we obtain the type of its realizers: the cartesian product in $s$.

$$\begin{array}{l}\textbf{data}\ \_\times\_ : s \to s \to s\ \textbf{where}\\ \quad (\_,\_) : \Pi\, \alpha\, \beta : s.\alpha \to \beta \to \alpha \times \beta\end{array}$$

Now we can apply our realizability construction to obtain a predicate telling what it means to realize a conjunction.

$$\begin{aligned}\textbf{data}\ \langle\wedge\rangle : \Pi(\alpha : s).(\alpha \to \lceil s\rceil) \to \\ \Pi(\beta : s).(\beta \to \lceil s\rceil) \to \\ \alpha \times \beta \to s\ \textbf{where} \\ \langle\mathsf{conj}\rangle : \Pi(\alpha : s)(P : \alpha \to \lceil s\rceil) \\ (\beta : s)(Q : \beta \to \lceil s\rceil)(x : \alpha)(y : \beta). \\ P\,x \to Q\,y \to \langle\wedge\rangle\,\alpha\,P\,\beta\,Q\,(x, y)\end{aligned}$$

By definition, $t \Vdash P \wedge Q$ means $\langle\wedge\rangle\,\lfloor P\rfloor\,\langle P\rangle\,\lfloor Q\rfloor\,\langle Q\rangle\,t$. We have

$$t \Vdash P \wedge Q \Leftrightarrow (\pi_1\,t) \Vdash P \wedge (\pi_2\,t) \Vdash Q$$

where $\pi_1$ and $\pi_2$ are projections upon Cartesian product.

We could build the realizers of other logical constructs in the same way: we would obtain a sum-type for the disjunction, an empty type for falsity, and a box type for the existential quantifier. All the following properties (corresponding to the usual definition of the realizability predicate) would then be satisfied:

- $t \Vdash P \vee Q \Leftrightarrow \textbf{case}\,t\,\textbf{with}\,\iota_1\,x \to x \Vdash P \,|\, \iota_2\,x \to x \Vdash Q$.
- $t \Vdash \bot \Leftrightarrow \bot$ and $t \Vdash \neg P \Leftrightarrow \Pi(x : \lfloor P\rfloor).\neg(x \Vdash P)$
- $t \Vdash \exists x : A.P \Leftrightarrow \exists x : A.(unbox\,t) \Vdash P$

where $\textbf{case}\ldots\textbf{with}\ldots$ is the destruction of the sum type, and *unbox* is the destructor of the box type.

### 5.2 Program extraction and computational irrelevance

An application of the theory developed so far is the extraction of programs from proofs. Indeed, an implication of the adequacy theorem is that the program $\lfloor A\rfloor$, obtained by projection of a proof $A$ of a formula $B$, corresponds to an implementation of $B$, viewed as a specification. One says that $\lfloor\cdot\rfloor$ implements program extraction.

For example, applying extraction to an expression involving vectors ($Vec : (A : \lceil\star\rceil) \to Nat \to \lceil\star\rceil$) yields a program over lists. This means that programs can be justified in the rich system $P^2$, and realized in the simple system $P$. Practical benefits include a reduction in memory usage: Brady et al. [6] measure an 80% reduction using a technique with similar goals.

While $P^2$ is already much more expressive than $P$, it is possible to further increase the expressive power of the system, while retaining the adequacy theorem, by allowing quantification of first-level terms by second-level terms.

**Definition 5** ($P^{2'}$). *Let $P = (\mathcal{S}, \mathcal{A}, \mathcal{R})$, we define $P^{2'} = (\mathcal{S}^{2'}, \mathcal{A}^{2'}, \mathcal{R}^{2'})$*

$$\begin{aligned}\mathcal{S}^{2'} &= \mathcal{S} \cup \{\lceil s\rceil \mid s \in \mathcal{S}\} \\ \mathcal{A}^{2'} &= \mathcal{A} \cup \{(\lceil s_1\rceil, \lceil s_2\rceil) \mid (s_1, s_2) \in \mathcal{A}\} \\ \mathcal{R}^{2'} &= \mathcal{R} \cup \{(\lceil s_1\rceil, \lceil s_2\rceil, \lceil s_3\rceil), (s_1, \lceil s_3\rceil, \lceil s_3\rceil), (\lceil s_1\rceil, s_3, s_3) \mid (s_1, s_2, s_3) \in \mathcal{R}\} \\ &\quad \cup \{(s_1, \lceil s_2\rceil, \lceil s_2\rceil), (\lceil s_1\rceil, s_2, s_2) \mid (s_1, s_2) \in \mathcal{A}\}\end{aligned}$$

The result is a symmetric system, with two copies of $P$. Within either side of the system, one can reason about terms belonging to the other side. Furthermore, either side has a computational interpretation where the terms of the other side are irrelevant. For the second level, this interpretation is given by $\lfloor \cdot \rfloor$.

Even though there is no separation between first and second level in $P^{2'}$, adequacy is preserved: the addition of rules of the form $(\lceil s_1 \rceil, s_2, s_3)$ only adds first level terms, which are removed by projection.

## 6 Related work and Conclusion

Our work is based on Krivine-style realizability [13] and Reynolds-style parametricity [23], which have both spawned large bodies of work.

*Logics for parametricity.* Study of parametricity is typically semantic, including the seminal work of Reynolds [23]. There, the concern is to capture the polymorphic character of $\lambda$-calculi (typically System F) in a model.

Mairson [16] pioneered a different angle of study, where the expressions of the programming language are (syntactically) translated to formulas describing the program. That style has then been picked by various authors before us, including Abadi et al. [1], Plotkin and Abadi [22], Bernardy et al. [5].

Plotkin and Abadi [22] introduce a logic for parametricity, similar to $\mathrm{F}^2$, but with several additions. The most important addition is that of a parametricity axiom. This addition allows to prove the initiality of Church-style encoding of types.

Wadler [29] defines essentially the same concepts as us, but in the special case of System F. He points out that realizability transforms unary parametricity into binary parametricity, but does not generalize to arbitrary arity. We find the $n = 0$ case particularly interesting, as it shows that parametricity can be constructed purely in terms of realizability and a trivial lifting to the second level. We additionally show that realizability can be obtained by composing realizability and projection, while Wadler only defines the realizability transformation as a separate construct.

The parametricity transformation and the abstraction theorem that we expose here are a modified version of [5]. The added benefits of the present version is that we handle finite PTSs, and we allow the target system to be different from the source. The possible separation of source and targets is already implicit in that paper though. The way we handle finite PTSs is by separating the treatment of types and programs.

*Realizability.* Our realizability construction can be understood as an extension of the work of Paulin-Mohring [20], providing a realizability interpretation for a variant of the Calculus of Construction. Paulin-Mohring [20] splits CC in two levels; one where $\star$ becomes *Prop* and one where it becomes *Spec*. Perhaps counter-intuitively, *Prop* lies in what we call the first level; and *Spec* lies in the second level. Indeed, *Prop* is removed from the realizers. The system is

symmetric, as the one we expose in Sec. 5.2, in the sense that there is both a rule $(\mathit{Spec}, \mathit{Prop}, \mathit{Prop})$ and $(\mathit{Prop}, \mathit{Spec}, \mathit{Spec})$. In order to see that Paulin-Mohring's construction as a special case of ours, it is necessary to recognize a number of small differences:

1. The sort *Spec* is transformed into *Prop* in the realizability transformation, whereas we would keep *Spec*.
2. The sorts of the original system use a different set of names (*Data* and *Order*). Therefore the sort *Spec* is transformed into *Data* in the projection, whereas we would use *Prop*.
3. The types of *Spec* and *Prop* inhabit the same sort, namely *Type*.
4. There is elimination from *Spec* to *Prop*, breaking the computational irrelevance in that direction.

The first two differences are essentially renamings, and thus unimportant.

*Connections.* We are unaware of previous work showing the connection between realizability and parametricity, at least as clearly as we do. Wadler [29] comes close, giving a version of Thm. 5 specialized to System F, but not its converse, Thm. 6. Mairson [16] mentions that his work on parametricity is directly inspired by that of Leivant [15] on realizability, but does not formalize the parallels.

*Conclusion.* We have given an account of parametricity and realizability in the framework of PTSs. The result is very concise: the definitions occupy only a dozen of lines. By recognizing the parallels between the two, we are able to further shrink the number of primitive concepts.

Our work points the way towards the transportation of every parametricity theory into a corresponding realizability theory, and *vice versa*.

*Acknowledgments.* Thanks to Andreas Abel, Thorsten Altenkirch, Thierry Coquand, Peter Dybjer and Guilhem Moulin for helpful comments and discussions.

## Bibliography

[1] M. Abadi, L. Cardelli, and P. Curien. Formal parametric polymorphism. In *Proc. of POPL'93*, pages 157–170. ACM, 1993.

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

[3] S. Berardi. *Type Dependence and Constructive Mathematics.* PhD thesis, Dipartimento di Informatica, Torino, 1989.

[4] J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In A. Gordon, editor, *Proc. of ESOP 2010*, volume 6012 of *LNCS*, pages 125–144. Springer, 2010.

[5] J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *Proc. of ICFP 2010*, pages 345–356. ACM, 2010.

[6] E. Brady, C. McBride, and J. McKinna. Inductive families need not store their indices. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *LNCS*, pages 115–129. Springer Berlin / Heidelberg, 2004.
[7] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proc. of FPCA*, pages 223–232. ACM, 1993.
[8] J.-Y. Girard. *Interprétation fonctionnelle et elimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'état, Université de Paris 7, 1972.
[9] R. Harrop. On disjunctions and existential statements in intuitionistic systems of logic. *Mathematische Annalen*, 132(4):347–361, 1956.
[10] S. C. Kleene. On the interpretation of intuitionistic number theory. *J. of Symbolic Logic*, 10(4):109–124, 1945.
[11] S. C. Kleene. *Introduction to metamathematics*. Wolters-Noordhoff, 1971.
[12] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In A. Heyting, editor, *Constructivity in mathematics*, pages 101–128, 1959.
[13] J.-L. Krivine. *Lambda-calcul types et modèles*. Masson, 1990.
[14] J.-L. Krivine and M. Parigot. Programming with proofs. *J. Inf. Process. Cybern.*, 26(3):149–167, 1990.
[15] D. Leivant. Contracting proofs to programs. In *Logic and Comp. Sci.*, pages 279–327, 1990.
[16] H. Mairson. Outline of a proof theory of parametricity. In *Proc. of FPCA 1991*, volume 523 of *LNCS*, pages 313–327. Springer-Verlag, 1991.
[17] C. McBride and J. McKinna. The view from the left. *J. Funct. Program.*, 14(01):69–111, 2004.
[18] R. Milner. Logic for Computable Functions: description of a machine implementation. *Artificial Intelligence*, 1972.
[19] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers Tekniska Högskola, 2007.
[20] C. Paulin-Mohring. Extracting F$\omega$'s programs from proofs in the calculus of constructions. In *POPL'89*, pages 89–104. ACM, 1989.
[21] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. PhD thesis, Université Paris 7, 1989.
[22] G. Plotkin and M. Abadi. A logic for parametric polymorphism. In *LNCS*, volume 664, page 361–375. Springer-Verlag, 1993.
[23] J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information processing*, 83(1):513–523, 1983.
[24] J. Staples. Combinator realizability of constructive finite type analysis. *Cambridge Summer School in Mathematical Logic*, pages 253–273, 1973.
[25] The Coq development team. The Coq proof assistant, 2010.
[26] A. Troelstra. *Handbook of proof theory*, chapter Realizability. Elsevier, 1998.
[27] J. Van Oosten. Realizability: a historical essay. *Mathematical Structures in Comp. Sci.*, 12(03):239–263, 2002.
[28] P. Wadler. Theorems for free! In *Proc. of FPCA 1989*, pages 347–359. ACM, 1989.
[29] P. Wadler. The Girard–Reynolds isomorphism. *Theor. Comp. Sci.*, 375 (1–3):201–226, 2007.