Complete Bidirectional Typing for the Calculus of Inductive Constructions

Meven Lennon-Bertrand

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 new judgement intermediate between the usual inference and checking, dubbed constrained inference, to handle the presence of computation in types. The key property of the system is its completeness with respect to the usual undirected one, which has been formally proven in Coq as a part of the MetaCoq project. Although it plays an important role in an ongoing completeness proof for a realistic typing algorithm, the interest of bidirectionality is wider, as it gives insights and structure when trying to prove properties on CIC or design variations and extensions. In particular, we put forward constrained inference, an intermediate between the usual inference and checking judgements, to handle the presence of computation in types.

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`

Acknowledgements Many thanks to Matthieu Sozeau for his help with MetaCoq and its nasty inductives, and to Chantal Keller, Nicolas Tabareau and the anonymous reviewers for their helpful comments on earlier versions of this article.

1 Introduction

In logical programming, an important characteristic of 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 Γ of this inspection are usually known, they are thus inputs (although some depart from this, see [12]). The mode of the type T, however, may vary: should it be inferred based upon Γ 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[1] typing $\Gamma \vdash t : T$ into two separate judgments $\Gamma \vdash t \triangleright T$ (inference) and $\Gamma \vdash t \triangleleft T$ (checking)[2], that differ only by modes. 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 this seems appealing, and despite advocacy by McBride [15, 16] to adopt this approach when designing type systems, most of the theoretical work on dependent typing to

[1] We call anything related to the $\Gamma \vdash t : T$ judgement undirected by contrast with the bidirectional typing.
[2] 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.

this day remains undirected. Others have described on paper bidirectional algorithms for dependent types, starting with Coquand [11] and continuing with Norell [17] or Abel [5]. However, all of these consider unannotated λ-abstractions, and use bidirectional typing as a way to remedy this lack of annotations. This is sensible when looking for lightness of the input syntax, but 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 from Asperti and the Matita team [7], who however concentrate on specific problems pertaining to unification and implementation of the Matita elaborator, without giving a general bidirectional framework. They also do not consider the problem of completeness with respect to a given undirected system, as it would fail in their setting due to the undecidability of higher order unification.

Thus, we wish to fill a gap in the literature, by describing a bidirectional type system that is complete with respect to the (undirected) Calculus of Inductive Constructions (CIC). By completeness, we mean that any term that is typable in the undirected system should also infer a type in the bidirectional one. This feature is very desirable when implementing kernels for proof assistants, whose algorithms should correspond to their undirected specification, never missing any typable term. The bidirectional systems we describe thus form intermediate steps between actual algorithms and undirected type systems. This step has proven useful in an ongoing completeness proof of MetaCoq's [23] type-checking algorithm[3]: rather than proving the algorithm complete directly, the idea is to prove it equivalent to the bidirectional type system, separating the implementation problems from the ones regarding the bidirectional structure.

But the interest of having a bidirectional type system equivalent to the undirected one is not limited to the link with algorithms, it is also purely theoretical. First, the structure of a bidirectional derivation is more constrained than that of an undirected one, especially regarding the uses of computation. This finer structure can make proofs easier, while the equivalence ensures that they can be transported to the undirected world. For instance, in a setting with cumulativity/subtyping, the inferred type for a term t should by construction be smaller than any other types against which t checks. This provides an easy proof of the existence of principal types in the undirected system. The bidirectional structure also provides a better base for new type systems. This was actually the starting point for this investigation: in [13], we quickly describe a bidirectional variant of CIC, as the usual undirected CIC is unfit for the gradual extension we envision due to the too high flexibility of a free-standing conversion rule. This is the system we wish to thoroughly describe and investigate here.

Outline We start by giving in Section 2 a general roadmap in the simple setting of pure type systems, including the introduction of a constrained inference judgement that had not been clearly singled out in previous works. With the ideas set clear, we go on to the real thing: a bidirectional type system proven equivalent to the Predicative Calculus of Cumulative Inductive Constructions – PCUIC, the extension of CIC nowadays at the heart of Coq. This equivalence has been formalised on top of MetaCoq [24][4] We next turn back to less technical considerations, as we believe that the bidirectional structure is of general theoretical interest. Section 4 thus describes the value of basing type systems on the bidirectional system directly

[3] A completeness bug in that algorithm – also present in the Coq kernel – has already been found, see Section 3 for details.

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

rather than on the equivalent undirected one. Finally Section 5 investigates related work, and in particular tries and identify the implicit presence of constrained inference in various earlier articles, showing how making it explicit clarifies those.

2 Warming up with CCω

2.1 Undirected CCω

As a starting point, let us consider the system CCω. It is the backbone of CIC, and we can already illustrate most of our methodology on it. CCω belongs to the wider class of pure type systems (PTS), that has been thoroughly studied and described, see for instance [8]. Since there are many presentational variations, let us first give a precise account of our conventions. *Terms* in CCω are given by the grammar

$$t ::= x \mid \square_i \mid \Pi\, x : t.t \mid \lambda\, x : t.t \mid t\ t$$

where the letter x denotes a variable (so will letters y and z), and the letter i is an integer (we will also use letters j, k and l for those). All other Latin letters will be used for terms, with the upper-case ones used to suggest the corresponding terms should be though of as types — although this is not a syntactical separation. We abbreviate $\Pi\, x : A.B$ by $A \to B$ when B does not depend on x, as is customary. On those terms, *reduction* $\rightsquigarrow$ is defined as the least congruence such that $(\lambda\, x : T.t)\ u \rightsquigarrow t[x := u]$, where $t[x := u]$ denotes *substitution*. *Conversion* $\equiv$ is the symmetric, reflexive, transitive closure of reduction. Finally, *contexts* are lists of variable declarations $x : t$ and are denoted using capital Greek letters. We write $\cdot$ for the empty list, $\Gamma, x : T$ for concatenation, and $(x : T) \in \Gamma$ if $(x : T)$ appears in Γ. Combining those, we can define *typing* $\Gamma \vdash t : T$ as in Figure 1, where $i \vee j$ denotes the maximum of i and j. We say a context Γ is *well-formed* if $\vdash \Gamma$, a type T is *well-formed* in a context Γ if there exists i such that $\Gamma \vdash T : \square_i$, and a term is *well-formed* in a context Γ if there exists T such that $\Gamma \vdash t : T$. We also use *well-typed* for the latter, and leave the context implicit for the last two when it is clear from context. These rules differ from more usual PTS presentations such as [8] on the Var and Sort rules so as to avoid general weakening (which is however admissible) and single out the context well-formedness judgment. Premises are not minimal in order to provide more generous inductive hypotheses when doing proofs by induction on derivations. However, this presentation can easily be seen to be equivalent to that of [8].

As any PTS, CCω has many desirable properties. We summarize the ones we rely on here. Detailed proofs in the context of PTS can be found in [8], and formalisation of the corresponding properties for PCUIC are an important part of MetaCoq.

▶ **Proposition 1** (Properties of CCω). *The type system CCω as just described enjoys the following properties:*

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

Transitivity *Conversion is transitive.*

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

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

2.2 Turning CCω Bidirectional

McBride's discipline To design our bidirectional type system, we follow a discipline exposed by McBride [15, 16]. The central point is to distinguish in a judgment between the subject,

$\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \cdot}\textsc{Empty} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A : \square_i}{\vdash \Gamma, x : A}\textsc{Ext}$$

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

$$\frac{\vdash \Gamma}{\Gamma \vdash \square_i : \square_{i+1}}\textsc{Sort} \qquad \frac{\vdash \Gamma \qquad (x : A \in \Gamma)}{\Gamma \vdash x : A}\textsc{Var} \qquad \frac{\Gamma \vdash A : \square_i \qquad \Gamma, x : A \vdash B : \square_j}{\Gamma \vdash \Pi\, x : A.B : \square_{i \vee j}}\textsc{Prod}$$

$$\frac{\Gamma \vdash \Pi\, x : A.B : \square_i \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda\, x : A.t : \Pi\, x : A.B}\textsc{Abs} \qquad \frac{\Gamma \vdash t : \Pi\, x : A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x := u]}\textsc{App}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : \square_i \qquad A \equiv B}{\Gamma \vdash t : B}\textsc{Conv}$$

Figure 1 Undirected typing for CCω

whose well-formedness is under scrutiny, from inputs, whose well-formedness is a condition for the judgment to behave well, and outputs, whose well-formedness is a consequence of the judgment. For instance, in inference $\Gamma \vdash t \triangleright T$, the subject is t, Γ is an input and T is an output. This means that one should consider whether $\Gamma \vdash t \triangleright T$ only in cases where $\vdash \Gamma$ is already known, and if the judgment is derivable it should be possible to conclude that both t and T are well-formed. All inference rules are to preserve this invariant. This means that inputs to a premise should be well-formed whenever the inputs to the conclusion and outputs and subjects of previous premises are. Similarly the outputs of the conclusion should be well-formed if the inputs of the conclusion and the subjects and outputs of the premises are assumed to be so.

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

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

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

$$\frac{}{\Gamma \vdash \square_i \triangleright \square_{i+1}}\textsc{Sort} \qquad \frac{(x : T) \in \Gamma}{\Gamma \vdash x \triangleright T}\textsc{Var} \qquad \frac{\Gamma \vdash A \triangleright_\square \square_i \qquad \Gamma, x : A \vdash B \triangleright_\square \square_j}{\Gamma \vdash \Pi\, x : A.B \triangleright \square_{i \vee j}}\textsc{Prod}$$

$$\frac{\Gamma \vdash A \triangleright_\square \square_i \qquad \Gamma, x : A \vdash t \triangleright B}{\Gamma \vdash \lambda\, x : A.t \triangleright \Pi\, x : A.B}\textsc{Abs} \qquad \frac{\Gamma \vdash t \triangleright_\Pi \Pi\, x : A.B \qquad \Gamma \vdash u \triangleleft A}{\Gamma \vdash t\ u \triangleright B[x := u]}\textsc{App}$$

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

$$\frac{\Gamma \vdash t \triangleright T' \qquad T' \equiv T}{\Gamma \vdash t \triangleleft T}\textsc{Check}$$

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

$$\frac{\Gamma \vdash t \triangleright T \qquad T \rightsquigarrow^* \square_i}{\Gamma \vdash t \triangleright_\square \square_i}\textsc{Sort-Inf} \qquad \frac{\Gamma \vdash t \triangleright T \qquad T \rightsquigarrow^* \Pi\, x : A.B}{\Gamma \vdash t \triangleright_\Pi \Pi\, x : A.B}\textsc{Prod-Inf}$$

Figure 2 Bidirectional typing for CCω

Structural rules To transform the rules of Figure 1 to those of Figure 2, we start by recalling that we wish to obtain a complete bidirectional type system. Therefore any term should infer a type, and thus all structural rules (i.e. all rules where the subject of the conclusion starts with a term constructor) should give rise to an inference rule. It thus remains to choose the judgements for the premises, which amounts to determining their modes. If a term in a premise appears as input in the conclusion or output of a previous premise, then it can be considered an input, otherwise it must be an output. Moreover, if a type output is unconstrained, then inference can be used, otherwise we must resort to constrained inference.

This applies straightforwardly to most rules but the PTS-style Abs. Indeed, neither $\Gamma \vdash \Pi\, x : A.B : \square_i$ nor $\Gamma, x : A \vdash t : B$ can be taken as the first bidirectional premise: the first one because B is not known from inputs to the conclusion, and the second because context $\Gamma, x : A$ is not known to be well-formed from the conclusion. For general PTS, this is quite problematic, as demonstrated by Pollack [19]. For CCω, however, the solution is simple. Replacing $\Gamma \vdash \Pi\, x : A.B : \square_i$ by the equivalent $\Gamma \vdash A : \square_j$ and $\Gamma, x : A \vdash B : \square_{j'}$, the former can become the first premise, ensuring that type inference for t is done in a well-formed context, and the latter can be dropped based upon the invariant that outputs – here the type B inferred for t — are well-formed.

Similarly, as the context is always supposed to be well-formed as an input to the conclusion, it is not useful to re-check it, and thus the premise of Sort and Var can be safely dropped. This is in line with implementations, where the context is not re-checked at leaves of a derivation tree, with performance issues in mind. The well-formedness invariants then ensure that any derivation starting with the (well-formed) empty context will only handle well-formed contexts.

Computation rules We are now left with the non-structural conversion rule. As we observed, there are two possible modes for computation: if both sides are inputs, conversion can be

used, but if only one is known one must resort to reduction, and the other side becomes an output instead. Rule Check corresponds to the first case, while rules Prod-Inf and Sort-Inf both are in the second case. This difference in turn introduces the need to separate between checking, that calls for the first rule, and constrained inference, that requires the others.

The need to split the conversion rule into a (weak-head) reduction and conversion subroutine depending on the mode is known to the implementors of proof assistants [2]. However, we wish to de-emphasize the role devoted specifically to reduction in the description of those algorithms, instead putting constrained inference forward. Indeed, reduction is only a means to determine whether a certain term fits into a certain category of types. In the setting of CCω, there is mainly one way to do so, which is to reduce its type until its head constructor is exposed. However, as soon as conversion is extended, for instance with unification [7], coercions [7, 22] or graduality [13], reduction is not enough any more. Singling out constrained inference then makes the required modification to the typing rules clearer. We come back to this more in length in Section 5.1.

2.3 Properties

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

2.3.1 Correctness

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

▶ **Theorem 2** (Correctness of bidirectional typing for CCω). *If* Γ *is well-formed and* $\Gamma \vdash t \triangleright T$ *or* $\Gamma \vdash t \triangleright_h T$ *then* $\Gamma \vdash t : T$*. If* Γ *and* T *are well-formed and* $\Gamma \vdash t \triangleleft T$ *then* $\Gamma \vdash t : T$*.*

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

Each rule of the bidirectional system can be replaced by the corresponding rule of the undirected system, with all three Check, Prod-Inf and Sort-Inf replaced by Conv, Abs using an extra Prod rule. In all cases, the induction hypothesis can be used on sub-derivations of the bidirectional judgment because the context extensions and checking are done with types that are known to be well-formed by induction hypothesis on previous premises and validity.

Some sub-derivations of the undirected rules that have no counterpart in the bidirectional ones are however missing. In rules Sort and Var the hypothesis that $\vdash \Gamma$ is enough to get the required premise. For rule Check, the well-formedness hypothesis on the type is needed to get the second premise of rule Conv. As for Prod-Inf and Sort-Inf, that second premise is obtained using the induction hypothesis, validity and subject reduction. Finally, the missing premise on the codomain of the product type in rule Abs is obtained by validity.

Uses of validity could alternatively be handled by strengthening the theorem to incorporate the well-formedness of types 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 CCω). *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. Rule Conv is a direct consequence of the induction hypothesis on its first premise, together with transitivity of conversion.

For rule Prod, we need the intermediate lemma that if T is a term such that $T \equiv \square_i$, then also $T \rightsquigarrow^* \square_i$. 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, we get $\Gamma \vdash \lambda\, x : A.t \triangleright \Pi\, x : A.B'$ 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 $T \rightsquigarrow^* \Pi\, x : A'.B'$ 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 crux 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 judgements. 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* $\rightsquigarrow^*$ *is replaced by weak-head reduction in rules* Sort-Inf *and* Prod-Inf, *then given a well-formed context* Γ *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* Γ *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 bidirectional rules, it amounts to structural recursion on the subject.

3 From CCω to PCUIC

CCω is already a powerful system, but today's proof assistants rely on much more complex features. There are two main areas of differences between CCω and the Predicative Calculus of Cumulative Inductive Constructions (PCUIC), the type theory nowadays behind the Coq proof assistant. Adapting to those is a good stress test for the bidirectional approach: seamlessly doing so is a good sign that the general methodology we presented is robust.

The first area of difference are the universes. While on paper those are simply integer, to handle typical ambiguity and polymorphic (co)-inductive types, PCUIC uses algebraic universes, containing level variables, algebraic $\vee$ and $+1$ operators, and a special level for the sort Prop. Moreover, those universes are cumulative, that is they behave as if smaller universes were included in larger ones. The precise handling of the algebraic universes is abstracted away in MetaCoq, and quite similar in the directed and undirected systems, so it did not prove too difficult to handle. Cumulativity, however, introduces some not-so-small differences with the previous presentation, so we spend some time on it in Section 3.1.

The second is the addition of new base type and term constructors. We describe the treatment of inductive types in Section 3.2. Co-inductive types and records behave very similarly to inductive types at the level of typing, so we do not dwell on them. The difference lies mainly at the level of reduction/conversion, but as our type system treats those as black boxes the differences have a negligible impact.

The interplay between those two areas is quite subtle, and we were able to uncover an incompleteness bug in the current kernel of Coq regarding pattern-matching of cumulative inductive types. This bug had gone unnoticed until our formalisation, but was causing subject reduction failures in some corner cases[5]. We try and give an intuition of the problem in Section 3.2.

3.1 Cumulativity

PCUIC incorporates a limited form of subtyping, corresponding to the intuition that smaller universes are included in larger ones. Conversion $\equiv$ is therefore replaced by *cumulativity* $\preceq$, the main difference being that the constraint on universes is relaxed. For conversions $\square_i \equiv \square_j$ only when $i = j$, but for cumulativity $\square_i \preceq \square_j$ whenever $i \leq j$ – and this extends by congruence through most constructors. The conversion rule is accordingly replaced by the following cumulativity rule

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : \square_i \qquad A \preceq B}{\Gamma \vdash t : B}\textsc{Cumul}$$

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

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

[5] The precise issue in the kernel is described in this git issue: `https://github.com/coq/coq/issues/13495`.

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

$$\frac{\Gamma \vdash t \triangleright A \qquad A \preceq B}{\Gamma \vdash t \triangleleft B}\textsc{Cumul}$$

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

▶ **Theorem 6** (Completeness with cumulativity). *If $\Gamma \vdash t : T$ using rules of Figure 1 replacing Conv with Cumul, then $\Gamma \vdash t \triangleright T'$ is derivable with rules of Figure 2 replacing Check with Cumul for some T' such that $T' \preceq T$.*

In that setting, even without fixing a reduction strategy as in Proposition 4, there is a weaker uniqueness property for inferred types.

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

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

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

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

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

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

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

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

$$\frac{\Gamma, z : \Sigma\, x : A.B \vdash P : \square_i \qquad \Gamma, x : A, y : B \vdash b : P[z := (x, y)] \qquad \Gamma \vdash s : \Sigma\, x : A.B}{\Gamma \vdash \mathrm{rec}_\Sigma(z.P, x.y.p, s) : P[z := s]}\,\Sigma\text{-REC}$$

Figure 3 Undirected sum type

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

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

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

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

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

$$\frac{\Gamma \vdash t \triangleright T \qquad T \rightsquigarrow^* \Sigma\, x : A.B}{\Gamma \vdash t \triangleright_\Sigma \Sigma\, x : A.B}\,\Sigma\text{-Inf}$$

Figure 4 Bidirectional sum type

3.2 Inductive Types

Sum type Before we turn to the general case of inductive types of the formalisation, let us present a simple inductive type: dependent sums. The undirected rules are given in Figure 3, and are inspired from the theoretical presentation of such dependent sums, such at the one of the Homotopy Type Theory book [26]. In particular, we use the same convention to write $y.P$ when variable y is bound in P. Note however that contrarily to [26], some typing information is kept on the pair constructor. Exactly as for the abstraction, this is to be able to infer a unique, most general type in the bidirectional system. Indeed, without that information a pair (a, b) could inhabit multiple types $\Sigma\, x : A.B$ because there are potentially many incomparable types B such that $B[x := a]$ is a type for b, as even if $B[x := a]$ and $B'[x := a]$ are convertible B and B' may be quite different, depending of which instances of a in $B[x := a]$ are abstracted to x.

To obtain the bidirectional rules of Figure 4, first notice that all undirected rules are structural and must thus become inference rules if we want the resulting system to be complete, just as in Section 2. The question therefore is again to know which modes to

choose for the premises. For Σ-TYPE and Σ-CONS this is straightforward: when the type appears in the conclusion, use checking, otherwise (constrained) inference. The case of the destructors is somewhat more complex. Handling the subterms of the destructor in the order in which they usually appear (predicate, branches and finally scrutinee) is not possible, as the parameters of the inductive type are needed to construct the context in which the predicate is typed. However those parameters can be inferred from the scrutinee. Thus, a type for the scrutinee is first obtained using a new constrained inference judgment, forcing the inferred type to be a Σ-type, but leaving its parameters free. Next, the obtained arguments can be used to construct the context to type the predicate. Finally, once the predicate is known to be well-formed, it can be used to type-check the branch.

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

$$\frac{}{\Gamma \vdash \mathbb{N} \triangleright \square_0} \qquad \frac{}{\Gamma \vdash 0 \triangleright \mathbb{N}} \qquad \frac{\Gamma \vdash n \triangleleft \mathbb{N}}{\Gamma \vdash \mathrm{S}(n) \triangleright \mathbb{N}}$$

$$\frac{\Gamma \vdash s \triangleright_{\mathbb{N}} \mathbb{N} \qquad \Gamma, z : \mathbb{N} \vdash P \triangleright_{\square} \square_i \qquad \Gamma \vdash b_0 \triangleleft P[z := 0] \qquad \Gamma, x : \mathbb{N}, p : P[z := x] \vdash b_{\mathrm{S}} \triangleleft P[z := \mathrm{S}(x)]}{\Gamma \vdash \mathrm{rec}_{\mathbb{N}}(z.P, b_0, x.p.b_{\mathrm{S}}, s) \triangleright P[z := s]}$$

$$\frac{\Gamma \vdash A \triangleright_{\square} \square_i \qquad \Gamma \vdash a \triangleleft A \qquad \Gamma \vdash a' \triangleleft A}{\Gamma \vdash \mathrm{Id}_A\ a\ a' \triangleright \square_i} \qquad \frac{\Gamma \vdash A \triangleright_{\square} \square_i \qquad \Gamma \vdash a \triangleleft A}{\Gamma \vdash \mathrm{refl}_A\ a \triangleright \mathrm{Id}_A\ a\ a}$$

$$\frac{\Gamma \vdash s \triangleright \mathrm{Id}_A\ a\ a' \qquad \Gamma, x : A, z : \mathrm{Id}_A\ a\ x \vdash P \triangleright_{\square} \square_i \qquad \Gamma \vdash b \triangleleft P[z := \mathrm{Id}_A\ a\ a][x := a]}{\Gamma \vdash \mathrm{rec}_{\mathrm{Id}}(x.z.P, b, s) \triangleright P[z := s][x := a']}$$

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

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

Figure 5 Other bidirectional inductive types

This same approach can be readily extended to other usual inductive types, with recursion or indices posing no specific problems, see Figure 5. The choice to use $\triangleright_{\square}$ rather than $\triangleleft$ for types is guided by the intuition that the universe level of e.g. A in $\mathrm{Id}_A\ a\ a'$ is free, similarly to what happens for sum types.

Polymorphic, Cumulative Inductive Types The account of general inductive types in PCUIC is quite different from the one we just gave. The main addition is universe polymorphism [25], which means that inductive types and constructors come with explicit universe levels. The Σ-type of the previous paragraph, for instance, would contain an explicit universe level i, and both A and B would be checked against $\square_i$ rather than having their level inferred. This makes the treatment of complex inductive types possible by using checking uniformly – rather than relying on constrained inference to infer universe levels – at the cost of possibly needless annotations, as here with Σ-types. To make that polymorphism more seamless, those polymorphic inductive types are also cumulative [27]: in much the same way as $\square_i \preceq \square_j$ if $i \leq j$, also $\mathbb{N}^{@i} \preceq \mathbb{N}^{@j}$, where $\mathbb{N}^{@i}$ denotes the polymorphic inductive $\mathbb{N}$ at universe level i.

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

$$\frac{}{\Gamma \vdash I^{@\mathbf{i}} \triangleright \Pi(\mathbf{x} : \mathbf{X}^{@\mathbf{i}})(\mathbf{y} : \mathbf{Y}^{@\mathbf{i}}), \square_{l^{@\mathbf{i}}}}\textsc{Ind} \qquad \frac{}{\Gamma \vdash c_k^{@\mathbf{i}} \triangleright \Pi(\mathbf{x} : \mathbf{X}^{@\mathbf{i}})(\mathbf{y} : \mathbf{Y_k}^{@\mathbf{i}}), I^{@\mathbf{i}}\ \mathbf{x}\ \mathbf{u_k}^{@\mathbf{i}}}\textsc{Cons}$$

$$\frac{\begin{array}{c}\Gamma \vdash s \triangleright_I I^{@\mathbf{i}'}\ \mathbf{a}\ \mathbf{b} \qquad \Gamma \vdash \mathbf{p}_k \triangleleft \mathbf{X}_k[\mathbf{x} := \mathbf{p}] \qquad \Gamma, \mathbf{y} : \mathbf{Y}^{@\mathbf{i}}[\mathbf{x} := \mathbf{p}], z : I^{@\mathbf{i}}\ \mathbf{p}\ \mathbf{y} \vdash P \triangleright_\square \square_j \\ I^{@\mathbf{i}'}\ \mathbf{a}\ \mathbf{b} \preceq I^{@\mathbf{i}}\ \mathbf{p}\ \mathbf{b} \qquad \Gamma, \mathbf{y} : \mathbf{Y_k}^{@\mathbf{i}}[\mathbf{x} := \mathbf{p}] \vdash \mathbf{t}_k \triangleleft P[z := c_k^{@\mathbf{i}}\ \mathbf{p}\ \mathbf{y}][\mathbf{y} := \mathbf{u_k}^{@\mathbf{i}}]\end{array}}{\Gamma \vdash \mathtt{match}\ s\ \mathtt{in}(I, \mathbf{i}, \mathbf{p})\ \mathtt{return}\ P\ \mathtt{with}[\mathbf{t}] \triangleright P[z := s][\mathbf{y} := \mathbf{b}]}\textsc{Match}$$

$$\frac{\Gamma \vdash T \triangleright_\square \square_i \qquad \Gamma, f : T \vdash t \triangleleft T \qquad \text{guard condition}}{\Gamma \vdash \mathtt{fix}\ f : T := t \triangleright T}\textsc{Fix}$$

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

$$\frac{\Gamma \vdash t \triangleright T \qquad T \rightsquigarrow I\ \mathbf{a}\ \mathbf{b}}{\Gamma \vdash t \triangleright_I I\ \mathbf{a}\ \mathbf{b}}$$

Figure 6 Bidirectional inductive type – PCUIC style

This enables lifting from a lower universe to a higher one, so that for instance $\vdash 0^{@i} : \mathbb{N}^{@j}$ if $i \leq j$. PCUIC as presented in MetaCoq also presents constructors and inductive types as functions, rather than requiring them to be fully applied, and it separates recursors into a pattern-matching and a fixpoint construct, the latter coming with a specific guard condition to keep the normalization property enjoyed by a system with recursors.

A sketch of the bidirectional rules is given in Figure 6, for a generic inductive I. We use bold characters to denote lists – for instance $\mathbf{a}$ is a list of terms – and indexes to denote a specific element – so that $\mathbf{a}_k$ is the k-th element of the previous. The considered inductive I has parameters of type $\mathbf{X}$, indices of type $\mathbf{Y}$ and inhabits some universe $\square_l$. Its constructors c_k are of types $\Pi(\mathbf{x} : \mathbf{X})(\mathbf{y} : \mathbf{Y_k}), I\ \mathbf{x}\ \mathbf{u}$, with $\mathbf{u_k}$ terms possibly depending on both $\mathbf{x}$ and $\mathbf{y}$. Since we are considering a polymorphic inductive type, all of those actually have to be instantiate with universe levels, an operation we denote with $^{@\mathbf{i}}$.

The two rules Ind, Cons are similar to those for variables, with the types pulled out of a global environment – not represented in our rules – rather than of the context. In particular, this presentation completely fixes the universe levels of the arguments. In rule Fix, the type of the fixpoint is checked to be well-formed, and then the body is checked against it. The guard condition, although complex, does not vary between the directed and undirected systems, and we thus do not dwell on it. The formalised version of this rule is complicated by the need to consider mutual (co-)fixpoints, but follows the same pattern.

Last but not least, Match follows the same structure as in Figures 4 and 5: first, the type of the scrutinee is inferred, then the predicate is verified to be a type and finally the branches are checked. An important point is how much information can be retrieved from the scrutinee s. Indeed, the universe levels $\mathbf{i}$ and the parameters $\mathbf{p}$ used to build the context in which the predicate P and branches $\mathbf{t}$ are typed are stored in the match constructor. For cumulative inductive types, this is crucial to retain equivalence between the undirected and bidirectional system, and wrongly building the context from the type inferred for the scrutinee led to the bug we discovered. The idea is that the match construction might need to be typed "higher" than the type of inferred for s to be able to type P and $\mathbf{t}$. Subsequently,

a cumulativity check not appearing in the examples above is needed to ensure that the scrutinee checks against the type constructed using **i** and **p**. In contrast with parameters, the inferred indices **b** can safely be used in the return type, but proving this is the most subtle part of the correctness proof.

3.3 The formalisation

Let us now go over the formalisation file by file.

BDEnvironmentTyping.v & BDTyping.v The first file refines a few definitions on contexts from EnvironmentTyping.v in order to account for the difference between checking and constrained inference. We expect this to eventually replace the less precise definitions.

The second contains the definition of the bidirectional type system as a mutually defined inductive type whose main part is `infering`. The best way to understand this inductive is probably to compare it with `typing`, the inductive predicate for undirected typing.

We then go on to proving by hand a custom induction principle, by first introducing a notion of size of a derivation. This induction principle is not as strong as we might expect, as it does not provide the extra induction hypothesis on context and type that would go with McBride's discipline. We did not try to go and prove such a strong induction principle, as we did not need it. Instead, reflecting the discipline in the choice of the predicates proven by induction was enough. But the main reason was that proving such an induction principle effectively corresponds to an inline proof of validity, a property that required quite an important amount of work to get. We still conjecture that such a strong induction principle should be provable, by reproducing some of the lemmas on the undirected typing, with extra care taken to the size of the obtained typing derivations, so as to be able to use e.g. substitutivity of typing together with an induction hypothesis.

BDToPCUIC.v & BDFromPCUIC.v The next two files prove the equivalence between both type system. Correctness (akin to Theorem 2) is `infering_typing` for inference and the following theorems for the other judgements. Completeness (akin to Theorem 6) is theorem `typing_infering`.

The bulk of both proofs is an induction on typing derivation whose most challenging part is the handling of the case constructor, especially the subtle issues around indices described in Section 3.2. Similarly to Section 2, correctness relies on the strong properties of validity and subject reduction to reconstruct missing premises, while completeness mostly requires transitivity of conversion and confluence of reduction.

BDUnique.v This last file contains the proofs of Proposition 7 – `uniqueness_inferred` – and Proposition 8 – `principal_type`. Apart from some lemmas on conversion that were only proved for cumulativity in MetaCoq, the induction itself is quite straightforward thanks to the bidirectional structure.

4 Beyond PCUIC

The use of our bidirectional structure is not limited to CIC or PCUIC. On the contrary, we found it crucial to have such a bidirectional type system when designing a gradual extension to CIC [13], for multiple reasons we try and detail below.

But let us first give a bit of context about this extension. The aim was to adapt the ideas of gradual typing [21] to CIC. Gradual typing aims at incorporating some level of dynamic

typing into a static typing discipline. To do so, a new type constructor ? is introduced to represent dynamic type information. At typing time, this ? should be seen as a wildcard that represents "any type" that is to be treated optimistically. In particular, ? should be considered convertible to any type. Conversion is thus replaced by a new relation, called consistency, that corresponds to this intuition. In effect it behaves somewhat similarly to unification, with each ? corresponding to a unification variable. This means in particular that consistency is *not* transitive, as if it were it would be useless: since any type T is consistent with ?, if consistency were transitive any two types would be related.

Localized computation The free-standing conversion rule Conv is powerful, but sometimes too much.

This was our first use for the bidirectional structure. Indeed, multiple uses of a consistency in a row would have allowed to change the type of a term to any other arbitrary type by going through ? using two conversion rules in a row. Thus, any term would have been typable! Being able to use the conversion rule unrestricted was too much. Instead, the bidirectional system enforces a more localized use of conversion: only once, at the interface between inference and checking. This restriction was enough to make the conversion rule meaningful again.

More generally, since the equivalence between the undirected and directed variants relies on the properties listed in Proposition 1, when one of these fails the equivalence is endangered. When one envisons a system where this would be the case, the bidirectional approach might be worth considering, as it could stay viable while its undirected counterpart might not.

Modes for the conversion rule The observation made in Section 2.2 that the unique Conv rule serves two different roles, which is clarified by the separation between checking and constrained inference, is an important one when toying with computation. Indeed, those two different aspects must be accounted for if one wishes to modify conversion and/or reduction. In particular, modifying the definition of conversion without accounting for the specific role of reduction would make rules for checking and constrained inference come out of sync, bringing trouble down the road.

Taking again the example of [13], the Check is modified by directly replacing conversion with the consistency relation usual in gradual typing. But this is not enough, because constrained inference must be handled as well. This is done by supplementing rule Sort-Inf by another rule to treat the case when the inferred type reduces to the wildcard ?, that can be used as a type – with some care taken. The same happens for all constrained inference rules.

Bidirectional elaboration In works such as [20, 7, 13], 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 corresponding t' in another 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 bidirectional structure readily adapts to those settings, with the extra term t' simply considered as an output of all judgements. As such, McBride's discipline as described in Section 2.2 demands that when, in a context Γ, the subject t elaborates to t' while inferring type T, we should have $\Gamma \vdash t' : T$ – 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

Traces of constrained inference 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 [19], $\Gamma \vdash t : T$ 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 [20] 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 [22] 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 $\{f : \mathbb{N} \to \mathbb{N} \mid f\ 0 = 0\}$ should be usable as a function of type $\mathbb{N} \to \mathbb{N}$. An erasure procedure is therefore required on top of reduction to remove subset types in the places where we use constrained inference.

Abel and Coquand [4] use a judgement written $\Delta \vdash V\delta \Uparrow \mathrm{Set} \rightsquigarrow i$, where a type V is checked to be well-formed, but with its exact level i free. This corresponds very closely to our use of $\triangleright_{\square}$.

Traces can also be found in the description of Matita's elaboration algorithm [7]. Indeed, the presence of meta-variables on top of coercions 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 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 [13] in Section 4, where the bidirectional structure is crucial in describing a gradual extension to CIC. In particular, and similarly to what happens with meta-variables in [7], all constrained inference rules are duplicated: there is one rule when the head constructor is the desired one, and a second one to handle the gradual wildcard.

5.2 Completeness

Quite a few articles tackle the problem of bidirectional typing in a setting with an untyped – so called Curry-style – abstraction. This is the case of early work by Coquand [11], the type system of Agda as described in [17], the systems considered by Abel in many of his papers [3, 4, 2, 5], and much of the work of McBride [14, 15, 16] on the topic. In such systems,

λ-abstractions can only be checked against a given type, but cannot infer one, so that only terms with no β-redexes are typable. Norell [17] argues that such β-redexes are uncommon in real-life programs, so that being unable to type them is not a strong limitation in practice. To circumvent this problem, McBride also adds the possibility of typing annotations to retain the typability of a term during reduction.

While this approach is adapted to programming languages, where the emphasis is on lightweight syntax, it is not tenable for a proof assistant kernel, where all valid terms should be accepted. Indeed, debugging a proof that is rejected because the kernel fails to accept a perfectly well-typed term the user never wrote – as most proofs are generated rather than written directly – is simply not an option.

In a setting with typed – Church-style – abstraction, if one wishes to give the possibility for seemingly untyped abstraction, another mechanism has to be resorted to, typically meta-variables. This is what is done in Matita [7], where the authors combine a rule similar to Abs – where the type of an abstraction is inferred – with another one, similar to the Curry-style one – where abstraction is checked – looking like this:

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

While such a rule would make a simple system such as that of Section 2 "over-complete", it is a useful addition to enable information from checking to be propagated upwards in the derivation. This is crucial in systems where completeness is lost, such as Matita's elaboration. Similar rules are described in [7] for let-bindings and constructors of inductive types.

Although only few authors consider the problem of a complete bidirectional algorithm for type-checking dependent types, we are not the first to attack it. Already Pollack [19] does, and the completeness proof for CCω of Section 2 is very close to one given in his article. Another proof of completeness for a more complex CIC-like system can be found in [22]. None of those however tackle as we do the whole complexity of PCUIC.

5.3 Inputs and outputs

We already credited the discipline we adopt on well-formedness of inputs and outputs to McBride [15, 16]. A similar idea has also appeared independently in [9]. Bauer and his co-authors introduce the notions of a (weakly) presuppositive type theory [9, Def. 5.6] and of well-presented premise-family and rule-boundary [9, Def. 6.16 and 6.17] to describe a discipline similar to ours, using what they call the boundary of a judgment as the equivalent of our inputs and outputs. Due to their setting being undirected, this is however more restrictive, because they are not able to distinguish inputs from outputs and thus cannot relax their condition to only demand inputs to be well-formed but not outputs.

6 Conclusion

We have described a judgmental presentation of the bidirectional structure of typing algorithms in the setting of dependent types. In particular, we identified a new family of judgements we called constrained inference. Those have no counterpart in the non-dependent setting, as they result from a choice of modes for the conversion rule, which is specific to the dependent setting. We proved our bidirectional presentation equivalent to an undirected one, both on paper on the simple case of CCω, and formally in the much more complex and realistic setting of PCUIC. Finally, we gave various arguments for the usefulness of

our presentation as a way to ease proofs, an intermediate between undirected type-systems and typing algorithms, a solid basis to design new type systems, and a tool to re-interpret previous work on type systems in a clearer way.

Regarding future work, a type-checking algorithm is already part of MetaCoq, and we should be able to use our bidirectional type system to give a pleasant completeness proof by separating the concerns pertaining to bidirectionality from the algorithmic problems, such as implementation of an efficient conversion check or proof of termination. More broadly, bidirectional type systems should be an interesting tool in the feat of incorporating in proof assistants features that have been satisfactorily investigated on the theoretical level while keeping a complete and correct kernel, avoiding the pitfall of cumulative inductive type's incomplete implementation in Coq. A first step would be to investigate the discrepancies between the two kinds of presentations of inductive types Section 3, and in particular if all informations currently stored in the match node are really needed or if a more concise presentation can be given. But we could go further and study how to handle cubical type theory [28], rewrite rules [10], setoid type theory [6], exceptional type theory [18], η-conversion... There might also be an interesting link to make with the current work on normalization by evaluation [1] as an alternative to weak-head reduction for constrained inference. Finally, we hope that our methodology will be adapted as a base for other theoretical investigations. As a way to ease this adoption, studying it in a general setting such as that of [9] might be a strong argument for adoption.

References

1 Andreas Abel. Towards normalization by evaluation for the $\beta\eta$-calculus of constructions. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, volume 6009 of *Lecture Notes in Computer Science*, pages 224–239. Springer-Verlag, 2010. doi:10.1007/978-3-642-12251-4.

2 Andreas Abel and Thorsten Altenkirch. A partial type checking algorithm for type:type. *Electronic Notes in Theoretical Computer Science*, 229(5):3–17, 2011. Proceedings of the Second Workshop on Mathematically Structured Functional Programming (MSFP 2008). doi:10.1016/j.entcs.2011.02.013.

3 Andreas Abel and Thierry Coquand. Untyped algorithmic equality for Martin-Löf's logical framework with surjective pairs. *Fundamenta Informaticae*, 77(4):345–395, 2007. TLCA'05 special issue.

4 Andreas Abel, Thierry Coquand, and Peter Dybjer. Verifying a semantic $\beta\eta$-conversion test for Martin-Löf type theory. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction*, pages 29–56, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

5 Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.*, December 2017. doi:10.1145/3158111.

6 Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. Setoid type theory - a syntactic translation. In *MPC 2019 - 13th International Conference on Mathematics of Program Construction*, volume 11825 of *LNCS*, pages 155–196. Springer, October 2019. URL: https://hal.inria.fr/hal-02281225, doi:10.1007/978-3-030-33636-3_7.

7 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions. *Logical Methods in Computer Science*, Volume 8, Issue 1, March 2012. doi:10.2168/LMCS-8(1:18)2012.

8 Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*. Oxford Universiy Press, 1992.

9 Andrej Bauer, Philipp G. Haselwarter, and Peter LeFanu Lumsdaine. A general definition of dependent type theories. Preprint, 2020. arXiv:2009.05539.

10 Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. The Taming of the Rew: A Type Theory with Computational Assumptions. *Proceedings of the ACM on Programming Languages*, 2021. doi:10.1145/3434341.

11 Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1), 1996. doi:10.1016/0167-6423(95)00021-6.

12 Trevor Jim. What are principal typings and what are they good for? In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, page 42–53, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/237721.237728.

13 Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. Gradualizing the calculus of inductive constructions. Preprint, 2020. arXiv:2011.10618.

14 Conor McBride. I got plenty o' nuttin'. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 207–233. Springer International Publishing, 2016. doi:10.1007/978-3-319-30936-1_12.

15 Conor McBride. Basics of bidirectionalism. Blog post, August 2018. URL: https://pigworker.wordpress.com/2018/08/06/basics-of-bidirectionalism/.

16 Conor McBride. Check the box! In *25th International Conference on Types for Proofs and Programs*, June 2019.

17 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, 9 2007.

18 Pierre-Marie Pédrot and Nicolas Tabareau. Failure is not an option an exceptional type theory. In *ESOP 2018 - 27th European Symposium on Programming*, volume 10801 of *LNCS*, pages 245–271, Thessaloniki, Greece, 2018. Springer. doi:10.1007/978-3-319-89884-1_9.

19 R. Pollack. Typechecking in Pure Type Systems. In *Informal Proceedings of the 1992 Workshop on Types for Proofs and Programs, Båstad, Sweden*, pages 271–288, 6 1992. URL: http://homepages.inf.ed.ac.uk/rpollack/export/BaastadTypechecking.ps.gz.

20 Amokrane Saïbi. Typing algorithm in type theory with inheritance. *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '97*, 1997. doi:10.1145/263699.263742.

21 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 274–293. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015. doi:10.4230/LIPIcs.SNAPL.2015.274.

22 Matthieu Sozeau. Subset coercions in coq. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, pages 237–252, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. doi:10.1007/978-3-540-74464-1_16.

23 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, 2 2020. URL: https://hal.inria.fr/hal-02167423, doi:10.1007/s10817-019-09540-0.

24 Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq. *Proceedings of the ACM on Programming Languages*, pages 1–28, January 2020. URL: https://hal.archives-ouvertes.fr/hal-02380196, doi:10.1145/3371076.

25 Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in Coq. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving*, pages 499–514, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-08970-6_32.

26 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

27 Amin Timany and Matthieu Sozeau. Cumulative Inductive Types In Coq. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*, volume 108 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1–16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.FSCD.2018.29.

28 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019. doi:10.1145/3341691.