From Formal Proofs to Mathematical Proofs: A safe, incremental way for building in first-order decision procedures

Frédéric Blanqui
LORIA, UMR , Projet INRIA PROTHEO, Campus Scientifique
BP 239, 54506 Vandoeuvre-lès-Nancy Cedex, France
Jean-Pierre Jouannaud and Pierre-Yves Strub
LIX, UMR 7161, Project INRIA TypiCal
École Polytechnique, 91128 Plaiseau, FRANCE

Abstract

It is commonly agreed that the success of future proof assistants will rely on their ability to incorporate computations within deductions in order 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 this paper, we investigate a new version of the Calculus of Inductive Constructions (CIC) on which the proof assistant Coq is based: the Calculus of Congruent Inductive Constructions (CCIC) which incorporates arbitrary decision procedures into deductions via the conversion rule of the calculus. A major technical innovation of this work lies in the fact that goals are sent to the decision procedure together with the set of user hypotheses available from the current proof context. A major conceptual innovation of this work is that decision procedures are used as blackboxes, hence can be obtained 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 CIC'kernel.

A detailed example shows that the resulting style of proofs becomes closer to that of the working mathematician.

Keywords. *Calculus of Inductive Constructions, Decision procedures, Proof assistants*

1 Introduction

It is commonly agreed that the success of future proof assistants will rely on their ability to incorporate computations within deductions in order 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.

Proof assistants based on the Curry-Howard isomorphism such as Coq [8] allow to build the proof of a proposition by applying appropriate proof tactics generating 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 inference and deduction rules of the logic on top of a term manipulation layer. Trusting the kernel is vital since the mathematical correctness of a proof development relies entirely on the kernel.

The (intuitionist) logic on which Coq is based is the Calculus of Constructions (CC) of Coquand and Huet [9], an impredicative type theory incorporating polymorphism, dependent types and type constructors. As other logics, CC enjoys a computation mechanism called cut-elimination, which is nothing but the β-reduction rule of the underlying λ-calculus. But unlike logics without dependent types, CC enjoys also a powerful type-checking rule, called *conversion*, which incorporates computations within deductions, making decidability of type-checking a non-trivial property of the calculus.

The traditional view that computations coincide with β-reductions suffers several drawbacks. A methodological one is that the user must encode other forms of computations as deductions, which is usually done by using appropriate, complex tactics. A practical one is that proofs become much larger than necessary, up to a point that they cannot be type-checked anymore. These questions become extremely important when carrying out complex developments involving a large amount of computation as the formal proof of the four colour (now proof-checked) theorem completed by Gonthier and Werner using Coq [15].

The Calculus of Inductive Constructions of Coquand and Paulin was a first attempt to solve this problem by introducing inductive types and the associated elimination rules [10]. The recent versions of Coq are based on a slight generalization of this calculus [14]. Besides the β-reduction rule, they also include the so-called ι-reductions which are recursors for terms and types. While the kernel of CC is extremely compact and simple enough to make it easily readable -hence trustable-, the kernel of CIC is much larger and quite complex. Trusting it would require a formal proof, which was done once [2]. Updating that proof for each new release of the system is however unrealistic. CIC does not solve our problem, though, since such a simple function as *reverse* of a *dependent list* cannot be defined in CIC because $a :: l$ and $l :: a$, assuming :: is list concatenation and the element a can be coerced to a list of length 1, have non-convertible types $list(n+1)$ and $list(1+n)$.

A more general attempt was carried out since the early 90's, by adding user-defined computations as rewrite rules, resulting in the Calculus of Algebraic Constructions [3]. Although conceptually quite powerful, since CAC captures CIC [4], this paradigm does not yet fulfill all needs, because the set of user-defined rewrite rules must satisfy several strong assumptions. No implementation of CAC has indeed been released because making type-checking efficient would require compiling the user-defined rules, a complex task resulting in a kernel too large to be trusted anymore.

The proof assistant PVS uses a potentially stronger paradigm than Coq by combining its deduction mechanism[1] with a notion of computation based on the powerful Shostak's method for combining decision procedures [19], a framework dubbed *little proof engines* by Shankar [18]: the *little engines of proof* are the decision procedures, required to be convex, combined by Shostak's algorithm. A given decision procedure encodes a fixed set of axioms P. But an important advantage of the method is that the relevant assumptions A present in the context of the proof are also used by the decision procedure to prove a goal G, and become therefore part of the notion of computation. For example, in the case where the little proof engine is the congruence closure algorithm, the fixed set of axioms P is made of the axioms for equality, A is the set of algebraic ground equalities declared in the context, while the goal G is an equality $s = t$ between two ground expressions. The congruence closure algorithm will then process A and $s = t$ together in order to decide whether or not $s = t$ follows from $P \cup A$. In the Calculus of Constructions, this proof must be constructed by a specific tactic called by the user, which applies the inference rules of CC to the axioms in P and the assumptions in A, and becomes then part of the proof term being built. Reflexion techniques allow to omit checking this proof term by proving the decision procedure itself, but the soundness of the entire mechanism cannot be guaranteed [11].

[1]PVS logic is not based on Curry-Howard and proof-checking is not even decidable making both frameworks very different and difficult to compare.

Two further steps in the direction of integrating decision procedures into the Calculus of Constructions are Stehr's Open Calculus of Constructions OCC [20] and Oury's Extensional Calculus of Constructions [16]. Implemented in Maude, OCC allows for the use of an arbitrary equationnal 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 achived by adding the extensionality and Stricher's axiom to CIC, hence the name of this calculus. Unfortunately, strong normalization and decidability of type checking are lost in ECC (and OCC), which shows that we should seek for more restrictive extensions. In a preliminary work, we also designed a new, quite restrictive framework, the Calculus of Congruent Constructions (CCC), which incorporates the congruence closure algorithm in CC's conversion [6], while preserving the good properties of the calculus, including the decidability of type checking. Further, a preliminary version of this work was published at the *Computer Science Logic* conference last september, describing an instance of CCIC, named $\mathrm{CC}_{\mathbb{N}}$, in which the decision procedure was Presburger arithmetic.

Problem. The main question investigated in this paper is the incorporation of a general mechanism calling a decision procedure for solving conversion-goals in the Calculus of Inductive Constructions which uses the relevant information available from the current context of the proof.

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 main properties: confluency, strong normalization, coherence and decidability of proof-checking are all preserved. Unlike previous calculi, the main difficulties here are confluence, which led to a complex definition of conversion as a fixpoint, and decidability of type checking, which requires restricting the congruence below recursors. We refer to Strub's PhD thesis for more detail about the meta-theory of CCIC.

Pratical 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, a proof checker that processes all proofs built by a user with the help of tactics that are available from existing libraries or can otherwise be developped for achieving a specific task. In the early days of Coq, the safety of its proof checker relied on its small size and its clear structure reflecting the inference rules of the intuitionistic type theory, the Calculus of Constructions, on which it was based. The slogan was that of a *readable kernel*. Moving later to the Calculus of Inductive Constructions allowed to ease the specification tasks, making the system very popular among proof developpers, 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 Coq' kernel 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, but its correctness proof was of course 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 garantee 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 basic calculus, the Calculus of Inductive Constructions in our case, which should satisfy the *readable* or *provable kernel* paradigm; a collection of independant decision procedures producing proof certificates. These proof certificates can then either be checked by a certificate checker which should itself satisfy the *readable* or *provable code* paradigm, or be used to generate a proof than can then be checked by the kernel of the basic calculus.

This paradigm has many advantages. First, it allows for a modular development of the system, once the basic calculus is stabilised, as it is the case with the Calculus of Inductive Constructions on which Coq is based. 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. Last but not least, it allows the user herself to use whatever 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 Nelsson and Oppen on the one hand, by Shostak on the other hand, and has generated much work since then. We must therefore build in the combination mechanism, and there are three possibilities: in the kernel, via a certificate generator and checker again, or by reflection. We have not made this design decision yet, although the second possibility seems to better fit with the spirit of our paradigm.

We assume familiarity with typed lambda calculi [1] and the Calculus of Inductive Constructions [21, 4].

2 Congruent Inductive Constructions

The Calculus of Congruent of Inductive Constructions is an extension of the Calculus of Inductive Constructions which embeds in its conversion rule the validity entailment of a fixed first order theory. First, we recall the bsics of the Calculus of Inductive Constructions before to introduce parametric multi-sorted algebras and then embed these first-order algebras into the Calculus of Inductive Constructions. We are then able to define our Calculus of Congruent Inductive Constructions 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 [5].

2.1 Calculus of Inductive Constructions

Terms. We start our presentation by first describing the terms algebra of the Calculus of Inductive Constructions.

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

As usual, following the presentation of *Pure Type Systems* [13], we use two classes of variables: $\mathcal{X}^\star$ and $\mathcal{X}^\Box$ are countably infinite sets of *term variables* and *predicate variables*) such that $\mathcal{X}^\star$ and $\mathcal{X}^\Box$ are disjoint. We write $\mathcal{X}$ for $\mathcal{X}^\star \cup \mathcal{X}^\Box$.

We use the following notations:

s	ranges over	$\mathcal{S}$
$x, y, \ldots$	–	$\mathcal{X}$
$X, Y, \ldots$	–	$\mathcal{X}^\Box$

We can now define the terms algebra of CIC:

Definition 2.1 (Pseudo-terms). The algebra $\mathcal{L}$ of *pseudo-terms* of CIC is defined by:

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

As a preparation for the embedding of parametric multi-sorted algebras into CIC, we have given here the symbols in Σ as particular constants of the calculus.

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). If $t \in \mathcal{L}$, 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 *occurs freely* 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 decribe 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:

$$\begin{aligned} &\mathrm{Elim}\mathbb{N}(\mathbf{0}, Q)\{v_{\mathbf{0}}, v_{\mathbf{S}}\} \xrightarrow{\iota} v_{\mathbf{0}} \\ &\mathrm{Elim}\mathbb{N}(\mathbf{S}\, x, Q)\{v_{\mathbf{0}}, v_{\mathbf{S}}\} \xrightarrow{\iota} v_{\mathbf{S}}\, x (\mathrm{Elim}\mathbb{N}(x, Q)\{v_{\mathbf{0}}, v_N S\}) \\ &\quad \text{with } Q : \mathbf{nat} \to s \in \mathcal{S}. \end{aligned}$$

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 parameterized lists, of respective types $\forall(T:\star).\, \mathbf{list}(T)$ and $\forall(T:\star).\, T \to \mathbf{list}(T) \to \mathbf{list}(T)$, obtained by replacing X by $\mathbf{list}(T)$ in X and $T \to X \to X$, and then abstracting over the type $T : \star$.
Elimination rules for $\mathbf{list}$ are:

$$\begin{aligned} &\mathrm{Elim}\mathbb{L}(\mathbf{car}, Q)\{v_{\mathbf{car}}, v_{\mathbf{cdr}}\} \xrightarrow{\iota} v_{\mathbf{car}} \\ &\mathrm{Elim}\mathbb{L}(\mathbf{cons}\, x\, l\,, Q)\{v_{\mathbf{car}}, v_{\mathbf{cdr}}\} \xrightarrow{\iota} \\ &\quad v_{\mathbf{cdr}}\, x\, l\, (\mathrm{Elim}\mathbb{L}(l, Q)\{v_{\mathbf{car}}, v_c dr\}) \\ &\quad \text{with } Q : \lambda[T:\star].\, \mathbf{list}(T) \to s \in \mathcal{S}. \end{aligned}$$

Finally, we define dependent words over an alphabet A:

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

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:

$$\begin{aligned} &\mathrm{Elim}\mathbb{W}(\epsilon, Q)\{v_\epsilon, v_{\mathbf{char}}, v_{\mathbf{app}}\} \xrightarrow{\iota} v_\epsilon \\ &\mathrm{Elim}\mathbb{W}(\mathbf{char}\, x, Q)\{v_\epsilon, v_{\mathbf{char}}, v_{\mathbf{app}}\} \xrightarrow{\iota} v_{\mathbf{char}}\, x \\ &\mathrm{Elim}\mathbb{W}(\mathbf{app}\, n\, m\, l\, l', Q)\{v_\epsilon, v_{\mathbf{char}}, v_{\mathbf{app}}\} \xrightarrow{\iota} v_{\mathbf{app}}\, n\, m\, l\, l' \\ &\qquad (\mathrm{Elim}\mathbb{W}(l, Q)\{v_\epsilon, v_{\mathbf{char}}, v_{\mathbf{app}}\}) \\ &\qquad (\mathrm{Elim}\mathbb{W}(l', Q)\{v_\epsilon, v_{\mathbf{char}}, v_{\mathbf{app}}\}) \\ &\quad \text{with } Q = \forall(n : \mathbf{nat}).\, \mathbf{word}\, n \to s \in \mathcal{S} \end{aligned}$$

Definitions by induction. We can now define functions by induction over natural numbers or over words. Using the CIC syntax being quite painful, we give only a quite simple example used in the Coq development presented in Section 3.

$$\begin{aligned} &\mathrm{toTlist} := \lambda[n:\mathbf{nat}][w : \mathbf{word}\, n]. \\ &\quad \mathrm{Elim}\mathbb{W}(w, Q) \left\{ \begin{array}{l} \epsilon, \\ \lambda[c:T].\, \mathbf{char}\, c, \\ \lambda[n\, p : \mathbf{nat}][wn : \mathbf{word}\, n]. \\ \quad \lambda[wp : \mathbf{word}\, p][In : Q\, n\, wn]. \\ \qquad \lambda[Ip : Q\, p\, zp].\, append\, T\, In\, Ip \end{array} \right\} \end{aligned}$$

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 [21].

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 judgements 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.

Typing rules of CIC are made of two subsets: the typing rules for the Calculus of Constructions given at Figure 1, and the typing rules for inductive types, given at Figure 2 for the particular case of $\mathbf{nat}$ and $\mathbf{list}(\_)$.

It can be easily verified as an exercice that both sides of the first ι-reduction for $\mathbf{nat}$ have type $Q\ \mathbf{0}$ while they have type $Q(\mathbf{S}\ t)$ for the second.

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

$$\frac{}{\vdash \star : \Box}\ [\textsc{Ax-1}]$$

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

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

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

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

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

Figure 1. CIC **Typing Rules (**CC **rules)**

inductive types $\mathbf{nat}$, $\mathbf{list}$ and $\mathbf{word}$. We refer to [17, 21] for the general case, and for the precise typing rule of $\mathrm{Elim}\mathbb{W}$.

2.2 Parametric sorted algebras

Signature. Order-sorted algebras were introduced as a formal framework for the OBJ language in [12], before to be generalized as *membership equational logic* in [7]. We use here a polymorphic version of a restriction of the latter, by assuming given: a signature Λ of sort constructors; a signature Σ of function symbols made of a set of constructors for each sort constructor, the corresponding destructors and totally defined symbols for which equations are not given. Destructors should be thought of as beeing at the same time total (at the *universe* level of membership equational logic), and partial (by applying to non-empty lists). As an example, we describe natural numbers and parametric (non-dependent) list using an OBJ-like syntax.

$\mathbf{nat}$	:	$*$
$\mathbf{list}$	:	$* \to *$

$\mathbf{0}$		$\to \mathbf{nat}$
$\mathbf{S}$	$\mathbf{nat}$	$\to \mathbf{nat}$
$\dot{+}$	$\mathbf{nat} \times \mathbf{nat}$	$\to \mathbf{nat}$

$$\frac{\vdash \tau_f : s \in \mathcal{S}}{\vdash f : \tau_f}\ [\textsc{Symb}]$$

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

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

Figure 2. CIC **Typing Rules for** $\mathbf{nat}$ **and** $\mathbf{list}(\_)$

var	α	$*$
nil		$\to \mathbf{list}(\alpha)$
cons	$\alpha \times \mathbf{list}(\alpha)$	$\to \mathbf{list}(\alpha)$
car	$\mathbf{list}(\alpha)$	$\to \alpha$
cdr	$\mathbf{list}(\alpha)$	$\to \mathbf{list}(\alpha)$

We use the notation $f : \forall\overline{\alpha}.\, \sigma_1 \times \cdots \times \sigma_n \to \tau$ as examplified above, making the input sort more precise in case of a destructor. **car** and **cdr** appear immediately after the constructor **cons** of which they are the destructors. The fact that they should apply to non-empty lists remains implicit.

In the following, we shall use $\alpha, \beta, \ldots$ for sort variables and $\sigma, \tau, \ldots$ for sort expressions.

Terms, Equations.

Definition 2.2 (Terms). For any sort σ, let $\mathcal{X}^\sigma$ be a countably infinite set of *variables of sort* σ, s.t. all the $\mathcal{X}^\sigma$'s are pairwise disjoint. Let $\mathcal{X} = \bigcup_\sigma \mathcal{X}^\sigma$. For any $x \in \mathcal{X}$, we say that x has sort σ if $x \in \mathcal{X}^\sigma$.

For any sort σ, the set $\mathcal{T}_\sigma(\Sigma, \mathcal{X})$ of *terms of sorts* σ *with variables* $\mathcal{X}$ is the smallest set s.t.:

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

We denote by $\mathcal{T}(\Sigma, \mathcal{X})$ the set $\bigcup_\sigma(\mathcal{T}_\sigma(\Sigma, \mathcal{X}))$. A term t has sort σ if $t \in \mathcal{T}_\sigma(\Sigma, \mathcal{X})$.

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

Example 2.1. Assuming x is a variable of sort $\mathbf{nat}$, then 0 and $0 + x$ are of sort $\mathbf{nat}$, and $\mathbf{nil}$ is of sort $\mathbf{list}(\alpha)$ and $\mathbf{list}(\mathbf{nat})$.

Equations $t =^{\sigma} u$ are pairs of terms of the same sort σ. For example, if x is a variable of sort $\mathbf{nat}$, $x + 0 =^{\mathbf{nat}} x$ is an equation of sort $\mathbf{nat}$, and if l is of sort $\mathbf{list}(\mathbf{list}((\mathbf{nat}))$, then $cons(x, nil) =^{list(\mathbf{nat})} car(l)$ is an equation of sort $\mathbf{list}(\mathbf{nat})$. Type supersripts may be omitted when they can be infered from the context.

We can therefore as usual build parameterized algebras for $\mathbf{list}$, algebras for $\mathbf{nat}$ and therefore get algebras for $\mathbf{nat}$, $\mathbf{list}(\mathbf{nat})$, etc. Satisfaction of an equation in these algebras is defined as usual.

2.3 Embedding parametric algebras in CIC

Our purpose here is to embed our parametric muti-sorted algebra into CIC. As a result, two different, but related kinds of symbols will coexist, in the Calculus of Inductive Constructions, and in the embedded albegraic subworld. We shall distinguish them by underlying symbols in the Calculus of Inductive Constructions.

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

$\underline{\mathbf{nat}} := \mathrm{Ind}(X : \star)\{C_1(X), C_2(X)\}$.

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 $C_2(X) = X \to X$. As expected, we obtain here the standard inductive definition of natural numbers given in Section 2.1: $\mathrm{Ind}(X : \star)\{X, X \to X\}$. 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 $\mathbf{list}$ is not very different. Being of arity 1, with two associated constructor symbols ($\mathbf{nil}$ and $\mathbf{cons}$), $\mathbf{list}$ is mapped to the already seen parameterized inductive type $\underline{\mathbf{list}} = \lambda[A : T].\,\mathrm{Ind}(\star)\{X, A \to X \to X\}$. Translation of constructor types is done the same way. We just need to care about curryfication of symbols, and to replace sort variables with CIC types variables. For instance, the constructor types for $\mathbf{cons} : \forall\alpha.\,\alpha \times \mathbf{list}(\alpha) \to \mathbf{list}(\alpha)$ is $\underline{\mathbf{cons}} = \lambda[A : *].\,(\underline{\mathbf{list}}\,A)^{[2]}$.

The next step is the translation of destructors. They are simply mapped to CIC functions with a guard requiring that the input value has the right form. For this, we use the usual encoding - a generalisation of the product following Heyting's semantics - of the existential quantifier in CIC:

$$\begin{gathered}\dot{\exists} := \\ \lambda[A : \star][P : A \to \star].\,\mathrm{Ind}(X : \star)\{\forall(x : A).\,P\,x \to X\}\end{gathered}$$

Notation 2.1. We write $\exists(x : A).\,P$ for $\dot{\exists}\,A\,(\lambda[x : A].\,P)$, and $\exists_e^{A,P,Q}(t, f)$ (or $\exists_e(t, f)$ when A, P, Q can be deduced from the context) for the elimination principle associated to $\dot{\exists}$:

$$\exists_e^{A,P,Q}(t, f) = \mathrm{Elim}(t : \dot{\exists}\,A\,P\,[] \to Q)\{f\}$$

Returning to our example, the translation $\underline{\mathbf{car}}$ of $\mathbf{car}$ is defined as:

$$\begin{gathered}\underline{\mathbf{car}} := \lambda[T : \star][l : \underline{\mathbf{list}}\,T][\begin{array}{c} p : \exists(x : T)(l' : \underline{\mathbf{list}}\,T). \\ l = \mathbf{cons}\,T\,x\,l' \end{array}]. \\ \exists_e(p, \lambda[x : T][l' : \mathbf{list}\,T][p : \_].\,x)\end{gathered}$$

Defined symbols are mapped to CIC defined symbols.

2.4 Building in a first-order theory

We now start describing our new calculus CCIC.

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

Let $\mathcal{A} = \{\mathrm{r}, \mathrm{u}\}$ a set of two constants, called *annotations*, totally ordered by $\mathrm{u} \prec_{\mathcal{A}} \mathrm{r}$, 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 2.3 (Pseudo-terms of CCIC). The algebra $\mathcal{L}$ of *pseudo-terms* of CCIC is defined by:

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

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

Compared with CIC, the differences are:

- annotations in products and abstractions, we will soon see their use in the modified typing rules
- the internalization of the equality predicate:
  1. $t \doteq_T u$ (or $t \doteq u$ when T is not relevant or can deduced from the context) denotes the equality of the two terms (of type T) t and u,
  2. $\mathrm{Eq}_T(t)$ will represent a proof by reflexivity of $t \doteq_T t$.
- the internalization of the function symbols $f \in \Sigma$ and sort constructor $\sigma \in \Lambda$ of $\mathcal{T}$.

$$\begin{array}{lcl}
\mathcal{O} & ::= & \mathcal{X}^\star \mid f \in \Sigma \mid \mathcal{O}\,\mathcal{O} \mid \mathcal{O}\,\mathcal{P} \mid \lambda[x^\star :^a \mathcal{P}].\,\mathcal{O} \mid \\
 & & \lambda[x^\Box :^a \mathcal{K}].\,\mathcal{O} \mid \mathrm{Elim}(\mathcal{O} : \mathcal{P}\,[\overline{\mathcal{O}}] \to \mathcal{O})\{\overline{\mathcal{O}}\} \\
\mathcal{P} & ::= & \mathcal{X}^\Box \mid \sigma \in \Lambda \mid \mathcal{P}\,\mathcal{O} \mid \mathcal{P}\,\mathcal{P} \mid \lambda[x^\star :^a \mathcal{P}].\,\mathcal{P} \mid \\
 & & \lambda[x^\Box :^a \mathcal{K}].\,\mathcal{P} \mid \mathrm{Elim}(\mathcal{O} : \mathcal{P}\,[\overline{\mathcal{O}}] \to \mathcal{P})\{\overline{\mathcal{P}}\} \mid \\
 & & \forall(x^\star :^a \mathcal{P}).\,\mathcal{P} \mid \forall(x^\Box :^a \mathcal{K}).\,\mathcal{P} \\
\mathcal{K} & ::= & \star \mid \mathcal{K}\,\mathcal{O} \mid \mathcal{K}\,\mathcal{P} \mid \lambda[x^\star :^a \mathcal{P}].\,\mathcal{K} \mid \lambda[x^\Box :^a \mathcal{K}].\,\mathcal{K} \mid \\
 & & \forall(x^\star :^a \mathcal{P}).\,\mathcal{K} \mid \forall(x^\Box :^a \mathcal{K}).\,\mathcal{K} \\
\Box & ::= & \Box
\end{array}$$

Figure 3. CCIC **term classes**

$$\frac{\begin{array}{c}\Gamma \vdash t : \forall(x :^a U).\,V \quad \Gamma \vdash u : U \\ \text{if } a = \mathrm{r} \text{ and } U \xrightarrow{\beta}_* t_1 \doteq_{\mathcal{T}} t_2 \text{ with } t_1, t_2 \in \mathcal{O} \\ \text{then } t_1 \sim_\Gamma t_2 \text{ must hold}\end{array}}{\Gamma \vdash t\,u : V\{x \mapsto u\}}\ [\textsc{App}]$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T' : s' \quad T \sim_\Gamma T'}{\Gamma \vdash t : T'}\ [\textsc{Conv}]$$

Figure 4. CCIC **Modified Typing Rules**

Notation 2.2. When x in not free in t, $\forall(x :^a T).\,t$ will be written $T \to^a t$. The default annotation, when not specified in a product or abstraction, is the *unrestricted* (u) one.

As usual, it is possible to define a layered set of syntactic classes for $\mathcal{L}$:

Definition 2.4 (Syntactic classes). The pairwise disjoint syntactic classes of CCIC called *objects* ($\mathcal{O}$), *predicates* ($\mathcal{P}$), *kinds* ($\mathcal{K}$), $\Box$ are defined in Figure 3.

This enumeration defined a postfixed successors function $+1$ on classes ($\mathcal{O} + 1 = \mathcal{P}$, $\mathcal{P} + 1 = \mathcal{K}$, $\cdots$, $\Box + 1 = \bot$). We also define $\mathrm{Class}(t) = \mathcal{D}$ if $t\mathcal{D}$ and $\mathcal{D} \in \{\mathcal{O}, \mathcal{P}, \mathcal{K}, \Box\}$, and $\mathrm{Class}(t) = \bot$ otherwise.

From now on, we only consider *well-constructed terms* (i.e. terms whose class is not $\bot$) 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 $\xrightarrow{\beta}$-reduction preserves term classes.

Typing judgement.

Definition 2.5 (Pseudo-contexts of CCIC). The typing environments of CIC are defined as $\Gamma, \Delta ::= [] \mid \Gamma, [x :^a T]$ s.t. a variable cannot appear 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 judgement $\Gamma \vdash t : T$ are the same as for CIC except the rules for application and conversion given at Figure 4.

2.5 Conversion

We are now left to define our conversion relation $\sim_\Gamma$.

- Our notion of algebraisation will be relative to the expected sort of the resulting first-order term. This is the aim of our next section.
- The conversion $\sim_\Gamma$ 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.

We start with our new notion of algebraisation.

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 $\mathcal{T}$ in its conversion. For this last point, goals are sent to the first order theory $\mathcal{T}$ together with a set of proof hypotheses extracted from the current context.

Algebraisation is the first part of this hypotheses extraction: it allows transforming a CCIC term into its first-order counterpart. We illustrate this on a example for $\mathcal{T}$ 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 $\mathbf{nat}$. What is expected is that the set of hypotheses sent to the theory $\mathcal{T}$ contains the two well formed $\mathcal{T}$-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 $x \in \mathcal{X}_{\mathbb{N}}$. The *algebraisation* $\mathcal{A}(t)$ of an algebraic term is then defined by induction: $\mathcal{A}(0) = 0$, $\mathcal{A}(S\,t) = S(\mathcal{A}(t))$, $\mathcal{A}(t + u) = \mathcal{A}(t) + \mathcal{A}(u)$ and $\mathcal{A}(x_{\mathbb{N}}) = x_{\mathbb{N}}$,
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 $\mathcal{A}(t) = \mathcal{A}(u)$.

The definition becomes a little harder for parametric signatures. The theory of lists give us a canonical example. From the definition of conversion of a polymorphic multi-sorted algebra to CIC, we know that the $\mathbf{car}$ symbol has type

$$\forall (T:\star)(l:\mathbf{list}\,T).\,(\exists (x:T)(l':\mathbf{list}\,T).\,l \doteq \mathbf{cons}\,T\,x\,l) \\ \to T$$

Thus, a fully applied, well formed term having the symbol **car** at head position must be of the form $\mathbf{car}\,T\,l\,p$, T being the type of the elements of the list and p a proof that l is non-empty. Algebraisation of such a term will erase all type parameters and retraction proofs: in our example, $\mathcal{A}(\mathbf{car}\,T\,l\,p) = \mathbf{car}(\mathcal{A}(l))$.

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 $\mathcal{T}$ 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 $\mathbf{list}(\mathbf{nat})$?
- *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 ?

The solution adopted here is to postpone decisions: $\mathcal{A}(t)$ will be a function from Λ to the terms of $\mathcal{T}$ s.t. $\mathcal{A}(t)(\sigma)$ will be 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 $\mathcal{A}(\cdot)$.

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

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

Definition 2.6 (Well applied term). A term is well applied if it is of one of the following forms:

- $f\,[\overline{T_\alpha}]_{\alpha\in\alpha}\,t_1\,\cdots\,t_n$ with $f : \forall\underline{\alpha}.\,\sigma_1 \times \cdots \times \sigma_n \to \sigma$ and $f \in \Sigma - \Sigma^{\mathcal{D}}$.
- $f\,[\overline{T_\alpha}]_{\alpha\in\underline{\alpha}}\,t\,p$ with $f : \forall\underline{\alpha}.\,\sigma \to \tau$ and $f \in \Sigma^{\mathcal{D}}$.

Example 2.2. Example of well applied terms are 0, $S\,t$, or $\mathbf{car}\,T\,l\,p$ - T being the type parameter and p the destructor guard. Note that we do not require the term to be well formed.

Definition 2.7 (Algebraisation). Let a term $t \in$ CCIC and $\mathcal{R}$ an equivalence relation. The *algebraisation of t modulo $\mathcal{R}$* is the function $\mathcal{A}_{\mathcal{R}}(t) : \Lambda \to \mathcal{T}(\mathcal{X}^\star \cup \mathcal{Y})$ defined by:

$$\begin{array}{rcl} \mathcal{A}_{\mathcal{R}}(x_\sigma)(\sigma) & = & x_\sigma \\ \mathcal{A}_{\mathcal{R}}(f\,\overline{T}\,[u_i]_{i\in n}\,\overline{p})(\tau\xi) & = & \\ \multicolumn{3}{r}{f(\mathcal{A}_{\mathcal{R}}(u_1)(\sigma_1\xi),\ldots,\mathcal{A}_{\mathcal{R}}(u_n)(\sigma_n\xi)) \quad \text{if } (*)} \\ \mathcal{A}_{\mathcal{R}}(t)(\tau) & = & \Pi^\tau_{\mathcal{R}}(t) \quad \text{otherwise} \end{array}$$

where $(*)$

1. $f\,\overline{T}\,[u_i]_{i\in n}\,\overline{p}$ is well applied,
2. f is of arity $\forall\underline{\alpha}.\,\sigma_1 \times \cdots \sigma_n \to \sigma$,
3. ξ is a Λ-substitution.

For any relation R, $\mathcal{A}_R$ is defined as $\mathcal{A}_{\mathcal{R}}$ where $\mathcal{R}$ is the smallest equivalence relation containing R. We call σ-*alien* (or *alien* when the context is clear) the subterms of t abstracted by a variable in $\mathcal{Y}_\sigma$.

Example 2.3. Let $t \equiv \mathbf{cons}\,T\,\mathbf{0}\,(\mathbf{cons}\,U\,(\mathbf{nil}\,V)\,(\mathbf{nil}\,U))$ and R a relation on the terms of CCIC. Then, assuming $\sigma = \mathbf{list}(\mathbf{nat})$, we have ($x_{\mathbf{nat}}, y_{\mathbf{list}}, z_{\mathbf{nat}}$ being abstraction variables)

$$\begin{aligned} \mathcal{A}_R(t)(\sigma) &= \mathbf{cons}\left(\begin{array}{l}\mathcal{A}_R(\mathbf{0})(\mathbf{nat}),\\ \mathcal{A}_R(\mathbf{cons}\,U(\mathbf{nil}\,V)\,(\mathbf{nil}\,U))(\sigma)\end{array}\right) \\ &= \mathbf{cons}(0, \mathbf{cons}\left(\begin{array}{l}\mathcal{A}_R(\mathbf{nil}\,V)(\mathbf{nat}),\\ \mathcal{A}_R(\mathbf{nil}\,U)(\sigma)\end{array}\right)) \\ &= \mathbf{cons}(0, \mathbf{cons}(x_{\mathbf{nat}}, \mathbf{nil})) \end{aligned}$$

whereas

$$\begin{aligned} \mathcal{A}_R(t)(\mathbf{list}(\sigma)) &= \mathbf{cons}\left(\begin{array}{l}\mathcal{A}_R(\mathbf{0})(\sigma), \mathcal{A}_R(\mathbf{cons}\\ U(\mathbf{nil}\,V)\,(\mathbf{nil}\,U))(\mathbf{list}(\sigma))\end{array}\right) \\ &= \mathbf{cons}\left(\begin{array}{l}y_{\mathbf{list}}, \mathbf{cons}(\mathcal{A}_R(\mathbf{nil}\,V)(\sigma),\\ \mathcal{A}_R(\mathbf{nil}\,U)(\mathbf{list}(\sigma)))\end{array}\right) \\ &= \mathbf{cons}(y_{\mathbf{list}}, \mathbf{cons}(\mathbf{nil}, \mathbf{nil})) \end{aligned}$$

and $\mathcal{A}_R(t)(\mathbf{nat}) = z_{\mathbf{nat}}$.

Note that, as explained before, the algebraisation does not only depend on the terms being transformed, but also on the expected sort of the result. This is clearly visible on the example: when abstracting the (heterogeneous and ill-formed) list $0\ ::\ \mathbf{nil}\ ::\ \mathbf{nil}$ as a list of lists, 0 is then seen as an alien which must be abstracted. When this list is abstracted as a list of natural numbers, 0 is considered algebraic but **nil** is then seen as an alien and abstracted. Of course, as seen in the last case, if the list is algebraised as a natural number, it is directly abstracted by a variable.

Also, the algebraisation of the previous list w.r.t. the sort $\mathbf{list}(\alpha)$ results in the abstraction of all its elements (0 and **nil**):

$$\mathcal{A}_{\mathcal{I}}(t)(\mathbf{list}(\alpha)) = \mathbf{cons}(x_\alpha, \mathbf{cons}(y_\alpha, \mathbf{nil}))$$

Weak and neutral 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 what will be a non weak term is

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

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. We know that there exists 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 $\omega = \lambda[x : \mathbf{nat}].\, x\,x$.

On the contrary, *weak* terms are defined s.t. they cannot lift inconsistencies from object level to a higher level.

Definition 2.8 (Weak terms). A term is said weak if i) it does not contain an applied type level variable, and ii) it does not contain open strong elimination (i.e. does not contain a term of the form $\mathrm{Elim}^{\mathcal{S}}(t : I\,[\overline{u}] \to Q)\{\overline{f}\}$ with t open).

We also distinguish a subclass of weak terms, based on the same restriction, but for weak elimination. The notion will play a crucial role is the decidability of our calculus.

Definition 2.9 (Neutral terms). A term is said neutral if i) it does not contain an applied variable, ii) it does not contain open (strong or weak) elimitation.

Extractable terms. From now on, let $\mathcal{O}^+$ 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 $\mathcal{O}^+$ will be considered.

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

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

would require - in the conversion check algorithm - to decide any statement of the form

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

which is well known to be impossible.

Reduction. Our calculus contains the standard notion of $\beta\iota$-reduction of CIC, but in our case, we need to sightly modify the definition of ι-reduction in order to take into account the first order construction symbols by adding explicitely the corresponding elimination rules.

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

Definition 2.10 (Conversion relation). Rules of Figure 5 defines 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 datas, 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 [5] are the following:

- The [App] rule has been splited into two rules: [$\mathrm{App}^{\mathcal{S}}$] and [$\mathrm{App}^{\mathcal{W}}$]. Conversion for strong terms is restricted to the $\beta\iota$-conversion.
- Following the same restriction as for the [App] rule, conversion for terms being strongly destructed 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 now only do one reductions step, which also eases proofs. Therefore $u \xleftrightarrow{\beta\iota}_* v$ should be understood as $\exists w$ s.t. $u \xrightarrow{\beta\iota} w$ and $v \xrightarrow{\beta\iota} w$.

2.6 Metatheoretical properties

CCIC enjoys all metatheoretrical properties needed (strong normalization, confluence, subsject reduction, etc), so that

Theorem 2.1. *There is no proof of* $\forall(x : \star).\, x$ *in the empty environment.*

Theorem 2.2. *Assuming* $\mathcal{O}^+$ *to be the set of terms which are* $\beta\iota$*-convertible to an algebraic term, then* $\sim_\Gamma$ *is decidable for any environment* Γ.

$$\frac{}{t \sim_\Gamma t}\ [\textsc{Refl}]$$

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

$$\frac{\begin{array}{c} E \vDash \mathrm{cap}_{\sim_\Gamma}(t)(\tau) = \mathrm{cap}_{\sim_\Gamma}(u)(\tau) \quad t, u \in \mathcal{O} \\ E = \left\{ \begin{array}{c} \mathrm{cap}_{\sim_\Gamma}(w_1)(\sigma) = \mathrm{cap}_{\sim_\Gamma}(w_2)(\sigma) \mid \\ w_1 \sim_\Gamma w_2, \sigma \in \Lambda, w_1, w_2 \in \mathcal{O} \end{array} \right\} \end{array}}{t \sim_\Gamma u \quad [\Gamma, t, u]}\ [\textsc{Ded}]$$

$$\frac{t \xrightarrow{\beta\iota} t' \quad t' \sim_\Gamma u}{t \sim_\Gamma u}\ [\beta\iota\text{-}\textsc{Left}]$$

$$\frac{u \xrightarrow{\beta\iota} u' \quad t \sim_\Gamma u'}{t \sim_\Gamma u}\ [\beta\iota\text{-}\textsc{Right}]$$

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

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

$$\frac{t_1 \sim_\Gamma u_1 \quad t_2 \overset{\beta\iota}{\longleftrightarrow}_* u_2}{t_1\, t_2 \sim_\Gamma u_1\, u_2}\ [\textsc{App}^{\mathcal{S}}]$$

$$\frac{t_1 \sim_\Gamma u_1 \quad t_2 \overset{\beta\iota}{\longleftrightarrow}_* u_2 \quad t_1, u_1 \text{ are weak}}{t_1\, t_2 \sim_\Gamma u_1\, u_2}\ [\textsc{App}^{\mathcal{W}}]$$

$$\frac{t_1 \sim_\Gamma u_1 \quad t_2 \sim_\Gamma u_2 \quad t_1, u_1 \text{ are neutral}}{t_1\, t_2 \sim_\Gamma u_1\, u_2}\ [\textsc{App}^{\mathcal{N}}]$$

$$\frac{t \overset{\beta\iota}{\longleftrightarrow}_* t' \quad I \sim_\Gamma I' \quad Q \sim_\Gamma Q' \quad \overline{v} \sim_\Gamma \overline{v}' \quad \overline{f} \sim_\Gamma \overline{f}'}{\mathrm{Elim}^{\mathcal{S}}(t : I\,[\overline{v}] \to Q)\{\overline{f}\} \sim_\Gamma \mathrm{Elim}^{\mathcal{S}}(t' : I'\,[\overline{v}'] \to Q')\{\overline{f}'\}}\ [\mathrm{S}]$$

$$\frac{t \sim_\Gamma t' \quad I \sim_\Gamma I' \quad Q \sim_\Gamma Q' \quad \overline{v} \sim_\Gamma \overline{v}' \quad \overline{f} \sim_\Gamma \overline{f}'}{\mathrm{Elim}^{\mathcal{W}}(t : I\,[\overline{v}] \to Q)\{\overline{f}\} \sim_\Gamma \mathrm{Elim}^{\mathcal{S}}(t' : I'\,[\overline{v}'] \to Q')\{\overline{f}'\}}\ [\mathrm{W}]$$

Figure 5. CCIC **conversion relation**

See `http://strub.nu` for proofs (thesis draft).

There are other possibilities for $\mathcal{O}^+$ that we are still exploring, which, we expect, allow for $\beta\iota$-reductions in the first argument of weak-eliminations. This would allow to enhance pattern matching definitions in CCIC, a feature much desired by Coq users.

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. Indentations have been modified by hand to make it fit.

```
Variable T : Set .

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

(* Test *)
Require Import List .

Definition Tlist := (list T) .

Fixpoint to_Tlist (n : nat) (w : word n)
  {struct w} : Tlist
:= match w with
| epsilon => nil
| char c => c :: nil
| append n1 n2 w1 w2 => (to_Tlist _ w1)
                     ++ (to_Tlist _ w2)
end .
Implicit Arguments to_Tlist [n] .

Variables a b c : T .

Eval compute in (
  to_Tlist (reverse
    (append (append epsilon (char a))
            (append (char b) (char c))))
) .

(* Print c :: b :: a :: nil *)
```

The example of *palindroms* as words satisfying the property `Eqword m reverse m\verb` is carried out in Strub's thesis (see 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.

First, the parameterized algebra:

```
Variable T : Set .

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

Fixpoint reverse (n : nat) (w : word n)
    {struct w} : word n := match w with
| epsilon            => epsilon
| char c             => char c
| append n1 n2 w1 w2 => append n2 n1 w2
                               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)
```

We now introduce palidroms and prove that reverse is the identity on palindroms.

```
Variables M : Set .
Variables f : forall n, word n -> M .
Implicit Arguments f [n] .

Definition word_eq
   n1 n2 (w1 : word n1) (w2 : word n2)
   := (f w1) = (f w2) .
Implicit Arguments word_eq [n1 n2] .
Notation "w1 == w2" := (word_eq w1 w2)
   (at level 70, no associativity) .

Lemma reverse_invol :
   forall n (w : word n),
     reverse (reverse w) = w.
Proof .
   intros w; induction w; simpl; trivial .
   rewrite -> IHw1; rewrite -> IHw2 .
   trivial .
Qed .
```

```
Definition is_pal n (w : word n) :=
  w == (reverse w) .
Implicit Arguments is_pal [n] .

Lemma is_pal_reverse :
   forall n (w : word n),
     is_pal w -> is_pal (reverse w) .
Proof .
   intros n w H .
   unfold is_pal; unfold is_pal in H .
   unfold word_eq; unfold word_eq in H .
   rewrite <- H;
     rewrite -> (reverse_invol w) .
   trivial .
Qed .
```

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 types especially in presence of types such as dependent lists which become easy to use; many proofs become automated, making the life of the user easier (developping the example of reverse for dependent lists in Coq took us a day of work); and proofs become 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.

So far, we have considered only decidable equality theories. But it is well-known that 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. This is so because of the decidability assumption.

There are still many directions to be investigated. A first is to embedd 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 more difficult question, which requires using our generalized notion of conversion in the main argument of an elimination, but would further help us addressing other weaknesses of Coq. This question was already mentionned in Section 2.6.

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. Jouannaud, and P. Strub. Building decision procedures in the calculus of inductive constructions. In *Proceedings 16th CSL 2007, Lausanne, Switzerland. LNCS 4646*, 2007.

[6] F. Blanqui, J.-P. Jouannaud, and P.-Y. Strub. A Calculus of Congruent Constructions. Unpublished draft, 2005.

[7] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Comput. Sci.*, 236:35–132, 2000.

[8] Coq-Development-Team. *The Coq Proof Assistant Reference Manual - Version 8.0*. INRIA, INRIA Rocquencourt, France, 2004. At URL `http://coq.inria.fr/`.

[9] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[10] T. Coquand and C. Paulin-Mohring. Inductively defined types. In Martin-Löf and G. Mints, editors, *Colog'-88, International Conference on Computer Logic*, volume 417 of *LNCS*, pages 50–66. Springer-Verlag, 1990.

[11] P. Corbineau. *Démonstration automatique en Théorie des Types*. PhD thesis, University of Paris IX, 2005.

[12] K. Futatsugi, J. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In B. Reid, editor, *Proceedings of 12th ACM Conference on Principles of Programming Languages*. ACM, 1985.

[13] 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.

[14] E. Giménez. Structural recursive definitions in type theory. In *Proceedings of ICALP'98*, volume 1443 of *LNCS*, pages 397–408, July 1998.

[15] G. Gonthier. The four color theorem in coq. In *TYPES 2004 International Workshop*, 2004.

[16] N. Oury. Extensionality in the calculus of constructions. In *Proceedings 18th TPHOL, Oxford, UK. LNCS 3603*, 2005.

[17] C. Paulin-Mohring. Inductive definitions in the system COQ. In *Typed Lambda Calculi and Applications*, pages 328–345. Springer Verlag, 1993. LNCS 664.

[18] 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. Invited Talk.

[19] R. E. Shostak. An efficient decision procedure for arithmetic with function symbols. *J. of the Association for Computing Machinery*, 26(2):351–360, 1979.

[20] 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). *To appear in Fundamenta Informaticae*, 2007.

[21] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, University of Paris VII, 1994.