



**HAL**  
open science

# Complete Bidirectional Typing for the Calculus of Inductive Constructions

Meven Lennon-Bertrand

► **To cite this version:**

Meven Lennon-Bertrand. Complete Bidirectional Typing for the Calculus of Inductive Constructions. 2021. hal-03139924v1

**HAL Id: hal-03139924**



**<https://hal.science/hal-03139924v1>**

Preprint submitted on 12 Feb 2021 (v1), last revised 19 Apr 2021 (v2)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Complete Bidirectional Typing for the Calculus of Inductive Constructions

Meven Lennon-Bertrand  

LS2N, Université de Nantes — Gallinette Project Team, Inria, France

## Abstract

This article presents a bidirectional type system for the Calculus of Inductive Constructions (CIC). It introduces a novel judgement intermediate between the usual inference and checking, dubbed constrained inference, to handle the presence of computation in types. The key property is the completeness of the system with respect to the usual undirected one, which has been formally proven in Coq as a part of the MetaCoq project. Although it plays a central role in an ongoing completeness proof for a realistic typing algorithm, the interest of bidirectionality is much wider, as it clarifies previous works in the area and gives strong insights and structure when trying to prove properties on CIC or design variations and extensions.

**2012 ACM Subject Classification** Theory of computation → Type theory

**Keywords and phrases** Bidirectional Typing, Calculus of Inductive Constructions, Coq, Proof Assistants

**Digital Object Identifier** 10.4230/LIPIcs.CVIT.2016.23

**Supplementary Material** *Software (Formalization)*: <https://github.com/MevenBertrand/metacoq/tree/itp-artefact>

## 1 Introduction

In logical programming, a very important information about judgements is the *mode* of the objects involved, i.e., which ones are considered inputs or outputs. When examining this distinction for a typing judgement  $\Gamma \vdash t : T$ , both the term  $t$  under inspection and the context  $\Gamma$  of this inspection are known, they are thus inputs. The mode of the type  $T$ , however, is much less clear: should it be inferred based upon  $\Gamma$  and  $t$ , or do we merely want to check whether  $t$  conforms to a given  $T$ ? Both are sensible approaches, and in fact typing algorithms for complex type systems usually alternate between them during the inspection of a single term/program. The bidirectional approach makes this difference between modes explicit, by decomposing undirected<sup>1</sup> typing  $\Gamma \vdash t : T$  into two separate judgments  $\Gamma \vdash t \triangleright T$  (inference) and  $\Gamma \vdash t \triangleleft T$  (checking)<sup>2</sup>, that differ only by modding. This decomposition allows theoretical work on practical typing algorithms, but also gives a finer grained structure to typing derivations, which can be of purely theoretical interest even without any algorithm in sight.

Although those seem appealing, and despite advocacy by McBride [9, 10] to adopt this approach when designing type systems, most of the dependent typing world to this day remains undirected. Some others than McBride’s appeal to bidirectionality, starting with Coquand [7] and continuing with Norell [12] or Abel [1]. However, all of these consider unannotated  $\lambda$ -abstractions. This lack of annotations, although sensible for lightness, poses an inherent completeness problem, as a term like  $(\lambda x.x) 0$  does not type-check against type  $\mathbb{N}$  in those systems. Very few have considered the case of annotated abstractions, apart

<sup>1</sup> We call anything related to the  $\Gamma \vdash t : T$  judgement undirected by contrast with the bidirectional typing.

<sup>2</sup> We chose  $\triangleright$  and  $\triangleleft$  rather than the more usual  $\Rightarrow$  and  $\Leftarrow$  to avoid confusion with implication on paper, and with the Coq notation for functions in the development.



© Meven Lennon-Bertrand;

licensed under Creative Commons License CC-BY 4.0

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:17

Leibniz International Proceedings in Informatics



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

41 from Asperti and the Matita team [3], who however concentrate mostly on specific problems  
42 pertaining to unification and implementation of the Matita elaborator, without giving a  
43 general bidirectional framework. They also do not consider the problem of completeness with  
44 respect to a given undirected system, as it would fail in their setting due to the undecidability  
45 of higher order unification.

46 Thus, we wish to fill a gap in the literature, by describing a bidirectional type system that  
47 is complete with respect to the (undirected) Calculus of Inductive Constructions (CIC). By  
48 completeness, we mean that any term that is typable in the undirected system should also infer  
49 a type in the bidirectional one. This feature is very desirable when implementing kernels for  
50 proof assistants, whose algorithms should correspond to their undirected specification, never  
51 missing any typable term. The bidirectional systems we describe thus forms intermediates  
52 between actual algorithms and undirected type systems. This step has proven useful in an  
53 ongoing completeness proof of MetaCoq's [17] type-checking algorithm<sup>3</sup>: rather than proving  
54 the algorithm complete directly, the idea is to prove it equivalent to the bidirectional type  
55 system, separating the implementation problems from the ones regarding the bidirectional  
56 structure.

57 But having a bidirectional type system equivalent to the undirected one has other purely  
58 theoretical interests. First, the structure of a bidirectional derivation is more constrained  
59 than that of an undirected one, especially regarding the uses of computation. This finer  
60 structure can make proofs easier, while the equivalence ensures they can be transported to  
61 the undirected world. For instance, in a setting with cumulativity/subtyping, the inferred  
62 type for a term  $t$  should by construction be smaller than any other types against which  $t$   
63 checks. This provides an easy proof of the existence of principal types in the undirected  
64 system. The bidirectional structure also provides a better base for extensions. This was  
65 actually the starting point for this investigation: in [8], we quickly describe a bidirectional  
66 variant of CIC, as the usual undirected CIC is unfit for the gradual extension we envision  
67 due to the too high flexibility of a free-standing conversion rule. This is the system we wish  
68 to thoroughly describe and investigate here.

69 Our main technical contributions are twofold. First the identification of a new constrained  
70 inference judgement introduced in Section 2 together with general ideas around bidirectional  
71 typing in the rather simple setting of pure type systems. Secondly, a formalized proof of  
72 equivalence<sup>4</sup> between PCUIC – the extension of CIC nowadays at the heart of Coq – and a  
73 bidirectional type system described on a high level in Section 3, built on top of MetaCoq.  
74 We next turn to less technical considerations, as we believe that the bidirectional structure  
75 is of general interest. Section 4 thus describes the interest of basing an extension of CIC  
76 on the bidirectional system directly rather than on the equivalent undirected one. Finally  
77 Section 5 investigates in length the related work, and in particular identifies the implicit  
78 presence of the bidirectional structure in various earlier articles, showing how making this  
79 structure explicit clarifies those.

---

<sup>3</sup> A completeness bug in that algorithm – also present in the Coq kernel – has already been found, see Section 3 for details.

<sup>4</sup> A version frozen as described in this article is available in the following git branch: <https://github.com/MevenBertrand/metacoq/tree/itp-artefact>.

$\boxed{\vdash \Gamma}$ 

$$\frac{}{\vdash \cdot} \text{EMPTY} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash A : \square_i}{\vdash \Gamma, x : A} \text{EXT}$$

 $\boxed{\Gamma \vdash t : T}$ 

$$\begin{array}{c} \frac{\vdash \Gamma}{\Gamma \vdash \square_i : \square_{i+1}} \text{SORT} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash A : \square_i}{\Gamma, x : A \vdash x : A} \text{VAR} \qquad \frac{\Gamma \vdash x : A \quad \Gamma \vdash B : \square_i}{\Gamma, y : B \vdash x : A} \text{WEAK} \\ \\ \frac{\Gamma \vdash A : \square_i \quad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Pi x : A.B : \square_{i \vee j}} \text{PROD} \qquad \frac{\Gamma \vdash \Pi x : A.B : \square_i \quad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B} \text{ABS} \\ \\ \frac{\Gamma \vdash t : \Pi x : A.B \quad \Gamma \vdash u : A}{\Gamma \vdash t u : B[x := u]} \text{APP} \qquad \frac{\Gamma \vdash t : A \quad \Gamma \vdash B : \square_i \quad A \equiv B}{\Gamma \vdash t : B} \text{CONV} \end{array}$$

■ **Figure 1** Undirected typing for  $\text{CC}\omega$  – PTS-style

## 80 2 Warming up with $\text{CC}\omega$

### 81 2.1 Undirected $\text{CC}\omega$

82 As a starting point, let us consider  $\text{CC}\omega$ . It is the backbone of CIC, and we can already  
 83 illustrate most of our methodology on it.  $\text{CC}\omega$  belongs to the wider class of pure type systems  
 84 (PTS), that has been thoroughly studied and described, see for instance [4]. Since there are  
 85 many presentational variations, let us first give a precise account of our conventions. *Terms*  
 86 in  $\text{CC}\omega$  are given by the grammar

$$87 \quad t ::= x \mid \square_i \mid \Pi x : t.t \mid \lambda x : t.t \mid t t$$

88 where the letter  $x$  denotes a variable (so will letters  $y$  and  $z$ ), and the letter  $i$  is an integer  
 89 (we will also use letters  $j$ ,  $k$  and  $l$  for those). All other Latin letters will be used for terms,  
 90 with the upper-case ones used to suggest the corresponding terms should be thought of as  
 91 types — although this is not a syntactical separation. We abbreviate  $\Pi x : A.B$  by  $A \rightarrow B$   
 92 when  $B$  does not depend on  $x$ , as is customary. On those terms, *reduction*  $\rightarrow$  is defined as  
 93 the least congruence such that  $(\lambda x : T.t) u \rightarrow t[x := u]$ , where  $t[x := u]$  denotes *substitution*.  
 94 *Conversion*  $\equiv$  is the symmetric, reflexive, transitive closure of reduction. Finally, *contexts* are  
 95 lists of variable declarations  $x : t$  and are denoted using capital Greek letters. We write  $\cdot$  for  
 96 the empty list,  $\Gamma, x : T$  for concatenation, and  $(x : T) \in \Gamma$  if  $(x : T)$  appears in  $\Gamma$ . Combining  
 97 those, we can define *typing*  $\Gamma \vdash t : T$  as in Figure 1, where  $i \vee j$  denotes the maximum of  $i$  and  
 98  $j$ . We use *well-formed* to denote  $\vdash \Gamma$  for a context, the existence of  $i$  such that  $\Gamma \vdash T : \square_i$   
 99 for a type  $T$  (in an implicit context), or the existence of  $T$  such that  $\Gamma \vdash t : T$  for a term  $t$   
 100 (again in an implicit context). We also say  $t$  is *well-typed* for the latter.

101 As any PTS,  $\text{CC}\omega$  has many desirable properties. We summarize the ones we need here,  
 102 see [4] for proofs.

103 ► **Proposition 1** (Properties of  $CC\omega$ ). *The type system  $CC\omega$  as just described enjoys the*  
 104 *following properties:*

105 **Confluence** *Reduction  $\rightarrow$  is confluent. As a direct consequence, two terms are convertible*  
 106 *just when they have a common reduct:  $t \equiv u$  if only if there exists  $t'$  such that  $t \rightarrow^* t'$*   
 107 *and  $u \rightarrow^* t'$ .*

108 **Transitivity** *Conversion is transitive.*

109 **Subject reduction** *If  $\Gamma \vdash t : T$  and  $t \rightarrow t'$  then  $\Gamma \vdash t' : T$ .*

110 **Validity** *If  $\Gamma \vdash t : T$  then  $T$  is well-formed, e.g. there exists some  $i$  such that  $\Gamma \vdash T : \square_i$ .*

## 111 2.2 Turning $CC\omega$ Bidirectional

112 **McBride’s discipline.** To design our bidirectional type system, we follow a discipline exposed  
 113 by McBride [9, 10]. The central point is to distinguish in a judgment between the subject,  
 114 whose well-formedness is under scrutiny, from inputs, whose well-formedness is a condition  
 115 for the judgment to behave well, and outputs, whose well-formedness is a consequence of  
 116 the judgment. For instance, in inference  $\Gamma \vdash t \triangleright T$ , the subject is  $t$ ,  $\Gamma$  is an input and  $T$  is  
 117 an output. This means that one should consider whether  $\Gamma \vdash t \triangleright T$  only in cases where  $\vdash \Gamma$   
 118 is already known, and if the judgment is derivable it should be possible to conclude that  
 119 both  $t$  and  $T$  are well-formed. All inference rules are to preserve this invariant. This means  
 120 that inputs to a premise should be well-formed whenever the inputs to the conclusion and  
 121 outputs and subjects of previous premises are. Similarly the outputs of the conclusion should  
 122 be well-formed if the inputs of the conclusion and the subjects and outputs of the premises  
 123 are assumed to be so.

124 This distinction also applies to the computation-related judgments, although those have  
 125 no subject. For conversion testing  $T \equiv T'$  both  $T$  and  $T'$  are inputs, and thus should be  
 126 known to be well-formed beforehand. For reduction  $T \rightarrow^* T'$ ,  $T$  is an input and  $T'$  is an  
 127 output, so only  $T$  needs to be well-formed, with the subject reduction property of the system  
 128 ensuring that the output  $T'$  is also well-formed.

129 **Constrained inference.** Beyond the already described inference and checking judgements  
 130 another one appears in the bidirectional typing rules of Figure 2: *constrained inference*,  
 131 written  $\Gamma \vdash t \triangleright_h T$ , where  $h$  is either  $\Pi$  or  $\square$  – and will be extended once we introduce  
 132 inductive types. Constrained inference is a judgement (or, rather, a family of judgements  
 133 indexed by  $h$ ) with the exact same modding as inference, but where the type output is not  
 134 completely free. Rather, as the name suggests, a constraint is imposed on it, namely that  
 135 its head constructor can only be the corresponding element of  $h$ . This is useful to handle  
 136 the behaviour absent in simple types that some terms might not have a desired type “on  
 137 the nose”. This is exemplified by the first premise  $\Gamma \vdash t \triangleright_{\Pi} \Pi x : A.B$  of the APP rule for  $t u$ .  
 138 Indeed, it would be too much to ask  $t$  to directly infer a  $\Pi$ -type, as some reduction might  
 139 be needed on  $T$  to uncover this  $\Pi$ . Checking also cannot be used, because the domain and  
 140 codomain of the tentative  $\Pi$ -type are not known at that point: they are to be inferred from  $t$ .

141 **Structural rules.** To transform the rules of Figure 1 to those of Figure 2, we start by  
 142 recalling that we wish to present a obtain bidirectional typing. Therefore any term should  
 143 infer a type, and thus all structural rules (i.e. all rules where the subject of the conclusion  
 144 starts with a term constructor) should give rise to an inference rule. It thus remains to chose  
 145 the judgements for the premises, which amounts to choosing how to mod them. If a term  
 146 in a premise appears as input in the conclusion or output of a previous premise, then it

Inference:  $\Gamma \vdash t \triangleright T$

$$\frac{}{\Gamma \vdash \square_i \triangleright \square_{i+1}} \text{SORT} \quad \frac{(x : T) \in \Gamma}{\Gamma \vdash x \triangleright T} \text{VAR} \quad \frac{\Gamma \vdash A \triangleright_{\square} \square_i \quad \Gamma, x : A \vdash B \triangleright_{\square} \square_j}{\Gamma \vdash \Pi x : A.B \triangleright_{\square_{i \vee j}}} \text{PROD}$$

$$\frac{\Gamma \vdash A \triangleright_{\square} \square_i \quad \Gamma, x : A \vdash t \triangleright B}{\Gamma \vdash \lambda x : A.t \triangleright \Pi x : A.B} \text{ABS} \quad \frac{\Gamma \vdash t \triangleright_{\Pi} \Pi x : A.B \quad \Gamma \vdash u \triangleleft A}{\Gamma \vdash t u \triangleright B[x := u]} \text{APP}$$

Checking:  $\Gamma \vdash t \triangleleft T$

$$\frac{\Gamma \vdash t \triangleright T' \quad T' \equiv T}{\Gamma \vdash t \triangleleft T} \text{CHECK}$$

Constrained inference:  $\Gamma \vdash t \triangleright_h T$

$$\frac{\Gamma \vdash t \triangleright T \quad T \rightarrow^* \square_i}{\Gamma \vdash t \triangleright_{\square} \square_i} \text{SORT-INF} \quad \frac{\Gamma \vdash t \triangleright T \quad T \rightarrow^* \Pi x : A.B}{\Gamma \vdash t \triangleright_{\Pi} \Pi x : A.B} \text{PROD-INF}$$

■ **Figure 2** Bidirectional typing for  $CC\omega$

147 can be considered an input, otherwise it must be an output. Moreover, if a type output is  
148 unconstrained, then inference can be used, otherwise we must resort to constrained inference.

149 This applies straightforwardly to most rules but the PST-style ABS rule. Indeed, if one  
150 looks at the undirected premises, the premise  $\Gamma \vdash \Pi x : A.B : \square_i$  needs  $A$  and  $B$  to be known,  
151 and only  $A$  is known from the conclusion, thus it cannot be the first premise. However, one  
152 also cannot put  $\Gamma, x : A \vdash t : B$  as the first premise, because  $A$  is not known to be well-formed  
153 at that point, thus  $\Gamma, x : A$  cannot be used as an input. The solution is to split the premise  
154  $\Gamma \vdash \Pi x : A.B : \square_i$  into the equivalent  $\Gamma \vdash A : \square_j$  and  $\Gamma, x : A \vdash B : \square_{j'}$ . The former can  
155 become the first premise, ensuring that type inference for  $t$  is done in a well-formed context,  
156 and the latter can be simply dropped based upon our invariant that outputs – here the type  
157  $B$  inferred for  $t$  – can be assumed to be well-formed.

158 Similarly, as the context is always supposed to be well-formed as an input to the conclusion,  
159 it is not useful to re-check it, and thus the premise to VAR can be dropped, and undirected  
160 rules VAR and WEAK can be fused into one single VAR. This is in line with implementations,  
161 where the context is not re-checked at leaves of a derivation tree, with performance issues in  
162 mind. The well-formedness invariants ensure that any derivation starting with the empty  
163 context will only use well-formed contexts.

164 **Computation rules.** We are now left with the non-structural conversion rule. As we  
165 observed, there are two ways to mode computation: if both sides are inputs, conversion can  
166 be used, but if only one is known one must resort to conversion, and the other side becomes  
167 an output instead. Rule CHECK corresponds to the first case, while rules PROD-INF and  
168 SORT-INF both are in the second case. This difference in turn introduces the need to separate  
169 between checking, that calls for the first rule, and constrained inference, that requires the  
170 others.

171 To the best of our knowledge, this difference in modding of conversion and the resulting  
 172 introduction of constrained inference have never been described on paper, although they  
 173 appear in the typing and elaboration algorithms of proof assistants based upon dependent  
 174 type theory, such as Coq, Lean or Agda. Instead, in presentations in print, constrained  
 175 inference has been inlined in some way, as is also often the case for checking, so that  $\Gamma \vdash t : T$   
 176 is used where we use  $\Gamma \vdash t \triangleright T$ , the bidirectional structure being left implicit. This is sensible  
 177 since in Figure 2 there is only one rule to derive checking and constrained inference. However,  
 178 as soon as typing features appear that complicate conversion, such as unification [3], coercions  
 179 [3, 16] or graduality [8], having singled out those judgements makes the structure clearer and  
 180 explains the choices made for the modification of typing that could appear ad-hoc otherwise.  
 181 We come back to this more in length in Section 5.1.

## 182 2.3 Properties

183 Let us now state the two main properties relating the bidirectional system to the undirected  
 184 one: it is both correct (terms typable in the bidirectional system are typable in the undirected  
 185 system) and complete (all terms typable in the undirected system are also typable in the  
 186 bidirectional system).

### 187 2.3.1 Correctness

188 A bidirectional derivation can be seen as a refinement of an undirected derivation. Indeed, the  
 189 bidirectional structure can be erased – replacing each bidirectional rule with the corresponding  
 190 undirected rule – to obtain an undirected derivation, but for missing sub-derivations, which  
 191 can be retrieved using the invariants on well-formedness of inputs and outputs. Thus,  
 192 we get the following correctness theorem – note how McBride’s discipline manifests as  
 193 well-formedness hypothesis on inputs.

194 ► **Theorem 2** (Correctness of bidirectional typing for  $CC\omega$ ). *If  $\Gamma$  is well-formed and  $\Gamma \vdash t \triangleright T$   
 195 or  $\Gamma \vdash t \triangleright_h T$  then  $\Gamma \vdash t : T$ . If  $\Gamma$  and  $T$  are well-formed and  $\Gamma \vdash t \triangleleft T$  then  $\Gamma \vdash t : T$ .*

196 **Proof.** The proof is by mutual induction on the bidirectional typing derivation.

197 Each rule of the bidirectional system can be replaced by the corresponding rule of the  
 198 undirected system, with all three CHECK, PROD-INF and SORT-INF replaced by CONV, ABS  
 199 using an extra PROD rule, and VAR using a succession of WEAK and a final VAR. In all  
 200 cases, the induction hypothesis can be used on sub-derivations of the bidirectional judgment  
 201 because the context is extended using types that are known to be well-formed, and similarly  
 202 checking is done against a type that is known to be well-formed by previous premises.

203 Some sub-derivations of the undirected rules that have no counterpart in the bidirectional  
 204 ones are however missing. In rules SORT and VAR the hypothesis that  $\vdash \Gamma$  is enough to  
 205 get the required premise. For rule CHECK, the well-formedness hypothesis on the type is  
 206 needed to get the second premise of rule CONV. As for PROD-INF and SORT-INF, that second  
 207 premise is obtained by subject reduction. Finally, the missing premise on the codomain of  
 208 the product type in rule ABS is obtained by validity of the undirected system, but could be  
 209 instead handled by strengthening the theorem to incorporate the well-formedness of types  
 210 when they are outputs.

### 2.3.2 Completeness

Let us now state the most important property of our bidirectional system: it does not miss any undirected derivation.

► **Theorem 3** (Completeness of bidirectional typing for  $CC\omega$ ). *If  $\Gamma \vdash t : T$  then there exists  $T'$  such that  $\Gamma \vdash t \triangleright T'$  and  $T' \equiv T$ .*

**Proof.** The proof is by induction on the undirected typing derivation.

Rules SORT and VAR are base cases, and can be replaced by the corresponding rules in the bidirectional world. Rules WEAK and CONV are both direct consequences of the induction hypothesis on their first premise, together with transitivity of conversion for the latter.

For rule PROD, we need the intermediate lemma that if  $T$  is a term such that  $T \equiv \square_i$ , then also  $T \rightarrow^* \square_i$ . This is a consequence of confluence of reduction. In turn, it implies that if  $\Gamma \vdash t \triangleright T$  and  $T \equiv \square_i$  then  $\Gamma \vdash t \triangleright_{\square} \square_i$ , and is enough to conclude for that rule.

In rule ABS, the induction hypothesis gives  $\Gamma \vdash \Pi x : A.B \triangleright T$  for some  $T$ , and an inversion on this gives  $\Gamma \vdash A \triangleright_{\square} \square_i$  for some  $i$ . Combined with the second induction hypothesis, it gives  $\Gamma \vdash \lambda x : A.t \triangleright \Pi x : A.B'$  for some  $B'$  such that  $B \equiv B'$ , and thus  $\Pi x : A.B \equiv \Pi x : A.B'$  as desired.

We are finally left with the APP rule. We know that  $\Gamma \vdash t \triangleright T$  with  $T \equiv \Pi x : A.B$ . Confluence then implies that  $T \rightarrow^* \Pi x : A'.B'$  for some  $A'$  and  $B'$  such that  $A \equiv A'$  and  $B \equiv B'$ . Thus  $\Gamma \vdash t \triangleright_{\Pi} \Pi x : A'.B'$ . But by induction hypothesis we also know that  $\Gamma \vdash u \triangleright A''$  with  $A'' \equiv A$  and so by transitivity of conversion  $\Gamma \vdash u \triangleleft A'$ . We can thus apply APP to conclude.

Contrarily to correctness, which kept a similar derivation structure, completeness is of a different nature. Because in bidirectional derivations the conversion rules are much less liberal than in undirected derivations, the bulk of the proof is to ensure that conversions can be permuted with structural rules, in order to concentrate them in the places where they are authorized in the bidirectional derivation. In a way, composing completeness with conversion gives a kind of normalization procedure that produces a canonical undirected derivation by pushing all conversions down as much as possible.

### 2.3.3 Reduction strategies

The judgements of Figure 2 are syntax-directed, in the sense that there is always at most one rule that can be used to derive a certain typing judgement. But with the rules as given there is still some indeterminacy. Indeed when appealing to reduction no strategy is fixed, thus two different reducts give different uses of the rule, resulting in different inferred types – although those are still convertible. However, a reduction strategy can be imposed to completely eliminate indeterminacy in typing, leading to the following.

► **Proposition 4** (Reduction strategy). *If  $\rightarrow^*$  is replaced by weak-head reduction in rules SORT-INF and PROD-INF, then given a well-formed context  $\Gamma$  and a term  $t$  there is at most one derivation of  $\Gamma \vdash t \triangleright T$  and  $\Gamma \vdash t \triangleright_h T$ , and so in particular such a  $T$  is unique. Similarly, given well-formed  $\Gamma$  and  $T$  and a term  $t$  there is at most one derivation of  $\Gamma \vdash t \triangleleft T$ . Moreover, the existence of those derivations is decidable.*

The algorithm for deciding the existence of the derivations is straightforward from the modded rules, it amounts to structural recursion on the subject.



### 256 **3 From $CC\omega$ to PCUIC**

257  $CC\omega$  is already a powerful system, but today’s proof assistants rely on much more complex  
 258 features. The Predicative Calculus of Cumulative Inductive Constructions (PCUIC), the type  
 259 theory nowadays behind the Coq proof assistant, for instance features the impredicative sort  
 260 Prop, the sort  $SProp$  of irrelevant propositions, algebraic universes, cumulativity, polymorphic  
 261 and mutual inductive and co-inductive types, (co-)fixpoints, primitive projections. . . This is  
 262 a good stress test for the bidirectional approach: being able to adapt seamlessly to those  
 263 features is a good sign that the methodology we presented should be able to handle other  
 264 extensions. In this section, we present some modifications and additions to the system of  
 265 Section 2 needed to treat the most usual features of PCUIC.

266 Bidirectional judgments incorporating the elements described in this section have been  
 267 formally proven correct<sup>5</sup> and complete<sup>6</sup> with respect to the description of PCUIC in the  
 268 MetaCoq project [17]. While working on this, we were able to uncover an incompleteness bug  
 269 in the current kernel of Coq regarding pattern-matching of cumulative inductive types. This  
 270 bug had gone unnoticed until our formalization, but was causing subject reduction failures  
 271 in corner cases with inductive types<sup>7</sup>.

272 As a demonstration of the use of bidirectionality for reasoning, the formalization also  
 273 contains a proof of the uniqueness of inferred types and of the existence of principal types as  
 274 a direct corollary.<sup>8</sup>

#### 275 **3.1 Cumulativity**

PCUIC incorporates a limited form of subtyping. Conversion  $\equiv$  is replaced by *cumulativity*  
 $\preceq$ , a very similar relation, but with the difference that it relaxes the constraint on universes:  
 for conversions  $\square_i \equiv \square_j$  only when  $i = j$ , but for cumulativity  $\square_i \preceq \square_j$  whenever  $i \leq j$ . The  
 conversion rule is accordingly replaced by the following cumulativity rule

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash B : \square_i \quad A \preceq B}{\Gamma \vdash t : B} \text{CUMUL}$$

276 This reflects the view that universes  $\square_i$  should be included one in the next when going up in  
 277 the hierarchy. In  $CC\omega$ , all types for a given term  $t$  in a fixed context  $\Gamma$  are equally good, as  
 278 they are all convertible. This is not the case any more in presence of cumulativity, as we can  
 279 have  $T \preceq T'$  but not  $T \equiv T'$ . Of particular interest are principal types, defined as follows.

280 **► Definition 5 (Principal type).** *The term  $T$  is called a principal type for term  $t$  in context*  
 281  *$\Gamma$  if it is a least type for  $t$  in  $\Gamma$ , that is if  $\Gamma \vdash t : T$  and for any  $T'$  such that  $\Gamma \vdash t : T'$  we*  
 282 *have  $T \preceq T'$ .*

283 The existence of such principal types is no so easy to prove directly but quite useful, as  
 284 they are in a sense the best types for any terms. Indeed, if  $T$  is a principal type for  $t$  in  
 285  $\Gamma$  and  $T'$  is any other type for  $t$ , the CUMUL rule can be used to deduce  $\Gamma \vdash t : T'$  from  
 286  $\Gamma \vdash t : T$ , which in general is not the case if  $T$  is not principal. Similarly, if  $T$  and  $T'$  are two  
 287 types for a term  $t$ , then they are not directly related, but the existence of principal types

<sup>5</sup> The formalized theorem is at line 419 and following of BDTToPCUIC.v.

<sup>6</sup> The formalized theorem is at line 387 and following of BDFromPCUIC.v.

<sup>7</sup> The precise technical problem is described in the following git issue: <https://github.com/coq/coq/issues/13495>.

<sup>8</sup> The corresponding theorems are respectively at line 347 and 355 of BDUnique.v.

288 ensures that there exists some  $T''$  that is a type for  $t$  and such that  $T \preceq T'$  and  $T \preceq T''$ ,  
 289 indirectly relating  $T'$  and  $T''$ .

Reflecting this modification in the bidirectional system of course calls for an update to the computation rules. The change to the CHECK rule is direct: simply replace conversion with cumulativity:

$$\frac{\Gamma \vdash t \triangleright A \quad A \preceq B}{\Gamma \vdash t \triangleleft B} \text{CUMUL}$$

290 As to the constrained inference rules, there is no need to modify them. Intuitively, this is  
 291 because there is no reason to degrade a type to a larger one when it is not needed. We  
 292 only resort to cumulativity when it is forced by a given input. In that setting, completeness  
 293 becomes the following:

294 ► **Theorem 6** (Completeness with cumulativity). *If  $\Gamma \vdash t : T$  using rules of Figure 1 replacing*  
 295 *CONV with CUMUL, then  $\Gamma \vdash t \triangleright T'$  is derivable with rules of Figure 2 replacing CHECK with*  
 296 *CUMUL for some  $T'$  such that  $T' \preceq T$ .*

297 In that setting, even without fixing a reduction strategy as in Proposition 4, there  
 298 is a weaker uniqueness property for inference types, that is vacuous in a setting without  
 299 cumulativity, where all types are convertible.

300 ► **Proposition 7** (Uniqueness of inferred type). *If  $\Gamma$  is well-formed,  $\Gamma \vdash t \triangleright T$  and  $\Gamma \vdash t \triangleright T'$*   
 301 *then  $T \equiv T'$ . Similarly if  $\Gamma$  is well-formed,  $\Gamma \vdash t \triangleright_h T$  and  $\Gamma \vdash t \triangleright_h T'$  then  $T \equiv T'$ .*

302 **Proof.** Mutual induction on the first derivation. It is key that constrained inference rules  
 303 only reduce a type, so that the type in the conclusion is convertible to the type in the premise,  
 304 rather than merely in cumulativity relation. ◀

305 In particular, those two properties with a correctness property akin to Theorem 2, we can  
 306 prove that any inferred type is principal, and so that they both exist and are computable since  
 307 the bidirectional judgement can still be turned into an algorithm in the spirit of Proposition 4.

308 ► **Proposition 8** (Principal types). *If  $\Gamma$  is well-formed and  $\Gamma \vdash t \triangleright T$  then  $T$  is a principal*  
 309 *type for  $t$  in  $\Gamma$ .*

310 **Proof.** If  $\Gamma \vdash t : T'$ , then by completeness there exists some  $T''$  such that  $\Gamma \vdash t \triangleright T''$  and  
 311 moreover  $T'' \preceq T'$ . But by uniqueness  $T \equiv T'' \preceq T'$  and thus  $T \preceq T'$ , and  $T$  is indeed a  
 312 principal type for  $t$  in  $\Gamma$ . ◀

313 Reasoning on the bidirectional derivation thus makes proofs easier, with the correctness  
 314 and completeness properties ensure they can be carried to the undirected system. Another  
 315 way to understand this is that seeing completeness followed by correction as a normalization  
 316 procedure on derivations, the produced canonical derivation is more structured and thus  
 317 more amenable to proofs. Here for instance the uniqueness of the inferred type translates to  
 318 the existence of principal types via completeness, and the normalization of the derivations  
 319 optimizes it to derive a principal type.

## 320 3.2 Inductive Types

321 **Sum type.** Before we turn to the general case of inductive types of the formalization, let us  
 322 present a simple inductive type: dependent sums. The undirected rules are given in Figure 3,  
 323 and are inspired from the theoretical presentation of such dependent sums, such at the one

$$\begin{array}{c}
\frac{\Gamma \vdash A : \square_i \quad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Sigma x : A.B : \square_{i \vee j}} \Sigma\text{-TYPE} \\
\\
\frac{\Gamma \vdash A : \square_i \quad \Gamma, x : A \vdash B : \square_j \quad \Gamma \vdash a : A \quad \Gamma \vdash b : B[x := a]}{\Gamma \vdash (a, b)_{A, x.B} : \Sigma x : A.B} \Sigma\text{-CONS} \\
\\
\frac{\Gamma, z : \Sigma x : A.B \vdash P : \square_i \quad \Gamma, x : A, y : B \vdash b : P[z := (x, y)] \quad \Gamma \vdash s : \Sigma x : A.B}{\Gamma \vdash \text{rec}_{\Sigma}(z.P, x.y.p, s) : P[z := s]} \Sigma\text{-REC}
\end{array}$$

■ **Figure 3** Undirected sum type

324 of the Homotopy Type Theory book [19]. In particular, we use the same convention to  
325 write  $y.P$  when variable  $y$  is bound in  $P$ . Note however that contrarily to [19], some typing  
326 information is kept on the pair constructor. Exactly as for the abstraction, this is to be  
327 able to infer a unique, most general type in the bidirectional system. Indeed, without that  
328 information a pair  $(a, b)$  could inhabit multiple types  $\Sigma x : A.B$  because there are potentially  
329 many incomparable types  $B$  such that  $B[x := a]$  is a type for  $b$ , as even if  $B[x := a]$  and  
330  $B'[x := a]$  are convertible  $B$  and  $B'$  may be quite different, depending of which instances of  
331  $a$  in  $B[x := a]$  are abstracted to  $x$ .

332 To obtain the bidirectional rules of Figure 4, first notice that all undirected rules are  
333 structural and must thus become inference rules if we want the resulting system to be  
334 complete, just as in Section 2. Thus the question is which modes to choose for the premises.  
335 For  $\Sigma\text{-TYPE}$  and  $\Sigma\text{-CONS}$  this is straightforward: when the type appears in the conclusion,  
336 use checking, otherwise (constrained) inference. The case of the destructors is somewhat  
337 more complex. Handling the subterms of the the destructor in the order in which they usually  
338 appear (predicate, branches and finally scrutinee) is not possible, as the context parameters  
339 of the inductive type are needed to construct the context for the predicate. However those  
340 can be inferred from the scrutinee. Thus, a type for the scrutinee is obtained first using a  
341 new constrained inference judgment, forcing the inferred type to be a  $\Sigma$ -type, but leaving  
342 its parameters free. Next, the obtained arguments can be used to construct the context to  
343 type the predicate. Finally, once the predicate is known to be well-formed, it can be used to  
344 type-check the branch.

345 This same approach can be readily extended to other usual inductive types, with recursion  
346 or indices posing no specific problems, see Figure 5.

347 **Polymorphic, Cumulative Inductive Types.** The account of inductive types in PCUIC is  
348 quite different from the one we just gave. On the theoretical side, the main addition is  
349 universe polymorphism [18], which means that inductive types and constructors come with  
350 explicit universe levels. The  $\Sigma$ -type of the previous paragraph, for instance, would contain an  
351 explicit universe level  $i$ , and both  $A$  and  $B$  would be checked against  $\square_i$  rather than having  
352 their level inferred. This makes the treatment of general inductive types easier, at the cost  
353 of possibly needless annotations, as here with  $\Sigma$ -types. To make that polymorphism more  
354 seamless, those polymorphic inductive types are also cumulative [20]: in much the same way  
355 as  $\square_i \preceq \square_j$  if  $i \leq j$ , also  $\mathbb{N}^{\textcircled{i}} \preceq \mathbb{N}^{\textcircled{j}}$ , where  $\textcircled{i}$  and  $\textcircled{j}$  are two different universe levels of the  
356 polymorphic inductive  $\mathbb{N}$ . This enables lifting from a lower inductive type to a higher one, so

$$\boxed{\Gamma \vdash t \triangleright T}$$

$$\frac{\Gamma \vdash A \triangleright_{\square} \square_i \quad \Gamma, x : A \vdash B \triangleright_{\square} \square_j}{\Gamma \vdash \Sigma x : A.B \triangleright_{\square} \square_{i \vee j}} \Sigma\text{-TYPE}$$

$$\frac{\Gamma \vdash A \triangleright_{\square} \square_i \quad \Gamma, x : A \vdash B \triangleright_{\square} \square_j \quad \Gamma \vdash a \triangleleft A \quad \Gamma \vdash b \triangleleft B[x := a]}{\Gamma \vdash (a, b)_{A.x.B} \triangleright \Sigma x : A.B} \Sigma\text{-CONS}$$

$$\frac{\Gamma \vdash s \triangleright_{\Sigma} \Sigma x : A.B \quad \Gamma, z : \Sigma x : A.B \vdash P \triangleright_{\square} \square_i \quad \Gamma, x : A, y : B \vdash b \triangleleft P[z := (x, y)]}{\Gamma \vdash \text{rec}_{\Sigma}(z.P, x.y.b, s) \triangleright P[z := s]} \Sigma\text{-REC}$$

$$\boxed{\Gamma \vdash t \triangleright_h T}$$

$$\frac{\Gamma \vdash t \triangleright T \quad T \rightarrow^* \Sigma x : A.B}{\Gamma \vdash t \triangleright_{\Sigma} \Sigma x : A.B} \Sigma\text{-INF}$$

■ **Figure 4** Bidirectional sum type

357 that for instance  $\vdash 0_i : \mathbb{N}_j$  if  $i \leq j$ .

358 Apart from that difference, PCUIC as presented in MetaCoq has constructors and  
 359 inductive types as functions, rather than requiring them to be fully applied. It also separates  
 360 recursors into a pattern-matching and a fixpoint construct, the latter coming with a specific  
 361 guard condition to keep the normalization property enjoyed by a system with recursors.

362 All those choices aim at making the system more flexible and practically usable, but they  
 363 come with a price: the complexity of the structure of terms is much higher. In particular,  
 364 contrarily to what happens in  $\Sigma\text{-REC}$ , the information needed to type the predicate  $P$  and  
 365 branch  $b$  cannot be simply inferred from the scrutinee  $s$  – thinking erroneously that this  
 366 was the case led to the incompleteness bug we mentioned. Instead the case constructor  
 367 must contain the universe instance and parameters that are used to type the predicate and  
 368 scrutinee.

369 A sketch of the resulting rules is given in Figure 6, for a generic inductive  $I$ . We use bold  
 370 characters to denote lists – for instance  $\mathbf{a}$  is a list of terms – and indexes to denote a specific  
 371 element – so that  $\mathbf{a}_k$  is the  $k$ -th element of the previous. The considered inductive  $I$  has  
 372 parameters of type  $\mathbf{X}$ , indices of type  $\mathbf{Y}$  and inhabits some universe  $\square_l$ . Its constructors  $c_k$   
 373 are of types  $\Pi(\mathbf{x} : \mathbf{X})(\mathbf{y} : \mathbf{Y}_k), I \mathbf{x} \mathbf{u}$ . Because we are considering a cumulative inductive  
 374 type, all of those actually have to be instantiate with universe levels, an operation we denote  
 375 with  $^{\circledast i}$ . Apart from the extra checking that the parameters given in the match construct  
 376 have the correct type, and the extra cumulativity check to compare the parameters obtained  
 377 from the scrutinee and the ones in the node, the structure of the match construct is quite  
 378 similar to that of the sum type. Concerning the fixpoint construct, the most important part  
 379 there is the guard condition, but as the bidirectional approach has nothing to add here we  
 380 leave it out.

## 23:12 Complete Bidirectional Typing for the Calculus of Inductive Constructions

$\boxed{\Gamma \vdash t \triangleright T}$

$$\begin{array}{c}
\frac{}{\Gamma \vdash \mathbb{N} \triangleright \square_0} \qquad \frac{}{\Gamma \vdash 0 \triangleright \mathbb{N}} \qquad \frac{\Gamma \vdash n \triangleleft \mathbb{N}}{\Gamma \vdash S(n) \triangleright \mathbb{N}} \\
\\
\frac{\Gamma, z : \mathbb{N} \vdash P \triangleright_{\square} \square_i \quad \Gamma \vdash b_0 \triangleleft P[z := 0] \quad \Gamma, x : \mathbb{N}, p : P[z := x] \vdash b_S \triangleleft P[z := S(x)] \quad \Gamma \vdash s \triangleright_{\mathbb{N}} \mathbb{N}}{\Gamma \vdash \text{rec}_{\mathbb{N}}(z.P, b_0, x.p.b_S, s) \triangleright P[z := s]} \\
\\
\frac{\Gamma \vdash A \triangleright_{\square} \square_i \quad \Gamma \vdash a \triangleleft A \quad \Gamma \vdash a' \triangleleft A}{\Gamma \vdash \text{Id}_A a a' \triangleright_{\square} \square_i} \qquad \frac{\Gamma \vdash A \triangleright_{\square} \square_i \quad \Gamma \vdash a \triangleleft A}{\Gamma \vdash \text{refl}_A a \triangleright \text{Id}_A a a} \\
\\
\frac{\Gamma \vdash s \triangleright \text{Id}_A a a' \quad \Gamma, x : A, z : \text{Id}_A a x \vdash P \triangleright_{\square} \square_i \quad \Gamma \vdash b \triangleleft P[z := \text{Id}_A a a][x := a]}{\Gamma \vdash \text{rec}_{\text{Id}}(x.z.P, b, s) \triangleright P[z := s][x := a']}
\end{array}$$

$\boxed{\Gamma \vdash t \triangleright_h T}$

$$\frac{\Gamma \vdash t \triangleright T \quad T \rightarrow^* \mathbb{N}}{\Gamma \vdash t \triangleright_{\mathbb{N}} \mathbb{N}} \qquad \frac{\Gamma \vdash t \triangleright T \quad T \rightarrow^* \text{Id}_A a a'}{\Gamma \vdash t \triangleright_{\text{Id}} \text{Id}_A a a'}$$

■ **Figure 5** Other bidirectional inductive types

### 381 **4 Beyond PCUIC: bidirectional extensions to CIC**

382 The use of our bidirectional structure is not limited to CIC or PCUIC. On the contrary, it  
383 forms a solid basis for extensions, as we illustrate now.

#### 384 **4.1 Localized computation**

385 The free-standing conversion rule `CONV` is very powerful, but sometimes too much. Indeed,  
386 the ability to stack as many conversion rules as desired at any place in an undirected  
387 derivation is reasonable only when types are compared using a transitive relation. When  
388 this is not the case, for instance when conversion is replaced by a unification-flavoured  
389 relation, the undirected system becomes inadequate, because repeated uses of `CONV` can  
390 drastically change a type in an undesired fashion. In such a setting, the equivalence between  
391 the undirected and the bidirectional system is lost. In such a setting, contrarily to the  
392 undirected system, the bidirectional system is still viable, as it enforces a localized use of  
393 conversion: only once, at the interface between inference and checking.

394 This is exactly what happens in [8]. In that paper, the conversion relation is relaxed to  
395 accommodate for an additional term `?` that behaves as a wildcard and should be considered  
396 convertible to any term. Conversion is therefore completely non-transitive, and the extension  
397 needs to be based on the bidirectional type system rather than the undirected one in order  
398 to ensure that the conversion rule is used in a meaningful way.

399 More generally, since the equivalence between the undirected and directed variants relies

$$\boxed{\Gamma \vdash t \triangleright T}$$

$$\frac{\frac{\frac{\Gamma \vdash I^{\textcircled{i}} \triangleright \Pi(\mathbf{x} : \mathbf{X}^{\textcircled{i}})(\mathbf{y} : \mathbf{Y}^{\textcircled{i}}), \square_{I^{\textcircled{i}}}}{\Gamma \vdash s \triangleright_I I^{\textcircled{i}'} \mathbf{a} \mathbf{b}} \quad \frac{\frac{\Gamma \vdash \mathbf{p}_k \triangleleft \mathbf{X}_k[\mathbf{x} := \mathbf{p}] \quad \Gamma, \mathbf{y} : \mathbf{Y}^{\textcircled{i}}[\mathbf{p} := \mathbf{x}], z : I^{\textcircled{i}} \mathbf{p} \mathbf{y} \vdash P \triangleright_{\square} \square_j}{I^{\textcircled{i}'} \mathbf{a} \mathbf{b} \preceq I^{\textcircled{i}} \mathbf{p} \mathbf{b}} \quad \Gamma, \mathbf{y} : \mathbf{Y}_k^{\textcircled{i}}[\mathbf{p} := \mathbf{x}] \vdash \mathbf{t}_k \triangleleft P[z := c_k^{\textcircled{i}} \mathbf{p} \mathbf{y}][\mathbf{y} := \mathbf{u}_k^{\textcircled{i}}]}{\Gamma \vdash \text{match } s \text{ in } (I, \mathbf{i}, \mathbf{p}) \text{ return } P \text{ with } [\mathbf{t}] \triangleright P[z := s][\mathbf{y} := \mathbf{b}]}}{\frac{\Gamma \vdash T \triangleright_{\square} \square_i \quad \Gamma, f : T \vdash t \triangleleft T \quad \text{guard condition}}{\Gamma \vdash \text{fix } f : T := t \triangleright T}}$$

$$\boxed{\Gamma \vdash t \triangleright_I T}$$

$$\frac{\Gamma \vdash t \triangleright T \quad T \rightarrow I \mathbf{a} \mathbf{b}}{\Gamma \vdash t \triangleright_I I \mathbf{a} \mathbf{b}}$$

■ **Figure 6** Bidirectional inductive type – PCIC style

400 on all properties of Proposition 1, when one of those fails the equivalence between the  
 401 undirected and bidirectional systems is endangered. This can be a sign that the bidirectional  
 402 system should be adapted, but it can also signal that the undirected system has become  
 403 meaningless and that the bidirectional version should be studied instead.

## 404 4.2 Modding the conversion rule

405 The fact that the unique conversion rule gives rise to multiple bidirectional ones is important:  
 406 it signals that there are in fact two ways to consider conversion, although the difference  
 407 between both is invisible in undirected presentations. But this difference might not be so  
 408 easily overlooked in extensions of CIC, which then need different treatment for them.

409 Taking again the example of [8], the CHECK can be kept as such, because the conversion  
 410 relation is directly modified in the new system. But this is not the case for partial inference.  
 411 In fact, rule SORT-INF has to be supplemented by another rule to treat the case when the  
 412 inferred type reduces to the wildcard ?, because such a term can be used as a type – with  
 413 some care taken. The same happens for all constrained inference rules.

414 Thus, the bidirectional structure clarifies a fact that might be overlooked by those who  
 415 do not dwell in the implementation of proof assistants: reduction does not only serve as a  
 416 subroutine of conversion checking, it is also directly needed to determine if a given type is a  
 417 sort, a product, an inductive. . . Which is quite different from checking that it is convertible  
 418 to a given sort or product type. Of course one could replace reduction by another machinery  
 419 to accomplish this task, but if one wishes to modify conversion, this specific role of reduction  
 420 must be accounted for. Otherwise, rules for  $\triangleleft$  and  $\triangleright_h$  would come out of sync, bringing  
 421 troubles down the road.

### 4.3 Bidirectional elaboration

In works such as [15, 3, 8], the procedure described is not typing but rather elaboration: the subject of the derivation  $t$  is in a kind of source syntax and the aim is not only to inspect  $t$ , but also to output a correspond  $t'$  in some kind of target syntax. The term  $t'$  is a more precise account of term  $t$ , for instance with solved meta-variables, inserted coercions, and so on. The structure we describe readily adapts to those settings, the extra term  $t'$  is simply considered as an output of all judgements. Since it is an output, McBride's discipline as described in Section 2.2 demands that when  $\Gamma \vdash t \rightsquigarrow t' \triangleright T$  (with input context  $\Gamma$ , the subject  $t$  elaborates to  $t'$  and infers type  $T$ ) we must ensure that  $\Gamma \vdash t' : T$ , and similarly for all other typing judgements. Having all rules locally preserve this invariant ensures that elaborated terms are always well-typed.

## 5 Related work

### 5.1 Constrained inference

Although explicit and systematic description of constrained inference in a bidirectional setting is new, traces of it in diverse seemingly ad-hoc workarounds can be found in various works around typing for CIC, illustrating that this notion, although overlooked, is of interest.

In [14],  $\Gamma \vdash t : T$  is used for what we write  $\Gamma \vdash t \triangleright T$ , but another judgment written  $\Gamma \vdash t : \geq T$  and denoting type inference followed by reduction is used to effectively inline the two hypothesis of our constrained inference rules. Checking is similarly inlined.

Saïbi [15] describes an elaboration mechanism inserting coercions between types. Those are inserted primarily in checking, when both types are known. However he acknowledges the presence of two special classes to handle the need to cast a term to a sort or a function type without more informations, exactly in the places where we resort to constrained inference rather than checking.

More recently, Sozeau [16] describes a system where conversion is augmented to handle coercion between subset types. Again,  $\Gamma \vdash t : T$  is used for inference, and the other judgments are inlined. Of interest is the fact that reduction is not enough to perform constrained inference, because type head constructors can be hidden by the subset construction: a term of subset type such as  $\{f : \mathbb{N} \rightarrow \mathbb{N} \mid f\ 0 = 0\}$  should be usable as a function of type  $\mathbb{N} \rightarrow \mathbb{N}$ . An erasure procedure is therefore required on top of reduction to remove subset types in the places where we use constrained inference.

These traces can also be found in the description of Matita's elaboration algorithm [3]. Indeed, the presence of meta-variables on top of coercions as in the two previous works makes it even clearer that specific treatment of what we identified as constrained inference is required. The authors introduce a special judgement they call type-level enforcing corresponding exactly to our  $\triangleright_{\square}$  judgement. As for  $\triangleright_{\Pi}$ , they have two rules to apply a function, one where its inferred type reduces to a product, corresponding to PROD-INF, and another one to handle the case when the inferred type instead reduces to a meta-variable. As Saïbi, they also need a special case for coercions of terms in function and type position. However, their solution is different. They rely on unification, which is available in their setting, to introduce new meta-variables for the domain and codomain of a product type whenever needed. For  $\triangleright_{\square}$  though this solution is not viable, as one would need a kind of universe meta-variable. Instead, they rely on backtracking to test multiple possible universe choices.

Finally, we have already mentioned [8] in Section 4, where the bidirectional structure is crucial in describing a gradual extension to CIC. In particular, and similarly to what

467 happens with meta-variables in [3], all constrained inference rules are duplicated: there is one  
 468 rule when the head constructor is the desired one, and a second one to handle the gradual  
 469 wildcard.

## 470 5.2 Completeness

471 Quite a few articles tackle the problem of bidirectional typing in a setting with an untyped  
 472 – so called Curry-style – abstraction. This is the case of early work by Coquand [7], the  
 473 type system of Agda as described in [12], the systems considered by Abel for instance [1],  
 474 and much of the work of McBride [11, 9, 10] on the topic. In such systems,  $\lambda$ -abstractions  
 475 can only be checked against a given type, but cannot infer one, so that only terms with no  
 476  $\beta$ -redexes are typable. Norell argues in [12] that such  $\beta$ -redexes are uncommon in real-life  
 477 programs, so that being unable to type them is not a strong limitation in practice. To  
 478 circumvent this problem, McBride also adds the possibility of typing annotations to retain  
 479 the typability of a term during reduction. While this approach is adapted to programming  
 480 languages, where the emphasis is on lightweight syntax, it is not tenable for a proof assistant  
 481 kernel, where all valid terms should be accepted. Indeed, debugging a proof that is rejected  
 482 because the kernel fails to accept a perfectly well-typed term the user never wrote – as most  
 483 proofs are generated rather than written directly – is simply not an option.

484 In a setting with typed – Church-style – abstraction, if one wishes to give the possibility  
 485 for seemingly untyped abstraction, another mechanism has to be resorted to, typically  
 486 meta-variables. This is what is done in Matita [3], where the authors combine a rule similar  
 487 to ABS – where the type of the abstraction is inferred – with another one, similar to the  
 488 Curry-style one – where abstraction is checked – looking like this:

$$489 \frac{T \rightarrow^* \Pi x : A'.B \quad \Gamma \vdash A \triangleright_{\square} \square_i \quad A \equiv A' \quad \Gamma, x : A \vdash t \triangleleft B}{\Gamma \vdash \lambda x : A.t \triangleleft T}$$

490 While such a rule would make a simple system such as that of Section 2 “over-complete”,  
 491 it is a useful addition to enable information from checking to be propagated upwards in  
 492 the derivation. This is crucial in extensions where completeness is lost, such as Matita’s  
 493 elaboration. Similar rules are described in [3] for let-bindings and constructors of inductive  
 494 types.

495 Although only few authors consider the problem of a complete bidirectional algorithm for  
 496 type-checking dependent types, we are not the first to attack it. Already Pollack [14] does,  
 497 and the completeness proof for  $CC\omega$  of Section 2 is very close to one given in his article.  
 498 Another proof of completeness for a more complex CIC-like system can be found in [16].  
 499 None of those however tackle as we do the whole complexity of PCUIC.

## 500 5.3 Inputs and outputs

501 We already credited the discipline we adopt on well-formedness of inputs and outputs to  
 502 McBride [9, 10]. A similar idea has also appeared independently in [5]. Bauer and his  
 503 co-authors introduce the notions of a (weakly) presuppositive type theory [5, Def. 5.6] and  
 504 of well-presented premise-family and rule-boundary [5, Def. 6.16 and 6.17] to describe a  
 505 discipline similar to ours, using what they call the boundary of a judgment as the equivalent  
 506 of our inputs and outputs. Due to their setting being undirected, this is however more  
 507 restrictive, because they are not able to distinguish inputs from outputs and thus cannot  
 508 relax their condition to only demand inputs to be well-formed but not outputs.



509 **6 Conclusion**

510 We have described a judgmental presentation of the bidirectional structure of typing al-  
 511 gorithms in the setting of dependent types. In particular, we identified a new family of  
 512 judgements we called constrained inference. Those have no counterpart in the non-dependent  
 513 setting, as they result from a choice of modding for the conversion rule, which is specific to  
 514 the dependent setting. We proved our bidirectional presentation equivalent to an undirected  
 515 one, both on paper on the simple case of  $CC\omega$ , and formally in the much more complex  
 516 and realistic setting of PCUIC. Finally, we gave various arguments for the usefulness of our  
 517 presentation as a way to ease proofs, an intermediate between undirected type-systems and  
 518 typing algorithms, a solid basis to design extensions, and a tool to re-interpret previous work  
 519 on type systems in a clearer way.

520 Regarding future work, a type-checking algorithm is already part of MetaCoq, and we  
 521 should be able to use our bidirectional type system to give a pleasant completeness proof by  
 522 separating the concerns pertaining to bidirectionality from the algorithmic problems, such as  
 523 implementation of an efficient conversion check or proof of termination. More broadly, our  
 524 bidirectional type system should be an interesting tool in the feat of incorporating in a proof  
 525 assistant features that have been satisfactorily investigated on the theoretical level while  
 526 keeping a complete and correct kernel, avoiding the pitfall of cumulative inductive type's  
 527 incomplete implementation in Coq. A first step would be to investigate the discrepancies  
 528 between the presentations of Section 3, and in particular if all informations currently stored  
 529 in the case node are really needed, or if a more concise presentation can be given. But we  
 530 could go further and study how to handle cubical type theory [21], rewrite rules [6], setoid  
 531 type theory [2], exceptional type theory [13],  $\eta$ -conversion... Finally, we hope that our  
 532 methodology will be adapted as a base for other theoretical investigations. As a way to ease  
 533 this adoption, studying it in a general setting such as that of [5] might be a strong argument  
 534 for adoption.

535 **References**

- 
- 536 **1** Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory  
 537 in type theory. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158111.
- 538 **2** Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. Setoid type  
 539 theory - a syntactic translation. In *MPC 2019 - 13th International Conference on Mathematics  
 540 of Program Construction*, volume 11825 of *LNCS*, pages 155–196. Springer. doi:10.1007/  
 541 978-3-030-33636-3\_7.
- 542 **3** Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A Bi-Directional  
 543 Refinement Algorithm for the Calculus of (Co)Inductive Constructions. Volume 8, Issue 1.  
 544 URL: <https://lmcs.episciences.org/1044>, doi:10.2168/LMCS-8(1:18)2012.
- 545 **4** Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*.
- 546 **5** Andrej Bauer, Philipp G. Haselwarter, and Peter LeFanu Lumsdaine. A general definition of  
 547 dependent type theories. 2020. arXiv:2009.05539.
- 548 **6** Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. The Taming of the Rew: A Type  
 549 Theory with Computational Assumptions. *Proceedings of the ACM on Programming Languages*,  
 550 2021. URL: <https://hal.archives-ouvertes.fr/hal-02901011>.
- 551 **7** Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer  
 552 Programming*, 26(1), 1996. URL: [http://www.sciencedirect.com/science/article/pii/  
 553 0167642395000216](http://www.sciencedirect.com/science/article/pii/0167642395000216), doi:[https://doi.org/10.1016/0167-6423\(95\)00021-6](https://doi.org/10.1016/0167-6423(95)00021-6).

- 554 8 Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. Gradualizing  
555 the calculus of inductive constructions, 2020. URL: <https://arxiv.org/abs/2011.10618>,  
556 arXiv:2011.10618.
- 557 9 Conor McBride. Basics of bidirectionality. URL: [https://pigworker.wordpress.com/2018/  
558 08/06/basics-of-bidirectionality/](https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectionality/).
- 559 10 Conor McBride. Check the box! In *25th International Conference on Types for Proofs and  
560 Programs*.
- 561 11 Conor McBride. *I Got Plenty o' Nuttin'*, pages 207–233. Springer International Publishing,  
562 2016. doi:10.1007/978-3-319-30936-1\_12.
- 563 12 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD  
564 thesis, Department of Computer Science and Engineering, Chalmers University of Technology,  
565 SE-412 96 Göteborg, Sweden, September 2007.
- 566 13 Pierre-Marie Pédro and Nicolas Tabareau. Failure is not an option in an exceptional type theory.  
567 In *ESOP 2018 - 27th European Symposium on Programming*, volume 10801 of *LNCS*, pages  
568 245–271. Springer. doi:10.1007/978-3-319-89884-1\_9.
- 569 14 R. Pollack. Typechecking in Pure Type Systems. In *Informal Proceedings of the 1992  
570 Workshop on Types for Proofs and Programs, Båstad, Sweden*, pages 271–288, June 1992. URL:  
571 <http://homepages.inf.ed.ac.uk/rpollack/export/BaastadTypechecking.ps.gz>.
- 572 15 Amokrane Saïbi. Typing algorithm in type theory with inheritance. doi:10.1145/263699.  
573 263742.
- 574 16 Matthieu Sozeau. Subset coercions in coq. In Thorsten Altenkirch and Conor McBride,  
575 editors, *Types for Proofs and Programs*, pages 237–252, Berlin, Heidelberg, 2007. Springer  
576 Berlin Heidelberg.
- 577 17 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian  
578 Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project.  
579 *Journal of Automated Reasoning*, February 2020. URL: <https://hal.inria.fr/hal-02167423>,  
580 doi:10.1007/s10817-019-09540-0.
- 581 18 Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in coq. In Gerwin Klein and  
582 Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 499–514. Springer International  
583 Publishing.
- 584 19 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of  
585 Mathematics*. <https://homotopytypetheory.org/book>.
- 586 20 Amin Timany and Matthieu Sozeau. Cumulative Inductive Types In Coq. In H el ene Kirchner,  
587 editor, *3rd International Conference on Formal Structures for Computation and Deduction  
588 (FSCD 2018)*, volume 108 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages  
589 29:1–29:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.  
590 URL: <http://drops.dagstuhl.de/opus/volltexte/2018/9199>, doi:10.4230/LIPIcs.FSCD.  
591 2018.29.
- 592 21 Andrea Vezzosi, Anders M ortberg, and Andreas Abel. Cubical agda: A dependently typed  
593 programming language with univalence and higher inductive types. *Proc. ACM Program.  
594 Lang.*, 3(ICFP), July 2019. doi:10.1145/3341691.