inria-00160586, version 2 - 9 Jul 2007

# Building Decision Procedures in the Calculus of Inductive Constructions

Frédéric Blanqui*, Jean-Pierre Jouannaud† and Pierre-Yves Strub†

**Abstract**

It is commonly agreed that the success of future proof assistants will rely on their ability to incorporate computations within deduction 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 which incorporates arbitrary decision procedures into deduction via the conversion rule of the calculus. The novelty of the problem in the context of the calculus of inductive constructions lies in the fact that the computation mechanism varies along proof-checking: 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 the calculus of constructions does not compromise its main properties: confluence, subject reduction, strong normalization and consistency are all preserved.

**Keywords.** Calculus of Inductive Constructions, Decision procedures, Theorem provers

## 1 Introduction

**Background.** It is commonly agreed that the success of future proof assistants will rely on their ability to incorporate computations within deduction 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 [9] 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.

[1]LORIA, UMR 7503 CNRS-INPL-INRIA-Nancy2-UHP, Equipe Protheo, Campus Scientifique, BP 239, 54506 Vandoeuvre-lès-Nancy Cedex, `blanqui@loria.fr`

[2]Projet LogiCal (Pôle Commun de Recherche en Informatique du Plateau de Saclay, CNRS, École Polytechnique, INRIA, Univ. Paris-Sud.), LIX, UMR CNRS 7161, École Polytechnique, 91128 Plaiseau, FRANCE, `{jouannaud,strub}@lix.polytechnique.fr`

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. As other logics, CC enjoys a computation mechanism called cut-elimination, which is nothing but the $\beta$-reduction rule of the underlying $\lambda$-calculus. But unlike logics without dependent types, CC enjoys also a powerful type-checking rule, called *conversion*, which incorporates computations within deduction, making decidability of type-checking a non-trivial property of the calculus.

The traditional view that computations coincide with $\beta$-reductions suffers several drawbacks. A methodological one is that the user must encode other forms of computations as deduction, 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 [14].

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 [11]. The recent versions of Coq are based on a slight generalization of this calculus [13]. Besides the $\beta$-reduction rule, they also include the so-called $\iota$-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 [3]. 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 [6]. Although conceptually quite powerful, since CAC captures CIC [7], 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 proof engines* 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 engines is the congruence closure algorithm, the fixed

[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.

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 [12].

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 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 axiom [15] 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 look 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 [8], while preserving the good properties of the calculus, including the decidability of type checking.

**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.

**Contributions.** Our main 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 deduction 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: confluence, strong normalization, coherence and decidability of proof-checking are all preserved. Unlike previous calculi, the main difficulty here is confluence, which led to a complex definition of conversion as a fixpoint. As a consequence of this definition, decidability of type checking becomes itself difficult.

Finally, we explain why the new system is still trustable, by leaving decision procedures *out* of its kernel, assuming that each procedure delivers a checkable *certificate* which becomes part of the proof. Certificate checkers become themselves part of the kernel, but are usually quite small and efficient and can be added one by one, making this approach a good compromise between CIC and the aforementioned extensions.

We assume some familiarity with typed lambda calculi [2] and the Calculus of Inductive Constructions.

## 2 The calculus

For ease of the presentation, we restrict ourselves to $\mathrm{CC}_{\mathbb{N}}$, a calculus of constructions with a type nat of natural numbers generated by its two constructors $\mathbf{0}$ and $\mathbf{S}$ and equipped with its weak recursor $\mathrm{Rec}_{\mathbb{N}}^{\mathcal{W}}$. The calculus is also equipped with a polymorphic equality symbol $\doteq$ for which we use here a mixfix notation, writing $t \doteq_T u$ (or even $t \doteq u$ when $T$ is not relevant) instead of $\doteq T\, t\, u$.

Let $\mathcal{S} = \{\star, \Box, \triangle\}$ the set of $\mathrm{CC}_{\mathbb{N}}$ *sorts*. For $s \in \{\star, \Box\}$, $\mathcal{X}^s$ denotes a countably infinite set of *s-sorted variables* s.t. $\mathcal{X}^\star \cap \mathcal{X}^\Box = \emptyset$. The union $\mathcal{X}^\star \cup \mathcal{X}^\Box$ will be written $\mathcal{X}$. For $x \in \mathcal{X}$, we write $s_x$ the sort of $x$. Let $\mathcal{A} = \{\mathbf{u}, \mathbf{r}\}$ a set of two constants called *annotations*, totally ordered by $\mathbf{u} \prec_{\mathcal{A}} \mathbf{r}$, where $\mathbf{r}$ stands for *restricted* and $\mathbf{u}$ for *unrestricted*. We use $a$ for an arbitrary annotation.

**Definition 2.1 (Pseudo-terms of $\mathrm{CC}_{\mathbb{N}}$).** *We define the* pseudo-terms *of* $\mathrm{CC}_{\mathbb{N}}$ *by the grammar rules:*

$$t, T := x \in \mathcal{X} \mid s \in \mathcal{S} \mid \mathrm{nat} \mid \doteq \mid \mathbf{0} \mid \mathbf{S} \mid \dot{+} \mid \mathrm{Eq}(t) \mid t\, u$$
$$\mid\ \lambda[x :^a T]t \mid \forall(x :^a T).\, t \mid \mathrm{Rec}_{\mathbb{N}}^{\mathcal{W}}(t, T)\{t_0, t_S\}$$

*We use* $\mathrm{FV}(t)$ *for the set of free variables of* $t$.

**Definition 2.2 (Pseudo-contexts of $\mathrm{CC}_{\mathbb{N}}$).** *The* typing environments *of* $\mathrm{CC}_{\mathbb{N}}$ *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* $\Gamma$ *and* $x\Gamma$ *for the type associated to* $x$ *in* $\Gamma$.

Remark that in our calculus, assumptions stored in the proof context always come along with an annotation used to control whether they can be used (in case the annotation is $\mathbf{r}$) or not in a conversion goal. We will later point out why this is necessary.

**Definition 2.3 (Syntactic classes).** *The pairwise disjoint syntactic classes of* $\mathrm{CC}_{\mathbb{N}}$*, called* objects *($\mathcal{O}$),* predicates *or* types *($\mathcal{P}$),* kinds *($\mathcal{K}$),* externs *($\mathcal{E}$) and* $\triangle$ *are defined in Figure 1.*

*This enumeration defines a postfixed successor function +1 on classes ($\mathcal{O} + 1 = \mathcal{P}$, $\mathcal{P} + 1 = \mathcal{K}$, ... $\Delta + 1 = \perp$) . We also define* $\mathrm{Class}(t) = \mathcal{D}$ *if* $t \in \mathcal{D}$ *and* $\mathcal{D} \in \{\mathcal{O}, \mathcal{P}, \mathcal{K}, \mathcal{E}, \triangle\}$ *and* $\mathrm{Class}(t) = \perp$ *otherwise.*

Our typing judgments are classically written $\Gamma \vdash t : T$, meaning that the *well formed term* $t$ is a proof of the proposition $T$ under the assumptions in the *well-formed* environment $\Gamma$. *Typing rules* are those of CIC restricted to the single inductive type of natural numbers, with one exception, [Conv], based on an equality relation called *conversion* defined in section 2.1.

**Definition 2.4 (Typing).** *Typing rules of* $\mathrm{CC}_{\mathbb{N}}$ *are defined in Figure 2.*

### 2.1 Computation by conversion

Our calculus has a complex notion of computation reflecting its rich structure made of three different ingredients, the typed lambda calculus, the type nat with its weak recursor and the Presburger arithmetic.

$$
\begin{aligned}
\mathcal{O} &:= \mathcal{X}^\star \mid \mathbf{0} \mid \mathbf{S} \mid \dot{+} \mid \mathcal{O}\,\mathcal{O} \mid \mathcal{O}\,\mathcal{P} \mid [\lambda\mathcal{X}^\star :^a \mathcal{P}]\mathcal{O} \mid \\
&:= [\lambda\mathcal{X}^\Box :^a \mathcal{K}]\mathcal{O} \mid \mathrm{Rec}_{\mathbb{N}}^{\mathcal{W}}(\mathcal{O}, \cdot)\{\mathcal{O}, \mathcal{O}\} \\
\mathcal{P} &:= \mathcal{X}^\Box \mid \mathrm{nat} \mid \mathcal{P}\,\mathcal{O} \mid \mathcal{P}\,\mathcal{P} \mid [\lambda\mathcal{X}^\star :^a \mathcal{P}]\mathcal{P} \mid \doteq \mid \\
&:= [\lambda\mathcal{X}^\Box :^a \mathcal{K}]\mathcal{P} \mid (\forall\mathcal{X}^\star :^a \mathcal{P})\mathcal{P} \mid (\forall\mathcal{X}^\Box :^a \mathcal{K})\mathcal{P} \\
\mathcal{K} &:= \star \mid \mathcal{K}\,\mathcal{O} \mid \mathcal{K}\,\mathcal{P} \mid [\lambda\mathcal{X}^\star :^a \mathcal{P}]\mathcal{K} \mid \\
&:= [\lambda\mathcal{X}^\Box :^a \mathcal{K}]\mathcal{K} \mid (\forall\mathcal{X}^\star :^a \mathcal{P})\mathcal{K} \mid (\forall\mathcal{X}^\Box :^a \mathcal{K})\mathcal{K} \\
\mathcal{E} &:= \Box \mid (\forall\mathcal{X}^\star :^a \mathcal{P})\mathcal{E} \mid (\forall\mathcal{X}^\Box :^a \mathcal{K})\mathcal{E} \\
\triangle &:= \triangle
\end{aligned}
$$

Figure 1: $\mathrm{CC}_{\mathbb{N}}$ terms classes

Our typed lambda calculus comes along with the $\beta$-rule. The $\eta$-rule raises known technical difficulties, see [22].

The type nat is generated by the two constructors $\mathbf{0}$ and $\mathbf{S}$ whose typing rules are given in Figure 2. We use $\mathrm{Rec}_{\mathbb{N}}^{\mathcal{W}}$ for its weak recursor whose typing rule is given in Figure 2 as well. Following CIC's tradition, we separate their arguments into two groups, using parentheses for the first two, and curly brackets for the two branches. The computation rules of nat are given below:

**Definition 2.5 ($\iota$-reduction).** *The $\iota$-reduction is defined by the following rewriting system:*

$$
\begin{aligned}
&\mathrm{Rec}_{\mathbb{N}}^{\mathcal{W}}(\mathbf{0}, Q)\{t_0, t_S\} &&\rightarrow_\iota t_0 \\
&\mathrm{Rec}_{\mathbb{N}}^{\mathcal{W}}(\mathbf{S}\,t, Q)\{t_0, t_S\} &&\rightarrow_\iota t_S\ t\ (\mathrm{Rec}_{\mathbb{N}}^{\mathcal{W}}(t, Q)\{t_0, t_S\})
\end{aligned}
$$

*where $t_0, t_S \in \mathcal{O}$.*

These rules are going to be part of the conversion $\sim_\Gamma$. Of course, we do not want to type-check terms at each single step of conversion, we want to type-check only the starting two terms forming the equality goal in [Conv]. But intermediate terms could then be non-typable and strong normalization be lost.

The constructors $\mathbf{0}$ and $\mathbf{S}$, as well as the additional first-order constant $\dot{+}$ are *also* used to build up expressions in the algebraic world of Presburger arithmetic, in which function symbols have arities. We therefore have two different possible views of terms of type nat, either as a term of the calculus of inductive constructions, or as an algebraic term of Presburger arithmetic. We now define precisely this algebraic world and explain in detail how to extract algebraic information from arbitrary terms of $\mathrm{CC}_{\mathbb{N}}$.

Let $\mathcal{T}$ be the theory of *Presburger arithmetic* defined on the signature $\Sigma = \{0, S(\_), \_+\_\}$ and $\mathcal{Y}$ a set of variables distinct from $\mathcal{X}$. Note that we syntactically distinguish the algebraic symbols from the $\mathrm{CC}_{\mathbb{N}}$ symbols by using a different font ($0$ and $S$ for the algebraic symbols, $\mathbf{0}$ and $\mathbf{S}$ for the constructors).

We write $\mathcal{T} \vDash F$ if $F$ is a valid formula in $\mathcal{T}$, and $\mathcal{T}, E \vDash F$ for $\mathcal{T} \vDash E \Rightarrow F$.

**Definition 2.6 (Algebraic terms).** *The set* **Alg** *of* $\mathrm{CC}_{\mathbb{N}}$ algebraic terms *is the smallest subset of $\mathcal{O}$ s.t. i) $\mathcal{X}^\star \subseteq$* **Alg***, ii) $\mathbf{0} \in$* **Alg***, iii) $\forall t \in \mathrm{CC}_{\mathbb{N}}.\, \mathbf{S}\,t \in$* **Alg***, iv) $\forall t, u \in \mathrm{CC}_{\mathbb{N}}.\, t \dot{+} u \in$* **Alg***.*

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

$$[\dot{=}\textsc{-Intro}]\ \frac{}{\vdash \dot{=} : \forall(T :^{u} \star).\, T \to T \to \star}$$

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

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

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

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

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

$$[\mathbf{0}\textsc{-Intro}]\ \frac{}{\vdash \mathbf{0} : \mathrm{nat}} \qquad [\mathbf{S}\textsc{-Intro}]\ \frac{}{\vdash \mathbf{S} : \mathrm{nat} \to \mathrm{nat}}$$

$$[\textsc{Nat}]\ \frac{}{\vdash \mathrm{nat} : \star} \qquad [\dot{+}\textsc{-Intro}]\ \frac{}{\vdash \dot{+} : \mathrm{nat} \to \mathrm{nat} \to \mathrm{nat}}$$

$$[\textsc{Eq-Intro}]\ \frac{\begin{array}{c}\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T \\ \Gamma \vdash p : \forall(P : T \to \star).\, P\, t_1 \to P\, t_2\end{array}}{\Gamma \vdash \mathrm{Eq}(p) : t_1 \doteq_T t_2}$$

$$[\iota\textsc{-Elim}]\ \frac{\begin{array}{c}\Gamma \vdash t : \mathrm{nat} \quad \Gamma \vdash Q : \mathrm{nat} \to \star \quad \Gamma \vdash f_0 : \mathrm{nat} \\ \Gamma \vdash f_S : \forall(n :^{\mathbf{u}} \mathrm{nat}).\, Q\, n \to Q\,(\mathbf{S}\, n)\end{array}}{\Gamma \vdash \mathrm{Rec}_{\mathbb{N}}^{\mathcal{W}}(t, Q)\{f_0, f_S\} : Q\, t}$$

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

Figure 2: Typing judgment of $CC_{\mathbb{N}}$

**Definition 2.7 (Algebraic cap and aliens).** *Given a relation $R$ on $\mathrm{CC}_{\mathbb{N}}$, let $\mathcal{R}$ be the smallest congruence on $\mathrm{CC}_{\mathbb{N}}$ containing $R$, and $\pi_R$ a function from $\mathrm{CC}_{\mathbb{N}}$ to $\mathcal{Y} \cup \mathcal{X}^\star$ such that $t \; \mathcal{R} \; u \iff \pi_R(t) = \pi_R(u)$.*

*The* algebraic cap of $t$ modulo $R$, $\mathrm{cap}_R(t)$, *is defined by:*

- $\mathrm{cap}_R(\mathbf{0}) = 0$, $\mathrm{cap}_R(\mathbf{S}\; u) = S(\mathrm{cap}_R(u))$, $\mathrm{cap}_R(u \dot{+} v) = \mathrm{cap}_R(u) + \mathrm{cap}_R(v)$,
- *otherwise,* $\mathrm{cap}_R(t) = t$ *if* $t \in \mathcal{X}^*$ *and else* $\pi_R(t)$.

*We call* aliens *the subterms of $t$ abstracted by a variable in $\mathcal{Y}$.*

Observe that a term not headed by an algebraic symbol is abstracted by a variable from our new set of variables $\mathcal{Y}$ in such a way that $\mathcal{R}$-equivalent terms are abstracted by the same variable.

We can now glue things together to define *conversion*.

**Definition 2.8 (Conversion relation).** *The family $\{\sim_\Gamma\}_\Gamma$ of* $\Gamma$-conversions *is defined by the rules of Figure 3.*

This definition is technically complex.

Being a congruence, $\sim_\Gamma$ includes congruence rules. However, all these rules are not quite congruence rules since crossing a binder increases the current context $\Gamma$ by the new assumption made inside the scope of the binding construct, resulting in a family of congruences. More questions are raised by the three different kinds of basic conversions.

First, $\sim_\Gamma$ includes the rules $\rightarrow_\beta$ and $\rightarrow_\iota$ of $\mathrm{CC}_{\mathbb{N}}$. Unlike the beta rule, $\rightarrow_\iota$ interacts with first-order rewriting, and therefore the Conv rule of Figure 2 cannot be expressed by $T \leftrightarrow^*_{\beta\iota} \sim_\Gamma \leftrightarrow^*_{\beta\iota} T'$ as one would expect.

Second, $\sim_\Gamma$ includes the relevant assumptions grabbed from the context, this is rule Eq. These assumptions must be of the form $[x :^{\mathbf{r}} T]$, with the appropriate annotation $r$, and $T$ must be an equality assumption or otherwise *reduce* to an equality assumption. Note that we use only $\rightarrow_\beta$ here. Using $\sim_\Gamma$ recursively instead is indeed an equivalent formulation under our assumptions. Without annotations, $\mathrm{CC}_{\mathbb{N}}$ does not enjoy subject reduction. Generating appropriate annotations is discussed in section 4.

Third, with rule [Ded], we can also generate new assumptions by using Presburger arithmetic. This rule here uses the property that two algebraic terms are equivalent in $\sim_\Gamma$ if their caps relative to $\sim_\Gamma$ are equivalent in $\sim_\Gamma$ (the converse being false). This is so because the abstraction function $\pi_{\sim_\Gamma}$ abstracts equivalent aliens by the same variable taken from $\mathcal{Y}$. It is therefore the case that deductions on caps made in Presburger arithmetic can be lifted to deductions on arbitrary terms via the abstraction function. As a consequence, the two definitions of the abstraction function $\pi_{\sim_\Gamma}$ and of the congruence $\sim_\Gamma$ are mutually inductive: our conversion relation is defined as a least fixpoint.

## 2.2 Two simple examples

**More automation - smaller proofs.** We start with a simple example illustrating how the equalities extracted from a context $\Gamma$ can be use to deduce new equalities in $\sim_\Gamma$.

$$[\beta\iota]\ \frac{t \leftrightarrow^*_{\beta\iota} u}{t \sim_\Gamma u} \qquad [\textsc{Eq}]\ \frac{[x :^{\mathbf{r}} T] \in \Gamma \quad T \rightarrow^*_\beta t_1 \doteq t_2 \quad t_1, t_2 \in \mathcal{O}}{t_1 \sim_\Gamma t_2}$$

$$[\textsc{Ded}]\ \frac{t_1, t_2 \in \mathcal{O} \quad \mathcal{T}, \{\mathrm{cap}_{\sim_\Gamma}(u_1) = \mathrm{cap}_{\sim_\Gamma}(u_2) \mid u_1 \sim_\Gamma u_2\} \models \mathrm{cap}_{\sim_\Gamma}(t_1) = \mathrm{cap}_{\sim_\Gamma}(t_2)}{t_1 \sim_\Gamma t_2}$$

$$[\textsc{Sym}]\ \frac{t \sim_\Gamma u}{u \sim_\Gamma t} \qquad [\textsc{Trans}]\ \frac{t \sim_\Gamma u \quad u \sim_\Gamma v}{t \sim_\Gamma v}$$

$$[\mathrm{CC}_{\mathbb{N}}\textsc{-Eq}]\ \frac{t \sim_\Gamma u}{\mathrm{Eq}(t) \sim_\Gamma \mathrm{Eq}(u)} \qquad [\textsc{App}]\ \frac{t_1 \sim_\Gamma t_2 \quad u_1 \sim_\Gamma u_2}{t_1\, u_1 \sim_\Gamma t_2\, u_2}$$

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

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

$$[\textsc{Elim-}\mathcal{W}]\ \frac{t \sim_\Gamma u \quad P \sim_\Gamma Q \quad t_0 \sim_\Gamma u_0 \quad t_S \sim_\Gamma u_S}{\mathrm{Rec}^{\mathcal{W}}_{\mathbb{N}}(t, P)\{t_0, t_S\} \sim_\Gamma \mathrm{Rec}^{\mathcal{W}}_{\mathbb{N}}(u, Q)\{u_0, u_S\}}$$

Figure 3: Conversion relation $\sim_\Gamma$

$$\begin{aligned}\Gamma = {} & [x\, y\, t :^{\mathbf{u}} \mathrm{nat}], [f :^{\mathbf{u}} \mathrm{nat} \rightarrow \mathrm{nat}],\\ & [p_1 :^{\mathbf{r}} t \doteq 2], [p_2 :^{\mathbf{r}} f\,(x \dot{+} 3) \doteq x \dot{+} 2],\\ & [p_3 :^{\mathbf{r}} f\,(y \dot{+} t) \dot{+} 2 \doteq y], [p_4 :^{\mathbf{r}} y \dot{+} 1 \doteq x \dot{+} 2]\end{aligned}$$

From $p_1$ and $p_4$ (extracted from the context by [Eq]), [Ded] will deduce that $y \dot{+} t \sim_\Gamma x \dot{+} 3$, and by congruence, $f\,(y \dot{+} t) \sim_\Gamma f\,(x \dot{+} 3)$. Therefore, $\pi_{\sim_\Gamma}$ will abstract $f(x \dot{+} 3)$ and $f(y \dot{+} t)$ by the same variable $z$, resulting in two new equations available for [Ded]: $z = x + 2$ and $z + 2 = y$. Now, $z = x + 2$, $z + 2 = y$ and $y + 1 = x + 2$ form a set of unsatisfiable equations and we deduce $0 \sim_\Gamma 1$ by the Ded rule: contradiction has been obtained. This shows that we can easily carry out a proof by contradiction in $\mathcal{T}$.

**More typable terms.** We continue with a second example showing that the new calculus can type more terms than CIC. For the sake of this example we assume that the calculus is extended by dependent lists on natural numbers. We denote by **list** (of type $\mathrm{nat} \rightarrow \star$) the type of dependent lists and by **nil** (of type $\mathbf{list}\,\mathbf{0}$) and **cons** (of type $\forall(n : \mathrm{nat}).\, \mathbf{list}\, n \rightarrow \mathrm{nat} \rightarrow \mathbf{list}\,(\mathbf{S}\, n)$) the lists constructors. We also add a weak recursor $\mathrm{Rec}^{\mathcal{W}}_{\mathbb{L}}$ such that, given $P : \forall(n : \mathrm{nat}).\, \mathbf{list}\, n \rightarrow \star$, $l_0 : P\,\mathbf{0}\,\mathbf{nil}$ and $l_S : \forall(n : \mathrm{nat})(l : \mathbf{list}\, n).\, P\, n\, l \rightarrow \forall(x : \mathrm{nat}).\, P\,(\mathbf{S}\, n)\,(\mathbf{cons}\, n\, x\, l)$, then $\mathrm{Rec}^{\mathcal{W}}_{\mathbb{L}}(l, P)\{l_0, l_S\}$ has type $P\, n\, l$ for any list $l$ of type $\mathbf{list}\, n$.

Assume now given a dependent reverse function (of type $\forall(n : \mathrm{nat}).\, \mathbf{list}\, n \rightarrow \mathbf{list}\, n$) and the list concatenation function @ (of type $\forall(n\, n' : \mathrm{nat}), \mathbf{list}\, n \rightarrow$

**list** $n' \to$ **list** $(n \dot{+} n')$). We can simply express that a list $l$ is a palindrome: $l$ is a palindrome if reverse $l \doteq l$.

Suppose now that one wants to prove that palindromes are closed under substitution of letters by palindromes. To make it easier, we will simply consider a particular case: the list $l_1 l_2 l_2 l_1$ is a palindrome if $l_1$ and $l_2$ are palindromes. The proof sketch is simple: it suffices to apply as many times as needed the lemma reverse$(ll')$ = reverse$(l')$@reverse$(l)$ $(*)$. What can be quite surprising is that Lemma $(*)$ is rejected by Coq. Indeed, if $l$ and $l'$ are of length $n$ and $n'$, it is easy to check that reverse$(ll')$ is of type **list** $(n \dot{+} n')$ and reverse$(l')$ :: reverse$(l)$ of type **list** $(n' \dot{+} n)$ which are clearly not $\beta\iota$-convertible. This is not true in our system: $n \dot{+} n'$ will of course be convertible to $n' \dot{+} n$ and lemma $(*)$ is therefore well-formed. Proving the more general property needs of course an additional induction on natural numbers to apply lemma $(*)$ the appropriate number of times, which can of course be carried out in our system.

Note that, although possible, writing a reverse function for dependent lists in Coq is not that simple. Indeed, a direct inductive definition of reverse will define reverse(**cons** $n\, a\, l$), of type **list** $(1 \dot{+} n)$, as reverse$(l)$ @ $a$, of type **list** $(n \dot{+} 1)$. Coq will reject such a definition since **list** $(1 \dot{+} n)$ and **list** $(n \dot{+} 1)$ are not convertible. Figure 4 shows how reverse can be defined in Coq.

```
Coq < Definition reverse: forall (n: nat), (list n) -> (list n) .
Coq <   assert (reverse_acc : forall (n m : nat),
Coq <             list n -> list m -> list (m+n)) .
Coq <   refine (fix reverse_acc (n m : nat) (from : list n) (to : list m)
Coq <             {struct from} : list (m+n) := _) .
Coq <   destruct from as [ | n' v rest ] .
Coq <     rewrite <- plus_n_O_transparent; exact to .
Coq <     rewrite <- plus_n_Sm_transparent;
Coq <       exact (reverse_acc n' (S m) rest (cons _ v to)) .
Coq <   intros n l . exact (reverse_acc _ _ l nil) .
Coq < Defined .
```

Figure 4: reverse function is Coq

# 3 Metatheorical properties

Most basic properties of Pure Type Systems (see [5]) are not too difficult. Those using substitution instances are more delicate. They rely on the annotations decorating the abstractions and products which were introduced for that purpose.

## 3.1 Stability by substitution

Assume that $\Gamma$ is a typing environment of the form $\Gamma_1, [p :^{\mathbf{r}} a \doteq b], \Gamma_2$ ($a$ and $b$ being two variables of type nat in $\Gamma$). The stability by substitution claims that if we have a typing derivation $\Gamma \vdash t : T$, then we can substitute $p$ by a term $P$ (of type $a \doteq b$ under $\Gamma_1$) in this derivation and obtain a proof of $\Gamma_1, \Gamma_2\theta \vdash t\theta : T\theta$, where $\theta$ is the substitution $\{p \mapsto P\}$. This property can easily be proved for Pure Type Systems as soon as the conversion relation is itself stable by substitution. In our example one can easily check that $a \sim_\Gamma b$, but $a \sim_{\Gamma_1,\Gamma_2\theta} b$ will not hold in general: the assumption $a \doteq b$ has been inlined and thus is no more extractable by the conversion relation. As a result, we need to strengthen the formulation of stability by substitution:

**Lemma 3.1.** *Let $\Gamma = \Gamma_1, [z :^a W], \Gamma_2$ and assume that i) $T \sim_\Gamma T'$, ii) if $a = \mathbf{r}$ and $W \to_\beta^* t_1 \doteq t_2$ then $t_1 \sim_{\Gamma_1} t_2$. Then, $T\theta \sim_\Delta T'\theta$ where $\theta = \{z \mapsto w\}$ and $\Delta = \Gamma_1, \Gamma_2\theta$*

**Corollary 3.2 (Stability by substitution).** *Let $\Gamma = \Gamma_1, [z :^a W], \Gamma_2$ and assume that i) $T \sim_\Gamma T'$ ii) if $a = \mathbf{r}$ and $W \to_\beta^* t_1 \doteq t_2$ then $t_1 \sim_{\Gamma_1} t_2$. Then, $\Delta \vdash t\theta : T\theta$ where $\theta = \{z \mapsto w\}$, $\Gamma_1 \vdash w : W$ and $\Delta = \Gamma_1, \Gamma_2\theta$.*

As usual, the substitutivity lemma is to be used in the proof of subject reduction (for $\to_{\beta\iota}$) to come later. Because it requires a specific typing property for the equality assumptions annotated by $\mathbf{r}$, we need to ensure this property in the application case of the coming subject reduction proof. This is indeed the origin of the similar condition arising in the typing rule [App].

## 3.2 Conversion as rewriting

We now turn conversion into a rewriting relation in order to prove that our system is logically consistent by analyzing a proof in normal form of $\forall(x :^{\mathbf{u}} \star).\, x$. The notion of a normal proof is of course more complicated than in CIC, since we must account for the congruence $\sim_\Gamma$ associated with an arbitrary context $\Gamma$. The difficulty is that the set of equalities assumed in a given environment $\Gamma$ together with the axioms of the theory $\mathcal{T}$ may be inconsistent, making all first-order terms equal in $\sim_\Gamma$ which could break strong normalization of our rewriting relation.

**Definition 3.3 ($\mathcal{T}$-consistent environment).** *A typing environment $\Gamma$ is* $\mathcal{T}$-consistent *if there exist two terms $t, u \in \mathcal{O}$ s.t. $\neg(t \sim_\Gamma u)$.*

**Lemma 3.4.** *If $\Gamma$ is $\mathcal{T}$-consistent then $\neg(\mathbf{0} \sim_\Gamma \mathbf{S}\, t)$ for any term $t$.*

**Definition 3.5 (Weak conversion).** *We inductively define a family of* weak conversion relations *$\{\cong_\Gamma\}_\Gamma$ as the smallest congruent relation s.t. $t \cong_\Gamma u$ if $\mathcal{T}, \mathrm{Eq}(\Gamma) \vDash \mathrm{cap}_\emptyset(t) = \mathrm{cap}_\emptyset(u)$, where $\mathrm{Eq}(\Gamma) = \{\mathrm{cap}_\emptyset(w_1) = \mathrm{cap}_\emptyset(w_2) \mid w_1, w_2 \in \mathcal{O}, [x :^{\mathbf{r}} w_1 \doteq w_2] \in \Gamma\}$.*

**Definition 3.6.** *We inductively define a family $\{\to_\Gamma\}_\Gamma$ of rewriting relations modulo weak-conversion as the smallest rewriting relations satisfying the rules of Figure 5.*

The first rule shows that rewriting is modulo weak conversion in a consistent environment. The second equates all object terms when the environment is inconsistent, replacing them by the new constant $\bullet$. The others are as expected.

**Lemma 3.7.** *1. The rewriting relation $\to_\Gamma$ is confluent.*

*2. If $t \sim_\Gamma u$ then $t \leftrightarrow_\Gamma^* u$.*

*3. If $t \leftrightarrow_\Gamma^* u$ with $\bullet \notin t$ and $\bullet \notin u$ then $t \sim_\Gamma u$.*

*4. If $\Gamma \vdash t : T$ with $\Gamma$ $\mathcal{T}$-consistent and $t \cong_\Gamma u$, then $\Gamma \vdash u : T$.*

$$[\text{Rw-Mod}]\ \frac{\Gamma \text{ is } \mathcal{T}\text{-consistent} \quad t \cong_\Gamma t' \rightarrow_\Gamma u' \cong_\Gamma u}{t \rightarrow_\Gamma u}$$

$$[\text{Rw-}\bullet]\ \frac{\Gamma \text{ is } \mathcal{T}\text{-inconsistent} \quad t \in \mathcal{O} \quad t \neq \bullet}{t \rightarrow_\Gamma \bullet}$$

$$[\text{Rw-}\beta\iota]\ \frac{t \rightarrow_{\beta\iota} u}{t \rightarrow_\Gamma u} \qquad [\text{Rw-Fwd}]\ \frac{t \rightarrow_\Delta u \quad \Gamma \rightarrow_\beta \Delta}{t \rightarrow_\Gamma u}$$

$$[\text{W-}\forall]\ \frac{t \rightarrow_{\Gamma,[x:^a T]} u \quad b \preceq a}{\forall(x :^b T).\, t \rightarrow_\Gamma \forall(x :^b T).\, u}$$

$$[\text{W-}\lambda]\ \frac{t \rightarrow_{\Gamma,[x:^a T]} u \quad b \preceq a}{\lambda[x :^b T].\, t \rightarrow_\Gamma \lambda[x :^b T].\, u}$$

Figure 5: Conversion as a rewriting system

**Lemma 3.8.** *If $\Gamma \vdash t : T$ and $t \rightarrow_\Gamma u$ with $\bullet \notin u$, then $\Gamma \vdash u : T$.*

*Proof.* The proof is standard, by induction on the type derivation of the left-hand side. The interesting case is when a $\beta$-reduction applies to the top of a term of the form $(\lambda[x :^a U]v)\ w$ and the typing rule is [App]: we then conclude by using Lemma 3.2. Note that the side condition of rule [App] provides us with the property needed for using Lemma 3.2. ☐

**Lemma 3.9.** *The rewriting relation $\rightarrow_\Gamma$ is strongly normalizing for well formed terms.*

*Proof.* The proof is a direct application of proof irrelevance [4], because $\sim_\Gamma$ is a congruence generated by equalities between object terms, apart from beta-reduction. What makes this true is that $\mathrm{Rec}_{\mathbb{N}}^{\mathcal{W}}$ is a weak recursor, working at the object level. Including strong elimination rules invalidates this argument. ☐ ☐

We finally conclude that $\mathrm{CC}_{\mathbb{N}}$ is consistent:

**Theorem 3.1.** *There is no proof of $\vdash t : \forall(x :^{\mathbf{u}} \star).\, x$.*

*Proof.* Assume that $\vdash t : \mathbf{0} \doteq \mathbf{S}\,\mathbf{0}$ where $t$ is $\rightarrow_\Gamma$-normal. Since $\mathbf{0} \doteq \mathbf{S}\,\mathbf{0}$ is not convertible to a sort, $t$ cannot be equal to nat, or a sort, or a product. Since $t$ is necessarily closed, $t$ is not a variable. Moreover, $t$ cannot be of the form $\mathrm{Rec}_{\mathbb{N}}^{\mathcal{W}}(u, Q)\{t_0, t_S\}$ since $t$ is closed and in $\rightarrow_\iota$-normal form.

If $t$ is an application, it is necessarily of the form $c\,\vec{u}$ with $c \in \{\mathbf{0}, \mathbf{S}, \dot{+}, \doteq\}$. By using inversion it suffices to check that in all these cases, $t$ has a type $T$ which is not convertible to $\mathbf{0} \doteq \mathbf{S}\,\mathbf{0}$.

If $t = \mathrm{Eq}(u)$, then $t$ has type $u \doteq u$ with $u$ of type nat and $u \doteq u$ convertible to $\mathbf{0} \doteq \mathbf{S}\,\mathbf{0}$. Thus $\mathbf{0} \sim_{[]} \mathbf{S}\,\mathbf{0}$, and $\mathcal{T} \vDash 0 = 1$, which is impossible. ☐ ☐

### 3.3 Decidability of type checking

**Theorem 3.2.** *Type checking of* $CC_{\mathbb{N}}$ *is decidable.*

Decidability of type checking needs two ingredients. First-of-all, eliminating [CONV], which is non-structural, by incorporating it to [APP]. This is classical, and it is easy to prove decidability of the transformed set of rules for type-checking, assuming $\sim_\Gamma$ is decidable.

Deciding $\sim_\Gamma$ is more complex. We cannot use the rewrite system $\rightarrow_\Gamma$ for that purpose since the first two rules use the $\mathcal{T}$-consistency of $\Gamma$ as a prerequisite. We use instead a saturation based algorithm. The method resembles very much the one used for combining first-order decision procedures operating on disjoint alphabets [17, 1]. Basic ingredients are: purification of formulas (here equations) by abstracting aliens by new variables; deriving new equalities among variables by using the appropriate decision procedure for pure formulas; propagating these new equalities to the other formulas.

# 4 Conclusion and discussion

$CC_{\mathbb{N}}$ is an extension of CIC (restricted to the weak elimination rules of the inductive type nat) by a fragment of Presburger arithmetic (without the natural strict order $\mathbb{N}$) in which conversion incorporates Presburger arithmetic, $\beta$-reduction and higher-order primitive recursion into a single mechanism. We now discuss in more details how this can be generalized to full CIC, how this can be used in practice, how useful that is, and whether the obtained kernel is trustable.

**Relevance.** Our second example shows very clearly the expressivity of our calculus with respect to CIC. However, what is done here by a typing rule could be done alternatively in CIC by a tactic. Besides, if one wants to avoid building a proof term which can be quite large and slow down the type-checker, it is possible to prove the tactic and then use a reflexion mechanism in order to avoid type-checking the proof each time the tactic is called. In both cases, however, the user must call the tactic explicitly. In our approach, this is completely transparent, and would remain transparent in case of a succession of uses of the decision procedure separated by eliminations, since conversion incorporates both, or in case of different decision procedures called successively.

**Extension to CIC.** Building decision procedures in a type-theoretic framework is not that easy. The main difficulty lies in the adequate definition of the congruence $\sim_\Gamma$. Once the definition is obtained, carrying out the technical development is not too difficult in the case of the pure Calculus of Constructions (the congruence becomes quite simpler in this case), difficult in the present case of $CC_{\mathbb{N}}$ (because of the presence of the weak recursors for nat), no more difficult when other decidable theories are introduced such as lists with their associated recursors, but much harder when including strong elimination rules which interact with the first-order theories. In this case, it is necessary to block the congruence below the strong recursor in order to avoid lifting an incoherence from the object level to the predicate level, which would immediately yield paradoxes [21].

**Annotations restriction** One may wonder how annotations can be handled in practice. As seen, annotations are used to forbid *inlining* (when a $\beta$-redex is contracted) of equational assumptions which are used by conversion. This could be seen as a restriction since our calculus, in order to avoid the creation of problematic $\beta$-redexes, forbids in most cases applications of symbols of type $\forall(p :^{\mathbf{r}} t \doteq u). T$.

This restriction can be removed by using the notion of *opaque definitions* (as opposed to *transparent definitions*) of Coq which allows the user to define symbols that the system cannot inline. In most cases, definitions having a computational behavior (like $\dot{+}$) are transparent whereas definitions representing lemmas (like the associativity of $\dot{+}$) are opaque. This convention is used in the standard library of Coq.

Returning to our previous example, if the user needs to prove a lemma of the form $\forall(p :^{\mathbf{r}} t \doteq u). T$, he or she should declare it as an opaque definition $P := \lambda[p :^{\mathbf{r}} t \doteq u]q$. The application of $P$ to a term $v$ should then be allowed: the term $P\,v$ cannot reduce to $q\{p \mapsto v\}$. Of course, if $P$ is defined transparently, the application $P\,v$ has to be forbidden.

Moreover, this gives us a simple heuristic to automatically tag products and abstractions: $\mathbf{r}$ annotation should by used by default when the user is defining an opaque symbol, whereas $\mathbf{u}$ annotation should be used everywhere else.

**Arbitrary decision procedures.** So far, we have considered only decidable equational theories. But it is well-known that a decidable theory can always be transformed into a decidable equational theory over the type Bool of truth values equipped with its usual operations. This is so because of the decidability assumption.

**Type levels equalities.** One may wonder whether the conversion relation of $CC_{\mathbb{N}}$ could use type level equalities (or hypotheses of the form $P \leftrightarrow Q$). The answer seems to be negative: extracting type levels equalities breaks subject reduction and $\beta$-strong normalization (see [16]), two properties needed for the decidability of our calculus.

**Trusting the kernel.** Decision procedures require complex coding. It took a lot of time to get a correct tactic for Presburger arithmetic in Coq. Including a tactic into the kernel of the system is therefore unrealistic, unless it is itself proved correct with a trustable proof assistant. On the other hand, most decision procedures can provide a *certificate* that is quite compact and can be verified by a *certificate-checker* which is usually small, and easy to write and read, and is therefore a trustable piece of code. The reason is that the procedure *searches* for a proof while the certificate-checker *verifies* that the certificate is correct. A certificate checker looks indeed like a proof-checker. It is then easy to modify the conversion rule so as to output a certificate each time a decision procedure is used. The kernel of $CC_{\mathbb{N}}$ should therefore include a certificate-checker for Presburger arithmetic. In case of CCIC with several decision procedures, the kernel would include one proof-checker for each decision procedure. Besides, the process is incremental: the procedures and the associated proof-checkers can be included one by one, because decision procedures for different inductive types operate on disjoint vocabularies, hence can be combined [17, 1].

An implementation of CCIC has started and should be available soon as a prototype in a version without certificate generation and checking.

## References


[1] F. Baader and K. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In Deepak Kapur, editor, *Proc. 11th Int. Conf. on Automated Deduction, Saratoga Springs, NY, LNAI 607*, 1992.

[2] H. Barendregt. *Lambda calculi with types*, volume 2 of *Handbook of logic in computer science*. Oxford University Press, 1992.

[3] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université de Paris VII, 1999.

[4] G. Barthe. The relevance of proof irrelevance. In *Proc. 24th Int. Coll. on Automata, Languages and Programming, LNCS 1443*, LNCS, 1998.

[5] F. Blanqui. *Type Theory and Rewriting*. PhD thesis, Université de Paris XI, Orsay, France, 2001.

[6] 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.

[7] F. Blanqui. Inductive types in the calculus of algebraic constructions. *Fundamenta Informaticae*, 65(1-2):61–86, 2005. Journal version of TLCA'03.

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

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

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

[11] Th. 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.

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

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

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

[15] M. Hofmann and T. Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford University Press, 1998.

[16] Nicolas Oury. Extensionality in the calculus of constructions. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2005.

[17] M. Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. *J. Symbolic Computation*, 8:51–99, 1989. Special issue on Unification.

[18] N. Shankar. Little engines of proof. In G. Plotkin, editor, *Proceedings of the Seventeenth Annual IEEE Symp. on Logic in Computer Science, LICS 2002*. 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.O. 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] P.-Y. Strub. *Type Theory and Decision Procedures.* PhD thesis, École Polytechnique, Palaiseau, France, Work in progress.

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