A BI-DIRECTIONAL REFINEMENT ALGORITHM FOR THE CALCULUS OF (CO)INDUCTIVE CONSTRUCTIONS

ANDREA ASPERTI, WILMER RICCIOTTI, CLAUDIO SACERDOTI COEN, AND ENRICO TASSI

Dipartimento di Scienze dell'informazione, Mura Anteo Zamboni 7, 40127, Bologna, Italy
e-mail address: asperti@cs.unibo.it

Dipartimento di Scienze dell'informazione, Mura Anteo Zamboni 7, 40127, Bologna, Italy
e-mail address: ricciott@cs.unibo.it

Dipartimento di Scienze dell'informazione, Mura Anteo Zamboni 7, 40127, Bologna, Italy
e-mail address: sacerdot@cs.unibo.it

Microsoft Research-INRIA Joint Centre, Building I, Parc Orsay Université, 28, rue Jean Rostand, 91893 Orsay Cedex
e-mail address: enrico.tassi@inria.fr

ABSTRACT. The paper describes the refinement algorithm for the Calculus of (Co)Inductive Constructions (CIC) implemented in the interactive theorem prover Matita.

The refinement algorithm is in charge of giving a meaning to the terms, types and proof terms directly written by the user or generated by using tactics, decision procedures or general automation. The terms are written in an "external syntax" meant to be user friendly that allows omission of information, untyped binders and a certain liberal use of user defined sub-typing. The refiner modifies the terms to obtain related well typed terms in the internal syntax understood by the kernel of the ITP. In particular, it acts as a type inference algorithm when all the binders are untyped.

The proposed algorithm is bi-directional: given a term in external syntax and a type expected for the term, it propagates as much typing information as possible towards the leaves of the term. Traditional mono-directional algorithms, instead, proceed in a bottom-up way by inferring the type of a sub-term and comparing (unifying) it with the type expected by its context only at the end. We propose some novel bi-directional rules for CIC that are particularly effective. Among the benefits of bi-directionality we have better error message reporting and better inference of dependent types. Moreover, thanks to bi-directionality, the coercion system for sub-typing is more effective and type inference generates simpler unification problems that are more likely to be solved by the inherently incomplete higher order unification algorithms implemented.

Finally we introduce in the external syntax the notion of vector of placeholders that enables to omit at once an arbitrary number of arguments. Vectors of placeholders allow a trivial implementation of implicit arguments and greatly simplify the implementation of primitive and simple tactics.

1. Introduction

In this paper we are interested in describing one of the key ingredients in the implementation of Interactive Theorem Provers (ITP) based on type theory.

The architecture of these tools is usually organized in layers and follows the so called de Bruijn principle: the correctness of the whole system solely depends on the innermost component called kernel. Nevertheless, from a user perspective, the most interesting layers are the external ones, the ones he directly interacts with. Among these, the *refiner* is the one in charge of giving a meaning to the terms and types he writes. The smarter the refiner is, the more freedom the user has in omitting pieces of information that can be reconstructed. The refiner is also the component generating the majority of error messages the user has to understand and react to in order to finish his proof or definition.

This paper is devoted to the description of a refinement algorithm for the Calculus of (Co)Inductive Constructions, the type theory on which the Matita [6], Coq [12] and Lego [19] ITPs are based on.

1.1. **Refinement.** In this and in the previous paper [4] we are interested in the implementation of interactive theorem provers (ITP) for dependently typed languages that are heavily based on the Curry-Howard isomorphism. Proofs are represented using lambda-terms. Proofs in progress are represented using lambda-terms containing metavariables that are implicitly existentially quantified. Progression in the proof is represented by instantiation of metavariables with terms. Metavariables are also useful to represent missing or partial information, like untyped lambda-abstractions or instantiation of polymorphic functions to omitted type arguments.

Agda [8] and Matita [6] are examples of systems implemented in this way. Arnaud Spiwack in his Ph.D. thesis [31] partially describes a forthcoming release of Coq 8.4 that will be implemented on the same principles.

The software architecture of these systems is usually built in layers. The innermost layer is the *kernel* of the ITP. The main algorithm implemented by the kernel is the *type checker*, which is based in turn on *conversion* and *reduction*. The type checker takes as input a (proof) term possibly containing metavariables and it verifies if the partial term is correct so far. To allow for type-checking, metavariables are associated to sequents, grouping their types together with the context (hypotheses) available to inhabit the type. The kernel does not alter metavariables since no instantiation takes place during reduction, conversion or type checking.

The kernel has the important role of reducing the trusted code base of the ITP. Indeed, the kernel eventually verifies all proofs produced by the outer layers, detecting incorrect proofs generated by bugs in those layers. Nevertheless, the user never interacts directly with the kernel and the output of the kernel is just a boolean that is never supposed to be false when the rest of the system is bug free. The most interesting layers from the user point of view are thus the outer layers. The implementation of a kernel for a variant of the Calculus of (Co)Inductive Constructions (CIC) has been described in [4] down to the gory details that make the implementation efficient.

The next layer is the *refiner* and is the topic of this paper. The main algorithm implemented by the refiner is the refinement algorithm that tries to infer as much information as it is needed to make its input meaningful. In other words it takes as input a partial term, written in an "external syntax", and tries to obtain a "corresponding" well typed term.

The input term can either be user provided or it can be a partial proof term generated by some proof command (called tactic) or automation procedure. The gap between the external and internal syntax is rather arbitrary and system dependent. Typical examples of external syntaxes allow for:

- Untyped abstractions. Hence the refiner must perform type inference to recover the explicit types given to bound variables. The polymorphism of CIC is such that binders are required to be typed to make type checking decidable.
- Omission of arguments, in particular omission of types used to instantiate polymorphic functions. Hence the refiner must recover the missing information during type inference to turn implicit into explicit polymorphism.
- Linear placeholders for missing terms that are not supposed to be discovered during type inference. For instance, a placeholder may be inserted by a tactic to represent a new proof obligation. Hence the refiner must turn the placeholder into a metavariable by constraining the set of free variables that may occur in it and the expected type.
- Implicit ad-hoc sub-typing determined by user provided cast functions (called coercions) between types or type families. Hence the refiner must modify the user provided term by explicitly inserting the casts in order to let the kernel completely ignore sub-typing.

Coercions are user provided functions and are thus free to completely ignore their input. Thus a refiner that handles coercions is actually able to arbitrarily patch wrong user provided terms turning them into arbitrarily different but well typed terms. Moreover, the insertion of a coercion between type families can also introduce new metavariables (the family indexes) that play the role of proof obligations for pre-conditions of the coercion. For instance, a coercion from lists to ordered lists can open a proof obligation that requires the list to be sorted.

The refiner is the most critical system component from the user point of view since it is responsible for the "intelligence" of the ITP: the more powerful the refiner is, the less information is required from the user and the simpler the outer layers become. For instance, a series of recent techniques that really improve the user experience have all been focused in the direction of making the refiner component more powerful and extensible by the user. Canonical structures [16], unification hints [5] and type classes [30] are devices that let the user drive some form of proof search that is seamlessly integrated in the refinement process. While the latter device is directly integrated into the refinement algorithm, the first two are found in the unification algorithm used by the refiner.

They all make it possible to achieve similar objectives, the second being more general than the first and the last two being incomparable from the point of view of efficiency (where the second is best) and expressiveness (where the third is more flexible). The implementation of type classes done in Coq is actually provided by an additional layer outside the refiner for historical reasons.

In this paper we will describe only the refinement algorithm implemented in a refiner for a variant of the Calculus of (Co)Inductive Constructions. The algorithm is used in the forthcoming major release of the Matita[1] ITP (1.0.x). The algorithm calls a unification algorithm that will be specified in this paper and described elsewhere. We do not consider type classes in our refinement algorithm since we prefer to assume the unification algorithm

---

[1]Matita is free software available at `http://matita.cs.unibo.it`

to implement unification hints. Nevertheless, type classes can be easily added to our algorithm with minor modifications and indeed the relevant bits that go into the refiner are implemented in Matita.

Before addressing bi-directionality, which is a peculiarity of the algorithm that has not been fully exploited yet[2] for the CIC, we just conclude our overview of an ITP architecture by talking about the next layer. The next layer after the refiner is that of *tactics*. This layer is responsible for implementing commands that help the user in producing valid proof terms by completely hiding to him the proof terms themselves. Tactics range from simple ones that capture the introduction and elimination rules of the connectives (called primitive tactics) to complicated proof automation procedures. The complexity of proof automation is inherent in the problem. On the other hand, primitive tactics should be as simple as building small partial proof terms. For instance, to reduce a proof of $A \Rightarrow B$ to a proof of B given A it is sufficient to instantiate the metavariable associated to the sequent $\vdash A \Rightarrow B$ with the term $\lambda x.?$ in external syntax where ? is a placeholder for a new proof obligation. This is possible when the refinement algorithm is powerful enough to refine $\lambda x.?$ to $\lambda x : A.?_1$ where $?_1$ is a new metavariable associated to the sequent $x : A \vdash B$. When this is not the case or when the refiner component is totally missing, the tactic is forced to first perform an analysis of the current goal, then explicitly create a new metavariable and its sequent, and then emit the new proof term $\lambda x : A.?_1$ directly in the internal syntax.

1.2. **Bi-directionality.** When the external syntax of our ITP allows to omit types in binders, the refinement algorithm must perform type inference. Type inference was originally studied in the context of lambda-calculi typed a la Curry, where no type information can be attached at all to the binders. The traditional algorithm for type inference, now called uni-directional, performs type inference by first traversing the term in a top-down way. When a binder is met, a new metavariable (usually called type or unification variable in this context) is introduced for the type of the bound variable. Then type constraints are solved traversing the term in a bottom-up way. When the variable or, more generally, a term is used in a given context, its type (called inferred type) is constrained to be compatible with the one expected by the context (called expected type). This triggers a unification problem.

Type inference, especially for the Hindley-Milner type system, gives the possibility to write extremely concise programs by omitting all types. Moreover, it often detects a higher degree of polymorphism than the one expected by the user. Unluckily, it has some drawbacks. A minor one is that types are useful for program documentation and thus the user desires to add types at least to top level functions. In practice, this is always allowed by concrete implementations. Another problem is error reporting: a typing error always manifests itself as a mismatch between an inferred and an expected type. Nevertheless, an error can be propagated to a very distant point in the code before being detected and the position where it is produced. The mismatch itself can be non informative about where the error actually is. Finally, unification quickly becomes undecidable when the expressive power of the type system increases. In particular, it is undecidable for higher order logic and for dependent types.

---

[2]The refinement algorithm of Coq 8.3, the most widespread implementation of CIC, is almost mono-directional with only the lambda-abstraction case handled in a bi-directional way. Many other interesting cases of bi-directionality are obtained in this paper for inductive types and constructors.

To avoid or mitigate the drawbacks of type inference, bi-directional type-checking algorithms have been introduced in the literature [24]. These algorithms take as input a λ-term typed a la Curry and an expected top-level type and they proceed in a top-down manner by propagating the expected type towards the leaves of the term. Additional expected types are given in local definitions, so that all functions are explicitly documented. Error detection is improved by making it more local. The need for unification is reduced and, for simple type systems, unification is totally avoided. Some terms, in particular β-redexes, are no longer accepted, but equivalent terms are (e.g. by using a local definition for the head). An alternative consists of accepting all terms by re-introducing a dependency over some form of unification.

Bi-directionality also makes sense for languages typed à la Church, like the one we consider here. In this case the motivations are slightly different. First of all, typing information is provided both in the binders and at the top-level, in the form of an expected type. Hence information can flow in both direction and, sooner or later, the need to compare the expected and inferred types arises. In the presence of implicit polymorphism, unification is thus unavoidable. Because of dependent types and metavariables for proof obligations, we need the full power of higher order unification. Moreover, again because of unification, the problem remains undecidable also via using a bi-directional algorithm. Hence avoiding unification is no longer a motivation for bi-directionality. The remaining motivations for designing a bi-directional refinement algorithm for CIC are the following:

Improved error messages. A typing error is issued every time a mismatch is found between the inferred and expected type. With a mono-directional algorithm, the mismatch is always found at the end, when the typing information reaches the expected type. In a bi-directional setting the expected type is propagated towards the leaves and the inferred type towards the root, the mismatch is localized in smaller sub-terms and the error message is simpler. For instance, instead of the message "*the provided function has type $A \Rightarrow List\ B$ but it is supposed to have type $A \Rightarrow List\ C$*" related to a whole function definition one could get the simpler message "*the list element has type B but it is supposed to have type C*" related to one particular position in the function body.

Improvement of the unification algorithm. To make the system responsive, the semi-decidable unification algorithm is restricted to always give an answer in a finite amount of time. Hence the algorithm could fail to find a solution even when a solution exists. For instance, the algorithms implemented in Coq and Matita are essentially backtracking free and they systematically favor projections over mimics: when unifying an applied metavariable $?_1\ a\ b\ c$ with $a\ b$ (for some a, b, c closed in a context Γ), the system instantiates $?_1$ with $\lambda x, y, z.y$ rather than $\lambda x, y, z.b$ (where $x, y, z \notin dom(\Gamma)$). Moreover, unification for CIC does not admit a most general unifier and it should recursively enumerate the set of solutions. However, it is usual in system implementations to let unification return just one solution and to avoid back-tracking in the refinement algorithm[3]. Thus, if the solution found by unification is correct locally, but not globally, refinement will fail. Thanks to bi-directionality, unification problems often become more instantiated and thus simpler, and they also admit fewer solutions. In particular, in the presence of dependent types, it is easy to find practical

---

[3]To the authors knowledge, Isabelle [18] is the only interactive prover implementing Huet's algorithm [17] capable of generating all second order unifiers

examples where the unification algorithm finds a solution only on the problems triggered by the bi-directional algorithm.

An interesting and practical example that motivated our investigation of bi-directionality is the following. Consider a dependently typed data-type (Term S) that represents the syntax of a programming language with binders. Type dependency is exploited to make explicit the set S of variables bound in the term and every variable occurrence must come with a proof that the variable occurs in the bound variables list: (Var S x I) has type (Term S) where x is a variable name, I is a proof of True and Var has type $\forall S.\forall a : \text{String}.x \in S \to \text{Term } S$ where $x \in S$ is a computable function that reduces to True when x belongs to S and to False otherwise. Consider now the term (Lambda ? x (Var ? x I)) in concrete syntax that represents $\lambda x.x$ in our programming language. Note that no information about the set of bound variables has been provided by the user. Thus it is possible to simply define notational macros so that the user actually writes $\lambda x.x$ and this is expanded[4] to Lambda ? x (Var ? x I). A uni-directional refiner is unlikely to accept the given term since it should guess the right value for the second placeholder ? such that $x \in$? reduces to True and ? is the set of variables actually bound in the term. The latter information is not local and it is still unknown in the bottom-up, uni-directional approach. On the other hand, a bi-directional refiner that tries to assign type Term $\emptyset$ to the term would simply propagate $\emptyset$ to the first placeholder and then propagate $\emptyset \cup \{x\}$ to the second one, since Lambda, which is a binder, has type $\forall S.\forall x.\text{Term } (S \cup \{x\}) \to \text{Term } S$. Finally, True is the inferred type for I, whose expected type is $x \in \emptyset \cup \{x\}$. The two types are convertible and the input is now accepted without any guessing.

Improvement of the coercion mechanism. Coercions are triggered when unification fails. They are explicit cast functions, declared by the user, used to fix the type of sub-terms. Simplifying the unification problem allows to retrieve more coercions. For instance, consider a list $[1; 2; 3]$ of natural numbers used as a list of integer numbers and assume the existence of a coercion function k from natural to integers. In the mono-directional problem, the failing unification problem is (List $\mathbb{N}$) vs (List $\mathbb{Z}$). The coercion required is the one obtained lifting k over lists. The lifting has to be performed manually by the user or by the system. In the latter case, the system needs to recognize that lists are containers and has to have code to lift coercions over containers, like in [10]. In the bi-directional case, however, the expected type (List $\mathbb{Z}$) would propagate to assign to each list element the expected type $\mathbb{Z}$ and the coercion k would be applied to all integers in the list without need of additional machinery. The bi-directional algorithm presented in this paper does not allow to remove the need for the coercion over lists in all situations, but it is sufficient in many practical ones, like the one just considered.

Introduction of vectors of placeholders ("...") in the external syntax. A very common use of dependently typed functions consists in explicitly passing to them an argument which is not the first one and have the system infer the previous arguments using type dependencies. For instance, if Cons : $\forall A.A \to \text{List } A \to \text{List } A$ and l is a list of integers, the user can simply write (Cons ? 2 l) and have the system infer that ? must be instantiated with the type of 2, which is $\mathbb{N}$.

---

[4]User provided notational macros are used to extend the external syntax of an ITP and they are expanded before refinement, yielding a term in external syntax to be refined.

This scenario is so common that many ITPs allow to mark some function arguments as implicit arguments and let the user systematically avoid passing them. This requires additional machinery implemented in the ITP and it has the unfortunate drawback that sometimes the user needs to explicitly pass the implicit arguments anyway, in particular in case of partial function applications. This special situation requires further ad-hoc syntax to turn the implicit argument into an explicit one. For instance, if we declare the first argument of Cons implicit, then the user can simply write (Cons 2 l) for the term presented above, but has to write something like (@Cons $\mathbb{N}$), in Coq syntax, to pass the partial function application to some higher order function expecting an argument of type $\mathbb{N} \to$ List $\mathbb{N} \to$ List $\mathbb{N}$.

An alternative to implicit arguments is to let the user explicitly insert the correct number of placeholders "?" to be inferred by the system. Series of placeholders are neither aesthetic nor robust to changes in the type of the function.

A similar case occurs during the implementation of tactics. Given a lemma $L : H_1 \to \ldots \to H_n \to C$, to apply it the tactic opens n new proof obligations by refining the term $(L\ ?\ \ldots\ ?)$ where the number of inserted placeholders must be exactly n.

In this paper we propose a new construct to be added to the external syntax of ITPs: a vector of placeholders to be denoted by $\vec{?}$ and to be used in argument position only. In the actual external syntax of Matita we use the evocative symbol "$\ldots$" in place of $\vec{?}$. The semantics associated to $\vec{?}$ is lazy: an $\vec{?}$ will be expanded to the sequence of placeholders of minimal length that makes the application refineable, so that its inferred type matches its expected type. In a uni-directional setting no expected type is known in advance and the implementation of the lazy semantics would require computationally expensive non-local backtracking, which is not necessary in the bi-directional case.

Thanks to vectors of placeholders the analysis phase of many primitive tactics implementation that was aimed at producing terms with the correct number of placeholders can now be totally omitted. Moreover, according to our experience, vectors of placeholders enable to avoid the implementation of implicit arguments: it is sufficient for the user to insert manually or by means of a notation a $\vec{?}$ before the arguments explicitly passed, with the benefit that the $\vec{?}$ automatically adapts to the case of partial function application. For example, using the infix notation :: for (Cons $\vec{?}$), the user can both write 2 :: l, which is expanded to (Cons $\vec{?}$ 2 l) and refined to (Cons $\mathbb{N}$ 2 l), and pass :: to an higher order function expecting an argument of type $\mathbb{N} \to$ List $\mathbb{N} \to$ List $\mathbb{N}$. In the latter case, :: is expanded to (Cons $\vec{?}$) that is refined to (Cons $\mathbb{N}$) because of the expected type. If :: is passed instead to a function expecting an argument of type $\forall A.A \to$ List $A \to$ List A, then (Cons $\vec{?}$) will be expanded simply to Cons whose inferred type is already the expected one.

The rest of the paper explains the bi-directional refinement algorithm implemented in Matita [6]. The algorithm is presented in a declarative programming style by means of deduction rules. Many of the rules are syntax directed and thus mutually exclusive. The implementation given for Matita in the functional OCaml language takes advantage of the latter observation to speed up the algorithm. We will clarify in the text what rules are mutually exclusive and what rules are to be tried in sequence in case of failure.

The refinement algorithm is presented progressively and in a modular way. In Section 3 we introduce the mono-directional type inference algorithm for CIC implemented following the kernel type checker code (that coincides with type inference if the term is ground) of Matita. The presentation is already adapted to be extended in Section 4 to bi-directional refinement. In these two sections the external and internal syntaxes coincides. In Section 5 we augment the external syntax with placeholders and vectors of placeholders. Finally, in Section 6 we add support for coercions. In all sections we will prove the correctness of the refinement algorithms by showing that a term in external syntax accepted by the refiner is turned into a new term that is accepted by the kernel and that has the expected type. Moreover, a precise correspondence is established between the input and output term to grant that the refined term corresponds to the input one.

For the sake of the reader, Appendix 8 is taken from [4] with minor modifications and it shows the type checking algorithm implemented by the kernel. The syntax of the calculus and some preliminary notions are also introduced in Section 2 before starting the description of the refinement algorithm.

2. Preliminaries

2.1. **Syntax.** We begin introducing the syntax for CIC terms and objects in Table 1 and some naming conventions.

To denote constants we shall use $c, c_1, c_2 \ldots$; the special case of (co)recursively defined constants will be also denoted using $f, f_1, f_2 \ldots$; we reserve $x, y, x_1, x_2 \ldots$ for variables; $t, u, v, t', t'', t_1, t_2 \ldots$ for terms; $T, U, V, E, L, R, T', T'', T_1, T_2 \ldots$ for types and we use $s, s', s_1 \ldots$ for sorts.

We denote by Γ a context made of variables declarations $(x : T)$ or typed definitions $(x := t : T)$. We denote the capture avoiding substitution of a variable x for a term t by $[x/t]$. The notation $[x_1/t_1; \ldots; x_n/t_n]$ is for simultaneous parallel substitution.

To refer to (possibly empty) sequences of entities of the same nature, we use an arrow notation (e.g. $\overrightarrow{t}$). For the sake of conciseness, it is sometimes convenient to make the length of a sequence explicit, while still referring to it with a single name: we write $\overrightarrow{t_n}$ to mean that $\overrightarrow{t_n}$ is a sequence of exactly n elements and, in particular, that it is a shorthand for $t_1 \, t_2 \ldots t_n$; the index n must be a natural number (therefore the notation $\overrightarrow{t_{n+1}}$ refers to a non-empty sequence). The arrow notation is extended to telescopes as in $\overrightarrow{(x : t)}$ or $\overrightarrow{(x_n : t_n)}$ and used in binders, (co)recursive definitions and pattern matching branches.

As usual, $\Pi x : T_1.T_2$ is abbreviated to $T_1 \to T_2$ when x is not a free variable in T_2. Applications are n-ary, consisting of a term applied to a non-empty sequence of terms.

Inductive types I_l are annotated with the number l of arguments that are homogeneous in the types of all constructors. For example consider the inductive type of vectors Vect of arity $\Pi A : \mathbf{Type}.\mathbb{N} \to \mathbf{Type}$. It takes two arguments, a type and a natural number representing the length of the vector. In the types of the two constructors, Vnil : Vect A 0 and Vcons : $\Pi m,$ Vect $A \, m \to A \to$ Vect $A \, (m+1)$, every occurrence of Vect is applied to the same argument A, that is also implicitly abstracted in the types of the constructors. Thus Vect has one homogeneous argument, and will be represented by the object

$$\Pi A : \mathbf{Type}. \, \mathbf{inductive} \, \text{Vect} \, : \, \mathbb{N} \to \mathbf{Type} :=$$
$$\text{Vnil} : \text{Vect} \, A \, 0 \mid \text{Vcons} : \Pi m, A \to \text{Vect} \, A \, m \to \text{Vect} \, A \, (m+1)$$

$$
\begin{array}{lll}
t & ::= & x & \text{identifiers} \\
& | & c & \text{constants} \\
& | & I_l & \text{inductive types} \\
& | & k & \text{inductive constructors} \\
& | & \mathbf{Prop} \mid \mathbf{Type}_u & \text{sorts} \\
& | & t\ \overrightarrow{t_{n+1}} & \text{n-ary application} \\
& | & \lambda x : t.t & \lambda\text{-abstraction} \\
& | & \mathbf{let}\ (x : t) := t\ \mathbf{in}\ t & \text{local definitions} \\
& | & \Pi x : t.t & \text{dependent product} \\
& | & \mathbf{match}\ t\ \mathbf{in}\ I_l\ \mathbf{return}\ t\ [k_1\ \overrightarrow{(x : t)} \Rightarrow t \mid \ldots \mid k_n\ \overrightarrow{(x : t)} \Rightarrow t] & \text{case analysis} \\
& | & ?_j[t\ ;\ \ldots\ ;\ t] & \text{metavariable occurrence}
\end{array}
$$

$$
\begin{array}{lll}
o & ::= & \mathbf{let\ rec}\ f_1\ \overrightarrow{(x : t)} : t := t\ \mathbf{and}\ \ldots\ \mathbf{and}\ f_n\ \overrightarrow{(x : t)} : t := t & \text{recursive definitions} \\
& | & \mathbf{let\ corec}\ f_1\ \overrightarrow{(x : t)} : t := t\ \mathbf{and}\ \ldots\ \mathbf{and}\ f_n\ \overrightarrow{(x : t)} : t := t & \text{co-recursive definitions} \\
& | & \mathbf{definition}\ c : t := t & \text{definitions} \\
& | & \mathbf{axiom}\ c : t & \text{axioms} \\
& | & \Pi \overrightarrow{x_l : t_l}.\ \mathbf{inductive}\quad I_l^1\ :\ A := k_{1,1} : t \mid \ldots \mid k_{1,m_1} : t & \text{inductives} \\
& & \qquad\qquad \mathbf{with}\ \ldots \\
& & \qquad\qquad \mathbf{with}\ I_l^n\ :\ A := k_{n,1} : t \mid \ldots \mid k_{n,m_n} : t
\end{array}
$$

Table 1: CIC terms and objects syntax

and referred to with Vect_1. This is relevant for the pattern matching construction, since the homogeneous arguments are not bound in the patterns because they are inferred from the type of the matched term. For example, to pattern match over a vector v of type (Vect $\mathbb{N}$ 3) the user writes

$\quad$ **match** v **in** Vect_1 **return** T [Vnil $\Rightarrow t_1$ | Vcons $(m : \mathbb{N})\ (x : \mathbb{N})\ (v' : \text{Vect}\ \mathbb{N}\ m) \Rightarrow t_2$]

The inductive type I_l in the pattern matching constructor is (almost) redundant, since distinct inductive types have distinct constructors; it is given for the sake of readability and to distinguish the inductive types with no constructors. In a concrete implementation it also allows to totally drop the names of the constructors by fixing an order over them: the i-th pattern will be performed on the i-th constructor of the I_l inductive type.

Since inductive types may have non homogeneous arguments, not every branch is required to have exactly the same type. The term introduced with the **return** keyword is a function that computes the type expected by a particular branch and also the type of the entire pattern matching. Variables $\overrightarrow{(x : t)}$ are abstracted in the right hand side terms of $\Rightarrow$.

The definitions of constants c (including (co)recursive constants f), inductive types I_l and constructors k are collected in the syntactic category of CIC objects o.

Metavariable occurrences, represented with $?_j[t_1\ ;\ \ldots\ ;\ t_n]$, are missing typed terms equipped with an explicit local substitution. The index j enables metavariables to occur non-linearly in the term. To give an intuition of the role played by the local substitution, the reader can think of $?_j[t_1\ ;\ \ldots\ ;\ t_n]$ as a call to the, still unknown, function $?_j$ with actual arguments $t_1 \ldots t_n$. The terms $t_1 \ldots t_n$ will be substituted for the formal arguments of the $?_j$ function inside its body only when it will be known.

We omit to write the local substitution when it is the identity substitution that sends all variables in the current context with themselves. Thus $?_j$ will be a shorthand for $?_j[x_1 ; \ldots ; x_n]$ when $x_1, \ldots, x_n$ are the variables bound in the right order in the context of the metavariable occurrence.

The CIC calculus extended with metavariables has been studied in [21] and the flavor of metavariables implemented in Matita is described in [26].

2.2. **Typing rules.** The kernel of Matita is able to handle the whole syntax presented in the previous section, metavariables included. While we report in the Appendix 8 the full set of typing rules implemented by the kernel, here we summarise only the ones that will be reused by the refinement algorithm. We will give a less formal but more intuitive presentation of these rules, defining them with a more concise syntax. Moreover, we will put our definition in linear order, while most of them are actually mutually recursive.

Definition 2.1 (Proof problem (Σ)). *A proof problem Σ is a finite list of typing declarations of the form $\Gamma_{?_j} \vdash ?_j : T_{?_j}$.*

A proof problem, as well as a CIC term, can refer to constants, that usually live in an environment that decorates every typing rule (as in the Appendix 8). In the following presentation we consider a global well formed environment Env, basically a collections of CIC objects defining all constants and inductive types and associating them to their respective types. No refinement rule will modify this environment that plays no role in this presentation. In fact it is the task of the kernel to enable well typed definitions, inductive types and (co-)recursive functions to enter the environment.

We thus omit the environment Env from the input of every judgment. We will fetch from it the type T of a constant, inductive type or constructor r writing $(r : T) \in$ Env.

We regard CIC as a Pure Type System [7], and we denote by PTS the set of axioms. We denote by $s \in$ PTS any sort of the PTS, with $(s_1 : s_2) \in$ PTS the fact that s_2 types s_1, and with $(s_1, s_2, s_3) \in$ PTS the fact that a product over s_1 to s_2 has sort s_3. CIC is a full but not functional PTS: all products are well formed but in $(s_1, s_2, s_3) \in$ PTS it may be $s_2 \neq s_3$. This is because the calculus is parameterized over a predicative hierarchy $\mathbf{Type}_u$ for u in a given set of universe indexes. In a predicative setting, given $s_1 = \mathbf{Type}_{u_1}$ and $s_2 = \mathbf{Type}_{u_2}$, s_3 is defined as $\mathbf{Type}_{\max\{u_1, u_2\}}$ according to some bounded partial order on the universe indexes. The details for the actual PTS used in Matita are given in [4]. We will often write simply $\mathbf{Type}$ when we are not interested in the universe index (e.g. in examples). We also write $\mathbf{Type}_\top$ for the biggest sort in the hierarchy, if any, or a variable universe to be later fixed to be big enough to satisfy all the required constraints.

We also write $(s_1, s_2) \in$ elim(PTS) to check if an element of an inductive type of sort s_1 can be eliminated to inhabit a type whose sort is s_2. This is relevant for CIC since the sort of propositions, $\mathbf{Prop}$, is non informative and cannot be eliminated to inhabit a data type of sort $\mathbf{Type}_u$ for any u (but for few exceptions described in [4] Section 6).

Proof problems do not only declare missing proofs (i.e. not all $T_{?_j}$ have sort $\mathbf{Prop}$) but also missing terms and, of particular interest for this paper, missing types.

Definition 2.2 (Metavariable substitution environment (Φ)). *A metavariable substitution environment Φ (called simply substitution when not ambiguous) is a list of judgments of the form*

$$\Gamma_{?_j} \vdash ?_j := t_{?_j} : T_{?_j}$$

stating that the term $t_{?_j}$ of type $T_{?_j}$ in $\Gamma_{?_j}$ has been assigned to $?_j$.

We now anticipate the typing judgment of the kernel. A formal definition of well formedness for Σ and Φ will follow.

Definition 2.3 (Typing judgment). *Given a term t, a proof problem Σ and a substitution Φ, all assumed to be well formed, we write*

$$\Sigma,\ \Phi,\ \Gamma \vdash t : T$$

to state that t is well typed of type T.

When $\Sigma,\ \Phi,\ \Gamma \vdash t : T$ the type T is well typed and its type is either a metavariable or a sort $s \in \mathsf{PTS}$.

The typing judgment implemented in our kernel is an extension of the regular typing judgment for CIC [35, 23, 13]. It is described in [4] and reported in the Appendix 8. Here we recall the main differences:

- Substitution of a regular variable x for a term t is extended with the following rule for metavariables:

$$?_j[t_1\ ;\ \ldots\ ;\ t_n][x/t] = ?_j[t_1[x/t]\ ;\ \ldots\ ;\ t_n[x/t]]$$

- The conversion relation (denoted by $\downarrow$) is enlarged allowing reduction to be performed inside explicit substitution for metavariables:

$$\frac{\Gamma \vdash t_i \downarrow t'_i \qquad i \in \{1 \ldots n\}}{\Gamma \vdash ?_j[t_1\ ;\ \ldots\ ;\ t_n] \downarrow ?_j[t'_1\ ;\ \ldots\ ;\ t'_n]}$$

- The following typing rules for metavariables are added:

$$\frac{\begin{array}{c} y_1 : T_1\ ;\ \ldots\ ;\ y_n : T_n \vdash ?_j : T_{?_j} \in \Sigma \\ \Gamma \vdash t_i : T_i[y_1/t_1\ ;\ \ldots\ ;\ y_{i-1}/t_{i-1}] \quad i \in \{1 \ldots n\} \end{array}}{\Gamma \vdash \mathcal{WF}(?_j[t_1\ ;\ \ldots\ ;\ t_n])}$$

$$\frac{(y_1 : T_1\ ;\ \ldots\ ;\ y_n : T_n \vdash ?_j : T_{?_j}) \in \Sigma \quad \Gamma \vdash \mathcal{WF}(?_j[t_1\ ;\ \ldots\ ;\ t_n])}{\Gamma \vdash ?_j[t_1\ ;\ \ldots\ ;\ t_n] : T_{?_j}[y_1/t_1\ ;\ \ldots\ ;\ y_n/t_n]}$$

Moreover, in many situations a metavariable occurrence is also accepted as a valid sort, marking it so that it cannot be instantiated with anything different from a sort. This additional labelling will be omitted, being marginal for the refinement algorithm.

The technical judgment $\Gamma \vdash \mathcal{WF}(?_j[t_1\ ;\ \ldots\ ;\ t_n])$ states that a metavariable occurrence $?_j[t_1\ ;\ \ldots\ ;\ t_n]$ is well formed in Γ.

In all the previous rules we assumed access to a global well formed proof problem Σ and substitution Φ. Both Σ and Φ are never modified by the judgments implemented in the kernel.

We now present the well formedness conditions, corresponding to the judgments $\vdash$ WF presented in the Appendix 8.

Definition 2.4 (Metavariables of term/context ($\mathcal{M}$))**.** *Given a term t, $\mathcal{M}(t)$ is the set of metavariables occurring in t. Given a context Γ, $\mathcal{M}(\Gamma)$ is the set of metavariables occurring in Γ.*

The function $\mathcal{M}$ is at the base of the order relation defined between metavariables.

Definition 2.5 (Metavariables order relation ($\ll_\Sigma$))**.** *Let Σ be a proof problem. Let $<_\Sigma$ be the relation defined as: $?n_1 <_\Sigma ?n_2$ iff $?n_1 \in \mathcal{M}(\Gamma_{?n_2}) \cup \mathcal{M}(T_{?n_2})$. Let $\ll_\Sigma$ be the transitive closure of $<_\Sigma$.*

Definition 2.6 (Valid proof problem)**.** *A proof problem Σ is a* valid proof problem *if and only if $\ll_\Sigma$ is a strict partial order (or, equivalently, if and only if $\ll_\Sigma$ is an irreflexive relation).*

The intuition behind $\ll_\Sigma$ is that the smallest $?_j$ (or one of them since there may be more than one) does not depend on any other metavariable (e.g. $\mathcal{M}(\Gamma_{?_j}) = \emptyset$ and $\mathcal{M}(T_{?_j}) = \emptyset$ where $\Gamma_{?_j} \vdash ?_j : T_{?_j} \in \Sigma$). Thus instantiating every minimal $?_j$ with a metavariable free term will give a new Σ in which there is at least one $?_j$ not depending on any other metavariable (or Σ is empty). This definition is the key to avoid circularity in the following definitions.

In the rules given in Appendix 8 the partial order is left implicit by presenting Σ as an ordered list. However, as proved by Strecker in his Ph.D. thesis [32], the order is not preserved by unification and thus in any realistic implementation Σ is to be implemented as a set and the fact that $\ll_\Sigma$ remains a partial order must be preserved as an invariant.

Definition 2.7 (Well formed context ($\mathcal{WF}(\Gamma)$))**.** *Given a well formed proof problem Σ, a context $\Gamma = y_1 : T_1, \ldots, y_n : T_n$ is well formed (denoted by $\mathcal{WF}(\Gamma)$) if $\mathcal{M}(\Gamma) \subseteq \Sigma$ and for every i*

$$y_1 : T_1, \ldots y_{i-1} : T_{i-1} \vdash y_i : T_i$$

Definition 2.8 (Well formed proof problem ($\mathcal{WF}(\Sigma)$))**.** *A valid proof problem Σ is a well-formed proof problem (denoted by $\mathcal{WF}(\Sigma)$) if an only if for all $(\Gamma_{?_j} \vdash ?_j : T_{?_j}) \in \Sigma$ we have $\Sigma, \Gamma_{?_j} \vdash T_{?_j} : s$ and $s \in \mathsf{PTS}$.*

Definition 2.9 (Well formed substitution ($\mathcal{WF}(\Phi)$))**.** *Given a well formed proof problem Σ, a substitution Φ is well formed (denoted by $\mathcal{WF}(\Phi)$) if for every $(\Gamma_{?_j} \vdash ?_j := t_{?_j} : T_{?_j}) \in \Phi$ we have $\Sigma, \emptyset, \Gamma_{?_j} \vdash t_{?_j} : T_{?_j}$.*

The well formedness definitions given so far are actually implemented by the kernel in a more precise but less intuitive way. We thus refer to the kernel judgments in the following definition, that will be used in the specification of all refinement rules.

Definition 2.10 (Well formed status ($\mathcal{WF}(\Sigma, \Phi, \Gamma)$))**.** *Given a proof problem Σ, a substitution Φ and a context Γ, the triple Σ, Φ, Γ is well formed (denoted by $\mathcal{WF}(\Sigma, \Phi, \Gamma)$) when $\mathcal{WF}(\Sigma)$ and $\mathcal{WF}(\Phi)$ and $\mathcal{WF}(\Gamma)$.*

We shall sometimes omit Γ, considering it equal to a default, well formed context, like the empty one. The recursive operation of applying a substitution Φ to a term t is denoted by $\Phi(t)$ and acts as the identity for any term but metavariables contained in Φ, on which it behaves as follows:

$$\Phi(?_j[t_1 ; \ldots ; t_n]) = t_{?_j}[y_1/t_1 ; \ldots ; y_n/t_n] \quad \text{when} \quad (y_1 : T_1; \ldots; y_n : T_n \vdash ?_j := t_{?_j} : T_{?_j}) \in \Phi$$

Note that, thanks to the extensions to the type checking rules made in Definition 2.3, substitution application is type preserving. Substitutions do apply also to well formed proof problems in the following way:

$$\Phi(\Gamma_{?_j} \vdash ?_j : T_{?_j}) = \Phi(\Gamma_{?_j}) \vdash ?_j : \Phi(T_{?_j}) \qquad \text{(for each } ?_j \in \Sigma)$$

The substitution application operation is seldom used explicitly, since all judgments take as input and give back a substitution. Nevertheless it will be used in the examples.

Definition 2.11 (Weak-head normalization ($\triangleright_{\text{whd}}$)). *Given a context Γ, substitution Φ and proof problem Σ, all assumed to be well formed, it computes the weak head normal form of a well typed term t according to the reduction rules of CIC. It is denoted by:*

$$\Sigma, \ \Phi, \ \Gamma \vdash t \triangleright_{\text{whd}} t'$$

Note that $?_j$ is in weak head normal form iff $?_j \notin \Phi$.

By abuse of notation we will write $\Sigma, \ \Phi, \ \Gamma \vdash t_1 \triangleright_{\text{whd}} \Pi x_1 : T_1 \ldots \Pi x_n : T_n.t_{n+1}$ to mean that for all $i \in \{1 \ldots n\}$ $\Sigma, \ \Phi, \ \Gamma; x_1 : T_1; \ldots; x_{i-1} : T_{i-1} \vdash t_i \triangleright_{\text{whd}} \Pi x_i : T_i.t_{i+1}$ and $\Sigma, \ \Phi, \ \Gamma \vdash t_{n+1} \triangleright_{\text{whd}} t_{n+1}$. Such repeated use of weak head computation to produce spines of dependent products occur frequently in the kernel and in the refinement rules, especially when dealing with inductive types.

Definition 2.12 (Conversion ($\downarrow$)). *Given a proof problem Σ, substitution Φ and context Γ, all assumed to be well formed, and two terms t_1 and t_2, it verifies if t_1 and t_2 have a common normal form according to the rules of CIC given in Appendix 8. It is denoted by:*

$$\Sigma, \ \Phi, \ \Gamma \vdash t_1 \downarrow t_2$$

3. Mono-directional refinement

We now present the mono-directional refinement algorithm for CIC implemented in the old versions of Matita (0.5.x) and directly inspired by the rules for type checking implemented in the kernel. In this section we assume the external syntax to coincide with the syntax of terms. Hence the algorithm actually performs just type inference. Nevertheless, we already organize the judgments in such a way that the latter extension to bi-directionality will be achieved just by adding new typing rules.

3.1. **Specification.** To specify what is a refinement algorithm we must first introduce the notion of proof problem refinement. Intuitively, a pair (proof problem, substitution) is refined by another pair when the second is obtained by reducing some proof obligations to new ones. It thus represents an advancement in the proof discovery process.

Definition 3.1 (Proof problem refinement ($\leq$)). *We say that Σ', Φ' refines Σ, Φ (denoted by $\Sigma', \Phi' \leq \Sigma, \Phi$) when $\Phi \subset \Phi'$ and for every $(\Gamma_{?_j} \vdash ?_j : T_{?_j}) \in \Sigma$ either $(\Gamma'_{?_j} \vdash ?_j : T'_{?_j}) \in \Sigma'$ or $(\Gamma'_{?_j} \vdash ?_j := t_{?_j} : T'_{?_j}) \in \Phi'$ where $\Gamma'_{?_j} = \Phi'(\Gamma_{?_j})$ and $T'_{?_j} = \Phi'(T_{?_j})$.*

Specification 3.2 (Refiner in type inference mode ($\mathcal{R}^{\Uparrow}$)). *A refiner algorithm $\mathcal{R}$ in type inference mode $\Uparrow$ takes as input a proof problem, substitution and context, all assumed to be well formed, and a term t. It fails or gives in output a new proof problem, a new substitution, a term t' and a type T'. It is denoted by:*

$$(\Sigma, \ \Phi) \ \Gamma \vdash t \overset{\mathcal{R}^{\Uparrow}}{\leadsto} t' : T' \ (\Sigma', \ \Phi')$$

Precondition:

$$\mathcal{WF}(\Sigma, \ \Phi, \ \Gamma)$$

Postcondition (parametric in $\preccurlyeq$):

$$\mathcal{WF}(\Sigma', \ \Phi') \quad \wedge \quad \Sigma', \Phi' \leq \Sigma, \Phi \quad \wedge \quad \Sigma', \ \Phi', \ \Gamma \vdash t' : T' \quad \wedge \quad t' \preccurlyeq t$$

The specification is parametric in the $\preccurlyeq$ relation that establishes a correspondence between the term t to be refined and the refiner output t'. In order to prove correctness, we are only interested in admissible $\preccurlyeq$ relations defined as follows.

Definition 3.3 (Admissible relations ($\preccurlyeq$)). *A partial order relation $\preccurlyeq$ is* admissible *when for every term t_1 in external syntax and t_2 and T in internal syntax and for every variable x occurring free only linearly in T we have that $t_1 \preccurlyeq t_2$ implies $T[x/t_1] \preccurlyeq T[x/t_2]$.*

Admissibility for equivalence relations correspond to asking the equivalence relation to be a congruence.

When the external syntax corresponds to the term syntax and coercions are not considered, we can provide an implementation that satisfies the specification by picking the identity for the $\preccurlyeq$ relation. Combined with $\Sigma', \Phi' \leq \Sigma, \Phi$, the two postconditions imply that $\Phi(t')$ must be obtained from t simply by instantiating some metavariables. In Sections 5 and 6, we shall use weaker definitions of $\preccurlyeq$ than the identity, allowing replacement of (vectors of) placeholders with (vectors of) terms and the insertion of coercions as results of the refinement process. All the $\preccurlyeq$ relations considered in the paper will be large partial orders over terms of the external syntax (that always include the internal syntax).

We will now proceed in presenting an implementation of a refinement algorithm in type inference mode $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$. The implementation is directly inspired by the type checking rules used in the kernel. However, since refinement deals with terms containing flexible parts, conversion tests need to be replaced with unification tests. In a higher order and dependently typed calculus like CIC, unification is in the general case undecidable. What is usually implemented in interactive theorem provers is an essentially fist order unification algorithm, handling only some simple higher order cases. The unification algorithm implemented in Matita goes beyond the scope of this paper, the interested reader can find more details in [26, 5]. Here we just specify the expected behavior of the unification algorithm.

Specification 3.4 (Unification ($\overset{?}{\equiv} - \mathcal{U}$)). *An unification algorithm takes as input a proof problem, a substitution and a context, all assumed to be well formed, and two well typed terms t_1 and t_2. It fails or gives in output a new proof problem and substitution. It is denoted using the following notation where $\bullet$ can either be $=$ or be omitted. In the former case universe cumulativity (a form of sub-typing) is not taken in account by unification.*

$$(\Sigma, \ \Phi) \ \Gamma \vdash t_1 \overset{?}{\equiv} t_2 \ \overset{\mathcal{U}_{\bullet}}{\rightsquigarrow} \ (\Sigma', \ \Phi')$$

Precondition:

$$\mathcal{WF}(\Sigma, \ \Phi, \ \Gamma) \quad \wedge \quad \Sigma, \ \Phi, \ \Gamma \vdash t_1 : T_1 \quad \wedge \quad \Sigma, \ \Phi, \ \Gamma \vdash t_2 : T_2$$

Postcondition:

$$\mathcal{WF}(\Sigma', \ \Phi') \quad \wedge \quad \Sigma', \Phi' \leq \Sigma, \Phi \quad \wedge \quad \Sigma', \ \Phi', \ \Gamma \vdash t_1' \downarrow_{\bullet} t_2'$$

3.2. **Implementation.**

3.2.1. *Additional judgments.* For the sake of clarity we prefer to keep the same structure for the mono and bi-directional refiners. We thus give the definition of some functions that are trivial in the mono-directional case, but will be replaced by more complex ones in the following sections.

Even if we presented the syntax of CIC using the same category terms, types and sorts, some primitive constructors (like the λ and Π abstractions) expect some arguments to be types or sorts, and not terms. A type level enforcing algorithm forces a term in external syntax to be refined to a valid type.

Specification 3.5 (Type level enforcing ($\mathcal{F}$)). *A type level enforcing algorithm takes as input a proof problem Σ, a substitution Φ and a context Γ, all assumed to be well formed, and a term T. It fails or it returns a new term T', a sort s, a new substitution Φ' and proof problem Σ'. It is denoted by:*

$$(\Sigma,\ \Phi)\ \Gamma \vdash T \overset{\mathcal{F}}{\rightsquigarrow} T' :\ s\ (\Sigma',\ \Phi')$$

Precondition:

$$\mathcal{WF}(\Sigma,\ \Phi,\ \Gamma)$$

Postcondition (parametric in $\preccurlyeq$):

$$\mathcal{WF}(\Sigma',\ \Phi')\quad \wedge\quad \Sigma',\Phi' \le \Sigma,\Phi \quad \wedge \quad \Sigma',\ \Phi',\ \Gamma \vdash T':s \quad \wedge \quad s \in \mathsf{PTS} \quad \wedge \quad T' \preccurlyeq T$$

Note that one may want to accept a metavariable as the sort s, eventually labelling it in such a way that the unification algorithm will refuse to instantiate it with a different term. The choice must be consistent with the one taken in the implementation of the kernel.

The task of checking if a term has the right type is called refinement in type forcing mode and it will be denoted by $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$. In the mono-directional case, $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ will be simply implemented calling the $\overset{\mathcal{C}}{\rightsquigarrow}$ algorithm that will handle coercions in Section 6 but which, at the moment, only verifies that no coercion is needed by calling the unification procedure.

Specification 3.6 (Explicit cast ($\mathcal{C}$)). *A cast algorithm takes as input a proof problem Σ, a substitution Φ and a context Γ, all assumed to be well formed, and a term t with its inferred type T and expected type T'. It fails or it returns a new term t' of type T', a new proof problem Σ' and substitution Φ'. It is denoted by:*

$$(\Sigma,\ \Phi)\ \Gamma \vdash t : T \overset{?}{\equiv} T' \overset{\mathcal{C}}{\rightsquigarrow} t'\ (\Sigma',\ \Phi')$$

Precondition:

$$\mathcal{WF}(\Sigma,\ \Phi,\ \Gamma) \quad \wedge \quad \Sigma,\ \Phi,\ \Gamma \vdash t : T \quad \wedge \quad \Sigma,\ \Phi,\ \Gamma \vdash T' : s$$

Postcondition (parametric in $\preccurlyeq$):

$$\mathcal{WF}(\Sigma',\ \Phi')\quad \wedge\quad \Sigma',\Phi' \le \Sigma,\Phi \quad \wedge \quad \Sigma',\ \Phi',\ \Gamma \vdash t':T' \quad \wedge \quad t' \preccurlyeq t$$

Specification 3.7 (Refiner in type forcing mode ($\mathcal{R}^{\Downarrow}$)). *A refiner algorithm $\mathcal{R}$ in type forcing mode $\Downarrow$ takes as input a proof problem Σ, a substitution Φ and a context Γ, all assumed to be well formed, and a term t together with its expected well formed type T. It fails or returns a term t' of type T, a new proof problem Σ' and substitution Φ'. It is denoted by:*

$$(\Sigma,\ \Phi)\ \Gamma \vdash t :\ T \overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} t'\ (\Sigma',\ \Phi')$$

Precondition:
$$\mathcal{WF}(\Sigma,\ \Phi,\ \Gamma) \quad \wedge \quad \Sigma,\ \Phi,\ \Gamma \vdash T : s$$

Postcondition (parametric in $\preccurlyeq$):
$$\mathcal{WF}(\Sigma',\ \Phi') \quad \wedge \quad \Sigma', \Phi' \leq \Sigma, \Phi \quad \wedge \quad \Sigma',\ \Phi',\ \Gamma \vdash t' : T \quad \wedge \quad t' \preccurlyeq t$$

3.2.2. *Notational conventions.* The arguments Σ and Φ will be taken as input and returned as output in all rules that define the refiner algorithm. To increase legibility we adopt the following notation, letting Σ and Φ be implicit. Each rule of the form

$$(rule) \quad \frac{\Gamma \vdash t \rightsquigarrow t' \quad \Gamma \vdash t' \rightsquigarrow t''}{\Gamma \vdash t \rightsquigarrow t''}$$

has to be interpreted as:

$$(rule) \quad \frac{(\Sigma,\ \Phi)\ \Gamma \vdash t \rightsquigarrow t'\ (\Sigma', \Phi') \quad (\Sigma',\ \Phi')\ \Gamma \vdash t' \rightsquigarrow t''\ (\Sigma'', \Phi'')}{(\Sigma,\ \Phi)\ \Gamma \vdash t \rightsquigarrow t''\ (\Sigma'', \Phi'')}$$

Moreover we will apply this convention also to rules not returning Σ or Φ as if they were returning the Σ or Φ taken as input.

Note that the Σ' and Φ' returned by all rules considered in this paper are well formed and are also a proof problem refinement of the Σ and Φ provided as input. Being a proof problem refinement is clearly a transitive relation. Thus we have for free that all the omitted pairs (proof problem, substitution) are refinements of the initial ones.

3.2.3. *Role of the relations and their interaction.* In this paragraph we shortly present the role played by the relations $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$, $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$, $\overset{\mathcal{C}}{\rightsquigarrow}$ and $\overset{\mathcal{F}}{\rightsquigarrow}$ introduced so far and the auxiliary ones $\overset{\mathcal{E}^T}{\rightsquigarrow}$ and $\overset{\mathcal{E}_t}{\rightsquigarrow}$ that will be specified when needed.

The relation $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ links a term with its inferred type, while $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ links a term with the type expected by its context. $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ will thus exploit the extra piece of information not only checking that the inferred type unifies with the expected one, but also propagating this information to its recursive calls on subterms (when possible). $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ and $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ will be defined in a mutually recursive way.

The relation $\overset{\mathcal{F}}{\rightsquigarrow}$ links a term with its refinement asserting that the refinement is a type. This is relevant when typing binders like $(\lambda x : t.t')$, where t is required to be a type. In its simplest formulation the relation is a simple assertion, linking a type with itself. In Section 6 the refinement relation $\preccurlyeq$ will admit to link a term t that is not a type with a function applied to t that turns its input into a type. For example t may be a record containing a type and $\overset{\mathcal{F}}{\rightsquigarrow}$ may link it with $(\pi_n\ t)$, where π_n is the projection extracting the type from the record. $\overset{\mathcal{F}}{\rightsquigarrow}$ is recursively defined in terms of $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$

The relation $\overset{\mathcal{C}}{\rightsquigarrow}$ links a term t, its inferred type T_1 and the type expected by its context T_2 with a refinement of the term t' asserting that the refined term has type T_2. In its simple formulation the relation is a simple assertion that T_1 and T_2 are the same and thus links t with itself. In Section 6 the refinement relation $\preccurlyeq$ will admit to explicitly cast t. For example a natural number n of type $\mathbb{N}$ may be casted into the rationals $\mathbb{Q}$ refining it to $(\lambda x : \mathbb{N}.x/1)\ n$. The $\overset{\mathcal{C}}{\rightsquigarrow}$ relation is non recursive.

The relations $\overset{\mathcal{E}^T}{\rightsquigarrow}$ and $\overset{\mathcal{E}_t}{\rightsquigarrow}$ are auxiliary relations only used to ease the presentation of the $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ and $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ relations in the case of applications. Both auxiliary relations are thus recursively defined with $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ and $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$.

3.2.4. *Rules for terms.* We now give an implementation for the refiner in both modes and for the auxiliary judgments. The implementation is parametric on the unification algorithm, that is not described in this paper.

$$
(\ \overset{\mathcal{C}}{\rightsquigarrow} -\mathrm{ok})\quad \frac{\Gamma \vdash T_1 \overset{?}{\equiv} T_2\ \overset{\mathcal{U}}{\rightsquigarrow}}{\Gamma \vdash t : T_1 \overset{?}{\equiv} T_2\ \overset{\mathcal{C}}{\rightsquigarrow}\ t}
\qquad
(\ \overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} -\mathrm{default})\quad \frac{\begin{array}{c}\Gamma \vdash t\ \overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}\ t' : T' \\ \Gamma \vdash t' : T' \overset{?}{\equiv} T\ \overset{\mathcal{C}}{\rightsquigarrow}\ t''\end{array}}{\Gamma \vdash t :\ T\ \overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}\ t''}
$$

$$
(\ \overset{\mathcal{F}}{\rightsquigarrow} -\mathrm{ok})\quad \frac{\begin{array}{c}\Gamma \vdash T\ \overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}\ T' :\ s \qquad s_1 \in \mathsf{PTS} \\ \Gamma \vdash T' : s \overset{?}{\equiv} s_1\ \overset{\mathcal{C}}{\rightsquigarrow}\ T''\end{array}}{\Gamma \vdash T\ \overset{\mathcal{F}}{\rightsquigarrow}\ T'' :\ s}
$$

Note that s_1 is arbitrary, and the actual code prefers the predicative sorts $\mathbf{Type}_u$ over **Prop**. This is the only rule defined in this section to be non syntax oriented: in case of an incorrect choice of s_1, backtracking is required. The actual algorithm implemented in Matita performs the choice of s_1 lazily to remain backtracking free[5].

$$
(\ \overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow} -\mathrm{variable})\quad \frac{(x : T) \in \Gamma \quad \mathrm{or}\quad (x := t : T) \in \Gamma}{\Gamma \vdash x\ \overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}\ x :\ T}
$$

$$
(\ \overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow} -\mathrm{constant})\quad \frac{(r : T) \in \mathsf{Env} \qquad r \in \{k, I, c\}}{\Gamma \vdash r\ \overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}\ r :\ T}
$$

$$
(\ \overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow} -\mathrm{sort})\quad \frac{(s_1 : s_2) \in \mathsf{PTS}}{\Gamma \vdash s_1\ \overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}\ s_1 :\ s_2}
$$

---

[5]Laziness will be no longer sufficient to avoid backtracking when we will add additional rules to handle coercions in Section 6.

$$(\Gamma_{?_j} \vdash ?_j : T_{?_j}) \in \Sigma \quad \text{or} \quad (\Gamma_{?_j} \vdash ?_j := t_{?_j} : T_{?_j}) \in \Phi$$
$$\Gamma_{?_j} = \overrightarrow{x_n : T_n}$$

$$(\overset{\mathcal{R}^{\Uparrow}}{\leadsto} -\text{meta}) \quad \frac{\Gamma \vdash t_i : \overrightarrow{T_i[x_{i-1}/t'_{i-1}]} \overset{\mathcal{R}^{\Downarrow}}{\leadsto} t'_i \quad i \in \{1 \dots n\}}{\Gamma \vdash ?_j[\overrightarrow{t_n}] \overset{\mathcal{R}^{\Uparrow}}{\leadsto} ?_j[\overrightarrow{t'_n}] : T_{?_j}[\overrightarrow{x_n/t'_n}]}$$

Note that the operation of firing a β-redex must commute with the operation of applying a substitution Φ. Consider for example the term $v = (\lambda x.?_j[x])\ u$ and the substitution $\Phi = \{x : T \vdash ?_j := t(x) : T(x)\}$. If one applies the substitution first, and then reduces the redex obtains $t(u)$, whose type is $T(u)$. If one fires the redex fist, the fact that x is substituted by u in $?_j$ is recorded in the local substitution attached to the metavariable instance. Indeed $\emptyset, \emptyset, \emptyset \vdash v \rhd_{\text{whd}} ?_j[u]$ and $\Phi(?_j[u]) = t(u) : T(u)$. Therefore $?_j[u]$ is given the type $T(u)$ by the rule $(\overset{\mathcal{R}^{\Uparrow}}{\leadsto} -\text{meta})$.

$$\Gamma \vdash T \overset{\mathcal{F}}{\leadsto} T' : s$$
$$\Gamma \vdash t : T' \overset{\mathcal{R}^{\Downarrow}}{\leadsto} t'$$

$$(\overset{\mathcal{R}^{\Uparrow}}{\leadsto} -\text{letin}) \quad \frac{\Gamma ; x := t' : T' \vdash u \overset{\mathcal{R}^{\Uparrow}}{\leadsto} u' : T_2}{\Gamma \vdash \mathbf{let}\ (x : T) := t\ \mathbf{in}\ u \overset{\mathcal{R}^{\Uparrow}}{\leadsto} \mathbf{let}\ (x : T') := t'\ \mathbf{in}\ u' : T_2[x/t'_1]}$$

$$\Gamma \vdash T_1 \overset{\mathcal{F}}{\leadsto} T'_1 : s_1$$

$$(\overset{\mathcal{R}^{\Uparrow}}{\leadsto} -\text{lambda}) \quad \frac{\Gamma ; x : T'_1 \vdash t \overset{\mathcal{R}^{\Uparrow}}{\leadsto} t' : T}{\Gamma \vdash \lambda x : T_1.t \overset{\mathcal{R}^{\Uparrow}}{\leadsto} \lambda x : T'_1.t' : \Pi x : T'_1.T}$$

$$\Gamma \vdash T_1 \overset{\mathcal{F}}{\leadsto} T'_1 : s_1$$
$$\Gamma ; x : T'_1 \vdash T_2 \overset{\mathcal{F}}{\leadsto} T'_2 : s_2$$

$$(\overset{\mathcal{R}^{\Uparrow}}{\leadsto} -\text{product}) \quad \frac{(s_1, s_2, s_3) \in \mathsf{PTS}}{\Gamma \vdash \Pi x : T_1.T_2 \overset{\mathcal{R}^{\Uparrow}}{\leadsto} \Pi x : T'_1.T'_2 : s_3}$$

We now state the correctness theorem holding for all the rules presented so far and for the few ones that will follow. The proof is partitioned in the following way: here we state the theorem, introduce the proof method we adopted and prove the theorem for the simple rules presented so far. Then we will introduce more complex rules, like the rule for application, and we will prove for each of them the correctness theorem.

Theorem 3.8 (Correctness). *The $\overset{\mathcal{C}}{\leadsto}$, $\overset{\mathcal{F}}{\leadsto}$, $\overset{\mathcal{R}^{\Uparrow}}{\leadsto}$, $\overset{\mathcal{R}^{\Downarrow}}{\leadsto}$, $\overset{\mathcal{E}^T}{\leadsto}$, and $\overset{\mathcal{E}_t}{\leadsto}$ algorithms defined by the set of rules presented in this section obey their specification for all admissible $\preccurlyeq$ relations*

that include the identity for terms in the internal syntax. In particular, the algorithms are correct when the identity relation is picked for $\preceq$.

Proof. We assume the unification algorithm to be correct w.r.t. its own specification. For every judgment, the proof is by induction on the proof tree. For each rule, we assume that the precondition of the judgment holds for the rule conclusion and that the appropriate postcondition holds by induction hypothesis for every hypothesis. We need to prove that the precondition of every hypothesis holds and that the postcondition of the conclusion holds too. The proofs are mostly trivial for the rules presented so far. In particular, the proof for each rule $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow} -name$ or $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} -name$ follows from the corresponding rule $\mathcal{K}-name$ reported in the Appendix 8.
We will shortly introduce the rules dealing with applicatios together with their correctness proofs since applications are handled slightly differently from the way they are processed by the kernel. $\square$

The next rule deals with applications which are n-ary in our implementation of CIC. In a calculus without dependent types, n-ary applications could be handled simply by putting the head function type in the form of a spine of n products and then by verifying that the type of each argument matches the corresponding expected type. In the presence of dependent types, however, it is possible to write functions whose arity depends on the arguments passed to the function. For instance, a function f could be given type $\forall n : \mathbb{N}.(\text{repeat } \mathbb{N} \ n)$ where $(\text{repeat } \mathbb{N} \ n)$ reduces to $\mathbb{N} \to \ldots \to \mathbb{N}$ where the number of products is exactly n. For this reason, the only possibility is to process applications one argument at a time, checking at every step if the function still accepts more arguments. We implement this with an additional judgment

$$\Gamma \vdash t \ \overrightarrow{(x_i := v_i : T_i)} : \ T \ \lrcorner \ \overrightarrow{u_n} \ \overset{\mathcal{E}^T}{\rightsquigarrow} \ v : \ V$$

called "eat products" to be specified and implemented immediately after the $(\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow} -\text{appl})$ rule.

$$(\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow} -\text{appl}) \quad \frac{\begin{array}{c} \Gamma \vdash t \ \overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow} \ t' : \ T \\ \Gamma \vdash t' : \ T \ \lrcorner \ \overrightarrow{u_{n+1}} \ \overset{\mathcal{E}^T}{\rightsquigarrow} \ v : \ V \end{array}}{\Gamma \vdash t \ \overrightarrow{u_{n+1}} \ \overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow} \ v : \ V}$$

Specification 3.9 (Eat products ($\mathcal{E}^T$))**.** *The $\overset{\mathcal{E}^T}{\rightsquigarrow}$ algorithm refines an n-ary application by consuming an argument at a time. It takes as input a proof problem Σ, a substitution Φ and a context Γ, all assumed to be well formed, the part of the already processed application $t \ (x_1 := v_1 : T_1) \ \ldots \ (x_r := v_r : T_r)$ together with its type T, and the list of arguments yet to be checked. The notation $(x_i := v_i : T_i)$ means that the i-th already processed argument has type T_i and is consumed by a product that binds the variable x_i. The algorithm fails or returns the refined application v together with its type V, a new substitution Φ' and proof problem Σ'. It is denoted by:*

$$(\Sigma, \ \Phi) \ \Gamma \vdash t \ \overrightarrow{(x_r := v_r : T_r)} : \ T \ \lrcorner \ \overrightarrow{u_k} \ \overset{\mathcal{E}^T}{\rightsquigarrow} \ v : \ V \ (\Sigma', \ \Phi')$$

Precondition:

$$\mathcal{WF}(\Sigma, \Phi, \Gamma) \quad \wedge \quad \Sigma, \Phi, \Gamma \vdash v_i : T_i \quad i \in \{1 \ldots r\} \quad \wedge \quad \Sigma, \Phi, \Gamma \vdash t \; v_1 \; \ldots \; v_r : T$$

Postcondition (parametric in $\preccurlyeq$):

$$\mathcal{WF}(\Sigma', \Phi') \quad \wedge \quad \Sigma', \Phi' \leq \Sigma, \Phi \quad \wedge \quad \Sigma', \Phi', \Gamma \vdash v : V \quad \wedge \quad v \preccurlyeq t \; v_1 \; \ldots \; v_r \; u_1 \; \ldots \; u_k$$

The applicative case is one of the two most complicated rules. Moreover, the refinement algorithm for the application does not mimic the one used in the kernel. Therefore we show the correctness of the ($\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ $-$appl) rule and of the implementation of the $\overset{\mathcal{E}^T}{\rightsquigarrow}$ algorithm.

Correctness of ($\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ $-$appl). The only rule precondition is $\mathcal{WF}(\Sigma, \Phi, \Gamma)$ that is also the precondition for the first premise. By induction hypothesis on the first premise we know $\Sigma', \Phi', \Gamma \vdash t' : T$ where Σ', Φ' are implicitly returned by the first call and passed to the second one. Moreover $\Sigma', \Phi' \leq \Sigma, \Phi$ and $t' \preccurlyeq t$. Therefore the preconditions for the second premise are satisfied. By induction hypothesis on the second premise we know $\Sigma'', \Phi'', \Gamma \vdash v : V$ where Σ'', Φ'' are implicitly returned by the second call and by the rule as a whole. Moreover $\Sigma'', \Phi'' \leq \Sigma', \Phi'$ and $v \preccurlyeq t' \; \overrightarrow{u_{n+1}}$. By transitivity of proof problem refinement, we also have $\Sigma'', \Phi'' \leq \Sigma, \Phi$. Moreover, since $\preccurlyeq$ is admissible, we also have $v \preccurlyeq t' \; \overrightarrow{u_{n+1}} \preccurlyeq t \; \overrightarrow{u_{n+1}}$. All post-conditions have been proved and therefore the rule is correct. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

The $\overset{\mathcal{E}^T}{\rightsquigarrow}$ algorithm is implemented as follows.

$$(\mathcal{E}^T-\text{empty}) \quad \frac{}{\Gamma \vdash t \; \overrightarrow{(x_r := v_r : T_r)} : T \; \lrcorner \; \overset{\mathcal{E}^T}{\rightsquigarrow} \; t \; \overrightarrow{v_r} : T}$$

Correctness of the ($\mathcal{E}^T-$empty) rule is trivial.

$$(\mathcal{E}^T-\text{prod}) \quad \frac{\begin{array}{c} \Gamma \vdash T \rhd_{\text{whd}} \Pi x : U_1.T_1 \\ \Gamma \vdash u_1 : U_1 \overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} u_1' \\ \Gamma \vdash t \; \overrightarrow{(x_r := v_r : T_r)} \; (x := u_1' : U_1) : T_1[x/u_1'] \; \lrcorner \; \overrightarrow{u_n} \overset{\mathcal{E}^T}{\rightsquigarrow} v : V \end{array}}{\Gamma \vdash t \; \overrightarrow{(x_r := v_r : T_r)} : T \; \lrcorner \; u_1 \; \overrightarrow{u_n} \overset{\mathcal{E}^T}{\rightsquigarrow} v : V}$$

Correctness of ($\mathcal{E}^T-$prod). Let Σ, Φ be the well formed pair taken as input by the rule and passed to the second premise, that returns the well formed pair Σ', Φ'. Similarly, let Σ'', Φ'' be the well formed pair given in output by the second premise and by the whole rule. By induction hypotheses $\Sigma', \Phi' \leq \Sigma, \Phi$ and $\Sigma'', \Phi'' \leq \Sigma', \Phi'$ and thus $\Sigma'', \Phi'' \leq \Sigma, \Phi$ as required. By the rule pre-condition, T is well typed in Σ, Φ, Γ and so are U_1 and T_1 obtained by reduction. Thus the premises of the second rule are all satisfied and, by induction hypothesis, $\Sigma', \Phi', \Gamma \vdash u_1' : U_1 \quad \wedge \quad u_1' \preccurlyeq u_1$. By rules $\mathcal{K}-\text{appl}-\text{rec}$ and $\mathcal{K}-\text{appl}-\text{base}$ applied to the rule pre-condition $t \; v_1 \; \ldots \; v_r : T$ we get $t \; v_1 \; \ldots \; v_r \; u_1' : T_1[x/u_1']$. Since all preconditions for the third premise are satisfied, by

induction hypothesis we know Σ'', Φ'', $\Gamma \vdash v : V$ and $v \preccurlyeq t\ v_1\ \ldots\ v_r\ u_1'\ u_2\ \ldots\ u_n$. By admissibility of $\preccurlyeq$ we conclude also $v \preccurlyeq t\ v_1\ \ldots\ v_r\ u_1\ \ldots\ u_n$. Since all post-conditions have been proved, the rule is correct. $\qquad\square$

$$(\mathcal{E}^T\text{-flexible}) \quad \dfrac{\begin{array}{c} \Gamma \vdash T \rhd_{\text{whd}} ?_j \quad \text{or} \quad \Gamma \vdash T \rhd_{\text{whd}} ?_j\ w_1\ \ldots w_l \\[2pt] \Gamma \vdash u_1 \overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow} u_1' : U_1 \\[2pt] \Sigma \rightsquigarrow \Sigma \cup \{\Gamma; \overrightarrow{x_r : T_r}; x : U_1 \vdash ?_k : \mathbf{Type}_{\top}\} \\[2pt] \Gamma \vdash T \overset{?}{\equiv} \Pi x : U_1.?_k[\overrightarrow{x_r/v_r}; x/x] \overset{\mathcal{U}}{\rightsquigarrow} \\[2pt] \Gamma \vdash t\ \overrightarrow{(x_r := v_r : T_r)}\ (x := u_1' : U_1) : ?_k[\overrightarrow{x_r/v_r}; x/u_1'] \downharpoonleft \overrightarrow{u_n} \overset{\mathcal{E}^T}{\rightsquigarrow} v : V \end{array}}{\Gamma \vdash t\ \overrightarrow{(x_r := v_r : T_r)} : T \downharpoonleft u_1\ \overrightarrow{u_n} \overset{\mathcal{E}^T}{\rightsquigarrow} v : V}$$

Correctness of $(\mathcal{E}^T\text{-flexible})$. The proof is similar to the one for the $\mathcal{E}^T$-prod rule. We only list the major differences here. The fact Σ', Φ', $\Gamma \vdash u_1' : U_1$ is now obtained by induction hypothesis on the second premise. The role of T_1 is now played by $?_k[\overrightarrow{x_r/v_r}; x/x]$. The induction hypothesis on the third premise yields Σ''', Φ'', $\Gamma \vdash T \downarrow \Pi x : U_1.?_k[\overrightarrow{x_r/v_r}; x/x]$ that was previously given directly by the rule pre-conditions (up to reduction of T). The rest of the proof follows without any changes. The only remaining check to be performed is the well-formedness of $?_k[\overrightarrow{x_r/v_r}; x/x]$ that follows from rule $\mathcal{K}$-meta using the rule pre-condition Σ, Φ, $\Gamma \vdash v_i : T_i \quad i \in \{1 \ldots r\}$. $\qquad\square$

Another reason for the complexity of the $\mathcal{E}^T$ algorithm is the need to infer a dependent type for the function f when its type is flexible (a metavariable). We now show an example of this scenario and an execution trace for the algorithm.

Example 3.10 (Inference of a maximally dependent type). *Consider the following input, where c_1, c_2, c_3, P_1, P_2 are such that $\vdash c_1 : \mathbb{N}$ and $\vdash c_2 : P_1(c_1)$ and $c_3 : P_2(c_1, c_2)$:*

$$\{\vdash ?_F : \mathbf{Type}\}, \emptyset, \emptyset \vdash \lambda f :?_F.f\ c_1\ c_2\ c_3$$

The rule $(\mathcal{E}^T\text{-flexible})$ *matches the input and since the argument c_1 has type $\mathbb{N}$, Σ is extended as follows:*

$$\Sigma = \{ \qquad\quad \vdash ?_F : \mathbf{Type}; \\ x : \mathbb{N}\ \vdash ?_S :?_T; \\ x : \mathbb{N}\ \vdash ?_T : \mathbf{Type}\}$$

Then $?_F$ gets unified with $\Pi x : \mathbb{N}.?_S$ obtaining the following substitution:

$$\Phi = \{\vdash ?_F := \Pi x : \mathbb{N}.?_S : \mathbf{Type}\}$$

The new type for the head of the application, morally $(f\ c_1)$, represented as $f\ (x := c_1)$, is $?_S[c_1/x]$. In the following call to $(\mathcal{E}^T\text{-flexible})$*, the argument c_2 has type $P_1(c_1)$. Σ is thus extended as follows:*

$$\Sigma = \{ \quad x : \mathbb{N}; y : P_1(c_1)\ \vdash ?_U :?_V; \\ x : \mathbb{N}; y : P_1(c_1)\ \vdash ?_V : \mathbf{Type}; \\ x : \mathbb{N}\ \vdash ?_S :?_T; \\ x : \mathbb{N}\ \vdash ?_T : \mathbf{Type}\}$$

Then $?_S[x/c_1]$ is unified with $\Pi y : P_1(c_1).?_U[x/c_1]$ obtaining

$$\Phi = \{ \qquad\qquad \vdash ?_F := \Pi x : \mathbb{N}.?_S : \textbf{\textit{Type}};$$
$$x : \mathbb{N} \quad \vdash ?_S := \Pi y : P_1(x).?_U :?_T;$$
$$x : \mathbb{N} \quad \vdash ?_T := \textbf{\textit{Type}} : \textbf{\textit{Type}}\}$$

The new type for the head of the application $(f\ c_1\ c_2)$ is $?_U[c_1/x; c_2/y]$. In the following call to $(\mathcal{E}^T-\text{flexible})$, the argument c_3 has type $P_2(c_1, c_2)$. Σ is thus extended as follows:

$$\Sigma = \{ \quad x : \mathbb{N}; y : P_1(c_1); z : P_2(c_1, c_2) \quad \vdash ?_Z :?_W;$$
$$x : \mathbb{N}; y : P_1(c_1); z : P_2(c_1, c_2) \quad \vdash ?_W : \textbf{\textit{Type}};$$
$$x : \mathbb{N}; y : P_1(c_1) \quad \vdash ?_U :?_V;$$
$$x : \mathbb{N}; y : P_1(c_1) \quad \vdash ?_V : \textbf{\textit{Type}}\}$$

Then $?_U[x/c_1; y/c_2]$ is unified with $\Pi z : P_2(c_1, c_2).?_Z[x/c_1; y/c_2]$ obtaining

$$\Phi = \{ \qquad\qquad\qquad \vdash ?_F := \Pi x : \mathbb{N}.?_S : \textbf{\textit{Type}};$$
$$x : \mathbb{N} \quad \vdash ?_S := \Pi y : P_1(x).?_U :?_T;$$
$$x : \mathbb{N} \quad \vdash ?_T := \textbf{\textit{Type}} : \textbf{\textit{Type}};$$
$$x : \mathbb{N}; y : P_1(c_1) \quad \vdash ?_U := \Pi z : P_2(x, y).?_Z :?_V;$$
$$x : \mathbb{N}; y : P_1(c_1) \quad \vdash ?_V := \textbf{\textit{Type}} : \textbf{\textit{Type}}\}$$

The final instantiation for $?_F$ is thus the maximally dependent type

$$\Phi(?_F) = \Pi x : \mathbb{N}.\Pi y : P_1(x).\Pi z : P_2(x, y).?_Z : \textbf{\textit{Type}}$$

where

$$\Sigma = \{ \quad x : \mathbb{N}; y : P_1(c_1); z : P_2(c_1, c_2) \quad \vdash ?_Z :?_W;$$
$$x : \mathbb{N}; y : P_1(c_1); z : P_2(c_1, c_2) \quad \vdash ?_W : \textbf{\textit{Type}}\}$$

$\square$

We conclude now the description of the refinement algorithm in type inference mode. The final missing rule is the most complicated one and deals with pattern matching.

$$(I_l : \Pi\overrightarrow{x_l : F_l}.\Pi\overrightarrow{y_r : G_r}.s) \in \mathsf{Env}$$
$$(k_j : \Pi\overrightarrow{x_l : F_l}.\Pi\overrightarrow{y_{n_j}^j : T_{n_j}^j}.I_l \; \overrightarrow{x_l} \; \overrightarrow{M_r^j}) \in \mathsf{Env} \qquad j \in \{1 \ldots n\}$$

$$\Sigma \leadsto \Sigma \cup \{\Gamma \vdash ?u_i : \; \overrightarrow{F_i[x_{i-1}/?u_{i-1}]}\} \qquad i \in \{1 \ldots l\}$$
$$\Sigma \leadsto \Sigma \cup \{\Gamma \vdash ?v_i : \; \overrightarrow{G_i[x_l/?u_l; y_{i-1}/?v_{i-1}]}\} \qquad i \in \{1 \ldots r\}$$
$$\Gamma \vdash t : \; I_l \; \overrightarrow{?u_l} \; \overrightarrow{?v_r} \; \overset{\mathcal{R}^{\Downarrow}}{\leadsto} \; t'$$

$$G_i' = G_i[\overrightarrow{x_l/?u_l}] \qquad i \in \{1 \ldots r\}$$
$$T_i'^j = T_i^j[\overrightarrow{x_l/?u_l}] \qquad j \in \{1 \ldots n\}, i \in \{1 \ldots n_j\}$$
$$M_i'^j = M_i^j[\overrightarrow{x_l/?u_l}] \qquad j \in \{1 \ldots n\}, i \in \{1 \ldots r\}$$

$$\Sigma \leadsto \Sigma \cup \{\Gamma' \vdash ?_1 : \; \mathbf{Type}_\top\}$$
$$\Gamma \vdash T : \; \overrightarrow{\Pi y_r : G_r'}.\Pi x : I_l \; \overrightarrow{?u_l} \; \overrightarrow{y_r}.?_1[] \; \overset{\mathcal{R}^{\Downarrow}}{\leadsto} \; T'$$
$$(s, \Phi(?_1)) \in \mathsf{elim}(\mathsf{PTS})$$

$$(\overset{\mathcal{R}^{\Uparrow}}{\leadsto} - \mathrm{match}) \quad \cfrac{\begin{array}{l} \overrightarrow{\Gamma; y_{n_j-1}^j : P_{n_j-1}^j} \vdash P_{n_j}^j \overset{?}{\equiv} T_{n_j}'^j \; \overset{\mathcal{U}}{\leadsto} \qquad j \in \{1 \ldots n\} \\ \overrightarrow{\Gamma; y_{n_j}^j : P_{n_j}^j} \vdash t_j : T' \; \overrightarrow{M_r'^j} \; (k_j \; \overrightarrow{?u_l} \; \overrightarrow{y_{n_j}^j}) \; \overset{\mathcal{R}^{\Downarrow}}{\leadsto} \; t_j' \quad j \in \{1 \ldots n\} \end{array}}{\Gamma \vdash \begin{pmatrix} \mathbf{match} \; t \; \mathbf{in} \; I_l \; \mathbf{return} \; T \\ [k_1 \; (\overrightarrow{y_{n_1}^1 : P_{n_1}^1}) \Rightarrow t_1 \mid \ldots \mid k_n \; (\overrightarrow{y_{n_n}^n : P_{n_n}^n}) \Rightarrow t_n] \end{pmatrix} \overset{\mathcal{R}^{\Uparrow}}{\leadsto}} }$$
$$\begin{pmatrix} \mathbf{match} \; t' \; \mathbf{in} \; I_l \; \mathbf{return} \; T' \\ [k_1 \; (\overrightarrow{y_{n_1}^1 : P_{n_1}'^1}) \Rightarrow t_1' \mid \ldots \mid k_n \; (\overrightarrow{y_{n_n}^n : P_{n_n}'^n}) \Rightarrow t_n'] \end{pmatrix} : T' \; \overrightarrow{?v_r} \; t'$$

The rule has been slightly simplified: in the actual implementation of Matita the test $(s, \Phi(?_1)) \in \mathsf{elim}(\mathsf{PTS})$ is relaxed to accept elimination of inhabitants of non informative data types in all cases under the restriction that the data type must be small. Intuitively, smallness corresponds to the idea that the inhabitant of the data type would be non informative even if declared in $\mathbf{Type}$. Typical examples are empty types and the Leibniz equality type. A precise definition of smallness together with the corresponding rules for pattern matching can be found in [4].

Note that the return type T' is usually an anonymous function, beginning with lambda abstractions. Thus the type inferred for the pattern match construct is a β-redex. In fact the actual code of Matita post-processes that type firing $(r + 1)$ β-redexes.

Theorem 3.11 (Termination). *The $\overset{\mathcal{R}^{\Uparrow}}{\leadsto}$ algorithm defined by the set of rules presented in this section including $\overset{\mathcal{R}^{\Downarrow}}{\leadsto}$, $\overset{\mathcal{C}}{\leadsto}$, $\overset{\mathcal{F}}{\leadsto}$ and $\overset{\mathcal{E}^T}{\leadsto}$ is terminating.*

Proof. The proof is by structural induction of the syntax of terms. The rules ($\overset{\mathcal{C}}{\leadsto} -\mathrm{ok}$), ($\overset{\mathcal{R}^{\Uparrow}}{\leadsto} -\mathrm{variable}$), ($\overset{\mathcal{R}^{\Uparrow}}{\leadsto} -\mathrm{sort}$) and ($\overset{\mathcal{R}^{\Uparrow}}{\leadsto} -\mathrm{constant}$) are base cases. The first one clearly terminates if the unification algorithm $\overset{?}{\equiv}$ terminates, while the others terminate since Γ and Env are finite and the test $(s_1, s_2) \in \mathsf{PTS}$ is also terminating.

Now that we proved that all the rules for $\overset{\mathcal{C}}{\rightsquigarrow}$, amounting to only one for the mono directional refiner, terminate, we can consider the ($\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ −default) and ($\overset{\mathcal{F}}{\rightsquigarrow}$ −ok) as aliases for $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$, as if we were inlining their code.

By induction hypothesis $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ (and $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ being now an alias) terminates when called on smaller terms. The rule ($\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ −meta) terminates because Φ and Σ are finite, thus lookups are terminating, and calls to $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ are done on smaller terms, so the induction hypothesis applies. The rule ($\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ −letin) calls $\overset{\mathcal{F}}{\rightsquigarrow}$, $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ and $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ on smaller terms, thus terminates by induction hypothesis. The same holds for ($\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ −lambda) and ($\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ −product). To prove that ($\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ −appl) terminates we use the induction hypothesis on the first premise and we are left to prove that $\overset{\mathcal{E}^{T}}{\rightsquigarrow}$ terminates as well. Note that $\overset{\mathcal{E}^{T}}{\rightsquigarrow}$ calls $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ and $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ on proper subterms of the n-ary application, thus the induction hypothesis applies and will be used in the next paragraph.

We show $\mathcal{E}^{T}$ terminates by induction on the list of arguments (i.e. the list of terms after $\downarrow$) assuming that the input term T is a well typed type. Thanks to the correctness property of $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$, ($\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ −appl) always passes to $\mathcal{E}^{T}$ a well typed type. The rule ($\mathcal{E}^{T}$−empty) clearly terminates. The recursive call in the rule ($\mathcal{E}^{T}$−prod) is on a shorter list of arguments, thus is terminating, and the call to $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ terminates by induction hypothesis. The term $T[x/u'_1]$ is a well typed type thanks to the subject reduction property of CIC, and the fact that the variable x and the term u'_1 have the same type (postcondition of $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$, called with expected type U_1). The call to $\triangleright_{\text{whd}}$ is terminating because T is well typed and CIC reduction rules, on well typed terms, form a terminating rewriting system.

The rule ($\mathcal{E}^{T}$−flexible) terminates because of the same arguments. The only non obvious step is that $\Pi x : U_1.?_k\overrightarrow{[x_r/v_r}; x/x]$ is a well typed type. The metavariable $?_k$ is declared in Σ of type $\textbf{Type}_\top$, thus cannot be instantiated with a term. Moreover, since CIC is a full PTS, the product $\Pi x : U_1.?_k$ is well typed of sort $\textbf{Type}_\top$.

The recursive calls in the last rule ($\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ −match) are always on smaller terms. We are left to prove that the expected type $(T' \ \overrightarrow{M'^{j}_r} \ (k_j \ \overrightarrow{?u_l} \ \overrightarrow{y^{j}_{n_j}}))$ passed to the recursive call made on the last line is indeed a well typed type. The term T' is obtained using $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$, and we thus know it is a function of type $\Pi\overrightarrow{y_r : G^{j}_r}.\Pi x : I_l \ \overrightarrow{?u_l} \ \overrightarrow{y_r}.?_1$. The arguments $\overrightarrow{M'^{j}_r}$ are as many as expected and have the right types according to the environment $\textsf{Env}$ (first two lines) and thanks to the fact that the substitutions $M'^{j}_i = M^{j}_i\overrightarrow{[x_l/?u_l]}$ preserves their types. Finally, the term $(k_j \ \overrightarrow{?u_l} \ \overrightarrow{y^{j}_{n_j}})$ has type $I_l \ \overrightarrow{?u_l} \ \overrightarrow{M^{j}_r}$ that is the expected one. Thus $(T' \ \overrightarrow{M'^{j}_r} \ (k_j \ \overrightarrow{?u_l} \ \overrightarrow{y^{j}_{n_j}}))$ has type $?_1$ that is a well typed type according to Σ. $\qquad\square$

3.3. Implementation remarks.

The choice of keeping Σ and Φ separate is important and motivated by the fact that their size is usually very different. While the number of proof problems in Σ is usually small, the substitution Φ may record, step by step, the whole proof input by the user and can grow to an arbitrary size. Each metavariable must be either declared in Σ or assigned in Φ, thus to know if a metavariable belongs to Φ it is enough to

test if it does not belong to Σ. Knowing if a metavariable is instantiated is a very common operation, needed for example by weak head normalization, and it must thus be possible to implement it efficiently.

Another important design choice is to design the kernel of the system so that it handles metavariables [4]. This enables to reuse a number of functionalities implemented in the kernel also during the refinement process, like an efficient reduction machinery. Also note that the extensions made to the type checker described in Definition 2.3 become dead code when the type checker is called on ground terms, and thus do not increase the size of the trusted code of the system.

Last, it is worth pointing out that the algorithm is mostly independent from the representation chosen for bound variables. Matita is entirely based on De Bruijn indexes, but the tedious *lift* function is mostly hidden inside reduction and only directly called twice in the actual implementation of this algorithm. In particular it is necessary only to deal with the types of variables that must be pulled from the context. This potentially moves the type under all the context entries following the variable declaration or definition, thus the type must be lifted accordingly.

3.3.1. *Rules for objects.* Objects are declarations and definitions of constants, inductive types and recursive and co-recursive functions that inhabit the environment Env. Exactly like terms, the user writes objects down using the external syntax and the objects need to be refined before passing them to the kernel for the final check before the insertion in the environment.

Definition 3.12 (Type checking for objects (Env $\vdash$ WF))**.** *The type checking algorithm for CIC objects takes as input a proof problem Σ and a substitution Φ, all assumed to be well formed, and an object o. It is denoted by:*

$$\mathsf{Env} \cup (\Sigma, \Phi, o) \vdash \mathit{WF}$$

and states that o is well typed.

This algorithm is part of the kernel, and described in Appendix 8. It is the basis for the construction of the corresponding refinement algorithm for objects, that is specified as follows.

Specification 3.13 (Refiner for objects ($\mathcal{R}$))**.** *A refiner algorithm $\mathcal{R}$ for CIC objects takes as input a proof problem Σ and a substitution Φ, all assumed to be well formed, and an object o. It fails or returns an object o', a proof problem Σ' and substitution Φ'. It is denoted by:*

$$(\Sigma, \; \Phi) \vdash o \;\; \overset{\mathcal{R}}{\rightsquigarrow} \; o' \; (\Sigma', \; \Phi')$$

Precondition:

$$\mathcal{WF}(\Sigma, \; \Phi)$$

Postcondition (parametric in $\preccurlyeq$):

$$\mathcal{WF}(\Sigma', \; \Phi') \quad \wedge \quad \Sigma', \Phi' \leq \Sigma, \Phi \quad \wedge \quad \mathsf{Env} \cup (\Sigma', \Phi', o') \vdash \mathit{WF} \quad \wedge \quad o' \preccurlyeq o$$

Note that an object can be a block of mutually recursive definitions or declarations, each one characterized by a different type. Thus the $\overset{\mathcal{R}}{\rightsquigarrow}$ rule does not return a single type, but a new object together with a new metavariable environment Σ' and substitution Φ'.

When Σ' is not empty, all the metavariables in Σ' correspond to proof obligations to be proved to complete the definition of the object, necessary to convince the system to accept the object definition. This is especially useful for instance in the formalization of category theory where definitions of concrete categories are made from definitions of terms (objects and morphisms) together with proofs that the categorical axioms hold. In such definitions, objects and functors are immediately fully specified, while the proof parts are turned into proof obligations. Spiwack's Ph.D. thesis [31] discusses this issue at length as a motivation for a complete re-design of the data type for proofs in Coq. The new data type is essentially the one used in this paper and it will be adopted in some forthcoming version of Coq. The old data type, instead, did not take the Curry-Howard isomorphism seriously in the sense that partial proofs were not represented by partial proof terms and the refinement of an object could not open proof obligations. This problem was already partially addressed by Sozeau [29] where he added a new system layer around the refiner to achieve the behavior that our refiner already provides.

$$
(\ \overset{\mathcal{R}}{\leadsto} -\text{axiom}) \quad \frac{\vdash T \overset{\mathcal{F}}{\leadsto} T' : s}{\vdash \mathbf{axiom}\ c : T \overset{\mathcal{R}}{\leadsto} \mathbf{axiom}\ c : T'}
$$

$$
(\ \overset{\mathcal{R}}{\leadsto} -\text{definition}) \quad \frac{\vdash T \overset{\mathcal{F}}{\leadsto} T' : s \qquad \vdash t : T' \overset{\mathcal{R}^{\Downarrow}}{\leadsto} t'}{\vdash \mathbf{definition}\ c : T := t \overset{\mathcal{R}}{\leadsto} \mathbf{definition}\ c : T' := t'}
$$

$$
(\ \overset{\mathcal{R}}{\leadsto} -\text{inductive})
$$

$$
\frac{
\left.
\begin{array}{l}
\vdash \Pi\overrightarrow{x_l : L_l}.A_i \overset{\mathcal{F}}{\leadsto} V_i' \\
\vdash V_i' \triangleright_{\text{whd}} \Pi\overrightarrow{x_l : L_l'}.A_i' \qquad \vdash A_i' \triangleright_{\text{whd}} \Pi\overrightarrow{y_{r_i} : R_{r_i}'}.s_i \\
\overrightarrow{x_l : L_l'}; \overrightarrow{I_n : V_n'} \vdash T_{i,k} \overset{\mathcal{F}}{\leadsto} T_{i,k}' : s_{i,k} \\
\overrightarrow{x_l : L_l'}; \overrightarrow{I_n : V_n'} \vdash T_{i,k}' \triangleright_{\text{whd}} \Pi\overrightarrow{z_{p_{i,k}} : V_{p_{i,k}}}.V_{i,k} \\
\Sigma \leadsto \Sigma \cup \{\overrightarrow{x_l : L_l'}; \overrightarrow{y_{j-1} : R_{j-1}'} \vdash ?_j : R_j'\} \quad j \in \{1 \ldots r_i\} \\
\overrightarrow{x_l : L_l'}; \overrightarrow{I_n : A_n'}; \overrightarrow{z_{p_{i,k}} : V_{p_{i,k}}} \vdash V_{i,k} \overset{?}{\equiv} I_i \overrightarrow{x_l} \overrightarrow{?_{r_i}} \overset{\mathcal{U}_=}{\leadsto}
\end{array}
\right\}
\begin{array}{l}
i \in \{1 \ldots n\} \\
\\
\\
i \in \{1 \ldots n\} \\
k \in \{1 \ldots m_n\}
\end{array}
}{
\vdash
\begin{pmatrix}
\Pi\overrightarrow{x_l : L_l}.\ \mathbf{inductive} \quad I_1\ :\ A_1 := k_{1,1} : T_{1,2} \mid \ldots \mid k_{1,m_1} : T_{1,m_1} \\
\qquad\qquad \mathbf{with}\ \ldots \\
\qquad\qquad \mathbf{with}\ I_n\ :\ A_n := k_{n,1} : T_{n,1} \mid \ldots \mid k_{n,m_n} : T_{n,m_n}
\end{pmatrix}
\overset{\mathcal{R}}{\leadsto}
}
$$

$$
\begin{pmatrix}
\Pi\overrightarrow{x_l : L_l'}.\ \mathbf{inductive} \quad I_1\ :\ A_1' := k_{1,1} : T_{1,2}' \mid \ldots \mid k_{1,m_1} : T_{1,m_1}' \\
\qquad\qquad \mathbf{with}\ \ldots \\
\qquad\qquad \mathbf{with}\ I_n\ :\ A_n' := k_{n,1} : T_{n,1}' \mid \ldots \mid k_{n,m_n} : T_{n,m_n}'
\end{pmatrix}
$$

The loop from 1 to n, ranges over all mutually inductive types of the block using the index i (for inductive). The other loop, from 1 to m_n, ranges over the m_n constructors of the n-th inductive type using index k (for constructor). $T_{i,k}$ is the type of the k-th constructor of the i-th inductive. Every inductive type in the block has the same number of homogeneous parameters l of type L_α for some α in $1 \ldots l$, and r_i extra arguments of type R_β for some β in $1 \ldots r_i$. As explained in Section 2, homogeneous arguments are not abstracted explicitly

in the types of the constructors, thus their context includes not only the inductive types $\overrightarrow{I_n}$ but also the homogeneous arguments $\overrightarrow{x_l}$. Note that in this rule we used I_n to mean the n-th inductive, in contrast with the rest of the paper where the index n means the number of homogeneous arguments, l here. The complete arity of the inductive types $\overrightarrow{V'}$ is a closed term. In fact that type is generated in an empty context in the first premise. This makes the context in which $T_{i,k}$ is processed valid: there is no variable capture when $\overrightarrow{x_l}$ is put before $\overrightarrow{I_n}$. Moreover the successful refinement of $\Pi\overrightarrow{x_l : L_l}.A_i$ in the empty context grants that the types of the homogeneous arguments do not depend on the inductive types $\overrightarrow{I_n}$. The last three premises just check that the type of each constructor is actually a product targeting the inductive type.

Whilst being already quite involved, this rule is only partial. It lacks the checks for positivity conditions, that are only implemented by the kernel. Since the kernel of Matita is able to deal with metavariables we can test for these conditions using directly the kernel after the refinement process. Nevertheless, when the inductive type fed to the kernel is partial, the checks cannot be precise: all non positive occurrences will be detected, but nothing will prevent the user from instantiating a missing part with a term containing a non positive occurrence. One could label metavariables in a such a way that the unification algorithm refuses to instantiate them with a term containing non positive occurrences of the inductive type, but our current implementation does not. Anyway, once the definition is completed by the user, another call to the kernel is made, and all non positive occurrences are detected.

Moreover, the kernel also checks that the sort $s_{i,k}$ of the type of every constructor $T_{i,k}$ is properly contained in the sort of the corresponding inductive s_i. Finally, one should also check that any occurrence of I_i in the types $T_{i,k}$ of the constructors is applied to $\overrightarrow{x_l}$. This test is also omitted since it is performed by the kernel during the test for positivity.

$$
(\ \overset{\mathcal{R}}{\leadsto}\ -\text{letrec})\ \ \cfrac{
\left.
\begin{array}{l}
\vdash \Pi\overrightarrow{x^i_{p_i} : T^i_{p_i}}.T^i_{p_i+1} \overset{\mathcal{F}}{\leadsto} T'_i : s_i \\[4pt]
\vdash T'_i \triangleright_{\text{whd}} \Pi\overrightarrow{x^i_{p_i} : T'^i_{p_i}}.T'^i_{p_i+1} \\[4pt]
\overrightarrow{f_n : T_n};\overrightarrow{x^i_{p_i} : T'^i_{p_i}} \vdash t_i : T''^i_{p_i+1} \overset{\mathcal{R}^{\Downarrow}}{\leadsto} t'_i
\end{array}
\right\} i \in \{1\ldots n\}
}{
\vdash \left(
\begin{array}{l}
\textbf{let rec } f_1(\overrightarrow{x^1_{p_1} : T^1_{p_1}}) : T^1_{p_1+1} := t_1 \textbf{ and } \ldots \\
\textbf{and } f_n(\overrightarrow{x^n_{p_n} : T^n_{p_n}}) : T^n_{p_n+1} := t_n
\end{array}
\right) \overset{\mathcal{R}}{\leadsto}
}
$$

$$
\left(
\begin{array}{l}
\textbf{let rec } f_1(\overrightarrow{x^1_{p_1} : T'^1_{p_1}}) : T'^1_{p_1+1} := t'_1 \textbf{ and } \ldots \\
\textbf{and } f_n(\overrightarrow{x^n_{p_n} : T'^n_{p_n}}) : T'^n_{p_n+1} := t'_n
\end{array}
\right)
$$

As for inductive types, this rule is only partial: it lacks the checks for guardedness conditions (termination or productivity tests), that are delegated to the kernel. We omit the rule for co-recursive functions, since it is identical to the one presented above.

4. Bi-directional refinement

To obtain a bi-directional implementation of the refiner, we add new rules to the $\overset{\mathcal{R}^{\Downarrow}}{\leadsto}$ algorithm. These ad-hoc rules for particular cases must take precedence over the generic

($\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ −default) rule. The ad-hoc rules are responsible for propagating information from the expected type towards the leaves of the term.

The new rule for lambda-abstraction is well known in the literature [24] and it is also the only one implemented in Coq. The rule for let-in statements is given to allow the system infer more concise types. The one for application of constructors is completely novel and it takes advantage of additional knowledge on the constant parameters of an inductive type. It is thus peculiar of the Calculus of (Co)Inductive Constructions. This is also the rule that, according to our experience, mostly affects the behavior of the refiner. It makes it possible to refine many more terms to be refined in frequently occurring situations where, using a mono directional algorithm, more typing information had to be given by hand.

Theorem 4.1 (Correctness). *The new rules given in this section do not alter the correctness of the $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ algorithm w.r.t. its specification for all admissible $\preccurlyeq$ relations that include the identity for terms in the internal syntax. In particular, the algorithm is correct when the identity relation is picked for $\preccurlyeq$.* □

$$
(\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} -\text{lambda}) \quad \frac{
\begin{array}{c}
\Gamma \vdash E \rhd_{\text{whd}} \Pi x : E_1.E_2 \\
\Gamma \vdash T \overset{\mathcal{F}}{\rightsquigarrow} T' : s \\
\Gamma \vdash T' \overset{?}{\equiv} E_1 \overset{\mathcal{U}_{\equiv}}{\rightsquigarrow} \\
\Gamma; x : T' \vdash t : E_2 \overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} t'
\end{array}
}{
\Gamma \vdash \lambda x : T.t : E \overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} \lambda x : T'.t'
}
$$

Note that to type t we push into the context the declared type for x and not its expected type E_1. This is to avoid displaying a confusing error message in case t is ill-typed, since the user declared x to have type T, and not E_1 (that in principle can be arbitrarily different from T).

$$
(\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} -\text{letin}) \quad \frac{
\begin{array}{c}
\Gamma \vdash T \overset{\mathcal{F}}{\rightsquigarrow} T' : s \\
\Gamma \vdash t : T' \overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} t' \\
\Gamma; x := t' : T' \vdash u : E[t'/x] \overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} u'
\end{array}
}{
\Gamma \vdash \textbf{let } (x : T) := t \textbf{ in } u : E \overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} \textbf{let } (x : T') := t' \textbf{ in } u'
}
$$

Where we denote by $[t'/x]$ the operation of substituting all occurrences of t' with x. Note that this operation behaves as an identity up to conversion (since x holds the value t'). Nevertheless, it enables the bi-directional type inference algorithm to propagate smaller types towards the leaves and, according to our observation, it leads to more readable inferred typed for sub-terms of u'.

Theorem 4.2 (Termination). *The $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ algorithm defined by the set of rules presented above with the addition of ($\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ −letin) and ($\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$ −lambda) terminates.*

Proof. The two functions terminate because all recursive calls are on smaller terms and because $\rhd_{\text{whd}}$, $\overset{\mathcal{F}}{\rightsquigarrow}$ and $\overset{?}{\equiv}$ terminate. □

The next rule deals with applications of constructors to arguments and it is only triggered when the expected type is an inductive type. In that case the application must be total. In CIC, the types of constructors of inductive types are constrained to have a particular shape. Up to reduction, their type must be of the form $\Pi x_1 : F_1 \ldots \Pi x_n : F_n . I\ x_1\ \ldots\ x_l\ t_{l+1}\ \ldots\ t_m$ where l is the number of uniform parameters of the inductive type. Therefore the application of a constructor to a list $u_1\ \ldots\ u_n$ of arguments has type $I\ u_1\ \ldots\ u_l\ v_{l+1}\ \ldots\ v_m$ for some vs. Reversing the reasoning, once we know that the expected type for the application of a constructor is $I\ u_1\ \ldots\ u_l\ v_{l+1}\ \ldots\ v_m$ we already know that the first l arguments of the application must be equal to $u_1\ \ldots\ u_l$ up to conversion. It is thus possible to propagate them following the bi-directional spirit. This is achieved by the following ($\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} -\mathrm{appl}-k$) that calls a new function denoted by $\mathcal{E}_t$ that consumes the first l arguments unifying them with the expected values. The remaining arguments are consumed as in the generic case of applications.

$$
(\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} -\mathrm{appl}-k) \quad \frac{
\begin{array}{l}
\Gamma \vdash E \triangleright_{\mathrm{whd}} I_l\ \overrightarrow{v_l}\ \overrightarrow{w_n} \\
\Gamma \vdash \overrightarrow{t_m} \overset{?}{\equiv} \overrightarrow{v_l}\ \overset{\mathcal{E}_t}{\rightsquigarrow}\ \overrightarrow{t_l'} \downarrow \overrightarrow{u_o} \\
(k : T) \in \mathsf{Env} \\
\Gamma \vdash T \triangleright_{\mathrm{whd}} \overrightarrow{\Pi x_l : S_l}.T' \\
\Gamma \vdash k\ \overrightarrow{t_l'} :\ T'[\overrightarrow{x_l/t_l'}] \downarrow \overrightarrow{u_o}\ \overset{\mathcal{E}^T}{\rightsquigarrow}\ r :\ R \\
\Gamma \vdash R \overset{?}{\equiv} I_l\ \overrightarrow{v_l}\ \overrightarrow{w_n}\ \overset{\mathcal{U}_{\equiv}}{\rightsquigarrow}
\end{array}
}{
\Gamma \vdash k\ \overrightarrow{t_m} :\ E\ \overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}\ r
}
$$

Note that if E does not reduce to an applied inductive type, the implemented algorithm falls back to the standard rule for application.

The rule presented only propagates information related to uniform parameters. Uniform parameters must be used consistently in every occurrence of the inductive type in the type of its constructors and not only in the occurrence at the end of the product spine (i.e. in the return type of the constructors). The variant of CIC implemented in Coq also considers non uniform parameters. Non uniform parameters must be used consistently only in the return type of the constructors and not in the premises. We do not consider non uniform parameters in this paper, but we remark that the ($\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} -\mathrm{appl}-k$) rule is also valid when the first l parameters are non uniform.

Specification 4.3 (Eat arguments ($\mathcal{E}_t$))**.** *The $\overset{\mathcal{E}_t}{\rightsquigarrow}$ algorithm takes a list of arguments for an application and a list of terms, and it verifies that an initial prefix of the arguments is equal to the given terms, up to unification. It takes as input a proof problem Σ, a substitution Φ and a context Γ, all assumed to be well formed, the list of arguments and the list of terms. It fails or it returns the list of arguments split into the consumed ones and the ones yet to be considered. It is denoted by:*

$$(\Sigma,\ \Phi)\ \Gamma \vdash t_1\ \ldots\ t_m \overset{?}{\equiv} v_1\ \ldots\ v_n\ \overset{\mathcal{E}_t}{\rightsquigarrow}\ t_1'\ \ldots\ t_n' \downarrow u_1\ \ldots\ u_l\ (\Sigma',\ \Phi')$$

Precondition:

$$\mathcal{WF}(\Sigma,\ \Phi,\ \Gamma)\quad \wedge\quad \Sigma,\ \Phi,\ \Gamma \vdash v_i : T_i\quad i \in \{1 \ldots n\}$$

Postcondition (parametric in $\preccurlyeq$):

$$\mathcal{WF}(\Sigma',\ \Phi') \quad \wedge \quad \Sigma',\Phi' \leq \Sigma,\Phi \quad \wedge \quad \Sigma',\ \Phi',\ \Gamma \vdash t'_i \downarrow v_i \quad i \in \{1\ldots n\} \quad \wedge$$
$$t'_1\ \ldots\ t'_n\ u_1\ \ldots\ u_l \preccurlyeq t_1\ \ldots\ t_m$$

$$(\mathcal{E}_t-\text{empty}) \quad \frac{}{\Gamma \vdash \overrightarrow{u_l} \overset{?}{\equiv} \overset{\mathcal{E}_t}{\rightsquigarrow} \ \boxed{\downarrow} \ \overrightarrow{u_l}}$$

$$(\mathcal{E}_t-\text{base}) \quad \frac{\begin{array}{c} \Gamma \vdash t_1 \overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow} t'_1 : T_1 \\ \Gamma \vdash t'_1 \overset{?}{\equiv} v_1 \overset{\mathcal{U}_=}{\rightsquigarrow} \\ \Gamma \vdash \overrightarrow{t_m} \overset{?}{\equiv} \overrightarrow{v_n} \overset{\mathcal{E}_t}{\rightsquigarrow} \overrightarrow{t'_n} \boxed{\downarrow} \overrightarrow{u_l} \end{array}}{\Gamma \vdash t_1 \overrightarrow{t_m} \overset{?}{\equiv} v_1 \overrightarrow{v_n} \overset{\mathcal{E}_t}{\rightsquigarrow} t'_1 \overrightarrow{t'_n} \boxed{\downarrow} \overrightarrow{u_l}}$$

Theorem 4.4 (Termination). *The $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ algorithm defined by the set of rules presented above with the addition of ($\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} -\text{appl} - k$), ($\mathcal{E}_t-\text{empty}$) and ($\mathcal{E}_t-\text{base}$) terminates.*

Proof. The rule ($\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} -\text{appl} - k$) terminates because $\overset{?}{\equiv}$ and $\triangleright_{\text{whd}}$ terminate and $\mathcal{E}^T$ is called on smaller terms. Moreover the term $T'[\overrightarrow{x_l/t'_l}]$ is a well typed type because $\mathcal{E}_t$ grants that $\overrightarrow{t'_l}$ are convertible with $\overrightarrow{v_l}$ and thus have the same types. Also notes that all the calls to $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ made by $\mathcal{E}_t$ are on sub-terms of the input of ($\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} -\text{appl} - k$).

We thus show that $\mathcal{E}_t$ terminates by induction on the second list of arguments (the one between $\overset{?}{\equiv}$ and $\overset{\mathcal{E}_t}{\rightsquigarrow}$). Rule ($\mathcal{E}_t-\text{empty}$) is the base case and clearly terminates. Rule ($\mathcal{E}_t-\text{base}$) terminates because $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ and $\overset{?}{\equiv}$ terminate and because the recursive call terminates by induction hypothesis. $\square$

4.1. **Remarks.** We present here a simple but frequently occurring case that explains why the bi-directional rule for application of constructors enables to refine many more terms w.r.t. the mono-directional algorithm. A more complicated example was already discussed in the introduction and deals with dependent data types to represent the syntax of languages with binders.

Consider the inductive type used to define the existential quantification.

$$\Pi T : \textbf{Type}.\Pi P : T \to \textbf{Prop. inductive } \text{Ex}\ :\ \textbf{Prop} :=$$
$$\text{Ex\_intro} : \Pi x : T.P\ x \to \text{Ex}\ T\ P$$

Note that T and P are homogeneous arguments.

Example 4.5 (Use of ($\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} -\text{appl}-k$)). *Consider the conjecture $\exists x : \mathbb{N}.x > 0$, encoded in CIC as*

$$Ex\ \mathbb{N}\ (\lambda x : \mathbb{N}.x > 0)$$

Given a context Γ containing the assumption p stating that $2 > 0$, one may want to use the following proof term to prove the conjecture

$$t = Ex\_intro\ ?_T\ ?_P\ ?_x\ p$$

A mono directional refiner encounters a hard unification problem involving the type of p, $2 > 0$, *and its expected type.*

$$\{\vdash ?_T \ : \ \textbf{\textit{Type}}; \ \vdash ?_P \ : ?_T \rightarrow \textbf{\textit{Prop}}; \ \vdash ?_x \ : ?_T\}, \emptyset, \Gamma \vdash 2 > 0 \ \stackrel{?}{\equiv} \ ?_P \ ?_x$$

Clearly, the desired solution is to instantiate $?_x$ *with* 2 *and* $?_P$ *with* $(\lambda x : \mathbb{N}.x > 0)$ *obtaining a proof term of type* $\exists x : \mathbb{N}.x > 0$. *Unfortunately, this is not the only possible solution. An undesired solution, but as reasonable as the correct one, is*

$$\Phi = \{?_T := \mathbb{N}; \ ?_x := 0; \ ?_P := \lambda x : \mathbb{N}.2 > x\}$$

under which the resulting proof term $\Phi(t)$ *has type* $\exists x : \mathbb{N}.2 > x$, *that is not the expected one. Why one should prefer the former to the latter is also unclear from a computer perspective. Thanks to the polymorphism of CIC, another undesired and less expected solution is also possible:*

$$\Phi' = \{?_T := \textbf{\textit{Prop}}; ?_x := 2 > 0; ?_P := \lambda x.x\}$$

The proof term $\Phi'(t)$ *would then be of type* $\exists x : \textbf{\textit{Prop}}.x$, *again different from the desired one.*

Using the expected type, $?_T$ *and* $?_P$ *are easily inferred looking at the homogeneous argument of the expected type.*

$$\Phi'' = \{?_T := \mathbb{N}; \ ?_P := \lambda x : \mathbb{N}.x > 0\}$$

Then inferring $?_x$ *is easy. Applying* Φ'' *to the right hand side of the unification problem we obtain:*

$$2 > 0 \ \stackrel{?}{\equiv} \ (\lambda x : \mathbb{N}.x > 0) \ ?_x$$

Then it is sufficient to reduce the right hand side and then perform a simple, first order, unification algorithm to obtain the desired instantiation for $?_x$. $\qquad\square$

The rule ($\stackrel{\mathcal{R}^{\Downarrow}}{\leadsto}-\text{appl}-k$) is only fired when the term to be refined is syntactically the application of a constructor. Because of conversion, the term under analysis could be reducible to an application of a constructor. However, we cannot reduce the term first to try to match the rule. The first motivation is that terms in the external syntax may contain placeholders (see Section 5) and may not be well typed. Duplication of placeholders and substitution into them is not admitted. Moreover, reducing an ill typed term may lead to divergence. Secondly, reduction of proof terms correspond to cut elimination that is known to yield proofs terms of arbitrary size.

5. EXTENSION TO PLACEHOLDERS

We consider here the first extension of our external syntax, obtained introducing linear placeholders for missing terms and for vectors of missing terms of unknown length. The latter are only accepted in argument position, even if we will enforce this only in the refinement algorithm and not in the syntax. The new syntax is obtained extending the one given in Table 1 with the new productions of Table 2. Placeholders are also called implicit arguments in the literature, but that terminology is ambiguous since it is also used for arguments that can be entirely omitted in the concrete syntax.

In a concrete implementation, user defined notations are used to further enlarge the external syntax. User defined notations behave as macros; macro expansion gives back a term in the external syntax we consider here. In particular, thanks to user defined notations,

$$
\begin{array}{rcl}
t & ::= & \ldots \\
& | & ? \qquad \text{placeholder} \\
& | & \overrightarrow{?} \qquad \text{arbitrarily many placeholder arguments}
\end{array}
$$

Table 2: CIC terms syntax II - Placeholders

it is possible to entirely omit the typing information in binders, like in calculi typed à la Curry. Omitted types are turned into placeholders during the macro expansion phase. Implicit arguments can also be simulated by defining notations that insert into applications a fixed number of placeholders or vectors of placeholders in appropriate positions.

A placeholder ? differs from a metavariable $?_i$ in the fact that it has no sequent associated to it. The intended associated sequent allows in ? occurrences of all variables in the context of ?. Moreover, the type of ? is meant to be the one determined by the context. This information is made explicit by the refinement algorithm that turns each placeholder into a corresponding metavariable.

Placeholders occur only linearly in the term (i.e. every occurrence of a placeholder is free to be instantiated with a different term). Non linear placeholders are not allowed since two occurrences could be in contexts that bind different set of variables and instantiation with terms that live in one context would make no sense in the other one.

Similarly, substitution is not allowed on placeholders since a placeholder occurrence does not have a corresponding explicit substitution.

For both previous reasons, reduction is not allowed on terms in the external syntax that contain placeholders: the reduction, conversion and unification judgements only make sense on refined terms.

Intuitively, a vector of placeholders can be instantiated by the refiner with zero or more metavariables. In our algorithm we adopt a lazy semantics: vectors of placeholders can only be used in argument position and a vector is expanded to the minimal number of metavariables that make the application well typed. Bi-directionality, i.e. the knowledge about the expected type for the application, is required for the lazy semantics. Indeed, without the expected type, an expansion could produce a locally well typed application whose inferred type will not match later on with the expected one.

We extend the $\overset{\mathcal{R}^{\Uparrow}}{\leadsto}$, $\overset{\mathcal{R}^{\Downarrow}}{\leadsto}$, $\overset{\mathcal{E}^{T}}{\leadsto}$ and $\overset{\mathcal{E}_t}{\leadsto}$ algorithms with new rules for single placeholders and for vectors of placeholders.

Theorem 5.1 (Correctness). *The $\overset{\mathcal{C}}{\leadsto}$, $\overset{\mathcal{F}}{\leadsto}$, $\overset{\mathcal{R}^{\Uparrow}}{\leadsto}$, $\overset{\mathcal{R}^{\Downarrow}}{\leadsto}$, $\overset{\mathcal{E}^{T}}{\leadsto}$, and $\overset{\mathcal{E}_t}{\leadsto}$ algorithms extended with the set of rules presented in this section obey their specification for all admissible $\preccurlyeq$ that include the $\preccurlyeq'$ relation defined as follows: $t' \preccurlyeq' t$ when t' is obtained from t by replacing single placeholders with terms and vectors of placeholders with vectors of terms. In particular, the algorithms are correct w.r.t. $\preccurlyeq'$.*

Proof. Every admissible $\preccurlyeq$ that includes $\preccurlyeq'$ also includes the identity. Thus we do not need to re-establish the result on the rules given in the previous sections. Correctness of the new rules given in this section is established by rule inspection. □

$$
(\overset{\mathcal{R}^{\Uparrow}}{\leadsto} -\text{placeholder}) \qquad \frac{\Sigma \leadsto \Sigma \cup \{\Gamma \vdash ?_l : \mathbf{Type}_\top \,,\; \Gamma \vdash ?_k : ?_l \,,\; \Gamma \vdash ?_j : ?_k\}}{\Gamma \vdash ? \overset{\mathcal{R}^{\Uparrow}}{\leadsto} ?_j : ?_k}
$$

$$(\overset{\mathcal{R}^{\Downarrow}}{\leadsto} -\text{placeholder}) \quad \dfrac{\Sigma \leadsto \Sigma \cup \{\Gamma \vdash ?_k : T\}}{\Gamma \vdash ? : T \overset{\mathcal{R}^{\Downarrow}}{\leadsto} ?_k}$$

$$(\mathcal{E}^T - \overrightarrow{\text{placeholder}} -0) \quad \dfrac{\Gamma \vdash t \, \overrightarrow{(x_r := v_r : T_r)} : T \, \big\lrcorner \, \overrightarrow{u_n} \overset{\mathcal{E}^T}{\leadsto} v : V}{\Gamma \vdash t \, \overrightarrow{(x_r := v_r : T_r)} : T \, \big\lrcorner \, \overrightarrow{?} \, \overrightarrow{u_n} \overset{\mathcal{E}^T}{\leadsto} v : V}$$

$$(\mathcal{E}^T - \overrightarrow{\text{placeholder}} +1) \quad \dfrac{\begin{array}{c}\Gamma \vdash T \rhd_{\text{whd}} \Pi x_1 : U_1.T_1 \\ \Gamma \vdash t \, \overrightarrow{(x_r := v_r : T_r)} : T \, \big\lrcorner \, ? \, \overrightarrow{?} \, \overrightarrow{u_n} \overset{\mathcal{E}^T}{\leadsto} v : V\end{array}}{\Gamma \vdash t \, \overrightarrow{(x_r := v_r : T_r)} : T \, \big\lrcorner \, \overrightarrow{?} \, \overrightarrow{u_n} \overset{\mathcal{E}^T}{\leadsto} v : V}$$

The rule $(\mathcal{E}^T - \overrightarrow{\text{placeholder}} -0)$ is meant to take precedence over $(\mathcal{E}^T - \overrightarrow{\text{placeholder}} +1)$. The second is applied when the first one fails (local backtracking).

Theorem 5.2 (Termination). *The $\overset{\mathcal{R}^{\Uparrow}}{\leadsto}$ algorithm defined by the set of rules presented above with the addition of $(\overset{\mathcal{R}^{\Uparrow}}{\leadsto} -\text{placeholder})$, $(\overset{\mathcal{R}^{\Downarrow}}{\leadsto} -\text{placeholder})$, $(\mathcal{E}^T - \overrightarrow{\text{placeholder}} -0)$ and $(\mathcal{E}^T - \overrightarrow{\text{placeholder}} +1)$ terminates.*

Proof. Rules $(\overset{\mathcal{R}^{\Uparrow}}{\leadsto} -\text{placeholder})$ and $(\overset{\mathcal{R}^{\Downarrow}}{\leadsto} -\text{placeholder})$ terminate. The proof that $\mathcal{E}^T$ terminates is, as before, by induction on the list of arguments that follow $\big\lrcorner$. The rule $(\mathcal{E}^T - \overrightarrow{\text{placeholder}} -0)$ terminates by induction hypothesis. The rule $(\mathcal{E}^T - \overrightarrow{\text{placeholder}} +1)$ deserves an accurate treatment. The check over T, asking it to be a product, is to avoid divergence. Since the input T is a well typed type, also T_1 is, and thus it admits a normal form T_1' in which x may occur. The recursive call does necessarily trigger the rule $(\mathcal{E}^T -\text{product})$ that will substitute a metavariable $?_j$ for x in T_1. Thanks to the reduction rules of CIC, reported in the appendix, substituting a variable for a metavariable declared in Σ (and not in Φ) does not change the normal form, meaning that $T_1'[x/?j] \rhd_{\text{whd}} T_1'[x/?j]$. Thus the rule $(\mathcal{E}^T - \overrightarrow{\text{placeholder}} +1)$ can be applied only a finite number of times, and the number of products in T is an upper bound. $\qquad\square$

The next two rules for the $\overset{\mathcal{E}_t}{\leadsto}$ judgment follow the same schema of the ones for $\overset{\mathcal{E}^T}{\leadsto}$.

$$(\mathcal{E}_t - \overrightarrow{\text{placeholder}} -0) \quad \dfrac{\Gamma \vdash \overrightarrow{t_m} \overset{?}{\equiv} \overrightarrow{v_n} \overset{\mathcal{E}_t}{\leadsto} \overrightarrow{t_n'} \big\lrcorner \overrightarrow{u_l}}{\Gamma \vdash \overrightarrow{?} \; \overrightarrow{t_m} \overset{?}{\equiv} \overrightarrow{v_n} \overset{\mathcal{E}_t}{\leadsto} \overrightarrow{t_n'} \big\lrcorner \overrightarrow{u_l}}$$

$$(\mathcal{E}_t - \overrightarrow{\text{placeholder}} +1) \quad \dfrac{\Gamma \vdash ? \; \overrightarrow{?} \; \overrightarrow{t_m} \overset{?}{\equiv} \overrightarrow{v_n} \overset{\mathcal{E}_t}{\leadsto} \overrightarrow{t_n'} \big\lrcorner \overrightarrow{u_l}}{\Gamma \vdash \overrightarrow{?} \; \overrightarrow{t_m} \overset{?}{\equiv} \overrightarrow{v_n} \overset{\mathcal{E}_t}{\leadsto} \overrightarrow{t_n'} \big\lrcorner \overrightarrow{u_l}}$$

Theorem 5.3 (Termination). *The $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$ algorithm defined by the set of rules presented above with the addition of $(\mathcal{E}_t-\overrightarrow{\text{placeholder}}-0)$ and $(\mathcal{E}_t-\overrightarrow{\text{placeholder}}+1)$ terminates.*

Proof. The rule $(\mathcal{E}_t-\overrightarrow{\text{placeholder}}-0)$ makes a recursive call on the same list of arguments $\overrightarrow{v_n}$ but consumes a $\overrightarrow{?}$, and no other rule of the refiner adds one, so it can be repeated only a finite number of times. The recursive call in the rule $(\mathcal{E}_t-\overrightarrow{\text{placeholder}}+1)$ can trigger only rules $(\mathcal{E}_t-\text{empty})$ and $(\mathcal{E}_t-\text{base})$. The former terminates immediately, the latter will do a recursive call consuming one argument in $\overrightarrow{v_n}$, and thus terminates. $\qquad\square$

Note that inlining the latter would lead to a rule whose termination is trivial to see, but we preferred to present the algorithm in a more modular way.

Example 5.4 (Vector of placeholders). *Assume a theorem $\tau \in \mathsf{Env}$ shows that $\forall x : \mathbb{N}.P\ x \to Q\ x$. The proof context may contain a natural number y and optionally a proof H that y validates P. Different proofs or proof styles may use the same theorem τ in different ways. For example, one may want to perform forward reasoning, and tell the system to assume $(Q\ y)$ providing the following proof for it*

$$y : \mathbb{N}; H : P\ y \vdash \tau\ H$$

Nevertheless, sometimes H is not known, and the user may want to tell the system he has intention to use the theorem τ on y, and prove $(P\ y)$ later.

$$y : \mathbb{N} \vdash \tau\ y$$

While the latter application is well typed, the first is not, since the first argument of τ must be of type $\mathbb{N}$. Nevertheless, the type of H depends on y, thus the term $(\tau\ ?\ H)$ would refine to the well typed term $(\tau\ y\ H)$ of type $(Q\ y)$.

The vector of placeholders enables the system to accept both terms originally written by the user. In the first case $(\tau\ \overrightarrow{?}\ H)$ would expand to $(\tau\ ?\ H)$ thanks to $(\mathcal{E}_t-\overrightarrow{\text{placeholder}}+1)$, and refine to $(\tau\ y\ H)$. In the second case $(\tau\ \overrightarrow{?}\ y)$ would refine to $(\tau\ y)$ thanks to $(\mathcal{E}_t-\overrightarrow{\text{placeholder}}-0)$. This suggests defining $(\tau\ \overrightarrow{?})$ as a notation for the theorem τ, obtaining a cheap implementation of what other systems call prenex implicit arguments: the first n arguments of an application whose head has a dependent type like $\overrightarrow{\Pi x_n : T_n}.\overrightarrow{\Pi y_m : P(\overrightarrow{x_n})}.T$ can be omitted, and are inferred thanks to the dependencies in the types of the m following arguments. As a bonus, in case the user wants to pass one of the implicit arguments there is no need to temporarily disable the mechanism, since the expansion of $\overrightarrow{?}$ is computed on the fly and automatically adapts to its context. $\qquad\square$

6. Coercions

Coercions are explicit type casts. While the literature [20] considers them mostly as a device to mimic sub-typing in a calculus lacking it, they have other interesting applications. The refiner of Matita inserts coercions in three locations:

- around the argument of an application

- around the head of an application
- around the type of an abstraction

The first case is the most common one, and is the one that can easily be explained in terms of sub-typing. For example, if one applies an operation defined over integers $\mathbb{Z}$ to an argument lying in the type of natural numbers $\mathbb{N}$, the system injects the argument into the right type by means of the obvious, user declared, cast function mapping naturals into the non negative fragment of $\mathbb{Z}$.

The second case is handy in two situations. First when the head of the application is implicit in the standard notation, like in $3x$ where the intended head constant is the multiplication but in the input it happens to be 3. The second is when the head constant has a non ambiguous interpretation as a function, but is not. For example a set may act as its characteristic function.

The last case is recurrent when algebraic structures are encoded as dependently typed records [25] embedding the type (or carrier) for the elements together with the operations and properties defining the structure. In that case, one may want to state a theorem quantifying over a structure G, say a group, and some elements in that group. However the statement $\forall G : \mathrm{Group}.\forall x, y : G.P(x, y)$ is ill-typed since G is a term (of type Group) but is used as a type for x and y. The intended meaning is clear: x and y lie in the carrier type of G. The system can thus insert around G the projection for the carrier component of the Group record.

Definition 6.1 (Coercion set (Δ)). *A coercion set Δ is a set of pairs (c, k) where c is a constant in* Env *and k is a natural number smaller than the arity of c (i.e. k points to a possible argument of c)*

In the literature the coercion set is usually represented as a graph. Given a coercion (c, k) such that $(c : \Pi x_1 : T_1 \dots \Pi x_k : T_k \dots \Pi x_n : T_n.T) \in$ Env, T_k and T are nodes in the graph, and c is an edge from T_k to T. Most coercion implementation, like the one of Coq, Lego and Plastic, assume Δ to be a graph validating a property called coherence. This property states that Δ is an acyclic graph with at most one path linking every pair of nodes. This property enables to employ a straightforward algorithm to look for a sequence of coercions linking two non adjacent nodes in the graph.

In Matita, for various reasons detailed in [34], Δ is not a graph, but a set of arcs for the transitive closure of the graph. Every time a coercion c is declared by the user, and thus added to Δ, the following set of automatically generated composite coercions is also added to Δ.

$$\{c_i \circ c \circ c_j | c_i \in \Delta \land c_j \in \Delta\} \cup \{c \circ c_j | c_j \in \Delta\} \cup \{c_i \circ c | c_i \in \Delta\}$$

Of course the $\circ$ operator here is partial, and only well typed composite coercions are actually considered. This design choice enables the coercion lookup operation to be single step, since the set is already transitively closed. Moreover, since composite coercions are defined constants in Env, the term resulting after a cast is smaller if compared with the one obtained inserting the corresponding chain of user declared coercions. Last, allowing k to differ from n is a peculiarity of Matita. When $k \neq n$ the application of the coercion creates new uninstantiated metavariables that correspond to proof obligations. This will be detailed later on.

The last detail worth mentioning is that, all systems known to the authors with the notable exception of Plastic [9], adopt some approximated representation for the nodes in the coercion graph, usually the name of the head constant of the source and target types. This

results in a faster lookup in the coercion graph, but the coherence check is also strengthened. In particular, in a calculus with dependent types, different, but similar, coercions may not be allowed to be declared. Matita drops the coherence check, or better changes it into a warning, and enables the user to attach to coercions a priority: coercions from and to the same approximation of types are all tried according to user defined priorities.

Specification 6.2 (Coercion lookup ($\rightarrowtail - \Delta$)). *Given a context Γ, substitution Φ and proof problem Σ, all assumed to be well formed, two types T_1 and T_2, this function returns an explicit cast $c\ ?_1\ \ldots\ ?_k\ \ldots\ ?_n$ for the metavariable of index k and its type T'. It is denoted by:*

$$(\Sigma,\ \Phi)\ \Gamma \vdash T_1 \rightarrowtail T_2\ \overset{\Delta}{\rightsquigarrow}\ k,\ c\ ?_1\ \ldots\ ?_k\ \ldots\ ?_n :\ T'\ (\Sigma', \Phi)$$

Precondition (parametric in $\approx$):

$$\mathcal{WF}(\Sigma,\ \Phi,\ \Gamma)\quad \wedge\quad (c, k) \in \Delta\quad \wedge\quad (c : \Pi x_1 : T_1 \ldots \Pi x_k : T_k \ldots \Pi x_n : T_n.T) \in \mathsf{Env}\quad \wedge$$

$$T_k \approx T_1\quad \wedge\quad T \approx T_2$$

Postcondition:

$$\mathcal{WF}(\Sigma')\quad \wedge\quad \Sigma',\ \Phi,\ \Gamma \vdash c\ ?_1\ \ldots\ ?_k\ \ldots\ ?_n : T'$$

We denoted by $\approx$ the approximated comparison test used to select from Δ a coercion c from T_1 to T_2. A proper definition of $\approx$ is not relevant for the present paper, but we can anyway say that Matita compares the first order skeleton of types obtained by dropping bound variables, metavariables and higher order terms, and that this skeleton can be made less precise on user request. We will give an account of this facility in the example that will follow.

The new metavariables $?_1, \ldots, ?_n$ generated by the lookup operation are all declared in the new proof problem Σ'. The number of metavariables to which c is applied to is defined when the coercion is declared and may be less than the arity of c. In the latter case T is a product and the coercion casts its k-th argument to be a function. The position k of the casted argument is user defined as well. The coerced term has then to be later unified with $?_k$.

Theorem 6.3 (Correctness). *The $\overset{\mathcal{C}}{\rightsquigarrow}$, $\overset{\mathcal{F}}{\rightsquigarrow}$, $\overset{\mathcal{R}^{\Uparrow}}{\rightsquigarrow}$, $\overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow}$, $\overset{\mathcal{E}^T}{\rightsquigarrow}$, and $\overset{\mathcal{E}_t}{\rightsquigarrow}$ algorithms extended with the set of rules presented in this section obey their specification where $\preccurlyeq''$ is the following admissible order relation: $t' \preccurlyeq'' t$ when t' is obtained from t by replacing single placeholders with terms, vectors of placeholders with vectors of terms, and terms u_k with terms convertible to $(c\ u_1\ \ldots\ u_k\ \ldots\ u_n)$ where c is a coercion declared in Δ for its kth argument.*

Proof. We do not need to re-establish correctness for the rules given in the previous sections since $\preccurlyeq''$ is admissible and includes $\preccurlyeq'$. Correctness of the new rules given in this section is established by rule inspection as usual. $\square$

In the following rule the coercion c is applied to its argument t unifying it with $?_k$. The returned term t' can still contains metavariables: $?_1 \ldots ?_{k-1}$ may appear in the type of $?_k$, thus unifying $?_k$ with t may instantiate them[6], but $?_{k+1} \ldots ?_n$ do not appear in the type of $?_1 \ldots ?_k$, and thus cannot be all instantiated. This rule is applied as a fall back in case $\overset{\mathcal{C}}{\rightsquigarrow} -$ok fails.

---

[6]In the case of dependent types the unification of the types is a necessary condition for the unification of the two terms, as claimed by Strecker [32].

$$\Gamma \vdash T_1 \rightarrowtail T_2 \overset{\Delta}{\rightsquigarrow} k, \ c \ \overrightarrow{?_m} \ ?_k \ \overrightarrow{?_n} : T_2'$$
$$\Gamma \vdash ?_k \overset{?}{\equiv} t \overset{\mathcal{U}_\equiv}{\rightsquigarrow}$$
$$(\overset{\mathcal{C}}{\rightsquigarrow} -\text{coercion}) \quad \dfrac{\Gamma \vdash T_2' \overset{?}{\equiv} T_2 \overset{\mathcal{U}}{\rightsquigarrow}}{\Gamma \vdash t : T_1 \overset{?}{\equiv} T_2 \overset{\mathcal{C}}{\rightsquigarrow} c \ \overrightarrow{?_m} \ ?_k \ \overrightarrow{?_n}}$$

Correctness of ($\overset{\mathcal{C}}{\rightsquigarrow}$ $-$coercion). Since $?_k$ is unified with t in the second premise of the rule, by definition of unification we have Σ, Φ, $\Gamma \vdash ?_k \downarrow t$, and thus $c \ \overrightarrow{?_m} \ ?_k \ \overrightarrow{?_n} \preccurlyeq'' t$. Moreover, the postconditions of coercion lookup $\overset{\Delta}{\rightsquigarrow}$ grant that $c \ \overrightarrow{?_m} \ ?_k \ \overrightarrow{?_n}$ has type T_2' that is later unified with T_2. Thus the postconditions of the unification algorithm allow us to prove that $c \ \overrightarrow{?_m} \ ?_k \ \overrightarrow{?_n}$ has a type convertible with T_2. $\qquad\square$

The $\overset{\mathcal{C}}{\rightsquigarrow}$ $-$coercion rule automatically takes care of the insertion of coercions around arguments of an application and around the types of an abstraction.

The following extension to $\overset{\mathcal{E}^T}{\rightsquigarrow}$ take cares of insertion around the head of an application.

$$\Sigma \rightsquigarrow \Sigma \cup \{\Gamma \vdash ?_{1'} : \mathbf{Type}_\top, \ \Gamma \vdash ?_1 : ?_{1'}\}$$
$$\Sigma \rightsquigarrow \Sigma \cup \{\Gamma; x_1 : ?_1 \vdash ?_{2'} : \mathbf{Type}_\top, \ \Gamma; x_1 : ?_1 \vdash ?_2 : ?_{2'}\}$$
$$\Gamma \vdash t \ \overrightarrow{v_r} : T \overset{?}{\equiv} \Pi x : ?_1.?_2 \overset{\mathcal{C}}{\rightsquigarrow} c \ \overrightarrow{w_s}$$
$$\Gamma \vdash w_i : W_i \quad i \in \{1 \dots s\}$$
$$(\mathcal{E}^T-\text{coercion}) \quad \dfrac{\Gamma \vdash c \ \overrightarrow{(x_s := w_s : W_s)} : \Pi x : ?_1.?_2 \ \downharpoonleft \ u_1 \ \overrightarrow{u_n} \overset{\mathcal{E}^T}{\rightsquigarrow} v : V}{\Gamma \vdash t \ \overrightarrow{(x_r := v_r : T_r)} : T \ \downharpoonleft \ u_1 \ \overrightarrow{u_n} \overset{\mathcal{E}^T}{\rightsquigarrow} v : V}$$

Correctness of $\mathcal{E}^T-$coercion. Its correctness follows trivially from the correctness of ($\overset{\mathcal{C}}{\rightsquigarrow}$ $-$coercion). $\qquad\square$

Theorem 6.4 (Termination). *The* $\overset{\mathcal{R}^\Uparrow}{\rightsquigarrow}$ *algorithm defined by the set of rules presented above with the addition of* ($\overset{\mathcal{C}}{\rightsquigarrow}$ $-$coercion) *and* ($\mathcal{E}^T-$coercion) *terminates.*

Proof. Rule ($\overset{\mathcal{C}}{\rightsquigarrow}$ $-$coercion) clearly terminates. The rule ($\mathcal{E}^T-$coercion) issues a recursive call to $\overset{\mathcal{E}^T}{\rightsquigarrow}$ without consuming u_1, but the only rule that can be triggered is ($\mathcal{E}^T-$product), that will immediately consume u_1. $\qquad\square$

Inlining ($\mathcal{E}^T-$product) would result in a rule that consumes some input and thus clearly terminates, but would be way less readable.

6.1. **Implementation remarks.** Since we allow coercion arguments not to be inferred automatically (like proof obligations) their type may depend on the coerced term (e.g. the proof that the coerced integer is greater than zero has an instance of the coerced integer in its type, and the corresponding metavariable will have index greater than k).

Example 6.5 (Coercion with side conditions). *Consider the following coercion set, declaring the coercion* v_to_nel *from vectors to non empty lists.*

$$\Delta = \{(\text{v\_to\_nel}, 3)\}$$

The environment holds the following type for the coercion:

$$(\text{v\_to\_nel} : \Pi A : \textbf{\textit{Type}}.\Pi n : \mathbb{N}.\Pi v : \text{Vect } A \ n.n > 0 \rightarrow \exists l : \text{List } A, \text{length } l > 0) \in \mathsf{Env}$$

Now consider the term $t = (\text{Vcons } \mathbb{N} \ 0 \ (\text{Vnil } \mathbb{N}) \ 2)$ *and the following coercion problem:*

$$
\cfrac{
\begin{array}{l}
\Gamma \vdash \text{Vect } \mathbb{N} \ (0+1) \rightarrowtail (\exists l : \text{List } \mathbb{N}, \text{length } l > 0) \ \overset{\Delta}{\leadsto} \\
\qquad 3, \ \text{v\_to\_nel } ?_1 \ ?_2 \ ?_3 \ ?_4 : \ (\exists l : \text{List } ?_1, \text{length } l > 0) \\
\Gamma \vdash ?_3 \overset{?}{\equiv} t \ \overset{\mathcal{U}_{\equiv}}{\leadsto} \\
\Gamma \vdash (\exists l : \text{List } ?_1, \text{length } l > 0) \overset{?}{\equiv} (\exists l : \text{List } \mathbb{N}, \text{length } l > 0) \ \overset{\mathcal{U}}{\leadsto}
\end{array}
}{
\Gamma \vdash t : \text{Vect } \mathbb{N} \ (0+1) \overset{?}{\equiv} (\exists l : \text{List } \mathbb{N}, \text{length } l > 0) \ \overset{\mathcal{C}}{\leadsto} \ \text{v\_to\_nel } ?_1 \ ?_2 \ ?_3 \ ?_4
}
$$

where the final proof problem and substitutions are:

$$\Sigma = \{\Gamma \vdash ?_4 : ?_2 > 0\}$$

$$\Phi = \{\Gamma \vdash ?_1 := \mathbb{N} : \textbf{\textit{Type}}, \ \Gamma \vdash ?_2 := 0 + 1 : \mathbb{N}, \ \Gamma \vdash ?_3 := t : \text{Vect } \mathbb{N} \ (0 + 1)\}$$

Note that $?_4$ *is still in* Σ, *thus it represent a proof obligation the user will be asked to solve. Also note that the following coercion could be declared as well, with a higher precedence. It is useful since it does not open a side condition when the type of the coerced vector is explicit enough to make the proof that it is not empty constant (not depending on* Γ *nor on the vector but just on its type) and thus embeddable in the body of the coercion.*

$$(\text{nev\_to\_nel}, \ \Pi A : \textbf{\textit{Type}}.\Pi n : \mathbb{N}.\Pi v : \text{Vect } A \ (n+1).\exists l : \text{List } A, \text{length } l > 0) \in \mathsf{Env}$$

The system would thus try nev_to_nel *first, and fall back to* v_to_nel *whenever needed.* □

Also note that this last coercion can be indexed as a cast from $(\text{Vect } \_ \ (\_ + 1))$ to $(\exists l : \text{List } \_, \text{length } l > 0)$ or in a less precise way. For example the approximation of the source type could be relaxed to $(\text{Vect } \_ \ \_)$. This will force the system to try to apply this coercion even if the casted term is a vector whose length is not explicitly mentioning $+1$, but is something that unifies with $?_j + 1$. For example the length $1 * 2$ would unify, since its normal form is $(0 + 1) + 1$.

7. Comparison with related work on Type Inference

Type inference is a very widely studied field of computer science. Nevertheless to the authors' knowledge there is no precise account of a type inference algorithm for the full CIC calculus in the literature.

The extension to the typing algorithm of CIC with explicit casts in [28] follows the same spirit of our refinement algorithm for raw terms. However the work by Saibi does not handle placeholders nor metavariables, and the presentation is in fact quite distant from the actual implementation in the Coq interactive prover.

Another work in topic is [22] where Norell describes the bi-directional type inference algorithm implemented in the Agda interactive prover. He presents the rules for a core dependently typed calculus enriched with dependent pairs. Unfortunately he omits the rules for its extension with inductive types. It is thus hard to tell if Agda exploits the type

expected by the context to type check inductive constructors as in rule ($\overset{\mathcal{R}^{\Downarrow}}{\leadsto} -\mathrm{appl}-k$).

Agda does not provide an explicit $\overrightarrow{?}$ placeholder but uses the expected type to know when it is necessary to pad an application with meta variables in order to reduce the arity of its type. In our setting this is equivalent to the following transformation: every application $(f\ \overrightarrow{a})$ is turned into $(f\ \overrightarrow{a}\ \overrightarrow{?})$ whenever its expected type is known (i.e. not a metavariable).

One aspect that allows for a direct comparison with Coq and Agda is the handling of implicit arguments. In both Agda and Coq, abstractions corresponding to arguments the user can freely omit are statically labelled as such. The systems automatically generate fresh metavariables as arguments to these binders and the type inference algorithm eventually instantiates them. Both systems give the user the possibility to locally override the implicit arguments mechanism. In Coq the user can prefix the name of a constant with the @ symbol, while in Agda the user can mark actual arguments as implicit enclosing them in curly braces. This escaping mechanism is required because many lemmas admit multiple and incompatible lists of implicit arguments.

As an example, consider a transitivity lemma eqt : $\forall x, y, z.x = y \to y = z \to x = z$. When used in a forward proof step the user is likely to pass as arguments a proof p that $a = b$ and a proof q that $b = c$ like in (eqt p q) to put in his context the additional fact $a = c$. In that case values for x, y and z are determined by the types of p and q. On the contrary if the lemma is used in backward proof step to prove that $a = c$, no value for c can be inferred, thus the user is likely to use the lemma as in (eqt b) and expect the system to open two new goals: $a = b$ and $b = c$. The two different uses of eqt make it impossible to statically attach to it a single list of implicit arguments and at the same time to never resort to an escape mechanism to temporarily forget that list.

In Matita the user can simply use the $\overrightarrow{?}$ placeholder, thus no escaping mechanism is required. In fact the type inference algorithm described in this paper lets the user write (eqt $\overrightarrow{?}$ p q) in the first case as well as (eqt $\overrightarrow{?}$ c) in the second one.[7]

The lack of a complete and formal study of type inference for raw CIC terms is probably due to the many peculiarities of the CIC type system, in particular inductive and dependent types, explicit polymorphism and the fact that type comparison is not structural, but up to computational equivalence. We thus try to position our work with respect to some of the main approaches adopted by type inference algorithms designed for programming languages.

7.1. **Greedy versus delayed constraint solving.** The most notable example of type inference algorithm based on constraint solving is the one adopted for the Agda system [22]. Agda is based on a dependently typed programming language quite similar to CIC, but is designed for programming and not for writing proofs. The type inference algorithm collects constraints and checks for their satisfiability. Nevertheless, their solution is not recorded in the terms. This enables the user to remove an arbitrary part of an already type checked term and have the typing of its context not influenced by the term just removed. While this "compositionality" property is desirable for programming in a language with dependent types, it is not vital for proof systems, where one seldom edits by hand type checked terms.

---

[7]A trailing $\overrightarrow{?}$ is automatically added to any term used in backward proof step.

A strong characteristic of constraint based type inference is precise error reporting, as described in [33]. Even if it the heuristics adopted in Matita [27] to discard spurious error reports are slightly more complex than the ones proposed by Stuckey, we believe that they provide a similar precision.

Greedy algorithms [14], like the one presented in this paper, are characterized by a very predictable behavior, at the cost of being forced to take early decisions leading to the rejection of some possibly well typed terms. Also remember that unification has to take computation into account, and user provided functions are known to be total only if they are well typed. Thus the resolution of type constraints cannot be delayed for long. According to our experience, predictability compensates for the extra type annotations the user is sometimes required to produce to drive the greedy algorithm towards a solution.

7.2. **Unification based versus local constraint solving.** Many algorithms to infer a polymorphic type for a program prefer to avoid the use of unification [24] since unification variables may represent type constraints coming from distant, loosely related, sub-terms. Moreover a bi-directional approach pushes the type constraints of the context towards sub-terms, making it effectively possible to drop unification altogether. These approaches also scaled up to types with some sort of dependency over terms, as in [15].

Interactive provers based on type theory are for (good and) historical reasons based on the two twin approaches. A small kernel based on decision procedures type checks (place-holder free) terms, and a refiner based on heuristics deals with terms with holes performing type inference. Since the kernel is the key component of the system, the one that must be trusted, the language is designed to allow the type checking algorithm to be as simple as possible. Explicit polymorphism makes type checking CIC terms decidable while allowing the same degree of polymorphism as $\mathcal{F}^\omega$. These explicit type annotations are usually left implicit by the user and represent long-distance constraints. In this context unification seems to be a necessary device. Moreover, the most characterizing feature of CIC is that types are compared taking computation into account, and that types can contain terms, in particular functions applications. Thus the kernel is equipped with a quite elaborate machinery to compute recursive functions and unfold definitions. Type inference has to provide a similar machinery, and possibly extend it to handle types containing metavariables. This extension is commonly named higher order unification, and it is a really critical component of an interactive prover. Recent important developments [16] heavily rely on a user-extensible unification algorithm [5], using it as a predictable form of Prolog-like inference engine. In other words, unification can be employed to infer terms (content) while type inference is employed to infer types and type annotations in the case of explicit polymorphism. For these many reasons, we believe that developing type inference on top of unification is a sound decision probably necessary to scale to a rich type system like CIC.

8. Conclusion

In this paper we studied the design of an effective refinement algorithm for the Calculus of (Co)Inductive Constructions. Its effectiveness has been validated in all the formalizations carried on using the Matita interactive theorem prover [11, 1, 2, 3], whose refiner is based on the algorithm described in this paper. Once again we stress that the refiner component, while not being critical for the correctness of the prover, is the user's main interlocutor, and is thus critical for the overall user's experience.

This algorithm is also the result of the complete rewrite the Matita ITP underwent in the last couple of years. The refiner algorithm described in this paper amounts to approximatively 1600 lines of OCaml code, calling the higher order unification algorithm that amounts to a bit less than 1900 lines. To give a term of comparison to the reader, the kernel of Matita, written by the same authors, amounts to 1500 lines of data structures definitions and basic operations on them, 550 lines of conversion algorithm and 1400 of type checking. More than 300 lines of the type checking algorithm are reused by the refiner for checking inductive types positivity conditions and recursive or co-recursive functions termination or productivity.

On top of this refinement algorithm all primitive proof commands have been reimplemented. In the old implementation they were not taking full advantage of the refiner, partially for historical reasons, partially because it was lacking support for placeholder vectors and bi-directionality was not always exploited. The size of the code is now 48.9% of what it used to be in the former implementation. In particular, it became possible to implement many proof commands as simple "notations" for lambda-terms in external syntax.

A particularity of this work is that the presented algorithm deals with completely raw terms, containing untyped placeholders, whose only precondition is to be syntactically well formed. In addition it also supports a very general form of coercive sub-typing, where inserting the explicit cast may leave uninstantiated metavariables to be later filled by the user. This eased the implementation of subset coercions in the style of [29], but that topic falls outside the scope of the present paper and is thus not discussed.

The algorithm could be enhanced adding more rules, capable of propagating more typing information. For instance, a specific type forcing rule for β-redexes (suggested by a referee) could be in the form

$$
(\overset{\mathcal{R}^{\Downarrow}}{\leadsto}-\text{beta}) \quad \frac{\begin{array}{l} \Gamma \vdash U \overset{\mathcal{F}}{\leadsto} U' : s \\ \Gamma \vdash u : U' \overset{\mathcal{R}^{\Downarrow}}{\leadsto} u' \\ \Gamma; x : U' \vdash t : E[u'/x] \overset{\mathcal{R}^{\Downarrow}}{\leadsto} t' \end{array}}{\Gamma \vdash (\lambda x : U.t)\ u : E \overset{\mathcal{R}^{\Downarrow}}{\leadsto} (\lambda x : U'.t')\ u'}
$$

enabling the system to propagate the expected type to the abstraction (compare this rule with $(\mathcal{R}^{\Downarrow}-\text{letin})$). In practice the advantage of this rule is limited, since it is quite infrequent for a user to write β-redexes. A type forcing rule for pattern matching based on the same principles, propagating the expected type to the return type of the **match**, could be of greater value, since this construct is more likely to come from the user input. We will consider adding such rules in a future implementation.

The refinement algorithm we presented already validates many desired properties, like correctness and termination. Nevertheless we did not even state the relative completeness theorem. In a simpler framework, admitting most general unifiers, one could have stated that given an oracle for unification, the algorithm outputs a well typed refinement every time it is possible and that any other refinement is less general than the produced one. Unluckily CIC is higher order and does not admit most general unifiers. To state the relative completeness theorem one has to make the oracle aware of the whole refinement procedure and the oracle has to guess a unifier (or all of them) such that the remaining refinement steps succeed. This makes the theorem way less interesting. Alternatively one would have to add backtracking to compensate for errors made by the oracle, and make the

algorithm distant from the implemented one, that is essentially greedy and backtracking free.

The algorithm presented in the paper is clearly not relatively complete. For example, the rules given in Section 3 do not accept the term $f\ c$ where $c : \mathbb{N}$ and $f :$ **match** $?_1$ **in** $\mathbb{N}$ **return** $\lambda x.$**Type** $[O \Rightarrow \mathbb{N} \mid S\ (x : \mathbb{N}) \Rightarrow \mathbb{N} \rightarrow \mathbb{N}]$. The term is however refineable, for instance by instantiating $?_1$ with $S\ O$. To obtain a relatively complete algorithm, we could add additional rules based on the invocation of the unifier on difficult problems. For instance, for the example just shown it would be sufficient to unify $\mathbb{N} \rightarrow ?_2$ with **match** $?_1$ **in** $\mathbb{N}$ **return** $\lambda x.$**Type** $[O \Rightarrow \mathbb{N} \mid S\ (x : \mathbb{N}) \Rightarrow \mathbb{N} \rightarrow \mathbb{N}]$ for a fresh metavariable $?_2$. However, we know in advance that the efficient but incomplete algorithm implemented in Matita always fails on such difficult unification problems. The same holds for the similar algorithm implemented in Coq. Therefore a relatively complete version of the algorithm would remain only of theoretical interest.

The following weaker theorem, which establishes completeness on well typed terms only, can be easily proved by recursion over the proof tree and by inspection of all cases under the hypothesis that every pair of convertible terms are unified by the identity metavariable instantiation.

Theorem 8.1 (Completeness for well typed terms). *For all well formed* Σ, Φ, Γ *and for all* t *and* T *such that* Σ, Φ, $\Gamma \vdash t : T$ *we have* $\Gamma \vdash t : T \overset{\mathcal{R}^{\Downarrow}}{\rightsquigarrow} t$. $\qquad\square$

Appendix A. Syntax-directed type-checking rules

The following appendix is an extract of the paper [4] in which the reader can find all the details of the type checking algorithm implemented in the Matita interactive prover. A few aesthetic changes have been made to the adopted syntax to increase its consistency with respect to the syntax adopted in this paper. The main differences are summarised in the following list:

- We use the membership relation over the PTS set to type sorts and products
- The check for the consistency of the metavariable local substitution has been inlined in the rule
- A new generic judgement $(r : T) \in \mathsf{Env}$ has been introduced to provide a more compact syntax for the lookup of the type of a generic object into the environment
- We inlined several auxiliary functions that were used in the presentation of the typechecking rule for case analysis.

  This was made possible by the following abuse of notation: $\mathsf{Env}, \Sigma, \Phi, \emptyset \vdash t_1 \triangleright_{\mathrm{whd}} \overrightarrow{\Pi x_i : T_i}.t_{n+1}$ is a shortcut to mean that for all $i \in \{1 \ldots n\}$ Σ, Φ, $\Gamma; x_1 : T_1; \ldots; x_{i-1} : T_{i-1} \vdash t_i \triangleright_{\mathrm{whd}} \Pi x_i : T_i.t_{i+1}$ and Σ, Φ, $\Gamma \vdash t_{n+1} \triangleright_{\mathrm{whd}} t_{n+1}$.

  Moreover, the rule presented in [4] is more liberal than the one presented here that just uses the test $(s, s') \in \mathsf{elim}(\mathsf{PTS})$ to check that a non informative data is never analyzed to obtain an informative one. The actual rules used in the kernel and the refiner of Matita also allow in every situation the elimination of inhabitants of singleton inductive types, whose definition is given in [4].

In this section, $\mathcal{I}$ will be short for

$$
\begin{aligned}
&\Pi \overrightarrow{x_l : U_l}.\textbf{inductive} \\
&\qquad I_l^1 : A_1 := k_1^1 : K_1^1 \ldots k_1^{m_1} : K_1^{m_1} \\
&\quad \textbf{with} \ \ldots \\
&\quad \textbf{with} \ I_l^n : A_n := k_n^1 : K_n^1 \ldots k_n^{m_n} : K_n^{m_n}
\end{aligned}
$$

A.1. **Environment formation rules.** Environment formation rules (judgement $\mathsf{Env} \vdash WF$, function typecheck_obj)

$$
\overline{\emptyset \vdash WF}
$$

$$
\frac{
\begin{array}{l}
\mathsf{Env} \vdash WF \quad d \text{ undefined in } \mathsf{Env} \quad \mathsf{Env}, \Sigma \vdash WF \quad \mathsf{Env}, \Sigma, \Phi \vdash WF \\
\mathsf{Env}, \Sigma, \Phi, \emptyset \vdash T : S \quad \mathsf{Env}, \Sigma, \Phi, \emptyset \vdash S \triangleright_{\mathrm{whd}} S' \text{ where } S' \text{ is a sort} \\
\mathsf{Env}, \Sigma, \Phi, \emptyset \vdash b : T' \quad \mathsf{Env}, \Sigma, \Phi, \emptyset \vdash T \downarrow T'
\end{array}
}{
\mathsf{Env} \cup (\Sigma, \Phi, \textbf{definition} \ d : T := b) \vdash WF
}
$$

$$
\frac{
\begin{array}{l}
\mathsf{Env} \vdash WF \quad d \text{ undefined in } \mathsf{Env} \quad \mathsf{Env}, \Sigma \vdash WF \quad \mathsf{Env}, \Sigma, \Phi \vdash WF \\
\mathsf{Env}, \Sigma, \Phi, \emptyset \vdash T : S \quad \mathsf{Env}, \Sigma, \Phi, \emptyset \vdash S \triangleright_{\mathrm{whd}} S' \text{ where } S' \text{ is a sort}
\end{array}
}{
\mathsf{Env} \cup (\Sigma, \Phi, \textbf{axiom} \ d : T) \vdash WF
}
$$

$$\mathsf{Env} \vdash WF \quad \overrightarrow{f_n} \text{ undefined in } \mathsf{Env} \quad \mathsf{Env}, \Sigma \vdash WF \quad \mathsf{Env}, \Sigma, \Phi \vdash WF$$

$$\mathsf{Env}, \Sigma, \Phi, \emptyset \vdash T_i : S_i \quad \mathsf{Env}, \Sigma, \Phi, \emptyset \vdash S_i \rhd_{\mathrm{whd}} S_i' \text{ where } S_i' \text{ is a sort}$$

$$T_i = \Pi \overrightarrow{x_{p_i}^i : T_{p_i}^i}.T_{p_i+1}^i$$

$$\left. \begin{array}{l} \mathsf{Env}, \Sigma, \Phi, [f_1 : T_1; \ldots; f_n : T_n; \overrightarrow{x_{p_i}^i : T_{p_i}^i}] \vdash t_i : T'^i_{p_i+1} \\ \mathsf{Env}, \Sigma, \Phi, [f_1 : T_1; \ldots; f_n : T_n; \overrightarrow{x_{p_i}^i : T_{p_i}^i}] \vdash T_{p_i+1}^i \downarrow T'^i_{p_i+1} \end{array} \right\} i \in \{1 \ldots n\}$$

$$\overrightarrow{t_n} \text{ guarded by destructors ([4], Sect. 6.3)}$$

$$\rule{10cm}{0.4pt}$$

$$\mathsf{Env} \cup \left(\begin{array}{l} \Sigma, \Phi, \\ \textbf{let rec } f_1(\overrightarrow{x_{p_1}^1 : T_{p_1}^1}) : T_{p_1+1}^1 := t_1 \textbf{ and } \ldots \\ \textbf{and } f_n(\overrightarrow{x_{p_n}^n : T_{p_n}^n}) : T_{p_n+1}^n := t_n \end{array} \right) \vdash WF$$

$$\mathsf{Env} \vdash WF \quad \overrightarrow{f_n} \text{ undefined in } \mathsf{Env} \quad \mathsf{Env}, \Sigma \vdash WF \quad \mathsf{Env}, \Sigma, \Phi \vdash WF$$

$$\mathsf{Env}, \Sigma, \Phi, \emptyset \vdash T_i : S_i \quad \mathsf{Env}, \Sigma, \Phi, \emptyset \vdash S_i \rhd_{\mathrm{whd}} S_i' \text{ where } S_i' \text{ is a sort}$$

$$T_i = \Pi \overrightarrow{x_{p_i}^i : T_{p_i}^i}.T_{p_i+1}^i$$

$$\left. \begin{array}{l} \mathsf{Env}, \Sigma, \Phi, [f_1 : T_1; \ldots; f_n : T_n; \overrightarrow{x_{p_i}^i : T_{p_i}^i}] \vdash t_i : T'^i_{p_i+1} \\ \mathsf{Env}, \Sigma, \Phi, [f_1 : T_1; \ldots; f_n : T_n; \overrightarrow{x_{p_i}^i : T_{p_i}^i}] \vdash T_{p_i+1}^i \downarrow T'^i_{p_i+1} \end{array} \right\} i \in \{1 \ldots n\}$$

$$\overrightarrow{t_n} \text{ guarded by constructors ([4], Sect. 6.3)}$$

$$\rule{10cm}{0.4pt}$$

$$\mathsf{Env} \cup \left(\begin{array}{l} \Sigma, \Phi, \\ \textbf{let corec } f_1(\overrightarrow{x_{p_1}^1 : T_{p_1}^1}) : T_{p_1+1}^1 := t_1 \textbf{ and } \ldots \\ \textbf{and } f_n(\overrightarrow{x_{p_n}^n : T_{p_n}^n}) : T_{p_n+1}^n := t_n \end{array} \right) \vdash WF$$

$$\mathsf{Env} \vdash WF \quad I_l^1, \ldots, I_l^n, k_1^1, \ldots, k_n^{m_n} \text{ undefined in } \mathsf{Env} \quad \mathsf{Env}, \Sigma \vdash WF \quad \mathsf{Env}, \Sigma, \Phi \vdash WF$$

$$\text{all the conditions in [4], Sect. 6.1 are satisfied}$$

$$\rule{10cm}{0.4pt}$$

$$\mathsf{Env} \cup (\Phi, \Sigma, \mathcal{I}) \vdash WF$$

A.2. **Metasenv formation rules.** Metasenv formation rules (judgement $\mathsf{Env}, \Sigma \vdash WF$, function typecheck_metasenv)

$$\overline{\mathsf{Env}, \emptyset \vdash WF}$$

$$\mathsf{Env}, \Sigma \vdash WF \quad ?_i \text{ undefined in } \Sigma \quad \mathsf{Env}, \Sigma, \emptyset, \Gamma \vdash WF$$

$$\frac{\mathsf{Env}, \Sigma, \emptyset, \Gamma \vdash T : S \quad \mathsf{Env}, \Sigma, \emptyset, \Gamma \vdash S \rhd_{\mathrm{whd}} S' \text{where } S' \text{ is a sort}}{\mathsf{Env}, \Sigma \cup (\Gamma \vdash ?_i : T) \vdash WF}$$

A.3. **Subst formation rules.** Subst formation rules (judgement $\mathsf{Env}, \Sigma, \Phi \vdash WF$, function typecheck_subst)

$$\overline{\mathsf{Env}, \Sigma, \emptyset \vdash WF}$$

$$\frac{\begin{array}{c} \mathsf{Env}, \Sigma, \Phi \vdash WF \quad ?_i \text{ undefined in } \Sigma \text{ and in } \Phi \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash WF \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T : S \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash S \triangleright_{\mathrm{whd}} S' \text{ where } S' \text{ is a sort} \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash t : T' \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T \downarrow T' \end{array}}{\mathsf{Env}, \Sigma, \Phi \cup (\Gamma \vdash ?_i : T := t) \vdash WF}$$

A.4. Context formation rules.
Context formation rules (judgement $\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash WF$, function typecheck_context)

$$\overline{\mathsf{Env}, \Sigma, \Phi, \emptyset \vdash WF}$$

$$\frac{\begin{array}{c} \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash WF \quad x \text{ is undefined in } \Gamma \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T : S \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash S \triangleright_{\mathrm{whd}} S' \text{ where } S' \text{ is a sort} \end{array}}{\mathsf{Env}, \Sigma, \Phi, \Gamma \cup (x : T) \vdash WF}$$

$$\frac{\begin{array}{c} \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash WF \quad x \text{ is undefined in } \Gamma \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T : S \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash S \triangleright_{\mathrm{whd}} S' \text{ where } S' \text{ is a sort} \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash t : T' \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T \downarrow T' \end{array}}{\mathsf{Env}, \Sigma, \Phi, \Gamma \cup (x : T := t) \vdash WF}$$

A.5. Term typechecking rules.
Term typechecking rules (judgement $\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash t : T$, function typeof)

$$(\mathcal{K}-\text{variable}) \quad \frac{(x : T) \in \Gamma \quad \text{or} \quad (x : T := t) \in \Gamma}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash x : T} \qquad (\mathcal{K}-\text{sort}) \quad \frac{(s_1, s_2) \in \mathsf{PTS}}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash s_1 : s_2}$$

$$(\mathcal{K}-\text{meta}) \quad \frac{\begin{array}{c} (x_1 : T_1; \ldots; x_n : T_n \vdash ?_i : T) \in \Sigma \quad \text{or} \quad (x_1 : T_1; \ldots; x_n : T_n \vdash ?_i : T := t) \in \Phi \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash t_i : T_i\overrightarrow{[x_{i-1}/t_{i-1}]} \quad i \in \{1 \ldots n\} \end{array}}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash ?_i[t_1; \ldots; t_n] : T\overrightarrow{[x_n/t_n]}}$$

$$(\mathcal{K}-\text{constant}) \quad \frac{(r : T) \in \mathsf{Env}}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash r : T}$$

$$(\mathcal{K}-\text{definition}) \quad \frac{\begin{array}{c} (\Sigma', \Phi', \textbf{definition } d : T := b) \in \mathsf{Env} \quad \text{or} \quad (\Sigma', \Phi', \textbf{axiom } d : T) \in \mathsf{Env} \\ \Sigma' = \emptyset \quad \Phi' = \emptyset \end{array}}{(d : T) \in \mathsf{Env}}$$

$$(\mathcal{K}-\text{letrec}) \quad \frac{\begin{pmatrix} \Sigma', \Phi', \\ \textbf{let rec } f_1(\overrightarrow{x_{p_1}^1 : T_{p_1}^1}) : T_{p_1+1}^1 := t_1 \textbf{ and } \ldots \\ \textbf{and } f_n(\overrightarrow{x_{p_n}^n : T_{p_n}^n}) : T_{p_n+1}^n := t_n \end{pmatrix} \in \mathsf{Env} \\ \Sigma' = \emptyset \quad \Phi' = \emptyset \quad 1 \le i \le n}{(f_i : \Pi\overrightarrow{x_{p_i}^i : T_{p_i}^i}.T_{p_i+1}^i) \in \mathsf{Env}}$$

$(\mathcal{K}-\text{letcorec})$
$$\dfrac{\begin{pmatrix} \Sigma', \Phi', \\ \textbf{let } \textbf{corec } f_1(\overrightarrow{x^1_{p_1} : T^1_{p_1}}) : T^1_{p_1+1} := t_1 \textbf{ and } \dots \\ \textbf{and } f_n(\overrightarrow{x^n_{p_n} : T^n_{p_n}}) : T^n_{p_n+1} := t_n \end{pmatrix} \in \mathsf{Env} \quad \Sigma' = \emptyset \quad \Phi' = \emptyset \quad 1 \le i \le n}{(f_i : T_i) \in \mathsf{Env}}$$

$(\mathcal{K}-\text{inductive})$
$$\dfrac{(\Sigma', \Phi', \mathcal{I}) \in \mathsf{Env} \quad \Sigma' = \emptyset \quad \Phi' = \emptyset \quad 1 \le p \le n}{(I^p_l : \Pi\overrightarrow{x_l : U_l}.A_p) \in \mathsf{Env}}$$

$(\mathcal{K}-\text{constructor})$
$$\dfrac{(\Sigma', \Phi', \mathcal{I}) \in \mathsf{Env} \quad \Sigma' = \emptyset \quad \Phi' = \emptyset \quad 1 \le p \le n \quad 1 \le j \le m_p}{(k^j_p : \Pi\overrightarrow{x_l : U_l}.K^j_p) \in \mathsf{Env}}$$

$(\mathcal{K}-\text{lambda})$
$$\dfrac{\begin{array}{c} \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T : S \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash S \rhd_{\mathrm{whd}} S' \quad S' \text{ is a sort or a meta} \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \cup (n : T) \vdash u : U \end{array}}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \lambda n : T.u : \Pi n : T.U}$$

$(\mathcal{K}-\text{product})$
$$\dfrac{\begin{array}{c} \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T : s_1 \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \cup (n : T) \vdash U : s_2 \\ (s_1, s_2, s_3) \in \mathsf{PTS} \end{array}}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \Pi n : T.U : s_3}$$

$(\mathcal{K}-\text{letin})$
$$\dfrac{\begin{array}{c} \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash t : T' \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T : S \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T \downarrow T' \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \cup (x : T := t) \vdash u : U \end{array}}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \textbf{let } (x : T) := t \textbf{ in } u : U[x/t]}$$

$(\mathcal{K}-\text{appl} - \text{base})$
$$\dfrac{\begin{array}{c} \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash h : \Pi x : T.U \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash t : T' \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T \downarrow T' \end{array}}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash h \ t : U[x/t]}$$

$(\mathcal{K}-\text{appl} - \text{rec})$
$$\dfrac{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash (h \ t_1) \ t_2 \cdots t_n : T}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash h \ t_1 \ t_2 \cdots t_n : T}$$

$(\mathcal{K}-\text{match})$
$$\dfrac{\begin{array}{c} (\Sigma', \Phi', \mathcal{I}) \in \mathsf{Env} \quad \Sigma' = \emptyset \quad \Phi' = \emptyset \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash t : T \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T \rhd_{\mathrm{whd}} I^p_l \ \overrightarrow{u_l} \ \overrightarrow{u_r} \\ A_p[\overrightarrow{x_l/u_l}] = \Pi\overrightarrow{y_r : Y_r}.s \quad K^j_p[\overrightarrow{x_l/u_l}] = \Pi\overrightarrow{x^j_{n_j} : Q^j_{n_j}}.I^p_l \ \overrightarrow{x_l} \ \overrightarrow{v_r} \qquad j = 1 \dots m_p \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash U : V \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash V \rhd_{\mathrm{whd}} \Pi\overrightarrow{z_r : Y_r}.\Pi z_{r+1} : I^p_l \ \overrightarrow{u_l} \ \overrightarrow{z_r}.s' \\ (s, s') \in \mathsf{elim}(\mathsf{PTS}) \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \lambda \overrightarrow{x^j_{n_j} : P^j_{n_j}}.t_j : T_j \qquad\qquad\qquad j = 1, \dots, m_p \\ \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T_j \downarrow \Pi\overrightarrow{x^j_{n_j} : Q^j_{n_j}}.U \ \overrightarrow{v_r} \ (k^p_j \ \overrightarrow{u_l} \ \overrightarrow{x^j_{n_j}}) \qquad\qquad j = 1, \dots, m_p \end{array}}{\begin{array}{c} \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \quad \textbf{match } t \textbf{ in } I^p_l \textbf{ return } U \\ [k^p_1 \ (\overrightarrow{x^1_{n_1} : P^1_{n_1}}) \Rightarrow t_1 \ | \dots \ |k^p_{m_p} \ (\overrightarrow{x^{m_p}_{n_{m_p}} : P^{m_p}_{n_{m_p}}}) \Rightarrow t_{m_p} \ ] : U \ \overrightarrow{u_r} \ t \end{array}}$$

A.6. **Term conversion rules.** Term conversion rules (judgement $\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T \downarrow T'$, function $\mathsf{are\_convertible}$; $\downarrow_=$ means $\mathsf{test\_eq\_only} = \mathsf{true}$; $\downarrow_\bullet$ means that the current rule must be intended as two rules, one with all the $\downarrow_\bullet$ replaced by $\downarrow$, the other with all the $\downarrow_\bullet$ replaced by $\downarrow_=$)

$$\frac{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T =_\alpha T'}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T \downarrow_= T'}$$

$$\frac{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T \downarrow_= T'}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T \downarrow T'}$$

$$\frac{\mathbf{Type}_u \leq \mathbf{Type}_v \quad \mathbf{Type}_v \leq \mathbf{Type}_u \quad \text{are declared constraints ([4], Sect. 4.3)}}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \mathbf{Type}_u \downarrow_= \mathbf{Type}_v}$$

$$\frac{\mathbf{Type}_u \leq \mathbf{Type}_v \quad \text{is a declared constraint ([4], Sect. 4.3)}}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \mathbf{Type}_u \downarrow \mathbf{Type}_v}$$

$$\frac{}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \mathbf{Prop} \downarrow \mathbf{Type}_u}$$

$$\frac{lc = t_1, \ldots, t_n \quad lc' = t'_1, \ldots, t'_n}{\mathsf{for \ all} \ i = 1, \ldots, n \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash t_i \downarrow_\bullet t'_i}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash ?_j[lc] \downarrow_\bullet ?_j[lc']}$$

$$\frac{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash T_1 \downarrow_= T'_1 \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \cup (x : T_1) \vdash T_2 \downarrow_\bullet T'_2}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \Pi x : T_1.T_2 \downarrow_\bullet \Pi x : T'_1.T'_2}$$

$$\frac{\mathsf{Env}, \Sigma, \Phi, \Gamma \cup (x : T) \vdash t \downarrow_\bullet t'}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \lambda x : T.t \downarrow_\bullet \lambda x : T'.t'}$$

In the rule above, no check is performed on the source of the abstractions, since we assume we are comparing well-typed terms whose types are convertible.

$$\frac{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash h \downarrow_\bullet h'}{\mathsf{for \ all} \ i = 1, \ldots, n \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash t_i \downarrow_= t'_i}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash h \overrightarrow{t_n} \downarrow_\bullet h' \overrightarrow{t'_n}}$$

$$\frac{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash t \downarrow_\bullet t' \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash U \downarrow_\bullet U'}{\mathsf{for \ all} \ i = 1, \ldots, m_p \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \lambda \overrightarrow{x^i_{n_i} : P^i_{n_i}}.t_i \downarrow_\bullet \lambda \overrightarrow{x^i_{n_i} : P'^i_{n_i}}.t'_i}{\begin{array}{c}\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \quad \mathbf{match} \ t \ \mathbf{in} \ I^p_l \ \mathbf{return} \ U \ [k^p_1 \ (\overrightarrow{x^1_{n_1} : P^1_{n_1}}) \Rightarrow t_1 | \ldots | k^p_{m_p} \ (\overrightarrow{x^{m_p}_{n_{m_p}} : P^{m_p}_{n_{m_p}}}) \Rightarrow t_{m_p}] \ \downarrow_\bullet \\ \mathbf{match} \ t' \ \mathbf{in} \ I^p_l \ \mathbf{return} \ U \ [k^p_1 \ (\overrightarrow{x^1_{n_1} : P'^1_{n_1}}) \Rightarrow t'_1 | \ldots | k^p_{m_p} \ (\overrightarrow{x^{m_p}_{n_{m_p}} : P'^{m_p}_{n_{m_p}}}) \Rightarrow t'_{m_p}]\end{array}}$$

$$\frac{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash t \triangleright_{\mathrm{whd}} t' \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash u \triangleright_{\mathrm{whd}} u' \quad \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash t' \downarrow_\bullet u'}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash t \downarrow_\bullet u}$$

In the previous rule, t' and u' need not be weak head normal forms: any term obtained from t (respectively, u) by reduction (even non-head reduction) will do. Indeed, the less reduction is performed, the more efficient the conversion test usually is.

A.7. **Term reduction rules.** Term reduction rules.

$$\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash (\lambda x : T.u)\ t\ \triangleright_\beta\ u[x/t]$$

$$\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash \mathbf{let}\ (x : T) := t\ \mathbf{in}\ u\ \triangleright_\zeta\ u[x/t]$$

$$\frac{(\emptyset, \emptyset, \mathbf{definition}\ d : T := b) \in \mathsf{Env}}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash d\ \triangleright_\delta\ b}$$

$$\frac{(\Gamma' \vdash ?_i : T := t) \in \Phi}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash ?_i[u_1\ ;\ \dots\ ;\ u_n]\ \triangleright_\delta\ t[\overrightarrow{dom(\Gamma')/\overrightarrow{u_n}}]}$$

$$\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash\ \mathbf{match}\ k_i^p\ \overrightarrow{t_l}\ \overrightarrow{t'_{n_i}}\ \mathbf{in}\ I_l^p\ \mathbf{return}\ U$$
$$[k_1^p\ (\overrightarrow{x_{n_1}^1 : P_{n_1}^1}) \Rightarrow u_1| \dots |k_{m_p}^p\ (\overrightarrow{x_{n_{m_p}}^{m_p} : P_{n_{m_p}}^{m_p}}) \Rightarrow u_{m_p}]\ \triangleright_\iota\ u_i[\overrightarrow{x_{n_i}^i}/\overrightarrow{t'_{n_i}}]$$

$$\frac{\left(\begin{array}{l} \emptyset, \emptyset, \\ \mathbf{let\ rec}\ f_1(\overrightarrow{x_{p_1}^1 : T_{p_1}^1}) : T_{p_1+1}^1 := t_1\ \mathbf{and}\ \dots \\ \quad \mathbf{and}\ f_n(\overrightarrow{x_{p_n}^n : T_{p_n}^n}) : T_{p_n+1}^n := t_n \end{array}\right) \in \mathsf{Env} \quad k \in \{1 \dots n\}}{\mathsf{Env}, \Sigma, \Phi, \Gamma \vdash f_k\ u_1\ \dots(k_j^i\ \overrightarrow{v_{n_j}})\dots u_m\ \triangleright_\mu\ t_k[\overrightarrow{x_m^k}/u_1, \dots, (k_j^i\ \overrightarrow{v_{n_j}}), \dots, u_m]}$$

Notice that $(k_j^i\ \overrightarrow{v_{n_j}})$ must occur in the position of the recursive argument of f_k. This implies that, for this reduction to be performed, f_k must be applied at least up to its recursive argument.

$$\frac{\left(\begin{array}{l} \emptyset, \emptyset, \\ \mathbf{let\ corec}\ f_1(\overrightarrow{x_{p_1}^1 : T_{p_1}^1}) : T_{p_1+1}^1 := t_1\ \mathbf{and}\ \dots \\ \quad \mathbf{and}\ f_n(\overrightarrow{x_{p_n}^n : T_{p_n}^n}) : T_{p_n+1}^n := t_n \end{array}\right) \in \mathsf{Env} \quad k \in \{1 \dots n\}}{\begin{array}{l} \mathsf{Env}, \Sigma, \Phi, \Gamma \vdash\ \mathbf{match}\ f_k\ \overrightarrow{u_q}\ \mathbf{in}\ I_l^p\ \mathbf{return}\ U \\ \qquad [k_1^p\ (\overrightarrow{y_{n_1}^1 : P_{n_1}^1}) \Rightarrow v_1| \dots |k_{m_p}^p\ (\overrightarrow{y_{n_{m_p}}^{m_p} : P_{n_{m_p}}^{m_p}}) \Rightarrow v_{m_p}]\ \triangleright_\nu \\ \qquad \mathbf{match}\ t_k[\overrightarrow{x_q^k}/\overrightarrow{u_q}]\ \mathbf{in}\ I_l^p\ \mathbf{return}\ U \\ \qquad [k_1^p\ (\overrightarrow{y_{n_1}^1 : P_{n_1}^1}) \Rightarrow v_1| \dots |k_{m_p}^p\ (\overrightarrow{y_{n_{m_p}}^{m_p} : P_{n_{m_p}}^{m_p}}) \Rightarrow v_{m_p}] \end{array}}$$

Notice that here q can be zero.

References

[1] Andrea Asperti and Cristian Armentano. A page in number theory. *Journal of Formalized Reasoning*, 1:1–23, 2008.

[2] Andrea Asperti and Wilmer Ricciotti. About the formalization of some results by Chebyshev in number theory. In *Proc. of TYPES'08*, volume 5497 of *LNCS*, pages 19–31. Springer-Verlag, 2009.

[3] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Formal metatheory of programming languages in the Matita interactive theorem prover. *Journal of Automated Reasoning: Special Issue on the Poplmark Challenge*. Published online, May 2011.

[4] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A compact kernel for the Calculus of Inductive Constructions. *Sadhana*, 34(1):71–144, 2009.

[5] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. Hints in unification. In *TPHOLs 2009*, volume 5674/2009 of *LNCS*, pages 84–98. Springer-Verlag, 2009.

[6] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. User interaction with the Matita proof assistant. *Journal of Automated Reasoning*, 39(2):109–139, 2007.

[7] Henk Barendregt. Lambda Calculi with Types. In Abramsky, Samson and others, editor, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.

[8] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - a functional language with dependent types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.

[9] P. Callaghan. Coherence checking of coercions in Plastic. In *In Proc. Workshop on Subtyping and Dependent Types in Programming*, 2000.

[10] Gang Chen. *Subtyping, Type Conversion and Transitivity Elimination*. PhD thesis, University Paris 7, 1998.

[11] Claudio Sacerdoti Coen and Enrico Tassi. Formalizing Overlap Algebras in Matita. *Mathematical Structures in Computer Science*, 21:1–31, 2011.

[12] The Coq proof-assistant.
`http://coq.inria.fr`, 2009.

[13] Thierry Coquand and Gérard P. Huet. The Calculus of Constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[14] Joshua Dunfield. Greedy bidirectional polymorphism. In *ML Workshop (ML '09)*, pages 15–26, August 2009. `http://www.cs.cmu.edu/~joshuad/papers/poly/`.

[15] Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In X. Leroy, editor, *Conference Record of the 31st Annual Symposium on Principles of Programming Languages (POPL'04)*, pages 281–292, 2004.

[16] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 327–342, Berlin, Heidelberg, 2009. Springer-Verlag.

[17] Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.

[18] The Isabelle proof-assistant.
`http://www.cl.cam.ac.uk/Research/HVG/Isabelle/`.

[19] The Lego proof-assistant.
`http://www.dcs.ed.ac.uk/home/lego/`.

[20] Zhaohui Luo. Coercive subtyping. *J. Logic and Computation*, 9(1):105–130, 1999.

[21] César Muñoz. *A Calculus of Substitutions for Incomplete-Proof Representation in Type Theory*. PhD thesis, INRIA, November 1997.

[22] 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, September 2007.

[23] Christine Paulin-Mohring. *Définitions Inductives en Théorie des Types d'Ordre Supérieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, December 1996.

[24] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22:1–44, January 2000.

[25] Robert Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002.

[26] Claudio Sacerdoti Coen. *Mathematical Knowledge Management and Interactive Theorem Proving*. PhD thesis, University of Bologna, 2004. Technical Report UBLCS 2004-5.

[27] Claudio Sacerdoti Coen and Stefano Zacchiroli. Spurious disambiguation errors and how to get rid of them. *Journal of Mathematics in Computer Science, special issue on Management of Mathematical Knowledge*, 2:355–378, 2008.

[28] Amokrane Saïbi. Typing algorithm in type theory with inheritance. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 292–301, New York, NY, USA, 1997. ACM.

[29] Matthieu Sozeau. Subset coercions in Coq. In *Types for Proofs and Programs*, volume 4502/2007 of *LNCS*, pages 237–252. Springer-Verlag, 2006.

[30] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Proceedings of TPHOLs*, pages 278–293, 2008.

[31] Arnaud Spiwack. *Verified Computing in Homological Algebra. A Journey Exploring the Power and Limits of Dependent Type Theory*. PhD thesis, École Polytechniqe, 2011.

[32] Martin Strecker. *Construction and Deduction in Type Theories*. PhD thesis, Universität Ulm, 1998.

[33] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Type processing by constraint reasoning. In *APLAS*, pages 1–25, 2006.

[34] Enrico Tassi. *Interactive Theorem Provers: issues faced as a user and tackled as a developer*. PhD thesis, University of Bologna, 2008.

[35] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Université Paris VII, May 1994.