On the strength of proof-irrelevant type theories

Benjamin Werner

INRIA-Futurs and LIX, Ecole Polytechnique, France
`Benjamin.Werner@inria.fr`

Abstract. We present a type theory with some proof-irrelevance built into the conversion rule. We argue that this feature is particularly useful when type theory is used as the logical formalism underlying a theorem prover. We also show a close relation with the subset types of the theory of PVS. Finally we show that in these theories, because of the additional extentionality, the axiom of choice implies the decidability of equality, that is, almost classical logic.

1 Introduction

A formal proof system, or proof assistant, implements a formalism in a similar way a compiler implements a programming language. Among existing systems, dependent type systems are quite widespread. This can be related to various pleasant features; among them :

1. Proofs are objects of the formalism. The syntax is therefore smoothly uniform, and proofs can be rechecked at will. Also, only the correctness of the type-checker, a relatively small and well-identified piece of software, is critical for the reliability of the system ("de Bruijn principle").
2. The objects of the formalism are programs (typed λ-terms) and are identified modulo computation (β-conversion). This makes the formalism well-adapted for problems dealing with program correctness. But also the conversion rule allows the computation steps not to appear in the proof; for instance $2+2=4$ is simply proved by one reflexivity step, since this proposition is identified with $4=4$ by conversion. In some cases this can lead to a dramatic space gain, using the result of certified computations inside a proof; spectacular recent applications include the formal proof of the four-color theorem [11] or formal primality proofs [14].
3. Finally, type theories are naturally constructive. This makes stating decidability results much easier. Furthermore, combining this remark with the two points above, one comes to *program extraction*: taking a proof of a proposition $\forall x : A.\exists y : B.P(x,y)$, one can *erase* pieces of the λ-term in order to obtain a functional program of type $A \rightarrow B$, whose input and result are certified to be related by P. Up to now however, program extraction was more an external feature of implemented proof systems[1]: programs certified by extraction are not anymore objects of the formalism and cannot be used anymore to assert facts like in the point above.

[1] Except NuPRL; see related work.

Some related formalisms only build on some of the points above. For example PVS implements a theory whose objects are functional programs, but where proofs are external to the formalism.

An important remark about (2) is that the more terms are identified by the conversion rule, the more powerful this rule is. In order to identify more terms it is therefore a tempting step to combine points (2) and (3) by integrating program extraction into the formalism so that the conversion rule does not require the *computationally irrelevant* parts of terms to be convertible.

In what follows, we present and argue in favor of a type-theory along this line. More precisely, we claim that such a feature is useful in at least two respects. For one, it gives a more comfortable type theory, especially in the way it handles equality. Furthermore it is a good starting point to build a platform for programming with dependent types, that is to use the theorem prover also as a programming environment. Finally, on a more theoretical level, we will also see that by making the theory more extensional, proof-irrelevance brings type theory closer to set-theory regarding the consequences of the axiom of choice.

The central idea of this work is certainly simple enough to be adjusted to various kinds of type theories, whether they are predicative or not, with various kinds of inductive types, more refined mechanisms to distinguish the computational parts of the proofs etc.... In what follows we illustrate it by using a marking of the computational content which is as simple as possible. We define it precisely, but omit most meta-theoretical proofs and do not detail the model construction.

Related work Almost surprisingly, proof-irrelevant type theories do not seem to enjoy wide use yet. In the literature, they are often not studied for themselves, but as a mean for proving properties of other systems. This is the case for the work of Altenkirch [2] and Barthe [4]. One very interesting work is Pfenning's modal type theory which involves proof-irrelevance and a sophisticated way to pinpoint which definitional equality is to be used for each part of a term; in comparision we here stick to much simpler extraction mechanism. Finally the NuPRL approach using a squash type [6] is very close to ours, but the extentional setting gives sometimes different results.

2 The Theory

2.1 The λ-terms

The core of our theory is a Pure Type System (PTS) extended with Σ-types and some inductive type definitions. As in PTS's, the type of types are *sorts*; the set of sorts is:

$$\mathcal{S} \equiv \{\mathsf{Prop}\} \cup \{\mathsf{Type}(i) | i \in \mathbb{N}\}.$$

As one can see, we keep the sort names of Coq. As usual, Prop is the impredicative sort and the sorts $\mathsf{Type}(i)$ give the hierarchy of predicative universes. It comes as no surprise that the system contains the usual syntactic constructs of PTSs; however it is comfortable, both for defining the conversion rule and constructing a model to *tag* the variables indicating whether they correspond to a computational piece of code or not; in our case this means whether they live in the impredicative or a predicative level (i.e. whether the type of their type is Prop or a $\mathsf{Type}(i)$). A similar tagging is done on the projections of Σ-types. Except for this detail, the backbone of the theory considered hereafter is essentially Luo's Extended Calculus of Constructions (ECC) [16].

The syntax of the ECC fragment is therefore:

$$
\begin{array}{lcl}
s & ::= & \mathsf{Prop} \mid \mathsf{Type}(i) \qquad\qquad \mathsf{s} \quad ::= \quad * \mid \diamond \\
t & ::= & s \mid x_{\mathsf{s}} \mid \lambda x_{\mathsf{s}} : t.t \ \mid (t\ t) \mid \Pi x_{\mathsf{s}} : t.t \mid \Sigma^{\mathsf{s}} x_{\mathsf{s}} : t.t \mid \ < t, t >_{\Sigma x:t.t} \\
 & & \mid \pi_1^{\mathsf{s}}(t) \mid \pi_2^{\mathsf{s}}(t) \\
\Gamma & ::= & [] \mid \Gamma(x : t).
\end{array}
$$

We will sometimes write x for x_s, $\Sigma x : A.B$ for $\Sigma^{\mathsf{s}} x : A.B$ or $\pi_2(t)$ for $\pi_2^s(t)$ omitting the tag s when it is not relevant or can be infered from the context.

The binding of variables is as usual. We write $t[x \setminus u]$ for the substitution of the free occurrences of variable x in t by u. As has become custom, we will not deal with α-conversion here, and leave open the choice between named variables and de Bruijn indices.

We also use the common practice of writing $A \rightarrow B$ (resp. $A \times B$) for $\Pi x : A.B$ (resp. $\Sigma x : A.B$) when x does not appear free in B. We also write $\Pi x, y : A.B$ (resp. $\lambda x, y : A.t$) for $\Pi x : A.\Pi y : A.B$ (resp. $\lambda x : A.\lambda y : A.t$).

2.2 Relaxed conversion

The aim of this work is the study of a relaxed conversion rule. While the idea is to identify terms with respect to typing information, the tagging of impredicative *vs.* predicative variables is sufficient to define such a conversion in a simple syntactic way. A variable is computationally irrelevant when tagged with the $*$ mark. The tag on the Σ-type construction are there to indicate whether the second component of pairs is irrelevant or not. This leads to the following definition.

Definition 1 (Extraction). *We can simply define the extraction relation $\rightarrow_\varepsilon$ as the contextual closure of the following rewriting equations:*

$$
\begin{array}{rr}
x_* \rightarrow_\varepsilon \varepsilon & \lambda x : A.\varepsilon \rightarrow_\varepsilon \varepsilon \\
(\varepsilon\ t) \rightarrow_\varepsilon \varepsilon & \pi_2^*(t) \rightarrow_\varepsilon \varepsilon.
\end{array}
$$

We write $\rightarrow_\varepsilon^*$ for the reflexive-transitive closure of $\rightarrow_\varepsilon$. We say that a term is of tag $*$ if $t \rightarrow_\varepsilon^* \varepsilon$ and of tag $\diamond$ if not. We write $s(t)$ for the tag of t.

Definition 2 (Reduction). *The β-reduction $\rhd_\beta$ is defined as the contextual closure of the following equations:*

$$\begin{aligned}
(\lambda x^{\mathsf{s}} : A.t\ u) \rhd_\beta t[x^{\mathsf{s}} \setminus u] \qquad & \text{if } s(u) = \mathsf{s}\\
\pi_1^{\mathsf{s}}(< a, b >_{\Sigma x:A.B}) \rhd_\beta a \qquad & \text{if } s(a) = \diamond\\
\pi_2^{\mathsf{s}}(< a, b >_{\Sigma x:A.B}) \rhd_\beta b \qquad & \text{if } s(b) = \mathsf{s}.
\end{aligned}$$

The restrictions on the right-hand side are there in order to ensure that the tag is preserved by reduction. Without them $(\lambda x_\diamond : \mathsf{Prop}.x_\diamond\ \mathsf{Prop})$ can reduce either to ε or to Prop which would falsify the Church-Rosser property. Actually these restrictions later appear to be always satisfied on well-typed terms, but are necessary in order to assert the meta-theoretic properties below. While they are specific to our way of marking computational terms, other methods will probably yield similar technical difficulties.

The relaxed reduction $\rhd_\varepsilon$ is the union of $\rhd_\beta$ and $\rightarrow_\varepsilon$. We write $=_\varepsilon$ for the reflexive, symmetric and transitive closure of $\rhd_\varepsilon$ and $\rhd_\varepsilon^*$ for the transitive-reflexive closure of $\rhd_\varepsilon$.

Lemma 1 (β-postponement). *If $t \rhd_\varepsilon^* t'$, then there exists t'' such that $t \rightarrow_\varepsilon^* t''$ and $t'' \rhd_\beta^* t'$.*

Lemma 2 (Church-Rosser). *For t a raw term, if $t \rhd_\varepsilon^* t_1$ and $t \rhd_\varepsilon^* t_2$, then there exists t_3 such that $t_2 \rhd_\varepsilon^* t_3$ and $t_2 \rhd_\varepsilon^* t_3$.*

Proof. By a slight adaptation of the usual Tait–Martin-Löf method.

Furthermore, $\rightarrow_\varepsilon$ is obviously strongly normalizing. One therefore can "pre-cook" all terms by $\rightarrow_\varepsilon$ when checking relaxed convertibility:

Lemma 3 (pre-cooking of terms). *Let t_1 and t_2 be terms. Let t_1' and t_2' be their respective $\rightarrow_\varepsilon$-normal forms. Then, $t_1 =_\varepsilon t_2$ if and only if $t_1' =_\beta t_2'$.*

While this property is important for implementation, its converse is also true and semantically understandable. Computationally relevant β-reductions are never blocked by not-yet-performed ε-reductions:

Lemma 4. *Let t_1 be any raw term. Suppose $t_1 \rightarrow_\varepsilon t_2 \rhd_\beta t_3$. Then there exists t_4 such that $t_1 \rhd_\beta t_4 \rightarrow_\varepsilon^* t3$.*

Proof. It is easy to see that $\rightarrow_\varepsilon$ cannot create new β-redexes, nor does it duplicate existing ones.

It is a good feature to have the predicative universes to be embedded in each other. It has been observed (Pollack, McKinna, Barras. . .) that a smooth way to present this is to define a syntactic subtyping relation which combines this with $=_\beta$ (or here $=_\varepsilon$). Note that this notion of subtyping should not be confused with, for instance, subtyping of subset types in the style of PVS.

Definition 3 (Syntactic subtyping). *The subtyping relation is defined on raw-terms as the transitive closure of the following equations:*

$$\mathsf{Type}(i) \leq \mathsf{Type}(i+1) \qquad T =_{\varepsilon} T' \Rightarrow T \leq T'$$
$$B \leq B' \Rightarrow \Pi x : A.B \leq \Pi x : A.B'.$$

2.3 Functional fragment typing rules

The typing rules for the kernel of our theory are given in PTS-style [3] and correspond to Luo's ECC. The differences are the use of subtyping in the conversion rule and the tagging of variables when they are "pushed" into the context.

The rules are given in figure 1. In the rule Prod, max is the maximum of two sorts for the order $\mathsf{Prop} < \mathsf{Type}(1) < \mathsf{Type}(2) < \ldots$

$$(\textsc{Prop})\frac{\Gamma \vdash \mathrm{wf}}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}(i)} \qquad (\textsc{Type})\frac{\Gamma \vdash \mathrm{wf}}{\Gamma \vdash \mathsf{Type}(i) : \mathsf{Type}(i+p)}$$

$$(\textsc{Base})\frac{}{[] \vdash \mathrm{wf}} \qquad (\textsc{Var})\frac{\Gamma \vdash \mathrm{wf}}{\Gamma \vdash x : A}\text{if } (x : A) \in \Gamma$$

$$(\textsc{Cont})\frac{\Gamma \vdash A : \mathsf{Type}(i)}{\Gamma(x_{\diamond} : A) \vdash \mathrm{wf}} \quad (\textsc{Cont*})\frac{\Gamma \vdash A : \mathsf{Prop}}{\Gamma(x_{*} : A) \vdash \mathrm{wf}}$$

$$(\textsc{Conv})\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B}\text{if } A \leq B$$

$$(\textsc{Prod})\frac{\Gamma \vdash A : s \qquad \Gamma(x_{\mathsf{S}} : A) \vdash B : \mathsf{Type}(i)}{\Gamma \vdash \Pi x_{\mathsf{S}} : A.B : \max(s, \mathsf{Type}(i))}$$

$$(\textsc{Prod*})\frac{\Gamma \vdash A : s \qquad \Gamma(x_{\mathsf{S}} : A) \vdash B : \mathsf{Prop}}{\Gamma \vdash \Pi x_{\mathsf{S}} : A.B : \mathsf{Prop}}$$

$$(\textsc{Lam})\frac{\Gamma \vdash \Pi x : A.B : s \qquad \Gamma(x : A) \vdash t : B}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B} \qquad (\textsc{App})\frac{\Gamma \vdash t : \Pi x : A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : B[x \setminus u]}$$

$$(\textsc{Sig})\frac{\Gamma \vdash A : \mathsf{Type}(i) \qquad \Gamma(x : A) \vdash B : \mathsf{Type}(i)}{\Gamma \vdash \Sigma^{\diamond} x : A.B : \mathsf{Type}(i)}$$

$$(\textsc{Sig}^{*})\frac{\Gamma \vdash A : \mathsf{Type}(i) \qquad \Gamma(x : A) \vdash B : \mathsf{Prop}}{\Gamma \vdash \Sigma^{*} x : A.B : \mathsf{Prop}}$$

$$(\textsc{Pair})\frac{\Gamma \vdash a : A \qquad \Gamma(x : A) \vdash b : B \qquad \Gamma \vdash \Sigma^{\mathsf{S}} x : A.B : \mathsf{Type}(i)}{\Gamma \vdash < a, b >_{\Sigma x:A.B} : \Sigma^{\mathsf{S}} x : A.B}$$

$$(\textsc{Proj1})\frac{\Gamma \vdash t : \Sigma^{\mathsf{S}} x : A.B}{\Gamma \vdash \pi_1^{\mathsf{S}}(t) : A} \qquad (\textsc{Proj2})\frac{\Gamma \vdash t : \Sigma^{\mathsf{S}} x : A.B}{\Gamma \vdash \pi_2^{\mathsf{S}}(t) : B[x \setminus \pi_1^{\mathsf{S}}(t)]}$$

Fig. 1. The ECC fragment

We sketch the basic meta-theory of the calculus defined up to here. As mentioned above, we cannot detail the proofs and the intermediate lemmas here. The proof techniques are relatively traditional, even if one has to take care of the more delicate behavior of relaxed reduction for the first lemmas (similarly to [21]).

Lemma 5 (Substitution). *If $\Gamma(x:A)\Delta \vdash t:T$ and $\Gamma \vdash a:A$ are derivable, then $\Gamma\Delta[x \setminus a] \vdash t[x \setminus a] : T[x \setminus a]$ is derivable.*

Of course, subject reduction holds only for $\rhd_\beta$-reduction, since ε is not meant to be typable.

Lemma 6 (Subject reduction). *If $\Gamma \vdash t : T$ is derivable, if $t \rhd_\beta t'$ (resp. $T \rhd_\beta T'$, $\Gamma \rhd_\beta \Gamma'$) by a well-sorted reduction, then $\Gamma \vdash t' : T$ (resp. $\Gamma \vdash t : T', \Gamma' \vdash t : T$).*

Lemma 7. *If $\Gamma \vdash t : T$ is derivable, then there exists a sort s such that $\Gamma \vdash T : s$; furthermore $\Gamma \vdash T : \mathsf{Prop}$ if and only if t is of tag $*$.*

A most important property is of course normalization. We do not claim any proof here, although we very strongly conjecture it. A smooth way to prove it is probably to build on top of a simple set-theoretical model using an interpretation of types as saturated Λ-sets as first proposed by Altenkirch [1, 20].

Conjecture 1 (Strong Normalization). If $\Gamma \vdash t : T$ is derivable, then t is strongly normalizing.

Stating strong normalization is important in the practice of proof-checker, since it entails decidability of type-checking and type-inference.

Corollary 1. *Given Γ, it is decidable whether $\Gamma \vdash wf$. Given Γ and a raw term t, it is decidable whether there exists T such that $\Gamma \vdash t : T$ holds.*

The other usual side-product of normalization is a syntactic assessement of constructivity.

Corollary 2. *If $[] \vdash t : \Sigma x : A.B$, then $t \rhd_\beta^* < a, b >_{\Sigma x:A.B}$ with $[] \vdash a : A$ and $[] \vdash b : B[x \setminus a]$.*

2.4 Data Types

In order to be practical, the theory needs to be extended by inductive definitions in the style of Coq, Lego and others. We do not detail the typing rules and liberally use integers, booleans, usual functions and predicates ranging over them. We refer to the coq documentation [8, 10]; for a possible more modern presentation [5] is interesting.

Two points are important though:

1. Data types live in Type. That is, for instance, $\mathsf{nat} : \mathsf{Type}(1)$; thus, their elements are of tag $\diamond$.
2. There is no primary need for inductive definitions in Prop. Logical connectors and inductive properties can be encoded using impredicativity. For instance, we write:

$$A \wedge B \equiv \Pi P : \mathsf{Prop}.(A \to B \to P) \to P.$$

2.5 Treatment of propositional equality

Propositional equality is a first example whose treatment changes when switching to a proof-irrelevant type theory. The definition itself is unchanged; two objects a and b of a given type A are equal if and only if they enjoy the same properties:

$$a =_A b \ \equiv\ \Pi P : A \rightarrow \mathsf{Prop}.(P\ a) \rightarrow (P\ b).$$

It is well-known that reflexivity, symmetry and transitivity of equality can easily be proved. When seen as an inductive definition, the definition of "$=_A$" is viewed as its own elimination principle.

Let us write refl for the canonical proof of reflexivity:

$$\mathsf{refl} \equiv \lambda A : \mathsf{Type}(i).\lambda x : A.\lambda P : A \rightarrow \mathsf{Prop}.\lambda p : (P\ x).p$$

In many cases, it is useful to extend this elimination over the computational levels:

$$\mathsf{Eq\_rec}_i : \Pi A : \mathsf{Type}(i).\Pi P : A \rightarrow \mathsf{Type}(i).\Pi a, b : A.(P\ a) \rightarrow a =_A b \rightarrow (P\ b).$$

There is however a particularity to $\mathsf{Eq\_rec}$: in Coq, it is defined by case analysis and therefore comes with a computation rule. The term $(\mathsf{Eq\_rec}\ A\ P\ a\ b\ p\ e)$ of type $(P\ b)$ reduces to p in the case where e is a canonical proof by reflexivity; in this case, a and b are convertible and thus coherence and normalization of the type theory are preserved.

As shown in the next section, such a reduction rule is useful, especially when programming with dependent types. In our proof-irrelevant theory however, we cannot rely on the information given by the equality proof e, since all equality proofs are treated as convertible. Furthermore, allowing, for any e, the reduction rule $(\mathsf{Eq\_rec}\ A\ P\ a\ b\ p\ e) \rhd p$ is too permissive, since it easily breaks the subject reduction property in incoherent contexts.

We therefore put the burden of checking convertibility between a and b on the reduction rule of $\mathsf{Eq\_rec}$ by extending reduction with the following, non-linear rule:

$$(\mathsf{Eq\_rec}\ A\ P\ a\ a\ p\ e) \rhd p$$

or the equivalent conditional rule:

$$(\mathsf{Eq\_rec}\ A\ P\ a\ b\ p\ e) \rhd p \qquad \text{if } a =_\varepsilon b$$

.

Again, we do not detail meta-theory here, but the various lemmas of the previous section still hold when $=_\varepsilon$ is enriched with this new reduction.

3 Programming with dependent types

We now list some applications of the relaxed conversion rule, which all follow the slogan that proof-irrelevance makes programming with dependent types more convenient and efficient. From now on, we will write $\{x : A|P\}$ for $\Sigma^* x : A.P$, that is for a Σ-type whose second component is non-computational.

3.1 Dependant equality

Programming with dependent types means that terms occur in the type of computational objects (i.e. not only in propositions). The way equality is handled over such families of types is thus a crucial point which is often problematic in intensional type theories.

Let us take a simple example. Consider we have defined a data-type of arrays over some type A. If n is a natural number, $(\mathsf{tab}\ n)$ is the type of arrays of size n. That is $\mathsf{tab} : \mathsf{nat} \to \mathsf{Type}(i)$.

Commutativity of addition can be proved in the theory: $\mathsf{com} : \Pi m, p : \mathsf{nat}.(m + p) = (p + m)$ is inhabited. Yet $\mathsf{tab}\ (m + p)$ and $\mathsf{tab}\ (p + m)$ are two distinct types with distinct inhabitants; the operator Eq_rec described above only allows to construct a translation function from one to the other:

$$tr : \Pi n : nat.(\mathsf{tab}\ (m + p)) \to (\mathsf{tab}\ (p + m)).$$

The problem is proving that this function indeed ultimately behaves like the identity; typically proving:

$$\Pi i : \mathsf{nat}.\Pi x : i < p + m \to (t\ i\ x) = (tr\ n\ t\ i\ (\mathsf{com}\ m\ p\ x)).$$

It is known [18, 15], that to do so, one needs the reduction rule for Eq_rec together with a proof that equality proofs are unique. The latter property being generally established by a variant of what Streicher calls the “K axiom”:

$$K : \Pi A : \mathsf{Type}.\Pi a : A.\Pi P : a =_A a \to \mathsf{Prop}.(P\ (\mathsf{refl}\ a)) \to \Pi e : a =_A a.(P\ e)$$

where refl stands for the canonical proof by reflexivity.

Here since equality proofs are also irrelevant to conversion, this axiom becomes trivial. Actually, since $(P\ e)$ and $(P\ (\mathsf{refl}\ a))$ are convertible, this statement does not even need to be mentioned anymore, and the associated reduction rule becomes superfluous.

In general, it should be interesting to transpose McBride’s work [18] in the framework of proof-irrelevant theories.

3.2 partial functions and equality over subset types

In the literature of type theory, subset types come in many flavors; they designate the restriction of a type to the elements verifying a certain predicate. The type $\{x : A|P\}$ can be viewed as the constructive statement "there exists an element of A verifying P", but also as the data-type A restricted to elements verifying P. In most current type theories, the latter approach is not very practical since equality is defined over it in a too narrow way. We have $< a, p > =_\beta < a', p' >$ only if $a =_\beta a'$ *and* $p =_\beta p'$; the problem is that one would like to get rid of the second condition. The same is true for propositional Leibniz equality and one can establish:

$$< a,p > \;=_{\{x:A|P\}}\; < a,p' > \;\rightarrow p =_{P[x\backslash a]} p'.$$

In general however, one is only interested in the validity of the assertion $(P\ a)$, not the way it is proved. A program awaiting an argument of type $\{x : A|P\}$ will behave identically if fed with $< a,p >$ or $< a,p' >$.

Therefore, each time a construct $\{x : A|P\}$ is used indeed as a data-type, one cannot use Leibniz equality in practice. Instead, one has to define a less restrictive equivalence relation $\simeq_{A,P}$ which simply states that the two first components of the pair are equal:

$$< a,p > \;\simeq_{A,P}\; < a',p' > \qquad \equiv \qquad a = a'.$$

But using $\simeq_{A,P}$ instead of $=_{\{x:A|P\}}$ quickly becomes very tedious; typically, for every function $f : \{x : A|P\} \rightarrow B$ one has to prove

$$\Pi c, c' : \{x : A|P\}\ .\ c \simeq_{A,P} c' \rightarrow (f\ c) =_B (f\ c')$$

and even more specific statements if B is itself a subset type.

In our theory one can prove without difficulties that $=_{\{x:A|P\}}$ and $\simeq_{A,P}$ are equivalent, and there is indeed no need anymore for defining $\simeq_{A,P}$. Furthermore, one has $< a,p > \;=_\varepsilon\; < a,p' >$, so the two terms are computationally identified which is stronger than Leibniz equality, avoiding the use of the deductive level and makes proofs and developments more concise.

Array bounds A typical example of the phenomenon above is observed when dealing with partial functions. For instance when an array t of size n is viewed as a function taking an index i as argument, together with a proof that i is less than n. That is:

$$t : (\mathsf{tab}\ n) \quad \text{with} \quad \mathsf{tab} \equiv \Pi i : \mathsf{nat}.i < n \rightarrow A.$$

In traditional type theory, this definition is cumbersome to use, since one has to state explicitly that the values $(t\ i\ p_i)$, where $p_i : i < n$ *do not depend upon* p_i. The type above is therefore not sufficient to describe an array; instead one needs the additional condition:

$$T_{irr} : \Pi i : nat.\Pi p_i, p'_i : i < n.(t\ i\ p_i) =_A (t\ i\ p'_i)$$

where $=_A$ stands for the propositional Leibniz equality.

This is again verbose and cumbersome since T_{irr} has to be invoked repeatedly. In our theory, not only the condition T_{irr} becomes trivial, since for any p_i and p'_i one has $(t\ i\ p_i) =_\varepsilon (t\ i\ p'_i)$, but this last coercion is stronger than propositional equality: there is no need anymore to have recourse to the deductive level and prove this equality. The proof terms are therefore clearer and smaller.

3.3 On-the-fly extraction

An important point, which we can only briefly mention here is the consequence for the implementation when switching to a proof-irrelevant theory. In a proof-checker, the environment consists of a sequence of definitions or lemmas which have been type-checked. If the proof-checker implements a proof-irrelevant theory, it is reasonable to keep two versions of each constant: the full proof-term, which can be printed or re-checked, and the extracted one (that is $\rightarrow_\varepsilon$-normalized) which is used for conversion check. This would be even more natural when building on recent Coq implementations which already use a dual storing of constants, the second representation being non-printable compiled code precisely used for fast conversion check.

In other words, a proof-system built upon a theory as the one presented here would allow the user to efficiently exploit the computational behavior of a constructive proof in order to prove new facts. This makes the benefits of program extraction technology available inside the system and helps transforming proof-system into viable programming environments.

4 Relating to PVS

Subset types also form the core of PVS. In this formalism the objects of type $\{x : A|P\}$ are also of type A. This makes type checking undecidable and is thus impossible in our setting. But we show that it is possible to build explicit coercions between the corresponding types of our theory which basically behave like the identity.

The following lemma states that the construction and destruction operations of our subset types can actually be omitted when checking conversion:

Lemma 8 (Singleton simplification). *The typing relation of our theory remains unchanged if we extend the $\rightarrow_\varepsilon$ reduction of our theory by :*

$$\begin{aligned} < a, p >_{\Sigma^* x:A.P} &\rightarrow_\varepsilon a \\ \pi_1^*(c) &\rightarrow_\varepsilon c. \end{aligned}$$

The following definition is directly transposed[2] from PVS [23]. We do not treat dependent types in full generality (see chapter 3 of [23]).

Definition 4 (Maximal super-type). *The maximal super-type is a partial function μ from terms to terms, recursively defined by the following equations. In all these equations, A and B are of type $\mathsf{Type}(i)$ in a given context.*

$$\mu(A) \equiv A \quad \textit{if } A \textit{ is a data-type} \qquad\qquad \mu(\{x : A|P\}) \equiv \mu(A)$$

$$\mu(A \rightarrow B) \equiv A \rightarrow \mu(B) \qquad\qquad \mu(A \times B) \equiv \mu(A) \times \mu(B).$$

[2] A difference is that in PVS, propositions and booleans are identified; but this point is independent with this study. It is however possible to do the same in our theory by assuming a computational version of excluded-middle.

Definition 5 (η-reduction). *The generalized η-reduction, written $\triangleright_\eta$, is the contextual closure of:*

$$\lambda x : A.(t\ x) \triangleright_\eta t \quad \textit{if } x \textit{ is not free in } t$$
$$< \pi_1(t), \pi_2(t) > \triangleright_\eta t.$$

We can now construct the coercion function between A and $\mu(A)$:

Lemma 9. *If $\Gamma \vdash A : \mathsf{Type}(i)$ and $\mu(A)$ is defined, then:*

- *$\Gamma \vdash \mu(A) : \mathsf{Type}(i)$,*
- *there exists a function $\overline{\mu}(A)$ which is of type $A \to \mu(A)$ in Γ,*
- *furthermore, when applying the singleton simplification $\mathcal{S}$ to $\overline{\mu}$ one obtains an η-expansion of the identity function; to be precise: $\mathcal{S}(\overline{\mu}) \to^*_{\varepsilon\eta} \lambda x : B.x$.*

Proof. It is almost trivial to check that $\Gamma \vdash \mu(A) : \mathsf{Type}(i)$. The two other clauses are proved by induction over the structure of A.

- If $f : A \to \mu(A)$, then $g \equiv \lambda x : \{x : A|P\}.(f\ \pi_1(x)) : \{x : A|P\} \to \mu(A)$. Furthermore $\pi_1(x)$ is here simplified to x, since $P : \mathsf{Prop}$. Since $(\mathcal{S}(f)\ x) \triangleright^*_{\varepsilon\eta} x$, $\mathcal{S}(g) \triangleright^*_{\varepsilon\eta} \lambda : x : \{x : A|P\}.x$.
- If $f : B \to \mu(B)$, then $g \equiv \lambda h : A \to B.\lambda x : A.(f\ (h\ x)) : A \to \mu(B)$. Since $(\mathcal{S}(f)\ (h\ x)) \triangleright^*_{\varepsilon\eta} (h\ x)$, we have $g \triangleright^*_{\varepsilon\eta} \lambda h : A \to B.h$.
- If $f_A : A \to \mu(A)$ and $f_B : B \to \mu(B)$, then $g \equiv \lambda x : A \times B. < (f_A\ \pi_1(x)), (f_B\ \pi_2(x)) >_{A\times B}$ is of the expected type. Again, the induction hypotheses assure that $g \triangleright^*_{\varepsilon\eta} \lambda x : A \times B.x$.

The opposite operation, going from from $\mu(A)$ to A, can only be performed when some conditions are verified (TCC's in PVS terminology). We can also transpose this to our theory, still keeping the simple computational behavior of the coercion function. This time however, our typing is less flexible than PVS', we have to define the coercion function and its type simultaneously; furthermore, in general, this operation is well-typed only if the type-theory supports generalized η-reduction.

This unfortunate restriction is typical when defining transformations over programs with dependent types. It should however not be taken too seriously, and we believe this cosmetic imperfection can generally be tackled in practice[3].

Lemma 10 (subtype constraint). *Given $\Gamma \vdash A : \mathsf{Type}(i)$, if $\mu(A)$ is defined, then one can define $\pi(A)$ and $\overline{\pi}(A)$ such that,* in the theory where conversion is extended with $\triangleright_\eta$, *one has:*

$$\Gamma \vdash \pi(A) : \mu(A) \to \mathsf{Prop} \quad \textit{and} \quad \Gamma \vdash \overline{\pi}(A) : \Pi x : \mu(A).(\pi(A)\ x) \to A.$$

Furthermore, $\overline{\pi}(A)$ $\triangleright_{\varepsilon\eta}$-normalizes to $\lambda x : \mu(A).\lambda p : (\pi(A)\ x).x$.

Proof. By straightforward induction. We only detail some the case where $A = B \to C$. Then $\pi(A) \equiv \lambda f : A \to \mu(B).\forall x : A.(\pi(B)\ (f\ x))$ and $\overline{\pi}(A) \equiv \lambda f : A \to \mu(B).\lambda p : \forall x : A.(\pi(B)\ (f\ x)).\lambda x : A.(\overline{\pi}(B)\ (f\ x)\ (p\ x))$.

[3] For one, in practical cases, η-does not seem necessary very often (only with some nested existentials). And even then, it should be possible to tackle the problem with by proving the corresponding equality on the deductive level.

5 A more extensional theory

Especially during the 1970ies and 1980ies, there was an intense debate about the respective advantages of intensional versus extensional type theories. The latter denomination seems to cover various features like replacing conversion by propositional equality in the conversion rule or adding primitive quotient types. In general, these features provide a more comfortable construction of some mathematical concepts and are closer to set-theoretical practice. But they break other desirable properties, like decidability of type-checking and strong normalization.

The theory presented here should therefore be considered as belonging to the intentional family. However, we retrieve some features usually understood as extensional.

5.1 The axiom of choice

Consider the usual form of the (typed) axiom of choice (AC):

$$(\forall x : A.\exists y : B.R(x, y)) \Rightarrow \exists f : A \to B.\forall x : A.R(x, f\ x).$$

When we transpose it into our type theory, we can chose to translate the existential quantifier either by a Σ-type, or the existential quantifier defined in Prop :

$$\exists x : A.P \equiv \Pi Q : \mathsf{Prop}.(\Pi x : A.P \to Q) \to Q \ \ : \mathsf{Prop}$$

If we use a Σ-type, we get a type which obviously inhabited, using the projections π_1 and π_2. However, if we read the existential quantifiers of AC as defined above, we obtain a (non-computational) proposition which is not provable in type theory.

Schematically, this propositions states that if $\Pi x : A.\exists y : B.R(x, y)$ is provable, then the corresponding function from A to B exists "in the model". This assumption is strong and allows to encode IZF set theory into type theory (see [26]).

What is new is that our proof-irrelevant type theory is extensional enough to perform the first part of Goodman and Myhill's proof based on Diaconescu's observation. Assuming AC, we can prove the decidability of equality. Consider any type A and two objects a and b of type A. We define:

$$\{a, b\} \equiv \{x : A | x = a \vee x = b\}$$

Let us write a' (resp. b') for the element of $\{a, b\}$ corresponding to a (resp. b); so $\pi_1(a') =_\varepsilon a$ and $\pi_1(b') =_\varepsilon b$. It is the easy to prove that

$$\Pi z : \{a, b\}.\exists e : \mathsf{bool}.(e = \mathsf{true} \wedge \pi_1(z) = a) \vee (e = \mathsf{false} \wedge \pi_1(z) = b)$$

and from the axiom of choice we deduce:

$$\exists f : \{a, b\} \to \mathsf{bool}.\Pi z : \{a, b\}.(f\ z = \mathsf{true} \wedge \pi_1(z) = a) \vee (f\ z = \mathsf{false} \wedge \pi_1(z) = b)$$

Finally given such a function f, one can compare $(f\ a')$ and $(f\ b')$, since both are booleans over which equality is decidable.

The key point is then that, thanks to proof-irrelevance, the equivalence between $a' = b'$ and $a = b$ is provable in the theory. Therefore, if $(f\ a')$ and $(f\ b')$ are different, so are a and b. On the other hand, if $(f\ a') = (f\ b') = \mathsf{true}$ then $\pi_1(b') = a$ and so $b = a$. In the same way, $(f\ a') = (f\ b') = \mathsf{false}$ entails $b = a$.

We thus deduce $a = b \vee a \neq b$ and by generalizing with respect to a, b and A we obtain:

$$\Pi A : \mathsf{Type}(i).\Pi a, b : A.a = b \vee a \neq b$$

which is a quite classical statement. We have formalized this proof in Coq, assuming proof-irrelevance as an axiom.

Note of course that this "decidability" is restricted to a disjunction in Prop and that it is not possible to build an actual generic decision function. Indeed, constructivity of results in the predicative fragment of the theory are preserved, even if assuming the excluded-middle in Prop.

5.2 Other classical non-computational axioms

At present, we have not been able to deduce the excluded middle from the statement above[4]. We leave this theoretical question to future investigations but it seems quite clear that in most cases, when admitting AC one will also be willing to admit EM. In fact both axioms are validated by the simple set-theoretical model and give a setting where the $\mathsf{Type}(i)$'s are inhabited by computational types (i.e. from $\{x : A|P\}$ we can compute x of type A) and Prop allows classical reasoning about those programs.

Another practical statement which is validated by the set-theoretical model is the axiom that point-wise equal functions are equal :

$$\Pi A, B : \mathsf{Type}(i).\Pi f, g : A \to B.\Pi x : A.f\ x = g\ x \to f = g.$$

Note that combining this axiom with AC (and thus decidability of equality) is already enough to prove (in Prop) the existence of a function deciding whether a Turing machine halts.

5.3 Quotients and normalized types

Quotient sets are a typically extensional concept whose adaptation to type theory has always been problematic. Again, one has to chose between "effective" quotients and decidability of type-checking. Searching for a possible compromise, Courtieu [9] ended up with an interesting notion of *normalized type*[5]. The idea is remarkably simple: given a function $f : A \to B$, we can define $\{(f\ x)|x : A\}$

[4] In set theory, decidability of equality entails the excluded middle, since $\{x \in \mathbb{N}|P\}$ is equal to $\mathbb{N}$ if and only if P holds.

[5] A similar notion has been developped for NuPRL [22].

which is the subtype of B corresponding to the codomain of f. His rules are straightforwardly translated into our theory by simply taking:

$$\{f(x)|x : A\} \equiv \{y : B|\exists x : A.y = f\ x\}$$

Courtieu also gives the typing rules for functions going from A to $\{f(x)|x : A\}$, and back in the case where f is actually of type $A \rightarrow A$.

The relation with quotients being that in the case $f : A \rightarrow A$ we can understand $\{f(x)|x : A\}$ as the type A quotiented by the relation

$$x\ R\ y \quad \Longleftrightarrow \quad f\ x = f\ y$$

In practice this appears to be often the case, and Courtieu describes several applications.

6 Conclusion and further work

We have tried to show that a relaxed conversion rule makes type theories more practical, without necessarily giving up normalization or decidable type checking. In particular, we have shown that this approach brings together the world of PVS and type theories of the Coq family.

We also view this as a contribution to closing the gap between proof systems like Coq and safe programming environments like Dependant ML or ATS [7, 27]. But this will only be assessed by practice; the first step is thus to implement such a theory.

7 Acknowledgements

Christine Paulin deserves special thanks, since she me gave the idea of a relaxed conversion rule long ago as well as the computation rule of paragraph 2.5. This work also benefited from discussions with Bruno Barras, Hugo Herbelin and Benjamin Grégoire. Anonymous referees gave numerous useful comments and hints.

References

1. T. Altenkirch. "Proving strong normalization for CC by modifying realizability semantics". In H. Barendregt and T. Nipkow Eds, *Types for Proofs and Programs*, LNCS 806, 1994.
2. T. Altenkirch. Extensional Equality in Intensional Type Theory. LICS, 1999.
3. H. Barendregt. Lambda Calculi with Types. Technical Report 91-19, Catholic University Nijmegen, 1991. In Handbook of Logic in Computer Science, Vol II, Elsevier, 1992.
4. G. Barthe. The relevance of proof-irrelevance. In K.G. Larsen, S. Skyum, and G. Winskel, editors, Proceedings of ICALP'98, volume 1443 of LNCS, pages 755-768. Springer-Verlag, 1998.

5. F. Blanqui. Definitions by rewriting in the Calculus of Constructions. MSCS, vol. 15(1), 2003.
6. J. Caldwell. "Moving Proofs-as-Programs into Practice", In Proceedings of the 12^{th} IEEE International Conference on Automated Software Engineering, IEEE, 1997.
7. Chiyan Chen and Hongwei Xi, Combining Programming with Theorem Proving. ICFP'05, 2005.
8. The Coq Development Team. *The Coq Proof-Assistant User's Manual*, INRIA. On `http://coq.inria.fr/`
9. P. Courtieu. "Normalized Types". *Proceedings of CSL 2001*, L. Fribourg Ed., LNCS 2142, Springer, 2001.
10. E. Gimenez. "A Tutorial on Recursive Types in Coq". INRIA Technical Report. 1999.
11. G. Gonthier. A computer-checked proof of the Four Colour Theorem. Manuscript, 2005.
12. B. Grégoire and X. Leroy. "A compiled implementation of strong reduction", proceedings of ICFP, 2002.
13. Compilation des termes de preuves: un (nouveau) mariage entre Coq et Ocaml. Thèse de doctorat, Université Paris 7, 2003.
14. A computational approach to Pocklington certificates in type theory. In proceedings of FLOPS 2006, M. Hagiya and P. Wadler (Eds), LNCS, Springer-Verlag, 2006.
15. M. Hofmann and T. Streicher. "A groupoid model refutes uniqueness of identity proofs". LICS'94, Paris. 1994.
16. Z. Luo. "ECC: An Extended Calculus of Constructions." Proc. of IEEE LICS'89. 1989.
17. P. Martin-Löf. *Intuitionistic Type Theory.* Studies in Proof Theory, Bibliopolis, 1984.
18. C. McBride. "Elimination with a Motive". Proceedings of TYPES'00, 2000.
19. J. McKinna and R. Pollack. "Pure Type Systems formalized", in TLCA'93, M. Bezem and J. F. Groote Eds, LNCS 664, Springer-Verlag, Berlin, 1993.
20. P.-A. Melliès and B. Werner. "A Generic Normalization Proof for Pure Type System" in TYPES'96, E. Gimenez and C. Paulin-Mohring Eds, LNCS 1512, Springer-Verlag, Berlin, 1998.
21. A. Miquel and B. Werner. "The not so simple proof-irrelevant model of CC". In Geuvers and Wiedijk (Eds.), Proceedings of TYPES 2002, LNCS 2646, 2003.
22. A. Nogin and A. Kopilov. "Formalizing Type Operations Using the "Image" Type Constructor". To appear in WoLLIC 2006, ENTCS.
23. S. Owre and N. Shankar. "The Formal Semantics of PVS". SRI Technical Report CSL-97-2R. Revised March 1999.
24. C. Paulin-Mohring. *Extraction de Programmes dans le Calcul des Constructions.* Thèse de doctorat, Université Paris 7, 1989.
25. F. Pfenning. "Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory". Proceedings of LICS, IEEE, 2001.
26. B. Werner. "Sets in Types, Types in Sets". In, M. Abadi and T. Itoh (Eds), Theoretical Aspects of Computer Science, TACS'97, LNCS 1281, Springer-Verlag, 1997.
27. Hongwei Xi, Dependent Types in Practical Programming, Ph.D, CMU, 1998.