

1 Gardening with the Pythia

2 A model of continuity in a dependent setting

3 Martin Baillon ✉

4 INRIA and LS2N, France

5 Assia Mahboubi ✉

6 INRIA and LS2N, France

7 Pierre-Marie Pédrot ✉

8 INRIA and LS2N, France

9 — Abstract —

10 We generalize to a rich dependent type theory a proof originally developed by Escardó that all
11 System T functionals are continuous. It relies on the definition of a syntactic model of Baclofen Type
12 Theory, a type theory where dependent elimination must be strict, into the Calculus of Inductive
13 Constructions. The model is given by three translations: the *axiom translation*, that adds an oracle
14 to the context; the *branching translation*, based on the dialogue monad, turning every type into a
15 tree; and finally, a layer of *algebraic binary parametricity*, binding together the two translations. In
16 the resulting type theory, every function $f : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ is externally continuous.

17 **2012 ACM Subject Classification** Theory of computation \rightarrow Type theory

18 **Keywords and phrases** Type theory, continuity, syntactic model

19 **Digital Object Identifier** 10.4230/LIPIcs.CSL.2022.13

20 Introduction

21 A folklore result from computability theory is that any computable function must be
22 continuous [4]. A more operational way to phrase this property is that a function can only
23 inspect a finite amount of its argument to produce a finite amount of output. There are
24 many ways to prove, or even merely state, this theorem, since it depends in particular on
25 how computable functions are represented [18, 37, 21]. Assuming we pick the λ -calculus
26 as our favourite computational system, a modern straightforward proof would boil down
27 to building a semantic model, typically some flavour of complete partial orders (cpos). By
28 construction, cpos are a specific kind of topological spaces, and all functions are interpreted
29 as continuous functions in the model. For some types simple enough, cpo-continuity implies
30 continuity in the traditional sense, thus proving the claim.

31 Instead of going down the semantic route, Escardó developed an alternative syntactic
32 technique called *effectful forcing* [11] to prove the continuity of all functionals $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$
33 definable in System T. While semantic models such as cpos are defined inside a non-
34 computational metatheory, Escardó’s technique amounts to building a model of System T
35 inside the dependent type theory MLTT, which is intrinsically a programming language with
36 a built-in notion of computation. The *effectful* epithet is justified by the fact that the model
37 construction extends System T with two different kinds of side-effects, and constrains those
38 two extensions by a logical relation.

39 A clear advantage of this approach is that there is a simple computational explanation for
40 why continuity holds in terms of elementary side-effects, which is not immediately apparent
41 in cpos. This computational aspect is reminiscent of a similar realizability model of NuPRL
42 internalizing continuity with a system of fresh exceptions [30]. But contrarily to the latter,
43 the purely syntactic nature of Escardó’s argument can actually be leveraged to interpret



© Martin Baillon, Assia Mahboubi and Pierre-Marie Pédrot;
licensed under Creative Commons License CC-BY 4.0

30th EACSL Annual Conference on Computer Science Logic (CSL 2022).

Editors: Florin Manea and Alex Simpson; Article No. 13; pp. 13:1–13:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

44 much richer languages than System T while preserving desirable properties that would be
45 lost with a semantic realizability model, such as decidability of type-checking.

46 Indeed, it happens that this technique can be formulated pretty much straightforwardly
47 as a syntactic model [13, 33]. From this initial observation, we show in this paper how it can
48 be generalized to a rich dependent type theory similar to MLTT, notably featuring universes
49 and a form of large dependent elimination. Unfortunately, since Escardó’s model introduces
50 observable side-effects in the sense of [25], the type theory resulting from our generalization
51 needs to be slightly weakened down or would otherwise be inconsistent. This effectively
52 means we provide a model of Baclofen Type Theory (BTT) rather than MLTT. The main
53 difference between those two theories lies in the typing rule for dependent elimination [28]. In
54 MLTT, the predicate of a dependent elimination is arbitrary, while it must be computationally
55 strict in BTT. This is discussed in detail in Section 1.2.

56 In the end we recover the continuity result of Escardó applied to BTT rather than
57 System T. That is, from any $\vdash_{\text{BTT}} f : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ we get a proof that it is continuous.

58 Plan of the paper.

59 Section 1 exposes preliminaries that are needed to understand this paper. In Section 2, we
60 describe a particular structure known as dialogue trees that will be critical for the rest of the
61 paper. Section 3 is dedicated to the model construction *per se*. Section 4 provides the proof
62 that all functions $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ of this model are indeed continuous. Section 5 frames our
63 result in a larger context and discusses potential extensions.

64 1 Preliminaries

65 1.1 Syntactic Conventions

66 In this paper, we will work with various flavours of type theory. Our base system will always
67 be CC_ω , a predicative version of the Calculus of Constructions featuring an infinite hierarchy
68 of universes \square_i and dependent functions, which is summarized in Figure 1. We add inductive
69 types to this negative fragment, leading to the Calculus of Inductive Constructions (CIC) or
70 Baclofen Type Theory (BTT) depending on the formulation of dependent elimination. We
71 will summarize the defining features of BTT in the next section. Since we will manipulate
72 several type theories, we will write $\text{T} := \text{CIC}$ as a notational device to make explicit that we
73 are referring to the ambient type theory.

74 For brevity, we will define inductive types in a Coq-like syntax, but we will use a pattern-
75 matching syntax à la Agda for definitions by induction. As an example, we give below the
76 definition of natural numbers and the resulting formal typing and conversion rules.

$$\begin{array}{c}
77 \quad \text{Inductive } \mathbb{N} := \text{O} : \mathbb{N} \mid \text{S} : \mathbb{N} \rightarrow \mathbb{N} \\
78 \quad \frac{\Gamma \vdash}{\Gamma \vdash \mathbb{N} : \square_i} \quad \frac{\Gamma \vdash}{\Gamma \vdash \text{O} : \mathbb{N}} \quad \frac{\Gamma \vdash}{\Gamma \vdash \text{S} : \mathbb{N} \rightarrow \mathbb{N}} \\
79 \quad \frac{\Gamma \vdash P : \mathbb{N} \rightarrow \square_i \quad \Gamma \vdash t_0 : P \text{ O} \quad \Gamma \vdash t_5 : \Pi n : \mathbb{N}. P n \rightarrow P (\text{S } n)}{\Gamma \vdash \mathbb{N}_{\text{ind}} P t_0 t_5 : \Pi n : \mathbb{N}. P n} \\
80 \quad \mathbb{N}_{\text{ind}} P t_0 t_5 \text{ O} \equiv t_0 \quad \mathbb{N}_{\text{ind}} P t_0 t_5 (\text{S } n) \equiv t_5 n (\mathbb{N}_{\text{ind}} P t_0 t_5 n)
\end{array}$$

80 We will mostly ignore universe constraints and silently rely on typical ambiguity for the
81 sake of readability. Definitions indexed by universe variables i, j are meant to be universe-

$$A, B, M, N ::= \square_i \mid x \mid M N \mid \lambda x : A. M \mid \Pi x : A. M \mid \Sigma x : A. B \mid M.\pi_1 \mid M.\pi_2 \mid (M, N)$$

$$\Gamma, \Delta ::= \cdot \mid \Gamma, x : A$$

$$\begin{array}{c}
\frac{}{\vdash \cdot} \quad \frac{\Gamma \vdash A : \square_i}{\vdash \Gamma, x : A} \quad \frac{\vdash \Gamma \quad (x : A) \in \Gamma}{\Gamma \vdash x : A} \quad \frac{\vdash \Gamma \quad i < j}{\Gamma \vdash \square_i : \square_j} \\
\frac{\Gamma \vdash A : \square_i \quad \Gamma \vdash M : B}{\Gamma, x : A \vdash M : B} \quad \frac{\Gamma \vdash A : \square_i \quad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Pi x : A. B : \square_{\max(i,j)}} \\
\frac{\Gamma \vdash M : \Pi x : A. B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B\{x := N\}} \quad \frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash \Pi x : A. B : \square_i}{\Gamma \vdash \lambda x : A. M : \Pi x : A. B} \\
\frac{\Gamma \vdash M : A \quad \Gamma \vdash B : \square_i \quad \Gamma \vdash A \equiv B}{\Gamma \vdash M : B} \\
\frac{\Gamma \vdash A : \square_i \quad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Sigma x : A. B : \square_{\max(i,j)}} \quad \frac{\Gamma \vdash M : \Sigma x : A. B}{\Gamma \vdash M.\pi_1 : A} \quad \frac{\Gamma \vdash M : \Sigma x : A. B}{\Gamma \vdash M.\pi_2 : B\{x := M.\pi_1\}} \\
\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B\{x := M\} \quad \Sigma x : A. B : \square_i}{\Gamma \vdash (M, N) : \Sigma x : A. B} \quad \text{(conversion omitted)}
\end{array}$$

■ **Figure 1** Syntax of CC_ω extended with Σ -types

82 polymorphic in those variables [34]. All the translations we will give can be annotated with
83 universe variables to handle an arbitrary hierarchy of universes, but we will refrain from
84 doing so. We sometimes use implicit function arguments, which we bind with braces in
85 definitions.

86 Writing explicit terms in type theory can quickly become cumbersome for proofs, hence
87 we will omit them when the computational content is not important and write instead an
88 underscore as in $\vdash _ : A$.

89 1.2 Dependence in an Effectful Setting

90 In this paper, we will build type theories that feature computational effects. Regrettably,
91 adding effects to dependent type theory is not without consequences. They make indeed
92 observable the difference between call-by-value and call-by-name [20], a phenomenon that
93 puts us in front of a dilemma [25]. If we stick to by-name, we preserve the behaviour of the
94 negative fragment, i.e. Π -types, but we break dependent elimination. If we stick to by-value,
95 we now preserve dependent elimination, but functions become quite different to what one is
96 used to, as substitution is now restricted to syntactic values. For historical reasons, there is
97 a clear bias in type theory towards by-name, and we will follow the same doctrine.

98 As explained above, an effectful call-by-name type theory does not support full-blown
99 dependent elimination in general. As dependent elimination is quite a critical feature [23], this
100 might look concerning. Thankfully, most effectful theories we know of support a restricted
101 form of it, which essentially amounts to forcing the predicate used in the eliminator to be

13:4 Gardening with the Pythia

102 *strict* in its inductive argument¹. The resulting theory is known as Baclofen Type Theory [28],
 103 or **BTT** for short.

104 Contrarily to MLTT, which has a single dependent eliminator \mathcal{I}_{ind} for any given inductive
 105 type \mathcal{I} , **BTT** has two eliminators: a non-dependent one \mathcal{I}_{cse} , and a strict dependent one
 106 \mathcal{I}_{rec} . These three eliminators enjoy the same computational ι -rules, i.e. they reduce on
 107 constructors. The difference lies in their typing rules. The predicate of \mathcal{I}_{cse} does not depend
 108 on its inductive argument, i.e. it is basically simply-typed. Meanwhile, the predicate of \mathcal{I}_{rec}
 109 is wrapped in a *storage operator* [19] \mathcal{I}_{str} that locally evaluates its argument in a by-value
 110 fashion. This guarantees that it will only ever be applied to values, and never to effectful, or
 111 non-standard, inductive terms. The important observation is that \mathcal{I}_{str} can be defined in a
 112 systematic way out of \mathcal{I}_{cse} , namely it is simply an η -expansion in CPS style. To make things
 113 self-contained, we recall below the **BTT** eliminators for \mathbb{N} .

$$\begin{array}{c}
 114 \quad \frac{\Gamma \vdash P : \square \quad \Gamma \vdash t_0 : P \quad \Gamma \vdash t_S : \mathbb{N} \rightarrow P \rightarrow P}{\Gamma \vdash \mathbb{N}_{\text{cse}} P t_0 t_S : \mathbb{N} \rightarrow P} \\
 115 \quad \frac{\Gamma \vdash P : \mathbb{N} \rightarrow \square \quad \Gamma \vdash t_0 : \mathbb{N}_{\text{str}} \mathbf{O} P \quad \Gamma \vdash t_S : \Pi(n : \mathbb{N}). \mathbb{N}_{\text{str}} n P \rightarrow \mathbb{N}_{\text{str}} (\mathbf{S} n) P}{\Gamma \vdash \mathbb{N}_{\text{rec}} P t_0 t_S : \Pi(n : \mathbb{N}). \mathbb{N}_{\text{str}} n P}
 \end{array}$$

116 where $\mathbb{N}_{\text{str}} (n : \mathbb{N}) (P : \mathbb{N} \rightarrow \square) : \square :=$
 $\mathbb{N}_{\text{cse}} ((\mathbb{N} \rightarrow \square) \rightarrow \square) (\lambda(Q : \mathbb{N} \rightarrow \square). Q \mathbf{O})$
 $(\lambda(m : \mathbb{N}) (_ : (\mathbb{N} \rightarrow \square) \rightarrow \square) (Q : \mathbb{N} \rightarrow \square). Q (\mathbf{S} m)) n P.$

117 From within CIC, one can prove that $\Pi(n : \mathbb{N}) (P : \mathbb{N} \rightarrow \square). \mathbb{N}_{\text{str}} n P = P n$. Hence, this
 118 strictification is akin to double-negation translation, in so far as **BTT** is finer-grained than
 119 CIC, just as LJ is finer-grained than LK where $\neg\neg A \leftrightarrow A$. Note that in particular **BTT** is a
 120 subset of CIC, a fact on which we will rely on silently in this paper.

121 1.3 Continuity

122 In the remainder of this article, we suppose given two types $\vdash_{\top} \mathbf{I} : \square_0$ and $\vdash_{\top} \mathbf{O} : \mathbf{I} \rightarrow \square_0$.
 123 For simplicity, we set them in the lowest universe level, but all of the constructions to come
 124 can handle an arbitrary base level by bumping them by an appropriate amount.

125 The type \mathbf{I} is to be understood as a type of input or questions to a black-box, called an
 126 oracle. Dually, \mathbf{O} is the type of output or answers from the oracle. Since \mathbf{O} depends on \mathbf{I} ,
 127 we can encode a pretty much arbitrary interaction. Finally, we define the type of oracles
 128 as $\mathbf{Q} := \Pi(i : \mathbf{I}). \mathbf{O} i$. A reader more inclined towards computer science could also consider
 129 that \mathbf{O} and \mathbf{I} describe an interface for system calls, and \mathbf{Q} is the type of operating systems
 130 implementing these calls.

131 Let us formally define the notion of continuity over \mathbf{Q} .

132 ► **Definition 1.** *Given $\alpha_1, \alpha_2 : \mathbf{Q}$ and $\ell : \text{list } \mathbf{I}$, we say that α_1 and α_2 are finitely equal on ℓ ,
 133 written $\alpha_1 \approx_{\ell} \alpha_2$ when the following inductively defined predicate holds.*

$$\begin{array}{c}
 134 \quad \frac{}{\alpha_1 \approx_{\text{nil}} \alpha_2} \quad \frac{\alpha_1 i = \alpha_2 i \quad \alpha_1 \approx_{\ell} \alpha_2}{\alpha_1 \approx_{(\text{cons } i \ell)} \alpha_2}
 \end{array}$$

¹ As in programming language theory, not as in higher category theory.

135 ▶ **Definition 2.** We say that a function is continuous when it satisfies the continuity predicate

$$136 \quad \begin{aligned} \mathcal{C} & : \quad \Pi\{A : \square\}. (\mathbf{Q} \rightarrow A) \rightarrow \square \\ \mathcal{C} f & := \quad \Pi(\alpha : \mathbf{Q}). \Sigma(\ell : \text{list } \mathbf{I}). \Pi(\beta : \mathbf{Q}). \alpha \approx_\ell \beta \rightarrow f \alpha = f \beta. \end{aligned}$$

137 This definition captures in a generic way the intuitive notion that a computable functional
138 only needs a finite amount of information from its argument to produce an output. Note that
139 in particular the list of points ℓ where the function is evaluated depends on the argument
140 α , so this notion of continuity is weaker than uniform continuity, where the two quantifiers
141 for ℓ and α are swapped. Depending on the expressivity of \mathbb{T} , one can also consider weaker
142 variants where the existential is squashed with various proof-irrelevant modalities [12, 30, 31].

143 2 Dialogue Trees and Intensionality

144 2.1 Talking with Trees

145 It is now time to justify the title of this article by giving some explanations on the links
146 between trees, oracles and functions. We consider an operator $\mathfrak{D} : \square \rightarrow \square$, which given a
147 type $A : \square$, associates the type of well-founded trees, with leaves labelled in A . Each inner
148 node is labelled with a certain $i : \mathbf{I}$ and has $\mathbf{O} i$ children. In \mathbb{T} , this amounts to the following
149 inductive definition:

$$150 \quad \text{Inductive } \mathfrak{D} (A : \square) : \square := \eta : A \rightarrow \mathfrak{D} A \mid \beta : \Pi(i : \mathbf{I}). (\mathbf{O} i \rightarrow \mathfrak{D} A) \rightarrow \mathfrak{D} A.$$

151 This type of *dialogue trees* is known under several other names and has a lot of close
152 relatives [36, 26, 22, 16, 39]. They can be easily interpreted as functionals of type $\mathbf{Q} \rightarrow A$.
153 Intuitively, every inner node is an inert call to an oracle $\alpha : \mathbf{Q}$, and the answer is the label of
154 the leaf. This interpretation is implemented by a recursively defined dialogue function.

$$155 \quad \begin{aligned} \partial & : \quad \Pi\{A : \square\} (\alpha : \mathbf{Q}) (d : \mathfrak{D} A). A \\ \partial \alpha (\eta x) & := \quad x \\ \partial \alpha (\beta i k) & := \quad \partial \alpha (k (\alpha i)). \end{aligned}$$

156 ▶ **Definition 3** (Eloquent functions). A function $f : \mathbf{Q} \rightarrow A$ is said to be eloquent if there is
157 a dialogue tree $d : \mathfrak{D} A$ and a proof that $\Pi \alpha : \mathbf{Q}. f \alpha = \partial \alpha d$.

158 Representing functions as trees is a well-known way to extract intensional content from
159 them [17, 14]. Moreover, elements of $\mathfrak{D} A$ being well-founded, we get the following.

160 ▶ **Theorem 4** (Continuity). *Eloquent functions are continuous.*

161 **Proof.** The proof of the theorem is straightforward by induction on the dialogue tree d . ◀

162 This theorem is the fundamental insight of the proof. The rest of the paper is devoted to the
163 construction of a model where every function is eloquent and therefore continuous.

164 2.2 Liberating the Dialogue Monad

165 In an extensional enough setting, the \mathfrak{D} type former turns out to be a monad. The η natural
166 transformation is already part of the definition, and we can recursively define a **bind** function:

$$167 \quad \begin{aligned} \text{bind} & : \quad \Pi\{A B : \square\} (f : A \rightarrow \mathfrak{D} B) (d : \mathfrak{D} A). \mathfrak{D} B \\ \text{bind } f (\eta x) & := \quad f x \\ \text{bind } f (\beta i k) & := \quad \beta i (\lambda(o : \mathbf{O} i). \text{bind } f (k o)) \end{aligned}$$

13:6 Gardening with the Pythia

168 ► **Lemma 5.** *Assuming function extensionality, $(\mathfrak{D}, \eta, \text{bind})$ is a monad.*

169 Since we want to build a model of dependent type theory, we need to preserve a call-by-
 170 name equational theory, i.e. generated by the unrestricted β -rule. Following [28], this means
 171 that we need to interpret types as some kind of \mathfrak{D} -algebras. Unfortunately, the standard
 172 categorical definition of monads and their algebras is not usable in our context because it
 173 fundamentally relies on `funext`, which is not available in CIC. Thankfully, even by categorical
 174 standards, \mathfrak{D} is a very particular monad.

175 ► **Definition 6.** *A free monad in CIC is a parameterized inductive type $\mathcal{M} : \square \rightarrow \square$ with a
 176 dedicated constructor $\eta : \Pi(A : \square). A \rightarrow \mathcal{M} A$ and a finite set of constructors*

$$177 \quad c_i : \Pi(A : \square). \Phi_i (\mathcal{M} A) \rightarrow \mathcal{M} A$$

178 where $\Phi_i : \square \rightarrow \square$ is a type former syntactically strictly positive in its argument.

179 Note that the formal definition of free monad from category theory requires a forgetful
 180 functor to specify against what the monad would be free. The closest thing to our definition
 181 would be a free monad relatively to pointed functors, but even there our definition is stricter.
 182 A free monad can be thought of as a way to extend a type with unspecified, inert side-effects,
 183 a trivial form of algebraic effects [27, 1]. Since we have neither QITs [2] nor HITs [38] in CIC,
 184 we cannot enforce equations on these effects but we can still go a long way.

185 Free monads in CIC enjoy a lot of interesting properties. As the name implies, they
 186 are indeed monads. Again, the η function is given by definition, and `bind` can be defined
 187 functorially by induction similarly to the \mathfrak{D} case. Furthermore, the algebras of a free monad
 188 can be described in an intensionally-friendly way.

189 ► **Definition 7.** *Given \mathcal{M} as above, the type of intensional \mathcal{M} -algebras is the record type*

$$190 \quad \square^{\mathcal{M}} := \{A : \square; \dots; p_i : \Phi_i A \rightarrow A; \dots\}.$$

191 where the Φ_i are the same as in Definition 6.

192 ► **Theorem 8.** *Assuming `funext`, $\square^{\mathcal{M}}$ is isomorphic to the usual definition of \mathcal{M} -algebras.*

193 Said otherwise, the p_i functions are equivalent to the usual morphism $h_A : \mathcal{M} A \rightarrow A$
 194 preserving the monadic structure, except that this presentation does not require any equation.
 195 This results in the main advantage of intensional algebras, namely that they are closed
 196 under product type in a purely intensional setting. That is, if $A : \square$ and $B : A \rightarrow \square^{\mathcal{M}}$ then
 197 $\Pi x : A. (B x).\pi_1$ can be equipped with an intensional algebra structure defined pointwise.
 198 This solves a similar issue encountered in [35].

199 It is clear that \mathfrak{D} is a free monad, so we can define similarly intensional \mathfrak{D} -algebras.

200 ► **Definition 9 (Pythias).** *A pythia for $A : \square$ is a term $p_A : \Pi(i : \mathbf{I}). (\bigcirc i \rightarrow A) \rightarrow A$.*

201 Per the above theorem, pythias for A are extensionally in one-to-one correspondence with
 202 \mathfrak{D} -algebra structures over A , but are much better behaved intensionally. This will be the
 203 crux of the branching translation from Section 3.3.

3 The Syntactic Model

3.1 Overview

We prove that all BTT functions are continuous using a generalization of Escardó's model. While the latter only provides a model of System T, a simply-typed language, our model accomodates not only dependent types, but also universes and inductive types equipped with a strict form of dependent elimination. It is given as a program translation, and thus belongs to the class of syntactic models [13, 7]. The final model is built in three stages, namely

1. An axiom model (Section 3.2),
2. A branching model (Section 3.3),
3. An algebraic parametricity model (Section 3.4).

The first two models are standalone, and the third one glues them together. Each model can be explained computationally. The axiom model adds an blackbox oracle as a global variable. Asking the oracle is just function application, so there is no internal way to observe calls to the oracle. The branching model does the exact converse, as it provides an oracle in a purely inert way. Every single call to the branching oracle is tracked as a node of a dialogue tree, a representation that is reminiscent of game semantics. Finally, the algebraic parametricity model internalizes the fact that these two interpretations are computing essentially the same thing, behaving like a proof-relevant logical relation.

3.2 Axiom Translation

Let us fix a reserved variable $\alpha : \mathbf{Q}$. The axiom translation simply consists in adding α as the first variable of the context. Everywhere else, this translation is transparent. Reserving a variable has no technical consequence, if we were to use De Bruijn indices it just amounts to shifting them all by one. We will also annotate both free and bound variables with an a subscript for readability of the future parts of the paper, where we mix together different translations. We formally give the translation of the negative fragment in Figure 2.

$$\begin{array}{ll}
 [x]_a & := x_a & [\square_i]_a & := \square_i \\
 [\lambda x : A. M]_a & := \lambda x_a : [A]_a. [M]_a & [[A]]_a & := [A]_a \\
 [M N]_a & := [M]_a [N]_a & [·]_a & := \alpha : \mathbf{Q} \\
 [\Pi x : A. B]_a & := \Pi x_a : [A]_a. [B]_a & [[\Gamma, x : A]]_a & := [[\Gamma]]_a, x_a : [A]_a
 \end{array}$$

Figure 2 Axiom Translation (negative fragment)

► **Theorem 10.** *The axiom translation is a trivial syntactic model of CIC and hence of BTT.*

3.3 Branching Translation

Using the results from Section 2.2, we can use a simplified form of the weaning construction [28] to define the *branching translation*. It all boils down to interpreting types as intensional \mathfrak{D} -algebras, whose type will be defined as

$$\square^b := \Sigma(A : \square). \Pi(i : \mathbf{I}). (\bigcirc i \rightarrow A) \rightarrow A.$$

13:8 Gardening with the Pythia

$$\begin{array}{llll}
[x]_b & := & x_b & \llbracket A \rrbracket_b & := & [A]_b.\pi_1 \\
[\lambda x : A. M]_b & := & \lambda x_b : \llbracket A \rrbracket_b. [M]_b & \llbracket \cdot \rrbracket_b & := & \cdot \\
[M N]_b & := & [M]_b [N]_b & \llbracket \Gamma, x : A \rrbracket_b & := & \llbracket \Gamma \rrbracket_b, x_b : \llbracket A \rrbracket_b \\
\llbracket \square \rrbracket_b & := & \square^b & & & \\
\beta_{\square} & := & \lambda(i : \mathbf{I}) (k : \mathbf{O} \ i \rightarrow \square^b). \mathcal{U}_b & & & \\
\llbracket \Pi x : A. B \rrbracket_b & := & \Pi x_b : \llbracket A \rrbracket_b. \llbracket B \rrbracket_b & & & \\
\beta_{\Pi x : A. B} & := & \lambda(i : \mathbf{I}) (k : \mathbf{O} \ i \rightarrow \Pi x : \llbracket A \rrbracket_b. \llbracket B \rrbracket_b) (x : \llbracket A \rrbracket_b). \beta_B \ i \ (\lambda o : \mathbf{O} \ i. k \ o \ x) & & &
\end{array}$$

■ **Figure 3** Branching Translation (negative fragment)

235 Figure 3 defines the negative branching translation, which translates a type A as $[A]_b : \square^b$,
236 i.e. a pair $(\llbracket A \rrbracket_b, \beta_A)$ where $\llbracket A \rrbracket_b : \square$ and β_A is a pythia for $\llbracket A \rrbracket_b$. For readability, we give
237 the translation of types as these two components through a slight abuse of notation.

238 The main difficulty is to endow \square^b with a \mathcal{D} -algebra structure. Since there is no constraint
239 on this structure, we simply assume as a parameter of the translation a dummy \mathcal{D} -algebra
240 $\mathcal{U}_b : \square^b$. We will similarly need an inhabitant $\omega_b : \mathcal{U}_b.\pi_1$ to define dependent elimination.
241 There are many possible choices for \mathcal{U}_b , the simplest one being the unit type which is trivially
242 inhabited and algebraic. As a simple instance of weaning, we get the following.

243 ► **Proposition 11** (CC_ω Soundness). *We have the following.*

- 244 ■ *If $M \equiv_{\text{CC}_\omega} N$ then $[M]_b \equiv_{\top} [N]_b$.*
- 245 ■ *If $\Gamma \vdash_{\text{CC}_\omega} M : A$ then $\llbracket \Gamma \rrbracket_b \vdash_{\top} [M]_b : \llbracket A \rrbracket_b$.*

246 The interpretation of inductive types is fairly straightforward. Given an inductive type
247 \mathcal{I} , we create an inductive type \mathcal{I}_b whose constructors are the pointwise translation of the
248 constructors of \mathcal{I} , together with an additional $\beta_{\mathcal{I}}$ constructor turning it into a free \mathcal{D} -algebra.
249 We give as an example below the translation of \mathbb{N} , which will be the running example for the
250 remainder of this paper. Parameters and indices present no additional difficulty and we refer
251 to [28] for more details.

$$252 \quad \text{Inductive } \mathbb{N}_b : \square := \mathbf{O}_b : \mathbb{N}_b \mid \mathbf{S}_b : \mathbb{N}_b \rightarrow \mathbb{N}_b \mid \beta_{\mathbb{N}} : \Pi(i : \mathbf{I}). (\mathbf{O} \ i \rightarrow \mathbb{N}_b) \rightarrow \mathbb{N}_b.$$

253 An astute reader would have remarked that $\llbracket \mathbb{N} \rrbracket_b$ is not defined in the same way as in
254 Escardó's proof. This particular fact and its consequences are further discussed in Section 5.1.

255 ► **Theorem 12.** *For any inductive type \mathcal{I} , its branching translation \mathcal{I}_b is well-typed and*
256 *satisfies the strict positivity criterion.*

257 We must now implement the eliminators. We first define the non-dependent ones.

$$\begin{array}{ll}
258 \quad \llbracket \mathbf{N}_{\text{cse}} \rrbracket_b & : \quad \Pi P : \square^b. \llbracket P \rrbracket_b \rightarrow (\mathbb{N}_b \rightarrow \llbracket P \rrbracket_b \rightarrow \llbracket P \rrbracket_b) \rightarrow \mathbb{N}_b \rightarrow \llbracket P \rrbracket_b \\
\llbracket \mathbf{N}_{\text{cse}} \rrbracket_b \ P \ p_{\mathbf{O}} \ p_{\mathbf{S}} \ \mathbf{O}_b & := \quad p_{\mathbf{O}} \\
\llbracket \mathbf{N}_{\text{cse}} \rrbracket_b \ P \ p_{\mathbf{O}} \ p_{\mathbf{S}} \ (\mathbf{S}_b \ n) & := \quad p_{\mathbf{S}} \ n \ (\llbracket \mathbf{N}_{\text{cse}} \rrbracket_b \ P \ p_{\mathbf{O}} \ p_{\mathbf{S}} \ n) \\
\llbracket \mathbf{N}_{\text{cse}} \rrbracket_b \ P \ p_{\mathbf{O}} \ p_{\mathbf{S}} \ (\beta_{\mathbb{N}} \ i \ k) & := \quad \beta_P \ i \ (\lambda(o : \mathbf{O} \ i). \llbracket \mathbf{N}_{\text{cse}} \rrbracket_b \ P \ p_{\mathbf{O}} \ p_{\mathbf{S}} \ (k \ o))
\end{array}$$

259 As $P : \llbracket \square \rrbracket_b$, it has a pythia $\beta_P : \Pi(i : \mathbf{I}). (\mathbf{O} \ i \rightarrow \llbracket P \rrbracket_b) \rightarrow \llbracket P \rrbracket_b$. Every time we encounter
260 a branching occurrence of $\beta_{\mathbb{N}}$, we can thus use β_P and propagate the call recursively in the
261 branches. This is the usual by-name semantics of recursors.

262 However, problems arise with dependent elimination. Given $P : \mathbb{N}_b \rightarrow \square^b$ and subproofs
263 for \mathbf{O}_b and \mathbf{S}_b , there is no clear way to produce a term of type $(P \ (\beta_{\mathbb{N}} \ i \ k)).\pi_1$. There is

264 actually a good reason for that: if it were possible, this would make \mathbb{T} inconsistent [25].
 265 Following [28], we therefore restrict ourselves to a strict dependent elimination, relying on
 266 the storage operator \mathbb{N}_{str} from Section 1.2. Since it is given in direct style, its translation is
 267 systematic.

268 ► **Lemma 13.** *We have the following conversions.*

- 269 1. $[\mathbb{N}_{\text{str}}]_b \text{O}_b P \equiv P \text{O}_b$
- 270 2. $[\mathbb{N}_{\text{str}}]_b (\text{S}_b n) P \equiv P (\text{S}_b n)$
- 271 3. $[\mathbb{N}_{\text{str}}]_b (\beta_{\mathbb{N}} i k) P \equiv \text{U}_b$

272 Note that the two first equations above are a consequence of the conversion rules of \mathbb{N}_{cse}
 273 and thus hold in any model of BTT. Only the last one is specific to the current model at
 274 hand. Using this, we define the dependent eliminator below. Thanks to the fact that the
 275 predicate is wrapped in a storage operator, it is able to return a dummy term when applied
 276 to an effectful argument.

$$\begin{array}{lcl} \mathbb{N}_{\text{rec}} & : & \Pi P : \mathbb{N} \rightarrow \square. P \text{O} \rightarrow \\ & & (\Pi n : \mathbb{N}. \mathbb{N}_{\text{str}} n P \rightarrow \mathbb{N}_{\text{str}} (\text{S } n) P) \rightarrow \Pi n : \mathbb{N}. \mathbb{N}_{\text{str}} n P \\ 277 \quad [\mathbb{N}_{\text{rec}}]_b P p_{\text{O}} p_{\text{S}} \text{O}_b & := & p_{\text{O}} \\ 278 \quad [\mathbb{N}_{\text{rec}}]_b P p_{\text{O}} p_{\text{S}} (\text{S}_b n) & := & p_{\text{S}} n ([\mathbb{N}_{\text{rec}}]_b P p_{\text{O}} p_{\text{S}} n) \\ 279 \quad [\mathbb{N}_{\text{rec}}]_b P p_{\text{O}} p_{\text{S}} (\beta_{\mathbb{N}} i k) & := & \omega_b \end{array}$$

278 ► **Theorem 14.** *The branching translation provides a syntactic model of BTT.*

279 3.4 Algebraic Parametricity Translation

280 Following Escardó, we now have to relate the two translations. We achieve this through a
 281 third layer of *algebraic parametricity*. There are two major differences compared to Escardó's
 282 model [11]. The first one is that the logical relation does not live in the metatheory anymore
 283 and is defined as a syntactic model similar to parametricity [5]. This is not unexpected, but
 284 it is needed to interpret dependent types in a satisfactory way. The second difference is that
 285 the parametricity predicate *itself* must be endowed with an algebraic structure. This was a
 286 much more surprising structure that happens to be required to interpret large dependent
 287 elimination.

288 Intuitively, every type $A : \square$ is translated as a predicate $\llbracket A \rrbracket_{\varepsilon} : \llbracket A \rrbracket_a \rightarrow \llbracket A \rrbracket_b \rightarrow \square$. Note
 289 that $\alpha : \mathbb{Q}$ is implicitly part of the context as in the axiom model. As explained above, we
 290 also ask for the predicate to be \mathfrak{D} -algebraic in the sense that it must be equipped with a
 291 proof

$$292 \quad \beta_A^{\varepsilon} : \Pi(x_a : \llbracket A \rrbracket_a) (i : \mathbb{I}) (k : \mathbb{O} i \rightarrow \llbracket A \rrbracket_b). \llbracket A \rrbracket_{\varepsilon} x_a (k (\alpha i)) \rightarrow \llbracket A \rrbracket_{\varepsilon} x_a (\beta_A i k).$$

293 We will write the type of such algebraic parametricity predicates as

$$\begin{array}{l} 294 \quad \square^{\varepsilon} (A_a : \llbracket \square \rrbracket_a) (A_b : \llbracket \square \rrbracket_b) := \Sigma(A_{\varepsilon} : \llbracket A \rrbracket_a \rightarrow \llbracket A \rrbracket_b \rightarrow \square). \\ \quad \Pi(x_a : \llbracket A \rrbracket_a) (i : \mathbb{I}) (k : \mathbb{O} i \rightarrow \llbracket A \rrbracket_b) (x_{\varepsilon} : A_{\varepsilon} x_a (k (\alpha i))). \\ \quad A_{\varepsilon} x_a (\beta_A i k) \end{array}$$

295 Just as we did for the branching translation, given $A : \square_i$ we define separately the
 296 predicate $\llbracket A \rrbracket_{\varepsilon}$ and the proof of parametric algebraicity β_A^{ε} . We define the translation in
 297 Figure 4. As before we also ask for a dummy algebraic predicate $\text{U}_{\varepsilon} : \Pi(A : \square). \square^{\varepsilon} A \text{U}_b$
 298 which can be taken to be always a trivially inhabited predicate, together with an arbitrary
 299 proof $\omega_{\varepsilon} : \Pi(A : \square) (x : A). (\text{U}_{\varepsilon} A). \pi_1 x \omega_b$.

13:10 Gardening with the Pythia

$$\begin{aligned}
\llbracket \square \rrbracket_\varepsilon &:= \lambda(A_a : \llbracket \square \rrbracket_a) (A_b : \llbracket \square \rrbracket_b). \square^\varepsilon A_a A_b \\
\beta_\square^\varepsilon &:= \lambda(A_a : \llbracket \square \rrbracket_a) (i : \mathbf{I}) (k : \mathbf{O} \ i \rightarrow \llbracket \square \rrbracket_b) (A_\varepsilon : \llbracket \square \rrbracket_\varepsilon A_a (k (\alpha \ i))). \mathcal{U}_\varepsilon A_a \\
[x]_\varepsilon &:= x_\varepsilon \\
\llbracket \lambda x : A. M \rrbracket_\varepsilon &:= \lambda(x_a : \llbracket A \rrbracket_a) (x_b : \llbracket A \rrbracket_b) (x_\varepsilon : \llbracket A \rrbracket_\varepsilon x_a x_b). \llbracket M \rrbracket_\varepsilon \\
\llbracket M \ N \rrbracket_\varepsilon &:= \llbracket M \rrbracket_\varepsilon \llbracket N \rrbracket_a \llbracket N \rrbracket_b \llbracket N \rrbracket_\varepsilon \\
\llbracket \Pi x : A. B \rrbracket_\varepsilon &:= \lambda(f_a : \llbracket \Pi x : A. B \rrbracket_a) (f_b : \llbracket \Pi x : A. B \rrbracket_b). \\
&\quad \Pi(x_a : \llbracket A \rrbracket_a) (x_b : \llbracket A \rrbracket_b) (x_\varepsilon : \llbracket A \rrbracket_\varepsilon x_a x_b). \llbracket B \rrbracket_\varepsilon (f_a x_a) (f_b x_b) \\
\beta_{\Pi x : A. B}^\varepsilon &:= \lambda(f_a : \llbracket \Pi x : A. B \rrbracket_a) (i : \mathbf{I}) (k : \mathbf{O} \ i \rightarrow \llbracket \Pi x : A. B \rrbracket_b). \\
&\quad \lambda(f_\varepsilon : \llbracket \Pi x : A. B \rrbracket_\varepsilon f_a (k (\alpha \ i))). \\
&\quad \lambda(x_a : \llbracket A \rrbracket_a) (x_b : \llbracket A \rrbracket_b) (x_\varepsilon : \llbracket A \rrbracket_\varepsilon x_a x_b). \\
&\quad \beta_B^\varepsilon (f_a x_a) i (\lambda(o : \mathbf{O} \ i). k \ o \ x_b) (f_\varepsilon x_a x_b x_\varepsilon) \\
\llbracket A \rrbracket_\varepsilon &:= [A]_\varepsilon . \pi_1 \\
\llbracket \cdot \rrbracket_\varepsilon &:= \alpha : \mathbf{Q} \\
\llbracket \Gamma, x : A \rrbracket_\varepsilon &:= \llbracket \Gamma \rrbracket_\varepsilon, x_a : \llbracket A \rrbracket_a, x_b : \llbracket A \rrbracket_b, x_\varepsilon : \llbracket A \rrbracket_\varepsilon x_a x_b
\end{aligned}$$

■ **Figure 4** Algebraic Parametricity Translation (negative fragment)

$$\begin{aligned}
\text{Inductive } \mathbb{N}_\varepsilon (\alpha : \mathbf{Q}) : \mathbb{N} \rightarrow \mathbb{N}_b \rightarrow \square := \\
| \mathbf{O}_\varepsilon : \mathbb{N}_\varepsilon \alpha \ \mathbf{O} \ \mathbf{O}_b \\
| \mathbf{S}_\varepsilon : \Pi(n_a : \mathbb{N}) (n_b : \mathbb{N}_b) (n_\varepsilon : \mathbb{N}_\varepsilon \alpha \ n_a \ n_b). \mathbb{N}_\varepsilon \alpha \ (\mathbf{S} \ n_a) \ (\mathbf{S}_b \ n_b) \\
| \beta_{\mathbb{N}}^\varepsilon : \Pi(n_a : \mathbb{N}) (i : \mathbf{I}) (k : \mathbf{O} \ i \rightarrow \mathbb{N}_b) (n_\varepsilon : \mathbb{N}_\varepsilon \alpha \ n_a \ (k (\alpha \ i))). \mathbb{N}_\varepsilon \alpha \ n_a \ (\beta_{\mathbb{N}} \ i \ k) \\
[\mathbb{N}]_\varepsilon := (\mathbb{N}_\varepsilon \ \alpha, \beta_{\mathbb{N}}^\varepsilon \ \alpha) \quad [\mathbf{O}]_\varepsilon := \mathbf{O}_\varepsilon \ \alpha \quad [\mathbf{S}]_\varepsilon := \mathbf{S}_\varepsilon \ \alpha
\end{aligned}$$

■ **Figure 5** Algebraic Parametricity for \mathbb{N}

- 300 ► **Theorem 15** (CC_ω Soundness). *We have the following.*
301 ■ *If $M \equiv_{\text{CC}_\omega} N$ then $\llbracket M \rrbracket_\varepsilon \equiv_{\top} \llbracket N \rrbracket_\varepsilon$.*
302 ■ *If $\Gamma \vdash_{\text{CC}_\omega} M : A$ then $\llbracket \Gamma \rrbracket_\varepsilon \vdash_{\top} \llbracket M \rrbracket_\varepsilon : \llbracket A \rrbracket_\varepsilon \llbracket M \rrbracket_a \llbracket M \rrbracket_b$.*

303 The algebraic parametric translation of inductive types sticks closely to the branching
304 one. Given an inductive type \mathcal{I} , we create an inductive type \mathcal{I}_ε whose constructors are the
305 pointwise $\llbracket \cdot \rrbracket_\varepsilon$ translation of those of \mathcal{I} . An additional constructor $\beta_{\mathcal{I}}^\varepsilon$ freely implements the
306 algebraicity requirement. Since $\alpha : \mathbf{Q}$ is implicitly part of the translated context, we have to
307 take it as a parameter of the translated inductive type and explicitly pass it as an argument
308 when interpreting those types and their proof of algebraicity. We give the translation on
309 our running example in Figure 5. Once again, parameters and indices present no particular
310 problem and are handled similarly to [28].

- 311 ► **Theorem 16.** *For any inductive type \mathcal{I} , its algebraic parametricity translation \mathcal{I}_ε is well*
312 *typed and satisfies the positivity criterion.*

313 As for the branching translation, we retrieve a restricted form of dependent elimination
314 based on storage operators. The argument is virtually the same, but now at the level of
315 parametricity, which makes the syntactic burden even heavier since we now have everything
316 repeated three times. To enhance readability, we will use the following shorthand for binders:

$$317 \quad \langle x : A \rangle := x_a : \llbracket A \rrbracket_a, x_b : \llbracket A \rrbracket_b, x_\varepsilon : \llbracket A \rrbracket_\varepsilon x_a x_b$$

318 and similarly for application to variables. We give the eliminators for our running example
319 in this lighter syntax, which is already the limit of what can be done on paper.

$$\begin{aligned}
320 \quad & \llbracket \mathbb{N}_{\text{cse}} \rrbracket_\varepsilon : \Pi \langle P : \square \rangle \langle p_O : P \rangle \langle p_S : \mathbb{N} \rightarrow P \rightarrow P \rangle \langle n : \mathbb{N} \rangle. \\
& \llbracket P \rrbracket_\varepsilon \llbracket \mathbb{N}_{\text{cse}} P p_O p_S n \rrbracket_a \llbracket \mathbb{N}_{\text{cse}} P p_O p_S n \rrbracket_b \\
& \llbracket \mathbb{N}_{\text{cse}} \rrbracket_\varepsilon \langle P \rangle \langle p_O \rangle \langle p_S \rangle _ _ \mathbf{O}_\varepsilon := p_{O\varepsilon} \\
& \llbracket \mathbb{N}_{\text{cse}} \rrbracket_\varepsilon \langle P \rangle \langle p_O \rangle \langle p_S \rangle _ _ (\mathbf{S}_\varepsilon \langle n \rangle) := p_{S\varepsilon} \langle n \rangle (\llbracket \mathbb{N}_{\text{cse}} \rrbracket_\varepsilon \langle P \rangle \langle p_O \rangle \langle p_S \rangle \langle n \rangle) \\
321 \quad & \llbracket \mathbb{N}_{\text{cse}} \rrbracket_\varepsilon \langle P \rangle \langle p_O \rangle \langle p_S \rangle _ _ (\beta_{\mathbb{N}}^\varepsilon n_a i k n_\varepsilon) := \beta_P^\varepsilon \\
& (\llbracket \mathbb{N}_{\text{cse}} P p_O p_S \rrbracket_a n_a) i \\
& (\lambda(o : \mathbf{O} i). \llbracket \mathbb{N}_{\text{cse}} P p_O p_S \rrbracket_b (k o)) \\
& (\llbracket \mathbb{N}_{\text{cse}} \rrbracket_\varepsilon \langle P \rangle \langle p_O \rangle \langle p_S \rangle n_a (k (\alpha i) n_\varepsilon))
\end{aligned}$$

322 Note that the $\beta_{\mathbb{N}}^\varepsilon$ case explicitly calls the global axiom α to relate the oracular term with
323 the branching one. This is one of the few places that introduce an actual use of the oracle in
324 the translation, by opposition to merely passing it around.

325 We define $\llbracket \mathbb{N}_{\text{str}} \rrbracket_\varepsilon$ as before, using the fact it is given directly in the source in terms of
326 \mathbb{N}_{cse} . In particular we do not have to write its translation explicitly. Finally, we can define
327 the dependent eliminators, following the same structure as before.

$$\begin{aligned}
328 \quad & \llbracket \mathbb{N}_{\text{rec}} \rrbracket_\varepsilon : \Pi \langle P : \mathbb{N} \rightarrow \square \rangle \langle p_O : P \mathbf{O} \rangle \langle p_S : \Pi(n : \mathbb{N}). \mathbb{N}_{\text{str}} n P \rightarrow \mathbb{N}_{\text{str}} (\mathbf{S} n) P \rangle. \\
& \Pi(n : \mathbb{N}). \llbracket \mathbb{N}_{\text{str}} n P \rrbracket_\varepsilon \llbracket \mathbb{N}_{\text{rec}} P p_O p_S n \rrbracket_a \llbracket \mathbb{N}_{\text{rec}} P p_O p_S n \rrbracket_b \\
& \llbracket \mathbb{N}_{\text{rec}} \rrbracket_\varepsilon \langle P \rangle \langle p_O \rangle \langle p_S \rangle _ _ \mathbf{O}_\varepsilon := p_{O\varepsilon} \\
329 \quad & \llbracket \mathbb{N}_{\text{rec}} \rrbracket_\varepsilon \langle P \rangle \langle p_O \rangle \langle p_S \rangle _ _ (\mathbf{S}_\varepsilon \langle n \rangle) := p_{S\varepsilon} \langle n \rangle (\llbracket \mathbb{N}_{\text{rec}} \rrbracket_\varepsilon \langle P \rangle \langle p_O \rangle \langle p_S \rangle \langle n \rangle) \\
& \llbracket \mathbb{N}_{\text{rec}} \rrbracket_\varepsilon \langle P \rangle \langle p_O \rangle \langle p_S \rangle _ _ (\beta_{\mathbb{N}}^\varepsilon n_a i k n_\varepsilon) := \omega_\varepsilon (P_a n_a) (\llbracket \mathbb{N}_{\text{rec}} P p_O p_S \rrbracket_a n_a)
\end{aligned}$$

330 Following the results from [28], this translation can be generalized to any inductive type,
331 potentially with parameters and indices. Indeed, it basically amounts to the composition of
332 weaning with binary parametricity.

333 **► Theorem 17.** *Algebraic parametricity is a syntactic model of BTT.*

334 4 Continuity of $(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$

335 This section is dedicated to the proof of the main theorem which we formally state below.

336 **► Theorem 18.** *If $\vdash_{\text{BTT}} f : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ then $\vdash_{\text{CIC}} _ : \mathcal{C} f$.*

337 **Proof.** The proof follows the same structure as Escardó's proof for System T, and requires a
338 clever instance of the model described above.

339 In short, we will define an element $\gamma_b : \mathbb{N}_b \rightarrow \mathbb{N}_b$ and lift it as a constant $\gamma : \mathbb{N} \rightarrow \mathbb{N}$ in the
340 source theory. Computationally, it behaves as an impure function that tracks the arguments it
341 is called on. We will then use it to prove that f is eloquent, with tree witness $[f \ \gamma]_b \equiv [f]_b \ \gamma_b$.
342 Before getting to the nitty-gritty, we will fix henceforth the oracular type parameters for the
343 remainder of this section as

$$344 \quad \mathbf{I} := \mathbb{N} \quad \text{and} \quad \mathbf{O} := \lambda(i : \mathbf{I}). \mathbb{N}.$$

345 Some results exposed in this section are still independent from this precise choice of oracle.
346 When this is the case, we will stick to the \mathbf{Q} notation to highlight this fact.

347 Since \mathbb{N}_b is essentially a free algebra, we can define a dialogue function $\partial^{\mathbb{N}}$ similar to the
348 one defined in Section 2.

$$\begin{aligned}
& \partial^{\mathbb{N}} : \mathbf{Q} \rightarrow \mathbb{N}_b \rightarrow \mathbb{N} \\
349 \quad & \partial^{\mathbb{N}} \alpha \mathbf{O}_b := \mathbf{O} \\
& \partial^{\mathbb{N}} \alpha (\mathbf{S}_b n_b) := \mathbf{S} (\partial^{\mathbb{N}} \alpha n_b) \\
& \partial^{\mathbb{N}} \alpha (\beta_{\mathbb{N}} i k) := \partial^{\mathbb{N}} \alpha (k (\alpha i)).
\end{aligned}$$

13:12 Gardening with the Pythia

350 ► **Proposition 19** (Unicity of specification). *There is a proof*

$$351 \quad \vdash_{\mathbb{T}} _ : \Pi(\alpha : \mathbf{Q}) \langle n : \mathbb{N} \rangle. n_a = \partial^{\mathbb{N}} \alpha n_b.$$

352 **Proof.** By induction on n_ε . ◀

353 ► **Proposition 20** (Generic parametricity). *There is a proof*

$$354 \quad \vdash_{\mathbb{T}} _ : \Pi(\alpha : \mathbf{Q}) (n_b : \mathbb{N}_b). \mathbb{N}_\varepsilon \alpha (\partial^{\mathbb{N}} \alpha n_b) n_b.$$

355 **Proof.** By induction on n_b . ◀

356 Let us now define our generic element $\gamma_b : \mathbb{N}_b \rightarrow \mathbb{N}_b$.

357 ► **Definition 21** (Generic tree). *We define in \mathbb{T} the generic tree \mathfrak{t} as*

$$358 \quad \begin{array}{llll} \mathfrak{t} & : & \mathbb{N} \rightarrow \mathbb{N}_b & \text{where } \eta_{\mathbb{N}} & : & \mathbb{N} \rightarrow \mathbb{N}_b \\ \mathfrak{t} & := & \lambda(n : \mathbb{N}). \beta_{\mathbb{N}} n \eta_{\mathbb{N}} & \eta_{\mathbb{N}} \mathbf{O} & := & \mathbf{O}_b \\ & & & \eta_{\mathbb{N}} (\mathbf{S} n) & := & \mathbf{S}_b (\eta_{\mathbb{N}} n). \end{array}$$

359 ► **Lemma 22** (Fundamental property of the generic tree). *We have a proof*

$$360 \quad \vdash_{\mathbb{T}} _ : \Pi(\alpha : \mathbb{N} \rightarrow \mathbb{N}) (n : \mathbb{N}). \partial^{\mathbb{N}} \alpha (\mathfrak{t} n) = \alpha n.$$

361 **Proof.** Immediate by the definition of the ∂ function. ◀

362 ► **Definition 23** (Generic element). *We define the generic element $\gamma_b : \mathbb{N}_b \rightarrow \mathbb{N}_b$ as follows.*

$$363 \quad \begin{array}{llll} \gamma_b n_b & := & \gamma_0 \mathbf{O} n_b & \text{where } \gamma_0 & : & \mathbb{N} \rightarrow \mathbb{N}_b \rightarrow \mathbb{N}_b \\ & & & \gamma_0 a \mathbf{O}_b & := & \mathfrak{t} a \\ & & & \gamma_0 a (\mathbf{S}_b n_b) & := & \gamma_0 (\mathbf{S} a) n_b \\ & & & \gamma_0 a (\beta_{\mathbb{N}} i k) & := & \beta_{\mathbb{N}} i (\lambda o : \mathbb{N}. \gamma_0 a (k o)). \end{array}$$

364 Intuitively, γ_b adds a layer to its argument, replacing each leaf by a $\mathfrak{t} n$, where n is the
365 number of \mathbf{S}_b encountered in the branch. It has the following property.

366 ► **Lemma 24** (Fundamental property of the generic element). *We have a proof*

$$367 \quad \vdash_{\mathbb{T}} _ : \Pi(\alpha : \mathbb{N} \rightarrow \mathbb{N}) (n_b : \mathbb{N}_b). \partial^{\mathbb{N}} \alpha (\gamma_b n_b) = \alpha (\partial^{\mathbb{N}} \alpha n_b).$$

368 **Proof.** Straightforward by induction on n_b , using Lemma 22 for the \mathbf{O}_b case. ◀

369 ► **Proposition 25.** *The γ_b term can be lifted to a function $\gamma : \mathbb{N} \rightarrow \mathbb{N}$ in the source theory.*

370 **Proof.** It is sufficient to derive the following sequents, the first two being trivial.

$$371 \quad \alpha : \mathbb{N} \rightarrow \mathbb{N} \vdash_{\mathbb{T}} \alpha : \llbracket \mathbb{N} \rightarrow \mathbb{N} \rrbracket_a \quad \vdash_{\mathbb{T}} \gamma_b : \llbracket \mathbb{N} \rightarrow \mathbb{N} \rrbracket_b \quad \alpha : \mathbb{N} \rightarrow \mathbb{N} \vdash_{\mathbb{T}} \gamma_\varepsilon : \llbracket \mathbb{N} \rightarrow \mathbb{N} \rrbracket_\varepsilon \alpha \gamma_b$$

372 For γ_ε , assuming $\langle n : \mathbb{N} \rangle$ we have to prove $\llbracket \mathbb{N} \rrbracket_\varepsilon (\alpha n_a) (\gamma_b n_b)$. By Proposition 19,
373 this is the same as $\llbracket \mathbb{N} \rrbracket_\varepsilon (\alpha (\partial^{\mathbb{N}} \alpha n_b)) (\gamma_b n_b)$. By Proposition 24, this is the same as
374 $\llbracket \mathbb{N} \rrbracket_\varepsilon (\partial^{\mathbb{N}} \alpha (\gamma_b n_b)) (\gamma_b n_b)$. We conclude by Proposition 20. ◀

375 We can now get to the proof of the main result. Let $\vdash_{\mathbf{BTT}} f : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$. Since
376 $\gamma : \mathbb{N} \rightarrow \mathbb{N}$ can be reflected from the model into \mathbf{BTT} , we can consider the term $\vdash_{\mathbf{BTT}} f \gamma : \mathbb{N}$.
377 By soundness, it results in the three terms below.

$$378 \quad \begin{array}{ll} \alpha : \mathbb{N} \rightarrow \mathbb{N} & \vdash_{\mathbb{T}} [f]_a \alpha : \mathbb{N} \\ & \vdash_{\mathbb{T}} [f]_b \gamma_b : \mathbb{N}_b \\ \alpha : \mathbb{N} \rightarrow \mathbb{N} & \vdash_{\mathbb{T}} [f]_\varepsilon \alpha \gamma_b \gamma_\varepsilon : \mathbb{N}_\varepsilon \alpha ([f]_a \alpha) ([f]_b \gamma_b) \end{array}$$

379 Applying Proposition 19 to $[f]_a$, $[f]_b$ and $[f]_\varepsilon$, we get:

$$\vdash_{\mathbb{T}} _ : \Pi(\alpha : \mathbb{N} \rightarrow \mathbb{N}). [f]_a \alpha = \partial^{\mathbb{N}} \alpha ([f]_b \gamma_b)$$

Since f is a term in BTT that does not use any impure extension of the model, it is easy to check that $[f]_a \equiv f$. Therefore, f is eloquent. By Theorem 4, this implies that f is continuous, which concludes our proof. \blacktriangleleft

5 Discussion and Related Work

5.1 Comparison with Similar Models

As already stated, our proof follows the argument given by Escardó [11] for System T, which can also be found as a close variant by Sterling that uses streams instead of trees [35]. Yet, in order to scale to BTT there are a few non-trivial technical differences in our version that ought to be highlighted.

The first obvious one is that Escardó's model does not really qualify as a syntactic model of System T. Rather, it is a model in a type-theoretic metatheory. The difference is subtle, and lies in the fact that the source language is an AST of the ambient type theory in Escardó's model, while there is no such thing in sight in our variant. Actually, this would not even have been possible because in order to internalize type theory inside itself, one needs some form of induction-recursion to handle universes. Morally, we got rid of the middle man of an overarching standard syntactic model of BTT [3].

Another major difference is that the parametricity predicates must be compatible with the \mathfrak{D} -algebra structure of the underlying types. This is needed to interpret large elimination, which is absent from System T. This requirement is thus void in Escardó's model. It was a surprising part of the model design, but in hindsight it is obvious that it would pop up eventually. Furthermore, both to preserve conversion and to scale to richer inductive types, the parametricity predicate needs to be given in an inductive way following the underlying source type, rather than as an ad-hoc equality between two terms.

We emphasized that our interpretation of \mathbb{N} is not the same as Escardó's, which uses instead $\llbracket \mathbb{N}' \rrbracket_b := \mathfrak{D} \mathbb{N}$. The reason for that has been already briefly observed in [35] but it is worth elaborating here. Said bluntly, Escardó's interpretation is actually *not* a model of System T. While it is indeed possible to write a simply-typed eliminator

$$\mathbb{N}'_{\text{cse}} : \Pi(P : \square). P \rightarrow (\mathbb{N}' \rightarrow P \rightarrow P) \rightarrow P$$

it does not enjoy the correct computational behaviour. Namely, in general

$$\mathbb{N}'_{\text{cse}} P p_0 p_S (S' n) \not\equiv p_S n (\mathbb{N}'_{\text{cse}} P p_0 p_S n).$$

A typical situation where this equation would break happens when n is an effectful term, i.e. its translation is of the form $\beta i k$. This can be explained by the fact that recursive constructors in effectful call-by-name need to thunk their arguments, i.e. pattern-matching on the head of an inductive term must not evaluate the subterms of the constructor. This is not the case for Escardó's interpretation, which is closer to a call-by-value embedding of \mathbb{N} in call-by-name. Since dependent type theory makes the requirement that this equation holds in the typing rules themselves, we need to pick the right interpretation of \mathbb{N} .

Escardó and Xu also gave related models to internalize uniform continuity [40, 10]. Contrarily to the above one, they build these models out of sheaves, which have also been used similarly by Coquand and Jaber [8, 9]. Sheaves form a locally closed cartesian category, hence they only implement a small fragment of MLTT. It is well-known that the universe

13:14 Gardening with the Pythia

of sheaves is not a sheaf in general, and in particular the existence of universes in the first model is an open problem. We have several remarks to make. First, assuming univalence and HITs in the target theory, it turns out to be straightforward to build a syntactic sheaf model of MLTT [32]. Univalence is typically needed to relax the strict uniqueness requirement of sheaves into its fibrant version.

More interestingly, a closer look at [32] shows that univalent sheafification is basically the HIT

$$\begin{aligned}
 & \text{Inductive } \mathfrak{S} (A : \square) : \square := \\
 & | \eta : A \rightarrow \mathfrak{S} A \\
 & | \beta : \Pi(i : \mathbf{I}). (\mathbf{O} i \rightarrow \mathfrak{S} A) \rightarrow \mathfrak{S} A \\
 & | \sigma_1 : \Pi(i : \mathbf{I}) (x : \mathfrak{S} A). \beta i (\lambda(o : \mathbf{O} i). x) = x \\
 & | \sigma_2 : \dots
 \end{aligned}$$

where $\mathbf{O} : \mathbf{I} \rightarrow \mathbf{hProp}$ and σ_2 is such that $(\beta, \sigma_1, \sigma_2)$ prove that $\lambda(x : \mathfrak{S} A) (_ : \mathbf{O} i). x$ defines an equivalence $\mathfrak{S} A \cong (\mathbf{O} i \rightarrow \mathfrak{S} A)$. The relationship to \mathfrak{D} is obvious, and leads us to challenge Escardó's claim that the dialogue model is not a sheaf model. The higher equalities are precisely what is missing to implement full dependent elimination, i.e. to ensure that sheafification preserves observational purity. Otherwise said, the dialogue monad is an impure variant of the sheafification monad, giving a curious and unexpected double entendre to the phrase *effectful forcing*.

Rahli et al. [30] give another proof of uniform continuity for NuPRL using a form of delimited exceptions. Computationally, their model tracks the accesses to the argument of functions by passing them exception-raising placeholders. The control flow is inverted w.r.t. Escardó's model, which requires non-terminating realizers, but we believe that the fundamental mechanism is similar. In the same context Rahli et al. [31] defines a sheaf model with bar induction in mind, but this principle is inextricably tied to uniform continuity [6].

5.2 Internalization

In this paper we have constructed a model of BTT that associates to every closed term $\vdash f : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ a proof in CIC that it is continuous. Can we do better? First, we know that there is a major limitation. Indeed, MLTT extended with the internal statement

$$\Pi f : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}. \mathcal{C} f$$

results in an inconsistent theory [12]. We will call this property *internal continuity* below. The proof crucially relies on two ingredients, namely congruence of conversion and large dependent elimination. Thus, there might be hope for BTT where the latter is restricted.

► **Theorem 26.** *Internal continuity holds in our model iff it holds in \mathbf{T} .*

This is obviously disappointing, since it implies that \mathbf{T} is inconsistent. One can then wonder if it is possible to aim for a middle ground, where we internalize the modulus of continuity itself, but keep the computation of this modulus in the target. That is, construct a term of type $\Pi(\alpha : \mathbb{N} \rightarrow \mathbb{N}) \langle f : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rangle. \llbracket \mathcal{C} f \rrbracket$, where $\llbracket A \rrbracket$ stands for the triple $\Sigma(x_a : \llbracket A \rrbracket_a) (x_b : \llbracket A \rrbracket_b). \llbracket A \rrbracket_\varepsilon x_a x_b$. The implication regarding the target theory is a bit more subtle.

► **Lemma 27.** *If we have $\vdash_{\mathbf{T}} _ : \Pi(\alpha : \mathbb{N} \rightarrow \mathbb{N}) \langle f : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rangle. \llbracket \mathcal{C} f \rrbracket$ then we can also get a proof that $\vdash_{\mathbf{T}} _ : \Pi f : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}. f \sim_{(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}} f \rightarrow \mathcal{C} f$, where $\sim_{(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}}$ is the canonical setoid equality on the functional type.*

461 **Proof.** Let $f : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ such that $f \sim_{(\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}} f$, and $\alpha : \mathbb{N} \rightarrow \mathbb{N}$ in \mathcal{T} . We define a
 462 term $\tilde{f} : \llbracket (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rrbracket$ as follows.

$$463 \quad [\tilde{f}]_a := f \quad [\tilde{f}]_b := \lambda(u_b : \llbracket \mathbb{N} \rrbracket_b \rightarrow \llbracket \mathbb{N} \rrbracket_b). \eta_{\mathbb{N}} (f (\lambda n : \mathbb{N}. \partial^{\mathbb{N}} \alpha (u_b (\eta_{\mathbb{N}} n))))$$

464 These two terms are proved to be in relation by the parametricity predicate by applying
 465 the preservation of pointwise equality followed by an induction on the parametricity proof
 466 of the argument. Finally, if we have a term of type $\Pi(\alpha : \mathbb{N} \rightarrow \mathbb{N}) \langle f : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rangle. \llbracket \mathcal{C} f \rrbracket$,
 467 then we have $\llbracket \mathcal{C} \tilde{f} \rrbracket$ and thus $\mathcal{C} f$ by projection. \blacktriangleleft

468 This lemma implies in particular that if our target theory features `funext`, internalization
 469 of the modulus of continuity implies continuity of all functions $f : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$ in \mathcal{T} .
 470 Thus, by the aforementioned theorem, our theory is inconsistent. Conversely, if our theory is
 471 consistent, internalization of the modulus of continuity is out of reach.

472 As `funext` is independent from `CIC`, internalization of the modulus of continuity is unat-
 473 tainable if our target theory is plain `CIC`. If our target theory does not feature `funext`, the
 474 diagonalization argument of Escardó and Xu does not work anymore.

475 However, in `BTT` it is unclear whether it is possible to construct a similar paradox, or if
 476 there exists a model of it which validates the internalization of the modulus of continuity.
 477 This is still an open question. We nonetheless conjecture that adding an additional layer of
 478 presheaves to allow a varying number of oracles in the context could be the key to realize
 479 such a model. Indeed, adding a modal type of exceptions to `MLTT` is precisely what permits
 480 to go from the external Markov's rule [29] to the internal Markov's principle [24]. If we were
 481 able to locally create a fresh generic element independent from all the previously allocated
 482 ones, it seems that we could turn the external continuity rule into an internal one, mimicking
 483 what happens for the implementation of Markov's principle. Fresh exceptions are precisely
 484 used by Rahli et al. [30] to get what amounts to an independent generic element at every
 485 call, so this argument does not seem far-fetched. We leave this to future work.

486 5.3 Coq Formalization

487 The results from this paper have been formalized in Coq using a presentation similar to
 488 category with families. It is a shallow embedding in the style of [15], hence in particular all
 489 conversions are interpreted as definitional equalities. The development relies on universe
 490 polymorphism to implement universes in the model, but it could have been avoided at the
 491 cost of duplicating the code for every level existing in the hierarchy. As usual, we use negative
 492 pairs to handle context extensions in a definitional way. Apart from this, the development
 493 does not make use of any fancier feature from the Coq kernel. The code can be found at
 494 <https://gitlab.inria.fr/mbaillon/gardening-with-the-pythia>.

495 Conclusion

496 This paper gives a purely syntactic proof that functionals of a rich dependent type theory
 497 are continuous. Not only is the argument syntactic, but it is also expressed as a program
 498 translation into another dependent type theory. Thus, everything computes by construction
 499 and conversion in the source is interpreted as conversion in the target. Despite being a
 500 generalization of a simpler proof by Escardó, the dependently-typed presentation gives more
 501 insight about the constraints one has to respect for it to work properly, and highlights a few
 502 hidden flaws of the original version. Finally, the model gives empirical foothold to the claim

13:16 Gardening with the Pythia

503 that BTT is a natural setting for dependently-typed effects. We believe it is not merely an
504 ad-hoc set of rules, but a system that keeps appearing in various contexts, and thus a generic
505 effectful type theory.

References

- 506 ———
- 507 1 Danel Ahman. Handling fibred algebraic effects. *Proc. ACM Program. Lang.*, 2(POPL):7:1–7:29,
508 2018. doi:10.1145/3158095.
- 509 2 Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall
510 Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors,
511 *Foundations of Software Science and Computation Structures - 21st International Conference,*
512 *FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice*
513 *of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume
514 10803 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2018. doi:10.1007/
515 978-3-319-89366-2_16.
- 516 3 Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive
517 types. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual*
518 *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016,*
519 *St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29. ACM, 2016. doi:10.1145/
520 2837614.2837638.
- 521 4 Michael Beeson. *Foundations of Constructive Mathematics: Metamathematical Studies*. Num-
522 ber 6 in *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer, Berlin Heidelberg New
523 York Tokyo, 1985.
- 524 5 Jean-Philippe Bernardy and Marc Lasson. Realizability and parametricity in pure type
525 systems. In Martin Hofmann, editor, *Foundations of Software Science and Computational*
526 *Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European*
527 *Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March*
528 *26-April 3, 2011. Proceedings*, volume 6604 of *Lecture Notes in Computer Science*, pages
529 108–122. Springer, 2011. doi:10.1007/978-3-642-19805-2_8.
- 530 6 Mark Bickford, Liron Cohen, Robert L. Constable, and Vincent Rahli. Computability beyond
531 Church-Turing via choice sequences. In Anuj Dawar and Erich Grädel, editors, *Proceedings of*
532 *the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford,*
533 *UK, July 09-12, 2018*, pages 245–254. ACM, 2018. doi:10.1145/3209108.3209200.
- 534 7 Simon Boulrier. Extending type theory with syntactic models. *Logic in Computer Science*
535 *[cs.LO]. École nationale supérieure Mines-Télécom Atlantique*, 2018, [https://tel.archives-](https://tel.archives-ouvertes.fr/tel-02007839)
536 [ouvertes.fr/tel-02007839](https://tel.archives-ouvertes.fr/tel-02007839).
- 537 8 Thierry Coquand and Guilhem Jaber. A note on forcing and type theory. *Fundam. Informaticae*,
538 100(1-4):43–52, 2010. doi:10.3233/FI-2010-262.
- 539 9 Thierry Coquand and Guilhem Jaber. A computational interpretation of forcing in type theory.
540 In Peter Dybjer, Sten Lindström, Erik Palmgren, and Göran Sundholm, editors, *Epistemology*
541 *versus Ontology - Essays on the Philosophy and Foundations of Mathematics in Honour of*
542 *Per Martin-Löf*, volume 27 of *Logic, Epistemology, and the Unity of Science*, pages 203–213.
543 Springer, 2012. doi:10.1007/978-94-007-4435-6_10.
- 544 10 Martín Escardó and Chuangjie Xu. A constructive manifestation of the Kleene-Kreisel
545 continuous functionals. *Ann. Pure Appl. Log.*, 167(9):770–793, 2016. doi:10.1016/j.apal.
546 2016.04.011.
- 547 11 Martin Hötzel Escardó. Continuity of Gödel’s System T definable functionals via effect-
548 ful forcing. *Proceedings of the Twenty-ninth Conference on the Mathematical Foundations*
549 *of Programming Semantics, MFPS 2013, New Orleans, LA, USA, June 23-25, 2013*, <https://doi.org/10.1016/j.entcs.2013.09.010>. doi:10.1016/j.entcs.2013.09.010.
- 550 12 Martin Hötzel Escardó and Chuangjie Xu. The inconsistency of a brouwerian continu-
551 ity principle with the Curry-Howard interpretation. *13th International Conference on*
552 *Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, <https://doi.org/10.4230/LIPIcs.TLCA.2015.153>. doi:10.4230/LIPIcs.TLCA.2015.153.
- 553 13 Martin Hofmann. *Extensional constructs in intensional type theory*. CPHC/BCS distinguished
554 dissertations. Springer, 1997.
- 555
- 556

- 557 14 J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: i, ii, and III. *Inf. Comput.*,
558 163(2):285–408, 2000. doi:10.1006/inco.2000.2917.
- 559 15 Ambrus Kaposi, András Kovács, and Nicolai Kraus. Shallow embedding of type theory
560 is morally correct. In Graham Hutton, editor, *Mathematics of Program Construction -*
561 *13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings*,
562 volume 11825 of *Lecture Notes in Computer Science*, pages 329–365. Springer, 2019. doi:
563 10.1007/978-3-030-33636-3_12.
- 564 16 Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In Ben Lippmeier, editor,
565 *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC,*
566 *Canada, September 3-4, 2015*, pages 94–105. ACM, 2015. doi:10.1145/2804302.2804319.
- 567 17 S.C. Kleene. Recursive functionals and quantifiers of finite types revisited I. In
568 J.E. Fenstad, R.O. Gandy, and G.E. Sacks, editors, *Generalized Recursion Theory II*,
569 volume 94 of *Studies in Logic and the Foundations of Mathematics*, pages 185–222. Elsevier,
570 1978. URL: <https://www.sciencedirect.com/science/article/pii/S0049237X08709339>,
571 doi:[https://doi.org/10.1016/S0049-237X\(08\)70933-9](https://doi.org/10.1016/S0049-237X(08)70933-9).
- 572 18 G. Kreisel, D. Lacombe, and J.R. Shoenfield. Partial recursive functionals and effective
573 operations. In A. Heyting, editor, *Constructivity in Mathematics*, Studies in Logic and the
574 Foundations of Mathematics, pages 290–297, Amsterdam, 1959. North-Holland. Proc. Colloq.,
575 Amsterdam, Aug. 26–31, 1957.
- 576 19 Jean-Louis Krivine. Opérateurs de mise en mémoire et traduction de Gödel. *Arch. Math. Log.*,
577 30(4):241–267, 1990. doi:10.1007/BF01792986.
- 578 20 Paul Blain Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of
579 *Semantics Structures in Computation*. Springer, 2004.
- 580 21 John Longley and Dag Normann. *Higher-Order Computability*. Theory and Applications of
581 Computability. Springer, 2015. doi:10.1007/978-3-662-47992-6.
- 582 22 Conor McBride. Turing-completeness totally free. In Ralf Hinze and Janis Voigtländer,
583 editors, *Mathematics of Program Construction - 12th International Conference, MPC 2015,*
584 *Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, volume 9129 of *Lecture Notes in*
585 *Computer Science*, pages 257–275. Springer, 2015. doi:10.1007/978-3-319-19797-5_13.
- 586 23 Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types*. Habilitation à
587 diriger des recherches, Université Claude Bernard - Lyon I, December 1996. URL: <https://tel.archives-ouvertes.fr/tel-00431817>.
588
- 589 24 Pierre-Marie Pédro. Russian constructivism in a prefascist theory. In Holger Hermanns,
590 Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS '20: 35th Annual ACM/IEEE*
591 *Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages
592 782–794. ACM, 2020. doi:10.1145/3373718.3394740.
- 593 25 Pierre-Marie Pédro and Nicolas Tabareau. The fire triangle: how to mix substitution,
594 dependent elimination, and effects. *Proc. ACM Program. Lang.*, 4(POPL):58:1–58:28, 2020.
595 doi:10.1145/3371126.
- 596 26 Maciej Piróg and Jeremy Gibbons. The coinductive resumption monad. In Bart Jacobs, Alex-
597 andra Silva, and Sam Staton, editors, *Proceedings of the 30th Conference on the Mathematical*
598 *Foundations of Programming Semantics, MFPS 2014, Ithaca, NY, USA, June 12-15, 2014*,
599 volume 308 of *Electronic Notes in Theoretical Computer Science*, pages 273–288. Elsevier,
600 2014. doi:10.1016/j.entcs.2014.10.015.
- 601 27 Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Log. Methods Comput.*
602 *Sci.*, 9(4), 2013. doi:10.2168/LMCS-9(4:23)2013.
- 603 28 Pierre-Marie Pédro and Nicolas Tabareau. An effectful way to eliminate addiction to
604 dependence. *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*
605 *2017, Reykjavik, Iceland, June 20-23, 2017*, 2017, <https://doi.org/10.1109/LICS.2017.8005113>.
606 doi:10.1109/LICS.2017.8005113.

- 607 29 Pierre-Marie Pédrot and Nicolas Tabareau. Failure is not an option an exceptional type theory.
608 *ESOP 2018 - 27th European Symposium on Programming*, 2018, [https://doi.org/10.1007/978-](https://doi.org/10.1007/978-3-319-89884-1_9)
609 [3-319-89884-1_9](https://doi.org/10.1007/978-3-319-89884-1_9). doi:[https://doi.org/10.1007/978-](https://doi.org/10.1007/978-3-319-89884-1_9)
- 610 30 Vincent Rahli and Mark Bickford. Validating Brouwer’s continuity principle for numbers
611 using named exceptions. *Math. Struct. Comput. Sci.*, 28(6):942–990, 2018. doi:[10.1017/](https://doi.org/10.1017/S0960129517000172)
612 [S0960129517000172](https://doi.org/10.1017/S0960129517000172).
- 613 31 Vincent Rahli, Mark Bickford, Liron Cohen, and Robert L. Constable. Bar induction is
614 compatible with constructive type theory. *J. ACM*, 66(2):13:1–13:35, 2019. doi:[10.1145/](https://doi.org/10.1145/3305261)
615 [3305261](https://doi.org/10.1145/3305261).
- 616 32 Egbert Rijke, Michael Shulman, and Bas Spitters. Modalities in homotopy type theory.
617 *Logical Methods in Computer Science*, Volume 16, Issue 1, January 2020. URL: [https:](https://lmcs.episciences.org/6015)
618 [//lmcs.episciences.org/6015](https://lmcs.episciences.org/6015), doi:[10.23638/LMCS-](https://doi.org/10.23638/LMCS-16(1:2)2020)
619 [16\(1:2\)2020](https://doi.org/10.23638/LMCS-16(1:2)2020).
- 620 33 Pierre-Marie Pédrot Simon Boulier and Nicolas Tabareau. The next 700 syntactical models of
621 type theory. *Certified Programs and Proofs (CPP 2017), Jan 2017, Paris, France. pp.182 -*
622 *194*, <https://doi.org/10.1016/j.entcs.2013.09.010>. doi:[10.1145/3018610.3018620](https://doi.org/10.1145/3018610.3018620).
- 623 34 Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in Coq. In Gerwin Klein
624 and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference,*
625 *ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July*
626 *14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514.
627 Springer, 2014. doi:[10.1007/978-3-319-08970-6_32](https://doi.org/10.1007/978-3-319-08970-6_32).
- 628 35 Jonathan Sterling. Higher order functions and Brouwer’s thesis. *Journal of Functional*
629 *Programming*, 31:e11, 2021. *Bob Harper Festschrift Collection*. arXiv:1608.03814, doi:
630 [10.1017/S0956796821000095](https://doi.org/10.1017/S0956796821000095).
- 631 36 Wouter Swierstra. Data types à la carte. *J. Funct. Program.*, 18(4):423–436, 2008. doi:
632 [10.1017/S0956796808006758](https://doi.org/10.1017/S0956796808006758).
- 633 37 G. S. Tseitin. Algorithmic operators in constructive metric spaces. In *Problems of the*
634 *constructive direction in mathematics. Part 2. Constructive mathematical analysis*, volume 67,
635 pages 295–361, Moscow–Leningrad, 1962. USSR Academy of Sciences.
- 636 38 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of*
637 *Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013.
- 638 39 Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce,
639 and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq.
640 *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. doi:[10.1145/3371119](https://doi.org/10.1145/3371119).
- 641 40 Chuangjie Xu and Martín Hötzel Escardó. A constructive model of uniform continuity.
642 In Masahito Hasegawa, editor, *Typed Lambda Calculi and Applications, 11th International*
643 *Conference, TLCA 2013, Eindhoven, The Netherlands, June 26-28, 2013. Proceedings*, volume
644 7941 of *Lecture Notes in Computer Science*, pages 236–249. Springer, 2013. doi:[10.1007/](https://doi.org/10.1007/978-3-642-38946-7_18)
[978-3-642-38946-7_18](https://doi.org/10.1007/978-3-642-38946-7_18).