From Formal Proofs to Mathematical Proofs: A Safe, Incremental Way for Building in First-order Decision Procedures

Fréderic Blanqui and Jean-Pierre Jouannaud and Pierre-Yves Strub

Abstract We investigate here a new version of the Calculus of Inductive Constructions (CIC) on which the proof assistant Coq is based: the Calculus of Congruent Inductive Constructions, which truly extends CIC by building in arbitrary first-order decision procedures: deduction is still in charge of the CIC kernel, while computation is outsourced to dedicated first-order decision procedures that can be taken from the shelves provided they deliver a proof certificate. The soundness of the whole system becomes an incremental property following from the soundness of the certificate checkers and that of the kernel. A detailed example shows that the resulting style of proofs becomes closer to that of the working mathematician.

1 Introduction

Proof assistants based on the Curry-Howard isomorphism such as Coq [9] allow to build the proof of a given proposition by applying appropriate proof tactics available from existing libraries or that can otherwise be developed for achieving a specific task. These tactics generate a proof term that can be checked with respect to the rules of logic. The proof-checker, also called the *kernel* of the proof assistant, implements the deduction rules of the logic on top of a term manipulation layer. In this model, the mathematical correctness of a proof development relies entirely on the kernel. Trusting the kernel is therefore vital.

The (intuitionist) logic on which Coq is based is the Calculus of Constructions (CC) of Coquand and Huet [10], an impredicative type theory incorporating polymorphism, dependent types and type constructors. Unlike logics without dependent types, CC enjoys a powerful type-checking rule, called *conversion*, which incorporates computations within deductions, making decidability of type-checking a non-trivial property of the calculus.

In CC, computation reduces to (pure) functional evaluation in the underlying lambda calculus. The notion of computation is richer in the Calculus of Inductive Constructions of Coquand and Paulin (CIC), obtained from CC by adding inductive

Fréderic Blanqui
INRIA & LORIA, Campus Scientifique, BP 239, 54506 Vandoeuvre-lès-Nancy Cedex, France, e-mail: blanqui@loria.fr

Jean-Pierre Jouannaud · Pierre-Yves Strub
LIX, UMR 7161, Project INRIA TypiCal, École Polytechnique, 91128 Palaiseau, France, e-mail: jouannaud,strub@lix.polytechnique.fr

types and the corresponding rules for higher-order primitive recursion [11]. The recent versions of Coq are based on a slight generalization of this calculus [15]. Still, such a simple function as *reverse* of a *dependent list* cannot be defined in CIC as one would expect, because $(reverse\ l :: l')$ and $(reverse\ l') :: (reverse\ l)$, assuming :: is list concatenation, have non-convertible types $list(n+m)$ and $list(m+n)$, assuming $(reverse\ l)$ has for type the type of its argument l. This is so because the usual definition of $+$ by induction on one of its arguments does not reduce the proof of $m+n = n+m$ to a computation.

We do believe that scaling up the proof development process requires being able to mimic the mathematician when replacing the proof of a proposition P by the proof of an equivalent proposition P' obtained from P thanks to possibly complex calculations in which *easy steps* are hidden away. It is our program to make this view a reality.

A way to incorporate decision procedures to Coq is by developing a tactic and then use a reflexion technique to omit checking the proof term being built by proving the decision procedure itself. But the soundness of the entire mechanism cannot be guaranteed in general [12]. Further, this does not answer the question of hiding easy steps away.

A first attempt towards our goal is the Calculus of Algebraic Constructions (CAC), obtained by adding to CC user-defined computations as rewrite rules [5, 3]. Although conceptually quite powerful since CAC captures CIC [4], this paradigm does not yet fulfill all needs. In particular, the user needs to hide away the easy steps by himself, that is by giving the necessary rewrite rules and by verifying that they satisfy the assumptions of the *general schema* [5, 3].

The proof assistant PVS uses a potentially stronger paradigm than Coq by combining its deduction mechanism with a notion of computation based on the powerful Shostak's method for combining decision procedures [20], a framework dubbed *little proof engines* by Shankar [19]. Indeed, the little engines of proof hide away the easy computational steps, without any user assistance. Unfortunately, proof-checking is not decidable in PVS. Further, since the little engines of proofs involve complex coding, as well as Shostak's algorithm itself, one can only *believe* a PVS proof, while one can *check* and *trust* a Coq proof.

Two steps in the direction of integrating decision procedures into CC are Stehr's Open Calculus of Constructions (OCC) [21] and Oury's Extensional Calculus of Constructions (ECC) [17]. Implemented in Maude, OCC allows for the use of an arbitrary equational theory in conversion. ECC can be seen as a particular case of OCC in which all provable equalities can be used in conversion, which can also be achieved by adding the extensionality and Streicher's axioms to CC [22], hence the name of this calculus. Unfortunately, strong normalization and decidability of type checking are then lost, which shows that we should seek for more restrictive extensions.

In a preliminary work, we designed a new, quite restrictive framework, the Calculus of Congruent Constructions (CCC), which incorporates the congruence closure algorithm in CC's conversion [7], while preserving the good properties of CC, including the decidability of type checking. In [6], we have described $CC_{\mathbb{N}}$, in which the decision procedure was Presburger arithmetic and strong elimination ruled out. The present work is a continuation of the latter.

Theoretical contribution. Our main theoretical contribution is the definition and the meta-theoretical investigation of the Calculus of Congruent Inductive Constructions (CCIC), which incorporates arbitrary *first-order theories* for which entailment is decidable into deductions via an abstract conversion rule of the calculus. A major technical innovation of this work lies in the computation mechanism: goals are sent to the decision procedure together with the set of user hypotheses available from the current context. Our main result shows that this extension of CIC does not compromise its properties: confluence, strong normalization, coherence and decidability of proof-checking are all preserved.

Unlike previous calculi, the difficulty with CCIC is not strong normalization, for which we have reused the strong normalization proof of CAC [3]. A major difficulty was a traditional step towards subject-reduction: compatibility of conversion with products. Decidability of type checking required restricting conversions below recursors [23].

Practical contribution. We give several examples showing the usefulness of this new calculus, in particular for using dependent types such as dependent lists, which has been an important weakness of Coq until now. Further studies are needed to explore other potential applications, to match inductive definition-by-case modulo theories of constructors-destructors, another very different weakness of Coq. A detailed example shows that the resulting style of proofs becomes closer to that of the working mathematician.

Methodological contribution. The safety of proof assistants is based on their kernel. In the early days of Coq, the safety of its kernel relied on its small size and its clear structure reflecting the inference rules of the intuitionist type theory, CC, on which it was based. The slogan was that of a *readable kernel*. Moving later to CIC allowed to ease the specification tasks, making the system very popular among proof developers, but resulted in a more complex kernel that can now hardly be read except by a few specialists. The slogan changed to a *provable kernel*, and indeed one version of it was once proved with an earlier version (using strong normalization as an assumption), and a new safe kernel extracted from that proof.

Of course, there has been many changes in the kernel since then, and its correctness proof was not maintained. This is a first weakness with the *readable kernel* paradigm: it does not resist changes. There is a second which relates directly to CCIC: there is no guarantee that a decision procedure taken from the shelf implements correctly the complex mathematical theorem on which it is based, since carrying out such a proof may require an entire PhD work. Therefore, these procedures *cannot* be part of the kernel.

Our solution to these problems is a new shift of paradigm to that of an *incremental kernel*. The calculus on which a proof assistant is based should come in two parts: a stable calculus implementing deduction, CIC in our case, which should satisfy the *readable* or *provable kernel* paradigm; a collection of independent decision procedures implementing computations, that produce checkable proof certificates.

The certificate checker should of course itself satisfy the *readable* or *provable code* paradigm. Note that a Coq proof is a particular case of a checkable certificate.

This paradigm has many advantages. First, it allows for a modular, cooperative development of the system, by separating the development of the kernel from that of the decision procedures. Second, it allows for an *unsafe mode* in case a decision procedure is used that does not have a certificate generator yet. Third, it allows to better trace errors in case the system rejects a proof, by using decision procedures that output *explanations* when they fail. Last, it allows the user to use any decision procedure she needs by simply hooking it to the system, possibly in unsafe mode.

This incremental schema is quite flexible, assuming that decision procedures come one by one. However, even so, they are not independent, they must be combined. Combining first-order decision procedures is not a new problem, it was considered in the early 80's by Nelson and Oppen on the one hand, by Shostak on the other hand, and has generated much work since then. There are several possibilities to build in this mechanism: in the kernel, via a certificate generator and checker again, or by reflection. This design decision has not been made yet.

2 Congruent Inductive Constructions

The Calculus of Congruent Inductive Constructions (CCIC) is an extension of CIC which embeds in its conversion rule the validity entailment of a fixed first order theory. First, we recall the basics of CIC before to introduce parametric multi-sorted algebras and then embed these first-order algebras into CIC. We are then able to define our calculus relative to a specific congruence that is defined last. For simplicity, we will only consider here the particular case of parametric lists and that of the natural numbers equipped with Presburger arithmetic. This simple case allows us to build lists of natural numbers, as well as lists of lists of natural numbers, and so on. It indeed has the complexity of the whole calculus, which is not at all the case when natural numbers only are considered as in [6].

2.1 *Calculus of Inductive Constructions*

Terms. We start our presentation by first describing the terms of CIC.

CIC uses two *sorts*: $\star$ (or Prop, or *object level universe*), $\Box$ (or Type, or *predicate level universe*) and $\triangle$. We denote $\{\star, \Box, \triangle\}$, the set of CIC sorts, by $\mathscr{S}$.

Following the presentation of *Pure Type Systems* (PTS) [14], we use two classes of variables: $\mathscr{X}^\star$ and $\mathscr{X}^\Box$ are countably infinite sets of *term variables* and *predicate variables* such that $\mathscr{X}^\star$ and $\mathscr{X}^\Box$ are disjoint. We write $\mathscr{X}$ for $\mathscr{X}^\star \cup \mathscr{X}^\Box$.

We shall use $\overline{u}$ for a list $(u_1, \ldots, u_n)$, s for a sort in $\mathscr{S}$, $x, y, \ldots$ for variables in $\mathscr{X}^\star$, $X, Y, \ldots$ for variables in $\mathscr{X}^\Box$.

Definition 1 (Pseudo-terms). The algebra $\mathscr{L}$ of *pseudo-terms* of CIC is defined by:

$$t, u, T, U, \ldots := s \in \mathscr{S} \mid x \in \mathscr{X} \mid \forall(x : T).\, t \mid \lambda[x : T].\, t$$
$$\mid \ t\, u \mid \mathrm{Ind}(X : t)\{\overline{T_i}\} \mid t^{[n]} \mid \mathrm{Elim}(t : T\, [\overline{u_i}] \to U)\{\overline{w_j}\}$$

The notion of free variables is as usual - the binders being λ, $\forall$ and Ind (in $\mathrm{Ind}(X:t)\{\overline{T_i}\}$, X is bound in the T_i's). We write $\mathrm{FV}(t)$ for the set of free variables of t. We say that t is closed if $\mathrm{FV}(t)=\emptyset$. A variable x *freely occurs* in t if $x\in\mathrm{FV}(t)$.

Inductive types. The novelty of CIC was to introduce inductive types, denoted by $I=\mathrm{Ind}(X:T)\{\overline{C_i}\}$ where the $\overline{C_i}$'s describe the types of the *constructors* of I, and T the type (or *arity*) of I which must be of the form $\forall(\overline{x_i:T_i}).\star$. The k-th constructor of the inductive type I, of type $C_k\{X\mapsto I\}$, will be denoted by $I^{[k]}$.

As an easy first example, we define natural numbers: $\mathbf{nat}:=\mathrm{Ind}(X:\star)\{X,X\to X\}$. We shall use $\mathbf{0}$ and $\mathbf{S}$ as constructors for natural numbers, of respective types $\mathbf{nat}$ and $\mathbf{nat}\to\mathbf{nat}$, obtained by replacing X by $\mathbf{nat}$ in the above two expressions X and $X\to X$. Elimination rules for $\mathbf{nat}$ are as follows:

$$\begin{array}{l}
\mathrm{Elim}\mathbb{N}(\mathbf{0},Q)\{v_{\mathbf{0}},v_{\mathbf{S}}\} \xrightarrow{\iota} v_{\mathbf{0}}\\
\mathrm{Elim}\mathbb{N}(\mathbf{S}x,Q)\{v_{\mathbf{0}},v_{\mathbf{S}}\} \xrightarrow{\iota} v_{\mathbf{S}}\,x\,(\mathrm{Elim}\mathbb{N}(x,Q)\{v_{\mathbf{0}},v_{\mathbf{S}}\}) \text{ with } Q:\mathbf{nat}\to s,\ \in\mathscr{S}.
\end{array}$$

Similarly, we now define parametric lists: $\mathbf{list}:=\lambda[T:\star].\,\mathrm{Ind}(X:\star)\{X,T\to X\to X\}$. We shall use $\mathbf{nil}$ and $\mathbf{cons}$ as constructors for parametrized lists, of respective types $\forall(T:\star).\,\mathbf{list}(T)$ and $\forall(T:\star).\,T\to\mathbf{list}(T)\to\mathbf{list}(T)$. Elimination rules for $\mathbf{list}$ are:

$$\begin{array}{ll}
\mathrm{Elim}\mathbb{L}(\mathbf{nil},Q)\{v_{\mathbf{nil}},v_{\mathbf{cons}}\} & \xrightarrow{\iota} v_{\mathbf{nil}}\\
\mathrm{Elim}\mathbb{L}(\mathbf{cons}\,x\,l,Q)\{v_{\mathbf{nil}},v_{\mathbf{cons}}\} & \xrightarrow{\iota} v_{\mathbf{cons}}\,x\,l\ \mathrm{Elim}\mathbb{L}(l,Q)\{v_{\mathbf{nil}},v_{\mathbf{cons}}\})
\end{array}$$

Finally, we define dependent words over an alphabet A:

$$\mathbf{word}\ =\ \mathrm{Ind}(X:\mathbf{nat}\to\star)\{X\,\mathbf{0},A\to X\ (\mathbf{S}\,\mathbf{0}),\forall(y,z:\mathbf{nat}).\,X\,y\to X\,z\to X(y+z)\}$$

We shall use ε, $\mathbf{char}$ and $\mathbf{app}$ for its three constructors, of respective types $\mathbf{word}\,\mathbf{0}$, $A\to\mathbf{word}\,(\mathbf{S}\,\mathbf{0})$, and $\forall(n,m:\mathbf{nat}).\,\mathbf{word}\,n\to\mathbf{word}\,m\to\mathbf{word}\,(n+m)$ obtained as previously by replacing X by $\mathbf{word}$ in the three expressions $X\,\mathbf{0},A\to X\ (\mathbf{S}\,\mathbf{0})$, and $\forall(y,z:\mathbf{nat}).\,X\,y\to X\,z\to X(y+z)$. Elimination rules for dependent words are:

$$\begin{array}{ll}
\mathrm{Elim}\mathbb{W}(\varepsilon,Q)\{v_\varepsilon,v_{\mathbf{char}},v_{\mathbf{app}}\} & \xrightarrow{\iota} v_\varepsilon\\
\mathrm{Elim}\mathbb{W}(\mathbf{char}\,x,Q)\{v_\varepsilon,v_{\mathbf{char}},v_{\mathbf{app}}\} & \xrightarrow{\iota} v_{\mathbf{char}}\,x\\
\mathrm{Elim}\mathbb{W}(\mathbf{app}\,n\,m\,l\,l',Q)\{v_\varepsilon,v_{\mathbf{char}},v_{\mathbf{app}}\} & \xrightarrow{\iota} v_{\mathbf{app}}\,n\,m\,l\,l'\,(\mathrm{Elim}\mathbb{W}(l,Q)\{v_\varepsilon,v_{\mathbf{char}},v_{\mathbf{app}}\})\\
& \qquad\qquad\qquad (\mathrm{Elim}\mathbb{W}(l',Q)\{v_\varepsilon,v_{\mathbf{char}},v_{\mathbf{app}}\})
\end{array}$$

Definitions by induction. We can now define functions by induction over natural numbers, lists or words. Since using the CIC syntax is a bit painful, we give only a quite simple example defining append (written @) for lists of natural numbers, of type $\forall(T:\star).\,\mathbf{list}(T)\to\mathbf{list}(T)\to\mathbf{list}(T)$:

$$@:=\lambda[l:\mathbf{list}\,\mathbf{nat}][l':\mathbf{list}\,\mathbf{nat}].\,\mathrm{Elim}\mathbb{L}(l,Q)\left\{\begin{array}{l}\quad\lambda[x:\mathbf{nat}][l'':\mathbf{list}\,\mathbf{nat}].\\ l',\quad\lambda[l1:\mathbf{list}\,\mathbf{nat}][l2:\mathbf{list}\,\mathbf{nat}].\\ \qquad\lambda[L:Q\,l1\,l2].\,\mathbf{cons}\,x\,L\end{array}\right\}$$

Strong and Weak reductions. CIC distinguishes *strong* ι-elimination when the type Q of terms constructed by induction is at predicate level, from weak ι-elimination when Q is at object level. Strong elimination is restricted to *small* inductive types to ensure logical consistency [24].

Typing judgments. A *typing environment* Γ is a sequence of pairs $x_i : T_i$ made of a variable x_i and a term T_i (we say that Γ binds x_i to the type T_i), such that Γ does not bind a variable twice. The typing judgments are classically written $\Gamma \vdash t : T$, meaning that the *well formed term* t is a proof of the proposition T (has type T) under the *well formed environment* Γ. $x\Gamma$ will denote the type associated to x in Γ, and we write $\mathrm{dom}(\Gamma)$ for the domain of Γ as well.

The typing rules of CIC given in 1 are made of the typing rules for CC and the typing rules for inductive types, given for the particular case of **nat** and **list**.

$$\frac{}{\vdash \star : \Box}\,[\textsc{Ax-1}] \qquad \frac{}{\vdash \Box : \triangle}\,[\textsc{Ax-2}]$$

$$\frac{\Gamma \vdash V : s \quad \Gamma \vdash t : T \quad s \in \{\star,\Box\} \quad x \in \mathscr{X}^s - \mathrm{dom}(\Gamma)}{\Gamma,[x:V] \vdash t : T}\,[\textsc{Weak}]$$

$$\frac{\Gamma \vdash T : s_T \quad \Gamma,[x:T] \vdash U : s_U}{\Gamma \vdash \forall(x:T).U : s_U}\,[\textsc{Prod}]$$

$$\frac{x \in \mathrm{dom}(\Gamma) \cap \mathscr{X}^{s_x} \quad \Gamma \vdash x\Gamma : s_x}{\Gamma \vdash x : x\Gamma}\,[\textsc{Var}]$$

$$\frac{\Gamma \vdash \forall(x:T).U : s \quad \Gamma,[x:T] \vdash u : U}{\Gamma \vdash \lambda[x:T].u : \forall(x:T).U}\,[\textsc{Abs}]$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T' : s' \quad T \overset{\beta\iota}{\longleftrightarrow}_* T'}{\Gamma \vdash t : T'}\,[\textsc{Conv}]$$

$$\frac{\Gamma \vdash t : \forall(x:U).V \quad \Gamma \vdash u : U}{\Gamma \vdash t\,u : V\{x \mapsto u\}}\,[\textsc{App}]$$

$$\frac{\vdash \tau_f : s \in \{\star,\Box\}}{\vdash f : \tau_f}\,[\textsc{Symb}]$$

$$\frac{\begin{array}{c}\Gamma \vdash Q : \mathbf{nat} \to s \in \{\star,\Box\}\\ \Gamma \vdash n : \mathbf{nat} \quad \Gamma \vdash v_{\mathbf{0}} : Q\,\mathbf{0}\\ \Gamma \vdash v_{\mathbf{S}} : \forall(p:\mathbf{nat}).\,Q\,p \to Q\,(\mathbf{S}\,p)\end{array}}{\mathrm{Elim}\mathbb{N}(n,Q)\{v_{\mathbf{0}},v_{\mathbf{S}}\} : Q\,n}\,[\textsc{Elim}]$$

$$\frac{\begin{array}{c}\Gamma \vdash T : \star \quad \Gamma \vdash p : \mathbf{nat} \quad \Gamma \vdash l : \mathbf{list}\,T\,p\\ \Gamma \vdash Q : \forall(n:nat).\,\mathbf{list}\,T\,n \to s \in \{\star,\Box\}\\ \Gamma \vdash v_{\mathbf{nil}} : Q\,\mathbf{0}\,(\mathbf{nil}\,T)\\ \Gamma \vdash v_{\mathbf{cons}} : \begin{array}{c}\forall(x:T)(n:\mathbf{nat})(l:\mathbf{list}\,T\,n).\\ Q\,n\,l \to Q\,(\mathbf{S}\,n)(\mathbf{cons}\,T\,x\,n\,l)\end{array}\end{array}}{\mathrm{Elim}\mathbb{L}(l,Q)\{v_{\mathbf{0}},v_{\mathbf{S}}\} : Q\,p\,l}\,[\textsc{Elim}]$$

Fig. 1 CIC typing rules for **nat** and **list**

We did not give the general typing elimination rule for arbitrary inductive types, which is quite complicated. Instead, we gave the elimination rules obtained for our three inductive types **nat**, **list** and **word**. We refer to [18, 24] for the general case, and for the precise typing rule of $\mathrm{Elim}\mathbb{W}$.

2.2 *Parametric sorted algebras*

Parametric sorted signature. Order-sorted algebras were introduced as a formal framework for the OBJ language in [13], before to be generalized as *membership equational logic* in [8]. We use here a polymorphic version of a restriction of the latter, by assuming given a signature (Λ, Σ), Λ for the sort constructors, and Σ for the function symbols made of a set of constructors for each sort constructor, and of a set of defined symbols. We shall use the notation $f : \forall\overline{\alpha}.\, \sigma_1 \times \cdots \times \sigma_n \to \tau$ for symbol declarations. As an example, we describe natural numbers and parametric (non-dependent) list using an OBJ-like syntax. We rule out here partiality, as introduced in practice by destructor symbols, for sake of clarity.

We shall use $\mathscr{V} = \{\alpha, \beta, \ldots\}$ for the set of sort variables, and $\mathscr{T}(\Sigma, \mathscr{V}) = \{\sigma, \tau, \ldots\}$ for the set of sort expressions.

sort **nat**	:	$*$
sort **list**	:	$* \to *$
svar α	:	$*$
cons **0**	:	**nat**

cons **S**	:	$\mathbf{nat} \to \mathbf{nat}$
fun $\dot{+}$	:	$\mathbf{nat} \times \mathbf{nat} \to \mathbf{nat}$
cons **nil**	:	$\mathbf{list}(\alpha)$
cons **cons**	:	$\alpha \times \mathbf{list}(\alpha) \to \mathbf{list}(\alpha)$
fun @	:	$\mathbf{list}(\alpha) \times \mathbf{list}(\alpha) \to \mathbf{list}(\alpha)$

Definition 2 (Terms). For any sort σ, let $\mathscr{X}^\sigma$ be a countably infinite set of *variables of sort* σ, s.t. all the $\mathscr{X}^\sigma$'s are pairwise disjoint. Let $\mathscr{X} = \bigcup_\sigma \mathscr{X}^\sigma$. For any $x \in \mathscr{X}$, we say that x has sort σ if $x \in \mathscr{X}^\sigma$. For any sort σ, the set $\mathscr{T}_\sigma(\Sigma, \mathscr{X})$ of *terms of sorts* σ *with variables* $\mathscr{X}$ is the smallest set s.t.:

1. if $x \in \mathscr{X}^\tau$, then $x \in \mathscr{T}_\tau(\Sigma)$,
2. if $t_1, \cdots, t_n \in \mathscr{T}_{\sigma_1\xi}(\Sigma, \mathscr{X}) \times \cdots \times \mathscr{T}_{\sigma_2\xi}(\Sigma, \mathscr{X})$ where $f : \forall\overline{\alpha}.\, \sigma_1 \times \cdots \times \sigma_n \to \tau$ and ξ is a sort substitution, then $f(t_1, \ldots, t_n) \in \mathscr{T}_{\tau\xi}(\Sigma, \mathscr{X})$.

Let $\mathscr{T}(\Sigma, \mathscr{X}) = \bigcup_\sigma(\mathscr{T}_\sigma(\Sigma, \mathscr{X}))$. A term t has sort σ if $t \in \mathscr{T}_\sigma(\Sigma, \mathscr{X})$.

Note that the sets $\mathscr{X}^\sigma$ play the role of a typing context.

Example 1. Assuming that x is a variable of sort **nat**, then 0 and $0 + x$ are of sort **nat**, while **nil** is of sort $\mathbf{list}(\alpha)$, $\mathbf{list}(\mathbf{nat})$, $\mathbf{list}(\mathbf{list}(\mathbf{nat}))$, etc.

Definition 3 (Equations). Equations $t =^\sigma u$ are pairs of terms of the same sort σ.

Example 2. Assuming x of sort **nat** and l of sort $\mathbf{list}(\mathbf{list}((\mathbf{nat}))$, $x + 0 =^{\mathbf{nat}} x$ is an equation of sort **nat** and $cons(x, nil) =^{list(\mathbf{nat})} car(l)$ is an equation of sort $\mathbf{list}(\mathbf{nat})$.

We can therefore as usual build parametrized algebras for **list**, algebras for **nat** and therefore get algebras for **nat**, $\mathbf{list}(\mathbf{nat})$, etc. Satisfaction of an equation in these algebras is defined as usual. In practice, type superscripts may be omitted when they can be infered from the context.

2.3 Embedding parametric algebras in CIC

Our purpose here is to embed parametric multi-sorted algebra into CIC. As a result, two different, but related kinds of symbols will coexist, in CIC and in the embedded algebraic sub-world. We shall distinguish them by underlying symbols in CIC.

The first step of the translation maps, respectively sort constructors and constructor symbols to CIC inductive types and constructors. We start with natural numbers and its sort constructor **nat**. Constructor symbols of **nat** are simply all the constructors symbols whose codomain is **nat**, i.e. here **0** and **S**. We thus define $\underline{\mathbf{nat}}$ (the CIC inductive type attached to **nat**) as an inductive type with two constructor types (one for **0**, and one for **S**): $\underline{\mathbf{nat}} := \mathrm{Ind}(X : \star)\{C_1(X), C_2(X)\}$.

The constructor types of $\underline{\mathbf{nat}}$ are simply the arities of $\mathbf{0}$ and $\mathbf{S}$ where $\mathbf{nat}$ is replaced with the constructor type variable: $C_1(X) = X$ and $C_2(X) = X \rightarrow X$. As expected, we obtain here the standard inductive definition of natural numbers given in Section 2.1: $\mathrm{Ind}(X : \star)\{X, X \rightarrow X\}$. The translation $\underline{\mathbf{0}}$ of $\mathbf{0}$ (resp. $\underline{\mathbf{S}}$ of $\mathbf{S}$) is then simply $\underline{\mathbf{nat}}^{[1]}$ (resp. $\underline{\mathbf{nat}}^{[2]}$).

Translating **list** is not very different. Being of arity 1, with two associated constructor symbols (**nil** and **cons**), **list** is mapped to the already seen parametrized inductive type $\underline{\mathbf{list}} = \lambda[A : T].\,\mathrm{Ind}(\star)\{X, A \rightarrow X \rightarrow X\}$. Translation of constructors is done the same way. We just need to care about curryfication of symbols, and to replace sort variables with CIC type variables.

Finally, defined symbols are mapped to CIC defined symbols, after translating their type appropriately.

2.4 Building in a first-order theory

We now start describing our new calculus CCIC.

Terms. CCIC uses the same set of sorts $\mathscr{S} = \{\star, \Box, \triangle\}$ and sets of variables $\mathscr{X} = \mathscr{X}^\star \cup \mathscr{X}^\Box$ of CIC. For any sort $\sigma \in \Lambda$, let $\mathscr{X}_\sigma \subseteq \mathscr{X}^\star$ a infinite set of variables of sort σ s.t. $\{\mathscr{X}_\sigma\}_\sigma$ is a family of pairwise disjoint sets. We also assume that $\mathscr{X} - \bigcup_\sigma \mathscr{X}_\sigma$ is infinite.

Let $\mathscr{A} = \{\mathrm{r}, \mathrm{u}\}$ a set of two constants, called *annotations*, totally ordered by $\mathrm{u} \prec_{\mathscr{A}} \mathrm{r}$, where r stands for *restricted* and u for *unrestricted*. We use a for an arbitrary annotation. The role of annotations will be explained later.

Definition 4 (Pseudo-terms of CCIC). Given a parametric sorted signature (Λ, Σ), the algebra $\mathscr{L}$ of *pseudo-terms* of CCIC is defined as:

$$t, u, T, U, \ldots := s \in \mathscr{S} \mid x \in \mathscr{X} \mid \forall(x :^a T).t \mid \lambda[x :^a T].t \mid t\,u \mid f \in \Sigma \mid \sigma \in \Lambda$$
$$\mid \doteq \mid \mathrm{Eq}_T(t) \mid \mathrm{Ind}(X : t)\{\overline{T_i}\} \mid t^{[n]} \mid \mathrm{Elim}(t : T\,[\overline{u_i}] \rightarrow U)\{\overline{w_j}\}$$

In order to make definitions more convenient, we assume in the following that Λ contains the symbols $\doteq$, **nat** and **list**, and that Σ contains the symbols $\mathbf{0}, \mathbf{S}$ and Eq.

Compared with CIC, the differences are:

- the internalization of the first-order symbols,
- the internalization of the equality predicate:
  - $t \doteq_T u$ denotes the equality of the two terms (of type T) t and u,
  - $\mathrm{Eq}_T(t)$ represents the reflexivity proof of $t \doteq_T t$.
- annotations in products and abstractions are used to control the formation of applications as it can be seen from the new [APP] rule given at Figure 2.

Notation 2.1 *When x is not free in t, $\forall(x :^a T).t$ is written $T \rightarrow^a t$. The default annotation, when not specified in a product or abstraction, is the* unrestricted *one.*

As usual, there is a layered set of syntactic classes for $\mathscr{L}$:

Definition 5 (Syntactic classes). The pairwise disjoint syntactic classes of CCIC called *objects* ($\mathscr{O}$), *predicates* ($\mathscr{P}$), *kinds* ($\mathscr{K}$), *kinds predicates* ($\mathscr{M}$), and $\triangle$ are defined as usual:

– $\mathscr{O} ::= \mathscr{X}^\star \mid f \in \Sigma \mid \mathscr{O}\,\mathscr{O} \mid \mathscr{O}\,\mathscr{P} \mid \lambda[x^\star :^a \mathscr{P}].\,\mathscr{O} \mid \lambda[x^\Box :^a \mathscr{K}].\,\mathscr{O} \mid \mathrm{Elim}(\mathscr{O} : \mathscr{P}\,[\overline{\mathscr{O}}] \to \mathscr{O})\{\overline{\mathscr{O}}\}$
– $\mathscr{P} ::= \mathscr{X}^\Box \mid \sigma \in \Lambda \mid \mathscr{P}\,\mathscr{O} \mid \mathscr{P}\,\mathscr{P} \mid \lambda[x^\star :^a \mathscr{P}].\,\mathscr{P} \mid \lambda[x^\Box :^a \mathscr{K}].\,\mathscr{P}$
$\quad \mid \mathrm{Elim}(\mathscr{O} : \mathscr{P}\,[\overline{\mathscr{O}}] \to \mathscr{P})\{\overline{\mathscr{P}}\} \mid \forall(x^\star :^a \mathscr{P}).\,\mathscr{P} \mid \forall(x^\Box :^a \mathscr{K}).\,\mathscr{P}$
– $\mathscr{K} ::= \star \mid \mathscr{K}\,\mathscr{O} \mid \mathscr{K}\,\mathscr{P} \mid \lambda[x^\star :^a \mathscr{P}].\mathscr{K} \mid \lambda[x^\Box :^a \mathscr{K}].\mathscr{K} \mid \forall(x^\star :^a \mathscr{P}).\mathscr{K} \mid \forall(x^\Box :^a \mathscr{K}).\mathscr{K}$
– $\mathscr{M} ::= \Box \mid \forall(x^\star :^a \mathscr{P}).\mathscr{M} \mid \forall(x^\Box :^a \mathscr{K}).\mathscr{M}$
– $\triangle ::= \triangle$

This enumeration defines a successor function $+1$ on classes ($\mathscr{O}+1=\mathscr{P}$, $\mathscr{P}+1=\mathscr{K}$, $\mathscr{K}+1=\mathscr{M}$, $\mathscr{M}+1=\triangle$). We also define $\mathrm{Class}(t)=\mathscr{D}$ if $t \in \mathscr{D}$ and $\mathscr{D} \in \{\mathscr{O},\mathscr{P},\mathscr{K},\mathscr{M},\triangle\}$.

From now on, we only consider *well-constructed terms* (i.e. terms whose class is not $\perp$) and *well-constructed substitution* (i.e. substitutions s.t. $\mathrm{Class}(x)=\mathrm{Class}(x\theta)$ for any x in its domain). It is easy to check that if t is a well-constructed term and θ a well-constructed substitution, then $\mathrm{Class}(t)=\mathrm{Class}(t\theta)$. It is also well-known that $\xrightarrow{\beta\iota}$-reduction preserves term classes.

Definition 6 (Pseudo-contexts of CCIC). The typing environments of CIC are defined as $\Gamma,\Delta ::= [] \mid \Gamma,[x :^a T]$ s.t. a variable cannot be declared twice. We use $\mathrm{dom}(\Gamma)$ for the domain of Γ and $x\Gamma$ for the type associated to x in Γ.

The rules defining the CCIC typing judgment $\Gamma \vdash t : T$ are the same as for CIC except the rules for application and conversion given at Figure 2.

$$\frac{\begin{array}{c}\Gamma \vdash t : \forall(x :^a U).V \quad \Gamma \vdash u : U \\ \text{if } a = \mathrm{r} \text{ and } U \xrightarrow{\beta}_{*} t_1 \doteq_T t_2 \text{ with } t_1, t_2 \in \mathscr{O} \\ \text{then } t_1 \sim_\Gamma t_2 \text{ must hold}\end{array}}{\Gamma \vdash t\,u : V\{x \mapsto u\}}\ [\textsc{App}] \qquad \frac{\Gamma \vdash t : T \quad \Gamma \vdash T' : s' \quad T \sim_\Gamma T'}{\Gamma \vdash t : T'}\ [\textsc{Conv}]$$

Fig. 2 CCIC modified typing rules

2.5 *Conversion*

We are now left with defining the conversion relation $\sim_\Gamma$, whose definition needs some preparation, since:

- conversion is defined on CCIC terms, but the first-order decision procedures operate on algebraic terms. We therefore need to translate CCIC terms into algebraic terms, a process we call *algebraisation*.
- conversion will operate on weak terms only, a notion introduced in Section 2.5. Non-weak terms will be converted with $\beta\iota$-reduction only, to forbid lifting up inconsistencies from the object level to the type level. This is crucial to avoid breaking strong normalization, and therefore decidability of type-checking in presence of inconsistent user's assumptions.

Algebraisation. Our calculus has a complex notion of computation reflecting its rich structure made of three ingredients: the typed lambda calculus, the inductive types with their recursors and the integration of the first order theory $\mathscr{T}$ in its conversion. To achieve this integration, goals are sent to the first order theory $\mathscr{T}$ together with a set of proof hypotheses extracted from the current context.

Algebraisation is the first step of this extraction: it allows transforming a CCIC term into its first-order counterpart. We illustrate this with an example, $\mathscr{T}$ being Presburger's arithmetic.

We begin by the simplest case, directly taken from $\mathrm{CC}_{\mathbb{N}}$, the extraction of pure algebraic, non parametric, equations. Suppose that the proof environment contains equations of the form $c \doteq 1+d$ and $d \doteq 2$ with c and d variables of sort **nat**. What is expected is that the set of hypotheses sent to the theory $\mathscr{T}$ contains the two well formed $\mathscr{T}$-formulas $c = 1+d$ and $d = 2$. This leads to a first definition of equations extraction:

1. a term is algebraic if it is of the form 0, or $S\,t$, or $t+u$, or $x \in \mathscr{X}_{\mathbb{N}}$. The *algebraisation* $\mathscr{A}(t)$ of an algebraic term is then defined by induction: $\mathscr{A}(0) = 0$, $\mathscr{A}(S\,t) = S(\mathscr{A}(t))$, $\mathscr{A}(t+u) = \mathscr{A}(t) + \mathscr{A}(u)$ and $\mathscr{A}(x_{\mathbb{N}}) = x_{\mathbb{N}}$,
2. a term is an extractable equation if it is of the form $t \doteq u$ with t and u algebraic terms. The extracted equation is then $\mathscr{A}(t) = \mathscr{A}(u)$.

The definition becomes harder for parametric signatures. The theory of lists gives us a paradigmatic example. From the definition of embedding a polymorphic multi-sorted algebra into CIC, we know that the symbol @ has $\forall(T:\star).\,\mathbf{list}\,T \to \mathbf{list}\,T \to \mathbf{list}\,T$ for type. Thus, a fully applied, well formed term having the symbol @ at head position must be of the form $(@\,T\,l1\,l2)$, T being the type of the elements of the lists $l1$ and $l2$. Algebraisation of such a term will erase all type parameters: in our example, $\mathscr{A}(@\,T\,l1\,l2) = @(\mathscr{A}(l1), \mathscr{A}(l2))$.

Algebraisation of non-pure algebraic terms is done by abstracting non-algebraic subterms with fresh variables. For example, algebraisation of $1+t$ with t non-algebraic will lead to $1+x_{\mathbf{nat}}$ where $x_{\mathbf{nat}}$ is an abstraction variable of sort **nat** for t. Of course, if the proof context contains two equations of the form $c \doteq 1+t$ and $d \doteq 1+u$ with t and u $\beta\iota$-convertible, t and u should be abstracted by a unique variable so that $c = d$ can be deduced in $\mathscr{T}$ from $c = 1+y_{\mathbf{nat}}$ and $d = 1+y_{\mathbf{nat}}$. The problem is harder for:

- *parametric symbols*: in $(\mathbf{cons}\,T\,t\,(\mathbf{nil}\,U))$ with t non algebraic, should t be abstracted by a variable of sort **nat** or **list**(**nat**) ?
- *ill-formed terms*: should $(\mathbf{cons}\,T\,0\,(\mathbf{cons}\,T\,(\mathbf{nil}\,U)\,(\mathbf{nil}\,T)))$ be abstracted as a list of natural numbers or as a list of lists ?

Our solution is to postpone decisions: $\mathscr{A}(t)$ will be a function from Λ to the terms of $\mathscr{T}$ s.t. $\mathscr{A}(t)(\sigma)$ is the algebraisation of t under the condition that t is a CCIC representation of a first order term of sort σ.

We now give the formal definition of $\mathscr{A}(\cdot)$. We assume:

- a Λ-sorted family $\{\mathscr{Y}_\sigma\}_\sigma$ of pairwise disjoint countable infinite sets of variables of sort σ. Let $\mathscr{Y} = \bigcup_\sigma \mathscr{Y}_\sigma$;

- for any equivalence relation $\mathscr{R}$ and sort $\sigma \in \Lambda$, we assume a function $\pi^\sigma_\mathscr{R}$: CCIC$(\mathscr{X}) \to \mathscr{Y}_\sigma$ s.t. $\pi^\sigma_\mathscr{R}(t) = \pi^\sigma_\mathscr{R}(u)$ if and only if $t\ \mathscr{R}\ u$ (i.e. $\pi^\sigma_\mathscr{R}(t)$ is the element of $\mathscr{Y}_\sigma$ representing the class of t modulo $\mathscr{R}$).

Definition 7 (Well applied term). A term is well applied if it is of the form $f\,[\overline{T_\alpha}]_{\alpha\in\overline{\alpha}}\,t_1\cdots t_n$ with $f : \forall\overline{\alpha}.\ \sigma_1 \times \cdots \times \sigma_n \to \sigma$.

Example 3. Example of well applied terms are 0, $S\,t$, or $\mathbf{cons}\,T\,x\,l$, T being the type parameter here. Note that we do not require the term to be well formed.

In case of partial symbols, such as **car** for lists, this definition must be changed slightly by adding a new argument, the proof that the input satisfies the appropriate guard, here that it is not **nil**.

Definition 8 (Algebraisation). The *algebraisation of $t \in CCIC$ modulo an equivalence relation $\mathscr{R}$* is the function $\mathscr{A}_\mathscr{R}(t) : \Lambda \to \mathscr{T}(\mathscr{X}^\star \cup \mathscr{Y})$ defined by:

$$\begin{aligned}
\mathscr{A}_\mathscr{R}(x_\sigma)(\sigma) &= x_\sigma\\
\mathscr{A}_\mathscr{R}(f\,\overline{T}\,[u_i]_{i\in n})(\tau\xi) &= f(\mathscr{A}_\mathscr{R}(u_1)(\sigma_1\xi),\ldots,\mathscr{A}_\mathscr{R}(u_n)(\sigma_n\xi))\\
\mathscr{A}_\mathscr{R}(t)(\tau) &= \pi^\tau_\mathscr{R}(t) \quad \text{otherwise}
\end{aligned}$$

where $f : \forall\overline{\alpha}.\ \sigma_1 \times \cdots \sigma_n \to \sigma$, $f\,\overline{T}\,[u_i]_{i\in n}$ is well applied, and ξ is a Λ-substitution.

For any relation R, $\mathscr{A}_R$ is defined as $\mathscr{A}_\mathscr{R}$ where $\mathscr{R}$ is the smallest equivalence relation containing R. We call σ*-alien* (or *alien* when the context is clear) a subterm of t abstracted by a variable in $\mathscr{Y}_\sigma$, and say that t is *algebraic* w.r.t. σ if contains no σ-alien. We denote by $\mathscr{A}\mathrm{lg}_\sigma$ the set of algebraic terms w.r.t. σ, and by $\mathscr{A}\mathrm{lg} = \bigcup_{\sigma\in\Lambda}\mathscr{A}\mathrm{lg}_\sigma$ the set of algebraic terms.

Example 4. Let $t \equiv \mathbf{cons}\,T\,\mathbf{0}\,(\mathbf{cons}\,U\,(\mathbf{nil}\,V)\,(\mathbf{nil}\,U))$, R be a relation on CCIC terms, $\sigma = \mathbf{list}(\mathbf{nat})$, and $x_\mathbf{nat}, y_\mathbf{list}, z_\mathbf{nat}, x_\alpha$ and y_α be abstraction variables. Then:

$$\begin{aligned}
\mathscr{A}_R(t)(\sigma) &= \mathbf{cons}(\mathscr{A}_R(\mathbf{0})(\mathbf{nat}), \mathscr{A}_R(\mathbf{cons}\,U(\mathbf{nil}\,V)\,(\mathbf{nil}\,U))(\sigma))\\
&= \mathbf{cons}(0, \mathbf{cons}(\mathscr{A}_R(\mathbf{nil}\,V)(\mathbf{nat}), \mathscr{A}_R(\mathbf{nil}\,U)(\sigma))) = \mathbf{cons}(0, \mathbf{cons}(x_\mathbf{nat}, \mathbf{nil}))\\
\mathscr{A}_R(t)(\mathbf{list}(\sigma)) &= \mathbf{cons}(\mathscr{A}_R(\mathbf{0})(\sigma), \mathscr{A}_R(\mathbf{cons}\,U(\mathbf{nil}\,V)\,(\mathbf{nil}\,U))(\mathbf{list}(\sigma)))\\
&= \mathbf{cons}(y_\mathbf{list}, \mathbf{cons}(\mathscr{A}_R(\mathbf{nil}\,V)(\sigma), \mathscr{A}_R(\mathbf{nil}\,U)(\mathbf{list}(\sigma)))) = \mathbf{cons}(y_\mathbf{list}, \mathbf{cons}(\mathbf{nil}, \mathbf{nil}))
\end{aligned}$$

$\mathscr{A}_R(t)(\mathbf{list}(\alpha)) = \mathbf{cons}(x_\alpha, \mathbf{cons}(y_\alpha, \mathbf{nil}))$ and $\mathscr{A}_R(t)(\mathbf{nat}) = z_\mathbf{nat}$.

It is clear from the above example that the algebraisation of a term depends on the expected sort of the result: when abstracting the (heterogeneous and ill-formed) list 0 :: **nil** :: **nil** as a list of lists, 0 is seen as an alien which must be abstracted. When this list is abstracted as a list of natural numbers or as a polymorphic list, 0 is considered algebraic and the first occurrence of **nil** as an alien to be abstracted. Finally, if the list is algebraised as a natural number, it is abstracted by a variable.

Weak terms. We first distinguish a class of terms called *weak*. This class of terms will play an important role in the following as they restrict the interaction between the conversion at object level and the strong ι-reduction.

An example of non weak term is

$$t = \lambda[x : \mathbf{nat}].\ \mathrm{Elim}^\mathscr{S}(x : \mathbf{nat}\,[] \to Q)\{\mathbf{nat}, \lambda[x : \mathbf{nat}][T : Q\,x].\ \mathbf{nat} \to \mathbf{nat}\}$$

Such a term is problematic in the sense that when applied to convertible terms, it can $\beta\iota$-reduce to type-level terms that are not $\beta\iota$-convertible. Suppose that the conversion relation is canonically extended to CCIC. Assume a typing environment Γ s.t. $\mathbf{0} \sim_\Gamma \mathbf{S}\,\mathbf{0}$, and hence, by congruence, $t\,\mathbf{0} \sim_\Gamma t\,(\mathbf{S}\,\mathbf{0})$. Now, it is easy to check that $t\,\mathbf{0} \xrightarrow{\beta\iota}_* \mathbf{nat}$ and $t\,(\mathbf{S}\,\mathbf{0}) \xrightarrow{\beta\iota}_* (\mathbf{nat} \to \mathbf{nat})$. Strong normalization of β-reduction is then broken by encoding the term $\omega = \lambda[x : \mathbf{nat}].\, x\,x$.

In contrast, *weak* terms lift no inconsistencies from object level to a higher level:

Definition 9 (Weak terms). A term is *weak* if it contains no i) applied type-level variable, and ii) term of the form $\mathrm{Elim}(t : I\,[\overline{u}] \to Q)\{\overline{f}\}$ with t open.

Extractable terms. From now on, let $\mathscr{O}^+$ be an arbitrary set of CCIC terms. This set will be used in the conversion definition to restrict the set of *extractable equations* of a given environment: only equation of the form $t \doteq u$ with t and u in $\mathscr{O}^+$ will be considered.

At the moment, we only require $\mathscr{O}^+$ to be a subset of $\mathscr{O}$. Note that taking $\mathscr{O}^+ = \mathscr{O}$ does not compromise the standard calculus properties (subject reduction, type unicity, strong normalization of $\beta\iota$-reduction, ...) but the decidability. E.g., if $\mathscr{T}$ is the Presburger arithmetic, allowing the extraction of

$$\lambda[x :^a \mathbf{nat}].\, f\,x \doteq \lambda[x :^a \mathbf{nat}].\, f\,(x \dotplus 2)$$

would require - for checking conversion - to decide any statement of the form

$$\mathscr{T} \vDash (\forall x.\, f(x) = f(x+2)) \to t = u,$$

which is well known to be impossible.

Conversion relation. We have now all necessary ingredients to define our conversion relation $\sim_\Gamma$:

Definition 10 (Conversion relation). Rules of Figure 3 define a family $\{\sim_\Gamma\}$ of CCIC binary relations indexed by a (non-necessarily well-formed) context Γ.

Note that the rule [DED] performing deductions in the first order theory, here Presburger arithmetic, outputs a certificate $[\_,\_,\_]$ made of the environment and the two terms to be proved equivalent under this environment, each time it is called. While this certificate must depend on these three data, it may of course carry additional information depending on the considered first-order theory.

The main differences with the calculus $\mathrm{CC}_{\mathbb{N}}$ defined in [6] are the following:

- The [APP] rule has been split into two rules: [$\mathrm{APP}^{\mathscr{S}}$] and [$\mathrm{APP}^{\mathscr{W}}$]. Conversion for strong terms is restricted to $\beta\iota$-conversion.
- Conversion for the first argument of an Elim is restricted to $\beta\iota$-conversion.
- The rules for transitivity and symmetry have been removed, which eases the proofs, notably that the deduction part of the conversion relation works at object level only. We prove later that the conversion relation is transitive and symmetric on well formed terms, thus recovering type unicity.
- The rules for $\beta\iota$-conversion perform one reduction step only, which also eases proofs. Therefore $u \xleftrightarrow{\beta\iota}_* v$ should be understood as $\exists w$ s.t. $u \xrightarrow{\beta\iota} w$ and $v \xrightarrow{\beta\iota} w$.

$$\frac{}{t \sim_\Gamma t}\,[\textsc{Refl}] \qquad \frac{[x :^{\mathrm{r}} T] \in \Gamma \quad T \xrightarrow{\beta\iota}_* t \doteq u \quad t,u \in \mathscr{O}^+}{t \sim_\Gamma u}\,[\textsc{Eq}]$$

$$\frac{T \sim_\Gamma U \quad t \sim_{\Gamma,[x:^a T]} u}{\lambda[x :^a T].t \sim_\Gamma \lambda[x :^a U].u}\,[\textsc{Lam}] \qquad \frac{T \sim_\Gamma U \quad t \sim_{\Gamma,[x:^a T]} u}{\forall(x :^a T).t \sim_\Gamma \forall(x :^a U).u}\,[\textsc{Prod}]$$

$$\frac{t \xrightarrow{\beta\iota} t' \quad t' \sim_\Gamma u}{t \sim_\Gamma u}\,[\beta\iota\text{-}\textsc{Left}] \qquad \frac{\begin{array}{c} t,t',f,f' \text{ are weak} \\ t \xleftrightarrow{\beta\iota}_* t' \quad I \sim_\Gamma I' \quad Q \sim_\Gamma Q' \quad \overline{v} \sim_\Gamma \overline{v}' \quad \overline{f} \sim_\Gamma \overline{f}' \end{array}}{\mathrm{Elim}(t : I[\overline{v}] \to Q)\{\overline{f}\} \sim_\Gamma \mathrm{Elim}(t' : I'[\overline{v}'] \to Q')\{\overline{f}'\}}$$

$$\frac{u \xrightarrow{\beta\iota} u' \quad t \sim_\Gamma u'}{t \sim_\Gamma u}\,[\beta\iota\text{-}\textsc{Right}] \qquad \frac{t_1 \sim_\Gamma u_1 \quad t_2 \sim_\Gamma u_2 \quad t_i, u_i \text{ are weak}}{t_1 t_2 \sim_\Gamma u_1 u_2}\,[\textsc{App}^{\mathscr{W}}]$$

$$\frac{\begin{array}{c} E \models \mathscr{A}_{\sim_\Gamma}(t)(\tau) = \mathscr{A}_{\sim_\Gamma}(u)(\tau) \quad t,u \in \mathscr{O}^+ \\ E = \{\mathscr{A}_{\sim_\Gamma}(w_1)(\sigma) = \mathscr{A}_{\sim_\Gamma}(w_2)(\sigma) \\ \mid w_1 \sim_\Gamma w_2, \sigma \in \Lambda, w_1, w_2 \in \mathscr{O}^+\} \end{array}}{t \sim_\Gamma u \quad [\Gamma, t, u]}\,[\textsc{Ded}]$$

Fig. 3 CCIC conversion relation

2.6 Decidability of type-checking

CCIC enjoys all needed meta-theoretical properties (strong normalization, confluence, subject reduction), and therefore consistency follows:

Theorem 1. *There is no proof of* $\forall(x : \star).x$ *in the empty environment.*

All proofs are similar to those made for PTSs with the same succession of meta-theoretical lemmas, but need more preparation. This is in particular the case with the substitution lemma which is much harder than usual.

As said, type-checking in a dependent type theory is non-trivial, since the rule [CONV] is not syntax-oriented. The classical solution to this problem is to eliminate [CONV] and replace [APP] by the following rule.The proof is not difficult.

$$\frac{\begin{array}{c}\Gamma \vdash t : \forall(x :^a U).V \quad \Gamma \vdash u : U' \quad U \sim_\Gamma U' \\ \text{if } a = \mathrm{r} \text{ and } U \xrightarrow{\beta}_* t_1 \doteq_T t_2 \text{ with } t_1, t_2 \in \mathscr{O} \text{then } t_1 \sim_\Gamma t_2 \text{ must hold}\end{array}}{\Gamma \vdash t\,u : V\{x \mapsto u\}}\,[\textsc{App}]$$

Decidability of type-checking in *CCIC* therefore reduces to decidability of $\sim_\Gamma$, the environment Γ being arbitrary, possibly containing ill-formed terms or even being inconsistent. To show that $\sim_\Gamma$ is decidable, we proceed as previously, by modifying the definition in order to make it syntax-oriented: we show that two arbitrary terms are convertible iff their $\beta\iota$-normal forms are convertible by the syntax-oriented *weak convertibility* relation $\approx_\Gamma$ given at Figure 4, in which, to any environment Γ, we associate the set $\mathrm{Eq}(\Gamma) = \{t = u \mid [x :^{\mathrm{u}} T] \in \Gamma, x\Gamma \to_* t \doteq u, t, u \in \mathscr{A}\}$.

Lemma 1. *Given Γ an environment and t,u two terms, $t \sim_\Gamma u$ iff $t\downarrow_{\beta\iota} \approx_\Gamma u\downarrow_{\beta\iota}$.*

This is the main technical result of the decidability proof, which proceeds by induction on the definition of $\sim_\Gamma$. Note that the numerous conditions of the form $\mathscr{T},\mathrm{Eq}(\Gamma) \not\models 0 = 1$ in the rules defining $\approx_\Gamma$ are required to make them mutually exclusive.

$$\frac{}{\star \approx_\Gamma \star}\,[\textsc{Refl-}\star] \qquad \frac{}{\square \approx_\Gamma \square}\,[\textsc{Refl-}\square] \qquad \frac{x \in \mathscr{X} \quad \mathscr{T},\mathrm{Eq}(\Gamma) \not\models 0=1 \text{ or } x \notin \mathscr{X}^\star}{x \approx_\Gamma x}\,[\textsc{Refl-}\mathscr{X}]$$

$$\frac{t,u \in \mathscr{O} \quad \mathscr{T},\mathrm{Eq}(\Gamma) \models 0=1}{t \approx_\Gamma u}\,[\textsc{Unsat}] \qquad \frac{\begin{array}{c} T \approx_\Gamma U \quad t \approx_{\Gamma,[x:^a T]} u \\ \mathscr{T},\mathrm{Eq}(\Gamma) \not\models 0=1 \text{ or} \\ \lambda[x :^a T].t \text{ and } \lambda[x :^a U].u \text{ not in } \mathscr{O} \end{array}}{\lambda[x :^a T].t \approx_\Gamma \lambda[x :^a U].u}\,[\textsc{Lam}]$$

$$\frac{T \approx_\Gamma U \quad t \approx_{\Gamma,[x:^a T]} u}{\forall(x :^a T).t \approx_\Gamma \forall(x :^a U).u}\,[\textsc{Prod}] \qquad \frac{\begin{array}{c} t = t' \quad I \approx_\Gamma I' \quad Q \approx_\Gamma Q' \quad \overline{v} \approx_\Gamma \overline{v}' \quad \overline{f} \approx_\Gamma \overline{f}' \\ t,t',\overline{f},\overline{f}' \text{ are weak } \mathscr{T},\mathrm{Eq}(\Gamma) \not\models 0=1 \text{ or} \\ \mathrm{Elim}(t,\ldots)\{\ldots\} \text{ and } \mathrm{Elim}(t',\ldots)\{\cdots\} \text{ not in } \mathscr{O} \end{array}}{\mathrm{Elim}(t : I[\overline{v}] \to Q)\{\overline{f}\} \approx_\Gamma \mathrm{Elim}(t' : I'[\overline{v}'] \to Q')\{\overline{f}'\}}\,[\mathscr{W}]$$

$$\frac{\begin{array}{c} t_1 \equiv u_1 \quad t_2 \equiv u_2 \\ \mathscr{T},\mathrm{Eq}(\Gamma) \not\models 0=1 \text{ or} \\ t_1\,t_2 \text{ and } u_1\,u_2 \text{ not in } \mathscr{O} \\ t_1\,t_2 \text{ or/and } u_1\,u_2 \text{ is not weak} \end{array}}{t_1\,t_2 \approx_\Gamma u_1\,u_2}\,[\textsc{App}^{\mathscr{S}}] \qquad \frac{\begin{array}{c} \mathscr{T},\mathrm{Eq}(\Gamma) \not\models 0=1) \\ t = C_t[a_1,\ldots,a_k] \quad u = C_u[a_{k+1},\ldots,a_{k+l}] \\ C_t \text{ or } C_u \text{ is a non-empty algebraic context} \\ \text{all the } a_i\text{'s have empty algebraic caps} \\ \text{the } c_i\text{'s are fresh constants s.t. } c_i = c_j \text{ iff } a_i \approx_\Gamma b_j \\ \mathscr{T},\mathrm{Eq}(\Gamma) \models C_t[c_1,\ldots,c_k] = C_u[c_{k+1},\ldots,c_{k+l}] \end{array}}{t \approx_\Gamma u}\,[\textsc{Ded}]$$

$$\frac{\begin{array}{c} t_1 \approx_\Gamma u_1 \quad t_2 \approx_\Gamma u_2 \quad t_i,u_i \text{ weak} \\ \mathscr{T},\mathrm{Eq}(\Gamma) \not\models 0=1 \text{ or} \\ t_1\,t_2 \text{ and } u_1\,u_2 \text{ not in } \mathscr{O} \end{array}}{t_1\,t_2 \approx_\Gamma u_1\,u_2}\,[\textsc{App}^{\mathscr{W}}]$$

Fig. 4 CCIC syntax-oriented conversion

Example 5. Let $\Gamma = [c : \mathbf{nat}],[p :^r (\lambda[x : \mathbf{nat}].x)\,\mathbf{0} \doteq c]$. Then $(\lambda[x : \mathbf{nat}].x \dot{+} x)\,\mathbf{0} \approx_\Gamma c$ and $(\lambda[x : \mathbf{nat}].x \dot{+} x)\,\mathbf{0} \approx_\Gamma c$, using congruence and deduction of $\sim_\Gamma$ and $\approx_\Gamma$.

In contrast, β-reducing $(\lambda[x : \mathbf{nat}].x \dot{+} x)\,\mathbf{0}$ yields $\mathbf{0} \dot{+} \mathbf{0} \sim_\Gamma c$, but not $\mathbf{0} \dot{+} \mathbf{0} \approx_\Gamma c$. Indeed, $(\lambda[x : \mathbf{nat}].x \dot{+} x)\,\mathbf{0}$ and $\mathbf{0} \dot{+} \mathbf{0}$ are no more $\approx_\Gamma$-convertible, a direct consequence of removing $\beta\iota$-reduction from $\sim_\Gamma$: the equation $(\lambda[x : \mathbf{nat}].x)\,\mathbf{0} \doteq c$ cannot be used anymore, since $\mathbf{0} \dot{+} \mathbf{0}$ is not $\approx_\Gamma$ convertible to $(\lambda[x : \mathbf{nat}].x)\,\mathbf{0}$).

Now, normalizing all terms as well as the environment Γ, we can recover convertibility for $\approx$: $\mathbf{0} \dot{+} \mathbf{0} \approx_{\Gamma\downarrow_{\beta\iota}} c$, the extractable equation of $\Gamma\downarrow_{\beta\iota}$ being now $\mathbf{0} \doteq c$.

As a consequence, we obtain:

Theorem 2. *$\sim_\Gamma$ is decidable for any environment Γ when taking for $\mathscr{O}^+$ the set of terms that are reducible to an algebraic terms.*

and therefore, our main result follows:

Theorem 3. *The type-checking relationship $\Gamma \vdash t : T$ is decidable in CCIC.*

3 Using CCIC

We give here a detailed example illustrating the advantages of CCIC, based on the inductive type of words introduced in Section 2.1.

In Coq. First, we give a development in Coq, therefore based on CIC.

```
Variable T : Set.

Inductive word : nat -> Set :=
| epsilon : word 0
| char : T -> word 1
| append : forall n p, word n -> word p -> word (n+p).

Lemma plus_n_0_transparent : forall n, n+0=n.
Proof. induction n as [| n IHn]; simpl;
  [idtac | rewrite -> IHn]; trivial. Defined.

Lemma plus_n_Sm_transparent: forall n m, n+(S m)=S(n+m).
Proof. intros n m; induction n as [| n IHn];
  simpl; [idtac | rewrite -> IHn]; trivial. Defined.

Lemma plus_assoc_transparent: forall n p q, (n+p)+q=n+(p+q).
Proof. intros n p q; elim n; [trivial | intros k].
  simpl; intros H; rewrite -> H; trivial. Defined.

Definition reverse_acc : forall n, word n -> forall p, word p -> word (p+n).
Proof. intros n wn; induction wn as [| c | n p wn IHwn wp IHwp];
  intros k wk. rewrite plus_n_0_transparent; exact wk.
  rewrite plus_n_Sm_transparent; rewrite plus_n_0_transparent;
    exact (append (char c) wk).
  rewrite <- plus_assoc_transparent; exact (IHwp _ (IHwn _ wk)). Defined.

Fixpoint reverse n (w : word n) {struct w} : word n :=
match w in word k return word k with
| epsilon => epsilon
| char c => char c
| append n1 n2 w1 w2 => reverse_acc w2 w1 end.
```

The example of *palindromes* as words satisfying the property `word_eq m reverse m` is carried out in Strub's thesis (see his website). It yields a much more complex Coq development than the above, since it involves the equality over (quotients) of words.

In CCIC. We now make the similar development in CCIC, using a self-explanatory syntax. The definition of `reverse` reduces then to:

```
Fixpoint reverse n (w : word n) {struct w} : word n := match w with
| epsilon          => epsilon
| char c           => char c
| append _ _ w1 w2 => append (reverse w2) (reverse w1) end.
```

Typing of the third clause of reverse will use here Presburger's arithmetic, since `append n1 n2 w1 w2` has type `word (n1 + n2)`, while `append n2 n1 w2 w1` has type `word (n2 + n1)`, two types that are not convertible in CIC, but which become convertible in CCIC. We can easily see with this example the immense benefit brought by internalizing Presburger's arithmetic. Note that a single certificate is generated for this conversion:

```
[n1 : nat, n2: nat, w1 : word n1, w2: word n2, n1 + n2, n2 + n1]
```

4 Conclusion

CCIC is an extension of CIC by arbitrary first-order decision procedures for equality. We have shown here with a detailed example using Presburger's arithmetic the benefit of the approach with respect to the current implementation of Coq based on CIC: more terms can be typed especially in presence of types such as dependent lists which become easy to use; many proofs get automated, making the life of the user easier (developing the example of reverse for dependent lists in the currently distributed version of Coq took us a day of work, and we don't believe this can be shrinked to one hour); and proofs are much smaller, some seemingly complex proofs becoming simple reflexivity proofs. We believe that the resulting style of proofs becomes much closer to that of the working mathematician.

We have also explained the advantage of the approach insofar as it allows to clearly separate computation from deduction, therefore allowing for an incremental development of the kernel of the system.

So far, we have considered only decidable -equality- theories. However, thanks to the decidability assumption, a decidable non-equality theory can always be transformed into a decidable equality theory over the type Bool of truth values equipped with its usual operations.

There are still many directions to be investigated. A first is to embed membership equational logic in CIC along the lines of the simpler embedding described here. A second is to consider the case of dependent algebras instead of the simpler parametric algebras. This is a much more difficult question, which requires using a stronger notion of conversion in the main argument of an elimination, but would further help us addressing other weaknesses of Coq.

Finally, we strongly believe that the use of decision procedures outputing certificates when they succeed and explanations when they fail will change our way of making formal, and enlarge the audience of proof assistants.

Acknowledgement. We thank the Coq group for many useful discussions and suggestions, and the referees for their useful remarks.

References

1. H. Barendregt. Lambda calculi with types. In S. Abramski, D. Gabba, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Oxford University Press, 1992.
2. B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, University of Paris VII, 1999.
3. F. Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005. Journal version of LICS'01.
4. F. Blanqui. Inductive types in the calculus of algebraic constructions. *Fundamenta Informaticae*, 65(1-2):61–86, 2005. Journal version of TLCA'03.
5. F. Blanqui, J.-P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions. In RTA, *Lecture Notes in Computer Science* 1631:301–316. Springer-Verlag, 1999.
6. F. Blanqui, J. Jouannaud, and P. Strub. Building decision procedures in the calculus of inductive constructions. In *Proceedings 16th CSL 2007. LNCS 4646*, 2007.
7. F. Blanqui, J.-P. Jouannaud, and P.-Y. Strub. A Calculus of Congruent Constructions. Unpublished draft, 2005.

8. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Comput. Sci.*, 236:35–132, 2000.
9. Coq-Development-Team. *The Coq Proof Assistant Reference Manual - Version 8.0*. INRIA, INRIA Rocquencourt, France, 2004. `http://coq.inria.fr/`.
10. T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988.
11. T. Coquand and C. Paulin-Mohring. Inductively defined types. *Colog'-88, International Conference on Computer Logic*, volume 417 of *LNCS*, pages 50–66. Springer-Verlag, 1990.
12. P. Corbineau. *Démonstration automatique en Théorie des Types*. PhD thesis, University of Paris IX, 2005.
13. K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. *Proceedings of 12th ACM Conference on Principles of Programming Languages*, 1985.
14. J. H. Geuvers and M. Nederhof. A modular proof of strong normalization for the calculus of constructions. *J. of Functional programming*, 1,2:155–189, 1991.
15. E. Giménez. Structural recursive definitions in type theory. In *Proceedings of ICALP'98*, volume 1443 of *LNCS*, pages 397–408, July 1998.
16. G. Gonthier. The four color theorem in Coq. In *TYPES 2004 International Workshop*, 2004.
17. N. Oury. Extensionality in the calculus of constructions. In *Proceedings 18th TPHOL, Oxford, UK. LNCS 3603*, 2005.
18. C. Paulin-Mohring. Inductive definitions in the system COQ. In *Typed Lambda Calculi and Applications*, pages 328–345. Springer Verlag, 1993. LNCS 664.
19. N. Shankar. Little engines of proof. In G. Plotkin, editor, *Proceedings of the Seventeenth Annual IEEE Symp. on Logic in Computer Science*. IEEE Computer Society Press, 2002.
20. R. E. Shostak. An efficient decision procedure for arithmetic with function symbols. *J. of the Association for Computing Machinery*, 26(2):351–360, 1979.
21. M. Stehr. The Open Calculus of Constructions: An equational type theory with dependent types for programming, specification, and interactive theorem proving (part I and II). *Fundamenta Informaticae* 68(1-2), p. 131-174, 2005.
22. T. Streicher. Investigations into intensional type theory, Habilitation, Münich University, 1993.
23. P.-Y. Strub. *The Calculus of Congruent Inductive Constructions*. PhD thesis, École Polytechnique, 2008.
24. B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, University Paris VII, 1994.