Extensionality in the Calculus of Constructions

Nicolas Oury

Laboratoire de Recherche en Informatique, UMR 8623 CNRS,
Université Paris–Sud, Orsay, France

Abstract. This paper presents a method to translate a proof in an extensional version of the Calculus of Constructions into a proof in the Calculus of Inductive Constructions extended with a few axioms. We use a specific equality in order to translate the extensional conversion relation into an intensional system.

In logical systems based on type theory, terms — and formulas — are identified modulo an equivalence relation which usually includes β–convertibility. Two equivalent terms or formulas are considered as exactly the same. There is also a propositional predicate for equality, such that one can state and possibly prove that two terms are equal. Two equivalent terms are obviously provably equal, the converse is not true in general. Actually, there is a distinction between so-called intensional and extensional type theories: in intensional type theories, two terms are equivalent only if they compute to the same value; in extensional type theory, two terms which are provably equal are equivalent. Having a larger class of equivalent terms leads to simpler proofs, the identification of provably equal terms is also an usual mathematical practice. However, because provability is not decidable, the equivalence relation becomes undecidable in extensional type theories and so is type checking. Except for the Nuprl system [3] and — with a weaker notion of conversion — HOL, all proof assistants based on type theories are indeed implementing an intensional type theory. Nevertheless, there is a growing interest in extending the equivalence relation to include more than β–reduction. Deduction modulo is a logical framework where the equivalence relation plays a central role in deduction, the Calculus of Algebraic Constructions extends also the usual computations with general rewrite rules still preserving normalisation and confluence. A natural question is to understand whether considering equal terms as equivalent significantly changes the logical system. The answer for a first-order system is negative, the extensional system is conservative over the intensional one, which means we can prove exactly the same theorems ([4]). However, the problem is more complicated in type theories with dependent types because having more equivalent terms extends the class of typable formula. M. Hofmann gives a semantical proof of conservativity of Extensional Martin–Lof's type theory into Intensional Martin–Lof's type theory extended with a few axioms. In this paper, we analyse the same problem in the framework of the Calculus of Constructions. Our contribution is to give a syntactic proof: it includes an effective process which translates an extensional proof into an intensional one (with additional axioms), this proof can consequently be checked by a

J. Hurd and T.F. Melham (Eds.): TPHOLs 2005, LNCS 3603, pp. 278–293, 2005.

computer. In our proof, we use the so-called John Major's equality introduced by C. McBride in order to compare two terms of different types ; it plays a central role in order to overcome the technical difficulty caused by dependent types.

In section 1, we give a precise presentation of the problem. In section 2, we present the extensional system and expose its properties. In section 3, we extend the intensional *Calculus of Inductive Constructions* in order to be able to translate the extensional calculus into it in section 4. The main result is to establish conservativity of this *Extensional Calculus of Constructions* on this extended *Calculus of Inductive Constructions*.

1 Presentation

The *Calculus of Inductive Constructions* (CIC) is a logical system implemented by the *Coq* proof assistant. CIC is a typed λ–calculus with dependent types and inductive definitions. There is an internal notion of convertibility over terms. This convertibility includes β–conversion and ι–conversion — which deals with pattern matching and fixpoints on inductive definitions. We denote this relation $\equiv$.

Convertibility is automatically used during typing. So some calculi are discharged by the system. For example, because $2+2 \equiv 4$, $2+2 = 4 \equiv 4 = 4$. So the obvious proof of $4 = 4$ is also a proof of $2 + 2 = 4$. This results in shorter proofs. In order for type checking to be decidable, we need $M \equiv N$ to be decidable when M and N are well typed. In the CIC, this is decidable. Indeed, the conversion is based on a reduction $\rightarrow$ strongly normalizing and confluent.

In CIC, it is also possible to define a propositional equality (*Leibniz* equality) defined as the smallest reflexive relation. Two terms that are convertible can be proved equal by reflexivity. But some terms, that are not convertible, can also be proved equal. For example, if we define addition on Peano numbers[1]:

```
plus O n = n
plus (S m) n = plus m (S n)
```

$O + n$ and n are convertible, so the proof of $\forall n, 0 + n = n$ is exactly the same as the proof of $\forall n, n = n$. But $n + O$ and n are not convertible. Indeed, `plus` is defined by pattern matching on its first argument. The first argument of $n + O$ is a free variable, so this term is in normal form. The conversion fails. We have to prove $\forall n, n + O = n$ by induction on n.

Moreover, the *Calculus of Inductive Constructions* allows *dependent types*. So values can appear in types. Hence, two types that depend on values which are equal but not convertible are not convertible. For example, assume we have defined $list\ n$, the type of lists of natural numbers of size n. Size of the list appears in the type. Now, we can define an `append` function to concatenate two of these lists. This function has type:

`append` : $\forall n, m : nat.(list\ n) \rightarrow (list\ m) \rightarrow (list\ (n + m))$

[1] For the sake of clarity, we use pseudo syntax here.

We may want to prove some properties of this function. Let us try to prove the property

$$\forall n : nat,\ l : (list\ n), (append\ l\ nil) = l.$$

Actually, we get into a problem. We can not even write this property. This equality is not well-typed. *append l nil* is of type $list\ (n + O)$ whereas l is of type $list\ n$. As shown earlier, these types are not convertible. So, the types are different. Whereas Leibniz equality only links two terms within the same type.

These errors are difficult to understand and even more difficult to solve. For example, the above property is difficult to formalize. Moreover, we lose modularity. Some properties or proofs relies on the implantation of *plus* and not only on its mathematical behaviour. In order to use natural numbers in a proof development, one need to know their implementation.

Usual mathematics identifies equal terms. So, ideally, a proof system should merge *convertibility* and *Leibniz* equality. Such a system — like *Nuprl* [3] — is said to be *extensional*. In such a system the following conversion would hold :

$$X + 0 \equiv X$$

$$0 + X \equiv X$$

This solves the problem described above.

Martin Hofmann has studied this problem in the context of Martin–Löf type theory [5](Section 3.2.5). He has shown this theory to be conservative over the usual theory extended with Streicher's axiom K and *functional extensionality*[2]. These axioms are similar but weaker than those which are used here. His proof is based on a semantical model of the system. Here, we syntactically translate typed terms in the usual system. We give a practical interest of an effective translation in section 5.

In the following, we drop inductive definitions in the Extensional Calculus of Inductive Constructions. We use proved equalities to reintroduce them later — see section 5.

2 Extensional Calculus of Constructions

We base the *Extensional Calculus of Constructions* (hereafter called CC_E) on the *Extended Calculus of Constructions*(ECC)[7]. This is a Pure Type System (PTS) [1] with a hierarchy of cumulative sorts Type_i and an impredicative sort Prop to express logical propositions.

Figure 1 shows typing rules of the *Extended Calculus of Constructions*. Rule (CONV) allows conversion during typing. The conversion rules for this system are those shown in figure 2 extended with reflexivity, symmetry and transitivity. This conversion is β-conversion. Nevertheless, for the sake of the clarity of our translation, we had rather decomposed it into head reductions and some congruence rules. In the following, we denote the empty context with $\emptyset$, product with

[2] This axiom states that two functions pointwisely equal are equal.

$$\text{(ECON)}\ \frac{}{\vdash \emptyset} \qquad \text{(I–CON)}\ \frac{\Gamma \vdash A : s}{\vdash \Gamma, x : A}\ s \in \mathcal{S}$$

$$\text{(TYPE)}\ \frac{}{\vdash \mathit{Type}_i : \mathit{Type}_{i+1}} \qquad \text{(PROP)}\ \frac{}{\Gamma \vdash \mathit{Prop} : \mathit{Type}_0}$$

$$\text{(VAR)}\ \frac{\vdash \Gamma}{\Gamma \vdash x : A}\ x : A \in \Gamma \qquad \text{(UNIV)}\ \frac{\Gamma \vdash A : \mathit{Type}_i}{\Gamma \vdash A : \mathit{Type}_{i+1}}$$

$$\text{(PROD)}\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : s'}{\Gamma \vdash \forall x : A.B : s'}\ (s, s') \in \mathcal{R}$$

$$\text{(APP)}\ \frac{\Gamma \vdash t : \forall x : A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : B\{x \leftarrow u\}}$$

$$\text{(LAM)}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\mathit{fun}\ x : A \Rightarrow t) : \forall x : A.B}$$

$$\text{(CONV)}\ \frac{\Gamma \vdash A \equiv B \qquad \Gamma \vdash t : A \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B}$$

Fig. 1. Typing rules of the Extended Calculus of Constructions

$\forall x.T$ and abstraction with $\mathit{fun}\ x \Rightarrow M$. A typing judgement is written $\Gamma \vdash t : T$. $\vdash \Gamma$ are judgements for well formed contexts. $\Gamma \vdash t \equiv t'$ are judgements of convertibility.

2.1 Extensionality Rule

We denote $=$ the usual, propositional *Leibniz* equality in *ECC*. CC_E consists in the Extended Calculus of Constructions extended with an *extensionality* rule:

$$\text{(EXT)}\ \frac{\Gamma \vdash_E t : A = B}{\Gamma \vdash_E A \equiv B}$$

In the following, judgements of CC_E are denoted $\vdash_E$.

If an equality is provable in a context, then the terms are convertible.[3] With this rule, we can type the example on size-dependant lists.

We can also prove that two functions equal pointwise are equal. The following derivation states this property and is a good example of the usage of the extensionality rule[4]:

[3] We keep the name *conversion* for this relation because it is used as a conversion in the usual system. We agree that this is much more powerful than an usual conversion.

[4] In this derivation, for the sake of clarity, we focus on types and forget terms.

$$(\beta)\ \frac{\Gamma \vdash (((fun\ x : A \Rightarrow u)v) : t}{\Gamma \vdash ((fun\ x : A \Rightarrow u)v) \equiv u\{x \leftarrow v\}}$$

$$(\text{CAPP})\ \frac{\Gamma \vdash u_1 \equiv u_1' \qquad \Gamma \vdash u_2 \equiv u_2'}{\Gamma \vdash (u_1\ u_2) \equiv (u_1'\ u_2')}$$

$$(\text{CPROD})\ \frac{\Gamma \vdash u_1 \equiv u_1' \qquad \Gamma,\ x : u_1 \vdash u_2 \equiv u_2'}{\Gamma \vdash \forall x : u_1.u_2 \equiv \forall x : u_1'.u_2'}$$

$$(\text{CLAM})\ \frac{\Gamma \vdash u_1 \equiv u_1' \qquad \Gamma,\ x : u_1 \vdash u_2 \equiv u_2'}{\Gamma \vdash (fun\ x : u_1 \Rightarrow u_2) \equiv (fun\ x : u_1' \Rightarrow u_2')}$$

Fig. 2. Conversion rules of the Extended Calculus of Constructions

$$(\text{EqIntro})\ \frac{(\text{CLAM})\ \dfrac{(\text{EXT})\ \dfrac{\dfrac{\Gamma \vdash_E \forall x : A, M = N}{\Gamma, x : A \vdash_E M = N}}{\Gamma, x : A \vdash_E M \equiv N}}{\Gamma \vdash_E fun\ x : A \Rightarrow M \equiv fun\ x : A \Rightarrow N}}{\Gamma \vdash_E fun\ x : A \Rightarrow M = fun\ x : A \Rightarrow N}$$

2.2 Undecidability

The extensional system allows simpler proofs but has some drawbacks. First of all, in this calculus, typing is undecidable. Intuitively, the extensionality rule (EXT) "forgets" a witness of an equality property. This proof that two terms are equal has been used but is not kept in the proof. In order to typecheck the term, the type system has to guess these equalities and proofs.

More precisely, on one hand, it is easy to prove that a type checker may have to decide for any given Γ, M and N, $\Gamma \vdash_E M \equiv N$. Indeed, let extend Γ with a term P of type $(nat \to nat) \to Prop$ and p of type $\forall x : nat \to nat.P\ x$. It is easy to check that $\Gamma' \vdash_E (fun\ y : (P\ M).y)(p\ N)$ is typable if and only if $\Gamma \vdash_E M \equiv N$.

On the other hand, we can encode the *halting problem* in a convertibility decision. Indeed, let now assume we have defined in CC_E a function T taking 3 arguments. $T\ n\ m\ p$ returns 1 if and only if the n^{th} Turing machine does not halt in p steps or less on entry m. This function can easily be written as a fixpoint on p in CC_E. Let also define, a function f such that $f\ p$ constantly equals 1. Using the derivation of previous section, we have that $\vdash_E T\ n\ m \equiv f$ if and only if $\vdash_E \forall p.T\ n\ m\ p \equiv f\ p$. This latter holds if and only if the n^{th} Turing machine does not halt on entry m. So the type system is able to decide an undecidable problem and so is undecidable.

2.3 Infinite Reductions

Moreover, this system is not strongly normalizing. Some terms can be typed and are not normalizing. For example, in a context containing a proof of $True = True \to True$[5], we can encode the whole λ–*calculus*, which is not normalising.

The Coq proof assistant checks proofs by typing. CC_E typing is undecidable. So this system seems not to be a good choice for this proof assistant. Nevertheless, it is useful as a superset of some decidable type systems.

2.4 A Model for Rewriting Extensions of the Calculus of Inductive Constructions

There is another possibility to solve the convertibility problems shown in section 1. CIC could be extended to support *rewriting rules*. Some extensions — like [2] or [9] — allows the user to add *symbols* and rewriting rules to convert these symbols. In such a system, for example, the user can define *symbols* `S`, `O` and `plus` and add the following *rewriting rules* :

```
O + n -> n
n + O -> n
S n + m -> S (n + m)
m + (S m) -> S (n + m)
```

Since more reductions are allowed, $n + O$ and $O + n$ are both convertible to n. This solves the problem described in section 1.

The system checked some criterias on *symbols* and *rules*. This ensures that the system stays strongly normalizing, confluent and consistent.

Nevertheless, these systems have some drawbacks. Logical power of these system with respect to *Calculus of Inductive Constructions* have not been much studied. Moreover, with such a system, it is not always clear whether we have defined the *same* data types as in the initial system. For example, we can look at the following rewriting system:

```
f 0 -> true
f 1 -> false
```

This rewriting system have an infinity of new normal forms — $f\ i, \forall i > 1$ — for *booleans*. This is not incoherent but very counter-intuitive.

Furthermore, it is difficult to *extract* programs from proofs using rewriting steps. Extraction is a process in the Coq proof assistant that translates a constructive proof to a functional program. Assume we want to extract a proof to a functional language. We know how to translate β–reductions and ι–reductions: they have their counter parts in most functional languages. But, some rewrite rules — like *non–linear* rules — can not be translated in a funcional languages.

We suggest a new way to add rewrite rules. First, the user proves or admits some equalities in the *Calculus of Inductive Constructions*. Then, one of the

[5] This property is not provable in CC_E but is consistent with this system. It holds in some *proof irrelevance* model.

criterias of rewriting extensions is used to check that the system is still *decidable*. The system where we add such *proved* rewrite rules is a subsystem of CC_E. So the translation described here can be used to translate it into a slight extension of the Calculus of Inductive Constructions. It allows to check the proof with a certified kernel. This practical interest will be fully discussed in section 5.

3 Translation of CC_E into CIC+

In section 2, we have shown a derivation that is provable in CC_E and does not hold in CIC. So CC_E is obviously non conservative over CIC. We introduce an extension of CIC, that is powerful enough to encode CC_E. We translate this *extensional* system into that extension of the *Calculus of Inductive Constructions* — thereafter called CIC+. CIC, and so CIC+ is based on the typing rules of ECC — see figures 1 and 2 —, extended with inductive definitions, pattern matching and fixpoints.

We keep the $\vdash$ notation for the deduction in CIC+ and $\vdash_E$ for the deduction in CC_E.

Next, we explain the principle of the translation. Then, we introduce the system CIC+.

3.1 Extensionality Rule

The difference between *extensional* and *intensional* systems is the *extensionality rule*. This rule transforms a proved equality into a *conversion*. Such a transformation is impossible in an *intensional* system like CIC or CIC+. So, we can not translate the conversion of the extensional system into the conversion of the intensional system. The best we can hope is to translate the conversion relation of CC_E into a proved equality in CIC+. We want to have:

$$\Gamma \vdash_E M \equiv N \Rightarrow \exists p, \Gamma \vdash p : M = N$$

This creates some difficulties. Indeed, let assume we have in CIC+ a function *subst* that rewrite with a proved equality.[6] The rule:

$$(\mathrm{CONV})\ \frac{\Gamma \vdash_E t : A \qquad \Gamma \vdash_E A \equiv B \qquad \Gamma \vdash_E B : s}{\Gamma \vdash_E t : B}$$

translates to:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash p : A = B \qquad \Gamma \vdash B : s}{\Gamma \vdash \mathit{subst}\ p\ t : B}$$

The translation changes the proof term. The equality step is made explicit. Since CC_E has dependent types, these explicit conversions can also appears in types. Moreover, the translation of a type depends on the derivation. A same type can have a lot of different translations with different explicit conversions.

[6] We will define this function in the next section.

Nevertheless, we manage to keep a key invariant in the translation.

Invariant. The translated term is the same as the initial term where some subterms t has been replaced by $subst\ p\ t$ for some proof p.

We stress here a point important to understand the whole translation process:

Remark 1. As type checking is undecidable, it can seem strange that there exists a translation into CIC+ — where type checking is decidable. Indeed, the process translates a *type tree* in CC_E. This type tree contains the information about equalities used in conversion. So, the translation have not to guess these equalities, which would be undecidable.

Before explaining the translation, we have to choose an equality for the translation of CC_E conversion relation. A same type in CC_E can have multiple translations. Moreover, *extensional* conversion can happen on two terms in two *intensionally* different types. So this equality has to link terms of different types.

3.2 Definition of = in CIC+

In CIC, the usual equality only allows to compare two terms that have the same type. In our case, we need to compare terms that have not, *a priori*, the same type. For example, $append\ l\ nil$ is of type $list\ (n+O)$ whereas l is of type $list\ n$.

$$(\text{JMType})\ \frac{\Gamma \vdash A, B : Type \qquad \Gamma \vdash x : A \qquad \Gamma \vdash y : B}{\Gamma \vdash x_A =_B y : Prop}$$

$$(\text{JMIntro})\ \frac{\Gamma \vdash x : A}{\Gamma \vdash refl\ x : x_A =_A x}$$

$$(\text{JMElim})\ \frac{\Gamma \vdash e : x_A =_B y \qquad \Gamma \vdash p : P\ A\ x \qquad \Gamma \vdash P : \forall A : Type, A \to Type}{\Gamma \vdash elim\ e\ P\ p : P\ B\ y}$$

$$(\text{JMLeibniz})\ \frac{\Gamma \vdash x, y : A}{\Gamma \vdash x_A =_A y \Rightarrow x =_L y}$$

$$(\text{JMLAM})\ \frac{\begin{array}{c}\Gamma \vdash p_1 : u_1 = u_1' \qquad \Gamma, x : u_1 \vdash u_2 : t \\ \Gamma, y : u_1' \vdash u_2' : t' \qquad \Gamma,\ x : u_1,\ y : u_1', p : x = y \vdash p_2 : u_2 = u_2'\end{array}}{\Gamma \vdash jmlam\ p_1\ p_2 : (fun\ x : u_1 \Rightarrow u_2) = (fun\ y : u_1' \Rightarrow u_2')}$$

$$(\text{JMAPP})\ \frac{\Gamma \vdash p_1 : u_1 = u_1' \qquad \Gamma \vdash p_2 : u_2 = u_2'}{\Gamma \vdash jmapp\ p_1\ p_2 : (u_1\ u_2) = (u_1'\ u_2')}$$

Fig. 3. Essential properties of = in CIC+

$$(\text{JMSET})\ \frac{\Gamma \vdash x_A =_B y}{\Gamma \vdash A =_L B}$$

$$(\text{JMSUBST1})\ \frac{\Gamma \vdash p : A = B \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathit{subst}\ p\ t : B}$$

$$(\text{JMSUBST2})\ \frac{\Gamma \vdash e : A = B \qquad \Gamma \vdash m : A}{\Gamma \vdash \mathit{subst}\ e\ m\ _B =_A m}$$

$$(\text{JMPROD})\ \frac{\begin{array}{c}\Gamma \vdash p_1 : u_1 = u_1' \qquad \Gamma, x : u_1 \vdash u_2 : s \\ \Gamma, y : u_1' \vdash u_2' : s \qquad \Gamma,\ x : u_1,\ y : u_1', p : x = y \vdash p_2 : u_2 = u_2'\end{array}}{\Gamma \vdash \mathit{jmprod}\ p_1\ p_2 : (\forall\ x : u_1.u_2) = (\forall\ y : u_1'.u_2')}$$

Fig. 4. Derivable properties of = in CIC+

Conversion of CC_E links these terms. Hence, in order to translate this relation into a *proved equality*, we have to extend equality to link these terms.

In [8], Conor Mc Bride introduced an equality that allows to compare terms in different types. Nevertheless, two terms can only be equal if they are in the same type. He has called this equality *John Major's equality*. It can be defined in CIC. Nevertheless, in CIC+, we need a slightly stronger equality. Figure 3 states the essential properties of this equality. All these properties are provable in CC_E for Leibniz equality and so need to be true in CIC+.

Definition. CIC+ consists of CIC extended with the equality axiomatised in figure 3.

Figure 4 states some properties — that are provable in CIC+ — that are used in the translation. This defines *subst*, a function that rewrites a term with an equality. One can define this function in CIC+ by using the primitive operation *elim*.

In section 5, we discuss the power of this equality compared to the John Major's equality that can be defined in the CIC.

As we often use this equality we write it =. When there is no ambiguity, we omit the type of the arguments. We write the usual *Leibniz* equality, $=_L$.

4 Proof of the Translation

4.1 Main Difficulty

We first explain the main problem with the translation of CC_E into CIC+. Let assume we inductively translate a derivation of CC_E. For example, one can look at the rule for application :

$$(\text{APP})\ \frac{\Gamma \vdash_E f : \forall x : A, B \qquad \Gamma \vdash_E t : A}{\Gamma \vdash_E f\ t : B}$$

By induction, we have, on one hand, a translation $\Gamma' \vdash f' : \forall x : A', B'$ and, on the other hand, a translation $\Gamma'' \vdash t' : A''$. We have, *a priori*, no link between, respectively, Γ' and Γ'', and A' and A''. Indeed, some explicit conversion — some *subst* p — may have been added at different positions during the translation. In order to use the rule for application in CCI+ and conclude the induction, we have to link them.

In order to solve this problem, we show that all translations of a same term are equal in CIC+. To have this result we have to introduce a new equivalence, linking all terms that are the same *modulo* some explicit conversion steps. This relation formalizes the key invariant of the translation.

4.2 Rewrited Terms Equivalence

Definition 1. *We define the equivalence relation* $\sim$ *on terms CIC+ by induction:*

- *for* x *variable:* $\dfrac{}{x \sim x}$
- *for* t_1, t_2 *and* p *terms,* $\dfrac{t_1 \sim t_2}{subst\ p\ t_1 \sim t_2}$ *and* $\dfrac{t_1 \sim t_2}{t_1 \sim subst\ p\ t_2}$
- *for* m_1, m_2, n_1 *and* n_2 *terms:* $\dfrac{m_1 \sim m_2 \qquad n_1 \sim n_2}{(m_1\ n_1) \sim (m_2\ n_2)}$
- *for* A_1, A_2, t_1 *and* t_2 *terms:* $\dfrac{A_1 \sim A_2 \qquad t_1 \sim t_2}{(fun\ x : A_1 \Rightarrow t_1) \sim (fun\ x : A_2 \Rightarrow t_2)}$ *and* $\dfrac{A_1 \sim A_2 \qquad t_1 \sim t_2}{(\forall x : A_1.t_1) \sim (\forall x : A_2.t_2)}$

This relation is canonically extended to contexts.

Remark 2. We stress the fact that this relation is purely syntactic and does not rely on typing.

Remark 3. subst is defined and one can be puzzled by the fact it could be inlined or reduced. The following translation can be done with an opaque parameter *subst′* — that would have the same properties as *subst* but can not be reduced. We can set this parameter to *subst* at the end of the translation.

We now prove that two typable equivalent terms are equal.

Lemma 1. *Let* t_1 *and* t_2 *be two terms. If* $t_1 \sim t_2$, $\Gamma \vdash t_1 : T_1$ *and* $\Gamma \vdash t_2 : T_2$, *then there exists* p *such that* $\Gamma \vdash p : t_1 = t_2$.

To prove this lemma, we need to introduce a larger relation. For all set E of couple of variables, we denote $\sim_E$ the relation $\sim$ extended by $x \sim y$ for every $(x, y) \in E$. In fact, we prove, by induction on the formation rules of $\sim_E$, that, for all E, for all Γ, if $\forall (x, y) \in E, \Gamma \vdash \_ : x = y$, $t_1 \sim_E t_2$, $\Gamma \vdash t_1 : \_$ and $\Gamma \vdash t_2 : \_$ then $\Gamma \vdash \_ : t_1 = t_2$.

- For $x \sim_E y$ the result is an hypothesis and for $x \sim_E x$ the result is an application of reflexivity.

- For $t_1 \sim_E \mathit{subst}\ p\ t_2$, we have by induction hypothesis $\Gamma \vdash \_ : t_1 = t_2$ and we conclude by use of property $(JMSubst2)$.
- For $(m_1 n_1) \sim_E (m_2 n_2)$, we have $\Gamma \vdash m_1 = m_2$ and $\Gamma \vdash n_1 = n_2$ by induction hypothesis, we conclude by $(JMProd)$.
- For $\forall x : T_1.m_1 \sim_E \forall x : T_2.m_2$, we use α–conversion and prove $\forall x : T_1.m_1 \sim_E \forall y : T_2.m_2 x \leftarrow y$. Let E' be $E \cup \{(x, y)\}$. By induction hypothesis, we have $\Gamma \vdash \_ : T_1 = T_2$ and $\Gamma, x : T_1, y : T_2, \_ : x = y \vdash \_ : m_1 = m_2$. We conclude by using the contextual rule of equality for product.
- The proof of λ–rule case is similar to the one of product.

We prove the original lemma by choosing $E = \emptyset$.

Lemma 2. *If $t_1 \sim t'_1$ and $t_2 \sim t'_2$ then $t_1\{x \leftarrow t_2\} \sim t'_1\{x \leftarrow t'_2\}$.*

By induction on the formation rules of $t_1 \sim t'_1$. For $y \sim y$, then we conclude because $y \sim y$ and $t_2 \sim t'_2$. For $\mathit{subst}\ p\ t \sim t'$, we have $t\{x \leftarrow t_2\} \sim t'\{x \leftarrow t'_2\}$.So we have $\mathit{subst}\ p\{x \leftarrow t_2\}\ t\{x \leftarrow t_2\} \sim t'\{x \leftarrow t'_2\}$. The other cases are simple applications of contextual rules.

4.3 Set of Terms with Explicit Conversions

We have now a tool to link judgements in CIC+ that are the same *modulo* some explicit conversions. We now translate a judgement in CC_E by a set of terms in CIC+ that are $\sim$–equivalent. Such a translation ensures — see the previous subsection — that all translations of a judgement can be proved to be equal in CIC+.

We have to translate three kinds of — mutually recursive — judgements: context formation, typing and conversion. We have to show that these interpretations sets are never empty — whatever choices have already been made in the translation process.

Definition 2. *For any $\vdash_E \Gamma$ we define a set $[\![\vdash_E \Gamma]\!]$ of judgements valid for CIC such that $\vdash \overline{\Gamma} \in [\![\vdash_E \Gamma]\!]$ if and only if $\overline{\Gamma} \sim \Gamma$ and Γ and $\overline{\Gamma}$ bind the same variable and each binded variable have the same kind.*

For any $\Gamma \vdash_E t : T$ we define a set $[\![\Gamma \vdash_E t : T]\!]$ of judgements valid for CIC such that $\overline{\Gamma} \vdash \overline{t} : \overline{T} \in [\![\Gamma \vdash_E t : T]\!]$ if and only if $\vdash \overline{\Gamma} \in [\![\vdash_E \Gamma]\!]$, $\overline{t} \sim t$ and $\overline{T} \sim T$.

Lemma 3. *We can always choose types $\overline{T}$ that have the same head constructor as T.*

Proof of the lemma Assume we have $\overline{\Gamma} \vdash \overline{t} : \overline{T}$. By definition of $\sim$, for any terms $T \sim \overline{T}$, $\overline{T}$ is shaped $\mathit{subst}\ p_1\ \ldots\ \mathit{subst}\ p_n \overline{T'}$ with $\overline{T'}$ having the same head constructor than T. Any subterm of a typable term is typable. So $\overline{T'}$ is typable. Moreover, from lemma 1, it is equal to $\overline{T}$. So there exists p of type $T = \overline{T'}$ such that $\overline{\Gamma} \vdash \mathit{subst}\ p\ \overline{t} : \overline{T'}$.

We now prove our translation process.

Theorem 1. *The following properties are valid:*

1. *If $\Gamma \vdash_E t : T$ then for any $\vdash \overline{\Gamma}$ in $[\![\vdash_E \Gamma]\!]$ there exists $\overline{t}$ and $\overline{T}$ such that $\overline{\Gamma} \vdash \overline{t} : \overline{T} \in [\![\Gamma \vdash_E t : T]\!]$.*

2. *If* $\vdash_E \Gamma$ *then there exists* $\vdash \overline{\Gamma} \in [\![\vdash_E \Gamma]\!]$.
3. *If* $\Gamma \vdash_E t_1 \equiv t_2$ *there exists* $\overline{\Gamma} \vdash \overline{t} : \overline{t_1} = \overline{t_2} \in [\![\Gamma \vdash t : t_1 = t_2]\!]$

Corollary 1. *For any* $\overline{\Gamma} \vdash \overline{T} : s$ *in* $[\![\Gamma \vdash_E T : s]\!]$, *and forall* t *such that* $\Gamma \vdash_E t : T$, *there exists* $\overline{t}$ *such that* $\overline{\Gamma} \vdash \overline{t} : \overline{T} \in [\![\Gamma \vdash_E t : T]\!]$. *In particular, if* $\Gamma \vdash T : s$ *there exists* $\overline{t}$ *such that* $\Gamma \vdash \overline{t} : T \in [\![\Gamma \vdash_E t : T]\!]$.

Proof of the corollary. By property 1 of the theorem, we have a translation. By lemma 2, we can choose $\overline{T}$.

This corollary induces the *conservativity* of CC_E over CIC+.

Proof of the theorem. We proceed by induction on the typing rules of CC_E.

1. (ECON) $\dfrac{}{\vdash_E \emptyset}$ become $\dfrac{}{\vdash \emptyset}$
2. (I-CON) $\dfrac{\Gamma \vdash_E A : s}{\Gamma, x : A \vdash_E}$. By induction hypothesis, there exists $\overline{\Gamma} \vdash \overline{A} : s$ in $[\![\Gamma \vdash_E A : s]\!]$. By the context formation rule we get $\vdash \Gamma, x : \overline{A}$. Moreover $\overline{A}$ and A have the same kind.
3. (TYPE) and (PROP) These rules translate to the same rules in CIC+.
4. (UNIV) $\dfrac{\Gamma \vdash A : Type_i}{\Gamma \vdash A : Type_{i+1}}$ Let $\vdash \overline{\Gamma}$ be in $[\![\vdash_E \Gamma]\!]$. By induction hypothesis, we have $\overline{\Gamma} \vdash \overline{A} : Type_i \in [\![\Gamma \vdash A : Type_i]\!]$. We conclude by using universe cumulativity in CIC.
5. (VAR) $\dfrac{\vdash_E \Gamma}{\Gamma \vdash_E x : A}$. Let $\vdash \overline{\Gamma}$ be in $[\![\vdash_E \Gamma]\!]$. Then we get by application of the same rule $\overline{\Gamma} \vdash x : \overline{A} \in [\![\Gamma \vdash_E x : A]\!]$.
6. (PROD) $\dfrac{\Gamma \vdash_E A : s \qquad \Gamma, x : A \vdash_E B : s'}{\Gamma \vdash_E \forall x : A.B : s'}$
   Let $\vdash \overline{\Gamma}$ be in $[\![\vdash_E \Gamma]\!]$. By induction hypothesis, there exists $\overline{A}$ such that $\overline{\Gamma} \vdash \overline{A} : s \in [\![\Gamma \vdash_E A : s]\!]$.
   Moreover, as $\vdash \overline{\Gamma}, x : \overline{A} \in [\![\vdash_E \Gamma, x : A]\!]$, there exists $\overline{B}$ such that $\overline{\Gamma}, x : \overline{A} \vdash \overline{B} : s' \in [\![\Gamma, x : A \vdash_E B : s']\!]$. As $(s, s') \in \mathcal{S}$, we conclude $\overline{\Gamma} \vdash \forall x : \overline{A}.\overline{B} : s' \in [\![\Gamma \vdash_E \forall x : A.B : s']\!]$.
7. (LAM) $\dfrac{\Gamma, x : A \vdash_E t : B}{\Gamma \vdash_E (fun\ x : A \Rightarrow t) : \forall x : A.B}$
   Let $\vdash \overline{\Gamma}, x : \overline{A}$ be in $[\![\vdash_E \Gamma, x : A]\!]$. By induction hypothesis, there exists $\overline{t}$ and $\overline{B}$ such that $\overline{\Gamma}, x : \overline{A} \vdash \overline{t} : \overline{B} \in [\![\Gamma, x : A \vdash_E t : B]\!]$. Then, with the λ–intro rule, we have : $\overline{\Gamma} \vdash (fun\ x : \overline{A} \Rightarrow \overline{t}) : \forall x : \overline{A}.\overline{B} \in [\![\Gamma \vdash_E (fun\ x : A \Rightarrow t) : \forall x : A.B]\!]$
8. (APP) $\dfrac{\Gamma \vdash_E t : \forall x : A.B \qquad \Gamma \vdash_E u : A}{\Gamma \vdash_E (t\ u) : B\{x \leftarrow u\}}$

Let $\vdash \overline{\Gamma}$ be in $[\![\vdash_E \Gamma]\!]$. By induction hypothesis and *lemma 3*, there exists $\overline{t}$, $\overline{A}$ and $\overline{B}$ such that $\overline{\Gamma} \vdash \overline{t} : \forall x : \overline{A}.\overline{B} \in [\![\Gamma \vdash_E t : \forall x : A.B]\!]$. Moreover, there exist u' and A' such that $\overline{\Gamma} \vdash u' : A' \in [\![\Gamma \vdash_E u : A]\!]$ As we have $\overline{\Gamma} \vdash \overline{A} = A'$, there exists $\overline{u}$ such that $\overline{\Gamma} \vdash \overline{u} : \overline{A} \in [\![\Gamma \vdash_E u : A]\!]$.
So, with the application rule, we conclude:

$$\overline{\Gamma} \vdash (\overline{t}\ \overline{u}) : \overline{B}[x \leftarrow \overline{u}] \in [\![\Gamma \vdash_E (t\ u) : B[x \leftarrow u]]\!]$$

9. (CONV) $\dfrac{\Gamma \vdash_E A \equiv B \qquad \Gamma \vdash_E t : A \qquad \Gamma \vdash_E B : s}{\Gamma \vdash_E t : B}$

Let $\vdash \overline{\Gamma}$ be in $[\![\vdash_E \Gamma]\!]$. By induction hypothesis and *lemma 3*, there exist p, $\overline{A}$ and $\overline{B}$, such that $\overline{\Gamma} \vdash p : \overline{A} = \overline{B} \in [\![\Gamma \vdash_E A \equiv B]\!]$
Moreover, there exist, by induction hypothesis, $\overline{t'}$ and $\overline{A'}$ such that $\overline{\Gamma} \vdash \overline{t'} : \overline{A'} \in [\![\Gamma \vdash_E t : A]\!]$.
Moreover, $\overline{\Gamma} \vdash \_ : \overline{A} = \overline{A'}$. We conclude, there exists $\overline{t}$ such that $\overline{\Gamma} \vdash \overline{t} : \overline{A} \in [\![\Gamma \vdash_E t : A]\!]$. So, $\overline{\Gamma} \vdash (\mathit{subst}\ p\ \overline{t}) : \overline{B} \in [\![\Gamma \vdash_E t : B]\!]$.

10. (EXT) $\dfrac{\Gamma \vdash_E t : A = B}{\Gamma \vdash_E A \equiv B}$

Let $\vdash \overline{\Gamma}$ be in $[\![\vdash_E \Gamma]\!]$. By induction hypothesis, there exist $\overline{t}$, $\overline{A}$ and $\overline{B}$ such that $\overline{\Gamma} \vdash \overline{t} : \overline{A} = \overline{B} \in [\![\Gamma \vdash_E A = B]\!]$. By definition, this judgement is also in $[\![\Gamma \vdash_E A \equiv B]\!]$.

11. (CAPP) $\dfrac{\Gamma \vdash_E u_1 \equiv u_1' \qquad \Gamma \vdash_E u_2 \equiv u_2'}{\Gamma \vdash_E (u_1\ u_2) \equiv (u_1'\ u_2')}$

We conclude by induction hypothesis and congruence rule (JMAPP).

12. (CPROD) $\dfrac{\Gamma \vdash_E u_1 \equiv u_1' \qquad \Gamma,\ x : u_1 \vdash_E u_2 \equiv u_2'}{\Gamma \vdash_E \forall x : u_1.u_2 \equiv \forall x : u_1'.u_2'}$

This rule is, in an extensional system, equivalent to:

$$\frac{\Gamma \vdash_E u_1 \equiv u_1' \qquad \Gamma,\ x : u_1,\ y : u_1', p : x = y \vdash_E u_2 \equiv u_2'\{x \leftarrow y\}}{\Gamma \vdash_E \forall x : u_1.u_2 \equiv \forall x : u_1'.u_2'}$$

We have, by induction hypothesis $\overline{\Gamma} \vdash \_ : \overline{u_1} = \overline{u_1'}$ and — as $x = y$ is typable in $\overline{\Gamma},\ x : \overline{u_1},\ y : \overline{u_1'}$ — $\overline{\Gamma},\ x : \overline{u_1},\ y : \overline{u_1'},\ \overline{p} : x = y \vdash \_ : \overline{u_2} = \overline{u_2'}\{x \leftarrow y\}$. We conclude by application of congruence rule (JMPROD).

13. (CLAM) The contextual rule for functions is similar to the rule for products.
14. (β)

$$\frac{\Gamma \vdash_E ((\mathit{fun}\ x : A \Rightarrow u)v) : \_}{\Gamma \vdash_E ((\mathit{fun}\ x : A \Rightarrow u)v) \equiv u\{x \leftarrow v\}}$$

From *lemma 3*, there exist $\overline{\Gamma}$, $\overline{A}$, $\overline{u}$, $\overline{B}$, $\overline{A'}$ et $\overline{v'}$ such that: $\overline{\Gamma} \vdash (\mathit{fun}\ x : \overline{A} \Rightarrow \overline{u}) : \forall x : \overline{A}.\overline{B}$ and $\Gamma \vdash \overline{v} : \overline{A'}$
As $\overline{\Gamma} \vdash \overline{A} = \overline{A'}$, there exist $\overline{v}$ such that: $\overline{\Gamma} \vdash ((\mathit{fun}\ x : \overline{A} \Rightarrow \overline{u})\overline{v}) : \overline{B} \in [\![\Gamma \vdash_E ((\mathit{fun}\ x : A \Rightarrow u)v) : \_]\!]$
By β–reduction, $\overline{\Gamma} \vdash \_ : ((\mathit{fun}\ x : \overline{A} \Rightarrow \overline{u})\overline{v}) = \overline{u}\{x \leftarrow \overline{v}\}$

5 Discussions and Conclusion

5.1 Comparison Between = in CIC+ and John Major's Equality Defined in CIC

In this translation, we have extended CIC to support a stronger equality. This equality links terms of different types and so it is similar to John Major's equality. One can define this latter in CIC but some of the properties of figure 3 do not hold.

- (JMType), (JMIntro) and (JMElim) holds.
- (JMLeibniz) does not hold in CIC. We need this axiom in order to prove (JMSubst2). It is equivalent to Streicher's axiom K. [6]
- (JMLam) does not hold in CIC. This is often called *functional extensionality* since it states that two functions pointwise equal are equal.
- (JMApp) can not be proved in CIC.

So CIC+ is CIC extended with three axioms.

1. Streicher's axiom K
2. Functional extensionality
3. (JMAPP), the equality of the results of two equal applications

In [5], only the two former are needed.

We now give a justification of these axioms by giving their interpretation in a *proof irrelevance* model of CIC in *set theory*.

- Axiom K states that the only proof of $t = t$ is $refl\ t$. It is clear in a *proof irrelevance* model, where two proofs of a proposition in Prop are always equal.
- Functional extensionality states that two functions that have equal results are equal. This is the definition of equality for functions in set theory.
- (JMAPP) states two equal functions applied to two equal arguments returns equal results. This holds for equality in set–theory.

5.2 Practical Interest of This Translation

This translation has some practical interests.

CC_E can be used as a superset of the rewriting extensions to CIC. Indeed, let assume the user has extended CIC with some new symbols and new rewrite rules linking these symbols. If the user can give an interpretation of these symbols in CIC and prove the equalities corresponding to these rewrite rules, then CC_E is a superset of CIC extended with these symbols and rewrite rules. So the translation process can translate a proof in this extended system into a proof in CIC+ that uses the equalities the user has proved. This allows a certified kernel to check these translated proofs. The extended system can construct a proof term that uses rewrite rules. This proof term can then be translated to be checked in the old, certified kernel.

Moreover, this translation gives a way to implement program extraction from proof in these extensions of CIC. Indeed, programs can be extracted from proofs

in CIC. Since axioms added in CIC+ are valid for *observational equivalence* of programs, the same process can be used to extract program in CIC+. Moreover, the shape of the translated terms guarantees that the extraction is safe in CC_E. Indeed, terms in CC_E can be translated into CIC+ by adding some explicit conversions *subst*. Since t and *subst* p t have the same extracted program, we can use the extraction process of CIC to extract proof in CC_E. Since CC_E can be used as a superset of rewriting extensions of CIC, this gives a safe way to extract programs from proofs in one of these extensions.

5.3 Generalizations

In the system we studied, some simplifications have been made. We work out these details here.

Universes Cumulativity. In the *Calculus of Inductive Constructions*, there is a hierarchy of universes $Type_0 \subset Type_1 \subset \ldots$. This is achieved by converting from $Type_i$ to $Type_j$ when $j > i$. But this means that conversion is not an equivalence relation anymore. This is a reduction order. As it would prevent us to encode conversion into equality, we choosed another rule: $\dfrac{\Gamma \vdash T : Type_i}{\Gamma \vdash T : Type_{i+1}}$

This was the original rule of the *Calculus of Constructions*. Luo explained in [7] that this is not equivalent to adding cumulativity in reduction. For example:

$$X : Type_0 \to Type_0 \nvdash X : Type_0 \to Type_1$$

Nevertheless, this restriction is not a restriction in our case. We still have:

$$X : Type_0 \to Type_0 \vdash (\mathit{fun}\ x \Rightarrow X\ x) : Type_0 \to Type_1$$

We also have by extensionality:

$$(\mathit{fun}\ x \Rightarrow X\ x) = X$$

So we can replace X everywhere. When a property contains $(\mathit{fun}\ x \Rightarrow X\ x)$, we can replace it by X with the extensionality rule. With this restriction we get a great technical simplification with no restriction on expressiveness.

Inductive Definitions. In this paper, we have decided, for the sake of simplicity, not to include *inductive types* in CC_E. In fact, since we can include equalities in convertibility, there is no need to include inductives in CC_E.

They can be introduced in CC_E. We can introduce axioms for constructors, *elimination*, *recursion* and *induction*. Then, we introduce axiomatised equalities for these symbols. Since this system is *extensional*, these equalities are automatically included in the conversion in CC_E. This axiomatisation will be used in the translation but is provable using inductive types in CIC+. So we can translate terms of CC_E extended with inductive types.

5.4 Conclusion

In this paper, we have shown a translation of an *Extensional Calculus of Constructions* into the usual *Calculus of Inductive Constructions* extended with Streicher's axiom K and contextual rules that allow John Major's equality to be a congruence.

This introduces a new approach to extend the Calculus of Inductive Constructions with rewrite rules. As soon as the set of rewriting rules is provable in CIC+, we can safely include these rewrite rules in CIC+. This leads to a method to safely introduce rewriting in the Coq proof assistant. First we provide a model and prove some equalities. Then we use a criteria to ensure that the equalities induces a normalising and decidable system. This process still allows to extract programs from proofs in an usual functional language.

References

1. Henk Barendregt. Typed lambda calculi. In Abramsky et al., editor, *Handbook of Logic in Computer Science*. Oxford Univ. Press, 1993.
2. F. Blanqui, J.-P. Jouannaud, and M. Okada. The Calculus of Algebraic Constructions. In Paliath Narendran and Michael Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, volume 1631, Trento, Italy, July 1999.
3. Robert L. Constable, Stuart F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, Douglas J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.
4. Gilles Dowek. *La part du calcul*. Thèse d'habilitation, Université Paris 7, 1999.
5. M. Hofmann. *Extensional concepts in intensional type theory*. Phd thesis, Edinburgh university, 1995.
6. Martin Hofmann and Thomas Streicher. A groupoid model refutes uniqueness of identity proofs. In *9th Symposium on Logic in Computer Science (LICS)*, Paris, 1994.
7. Zhaohui Luo. ECC an Extended Calculus of Constructions. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, Pacific Grove, California, 1989.
8. Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. Available from `http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/`.
9. Daria Walukiewicz-Chrzaszcz. Termination of rewriting in the Calculus of Constructions. In *Proceedings of the Workshop on Logical Frameworks and Meta-languages, Santa Barbara, California*, 2000. Part of the LICS'2000.