Logical Methods in Computer Science
Vol. 4 (3:13) 2008, pp. 1–20
www.lmcs-online.org

Submitted Feb. 13, 2007
Published Sep. 26, 2008

ON THE STRENGTH OF PROOF-IRRELEVANT TYPE THEORIES

BENJAMIN WERNER

INRIA Saclay – Île-de-France and LIX, Ecole Polytechnique, 91128 PALAISEAU cedex, France
e-mail address: 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 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. 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. Finally we describe a simple set-theoretic semantics.

1. Introduction

A formal proof system, or proof assistant, implements a formalism in a similar way a compiler implements a programming language. Among existing formalisms, 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 (the "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 [15] or formal primality proofs [18].
(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 $A \to B$, whose input and result are certified to be related by P. Up to now however, program extraction

1998 ACM Subject Classification: F.4.1,F.3.1.
Key words and phrases: logic, proofs, coq, lambda-calculus, types.

DOI:10.2168/LMCS-4 (3:13) 2008

was more an external feature of implemented proof systems[1]: programs certified by extraction are no longer objects of the formalism and cannot be used to assert facts like in the point above.

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 not objects of 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 thus is tempting 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. The extraction function we use is quite close to Letouzey's [21, 22], except that we discard the inclusion rule $\mathsf{Prop} \subset \mathsf{Type}$, which would complicate the definition of the type theory and the semantics (see [29] for the last point).

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 means for proving properties of other systems. This is the case for the work of Altenkirch [3] and Barthe [6]. 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. The NuPRL approach using a squash type [9] is very close to ours, but the extentional setting gives somewhat different results. Finally, let us mention recent work [5] by Barras and Bernardo who present a type theory with implicit arguments. This interesting proposal can be understood as a theory with proof-irrelevance, where the computational fragment is precisely Miquel's calculus [28]. Their proposal can be understood as a theory similar to ours, but with a more sophisticated way to mark what is computational and what is not.

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. In PTS's, the types of types are *sorts*; the set of sorts is

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

[1] Except NuPRL; see related work.

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 to indicate 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) [23].

The syntax of the ECC fragment is therefore

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

We sometimes call *raw terms* these terms, when we want to stress that they are considered independently of typing issues. The tagging of Σ is there to indicate whether the second component of the pair is computational or not (the first component will always be). For the same technical reason, we also tag the second projection π_2.

We will sometimes write x for x_{s}, $\Sigma x : A.B$ for $\Sigma^{\mathsf{s}} x : A.B$ or $\pi_2(t)$ for $\pi_2^{\mathsf{s}}(t)$ 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 or a second projection $\pi_2(t)$ is computationally irrelevant when tagged with the $*$ mark. This leads to the following definition.

Definition 2.1 (Extraction)**.** We can simply define the extraction relation $\rhd_\varepsilon$ as the contextual closure of the following rewriting equations

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

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

Definition 2.2 (Reduction)**.** The β-reduction $\rhd_\beta$ is defined as the contextual closure of the following equations

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

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 we will see that these restrictions are always satisfied on well-typed terms, but are necessary in order to assert the meta-theoretic properties below. While these restrictions are specific to our way of marking computational terms, other methods will probably yield similar technical difficulties.

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

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 $=_{\beta\varepsilon}$). Note that this notion of subtyping should not be confused with, for instance, subtyping of subset types in the style of PVS.

Definition 2.3 (Syntactic subtyping)**.** The subtyping relation is defined on raw-terms as the transitive closure of the following equations

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

2.3. **Functional fragment typing rules.** The typing rules for the kernel of our theory are given in PTS-style [4] 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 $\mathsf{Prop} < \mathsf{Type}(0) < \mathsf{Type}(1) < \dots$

2.4. **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

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

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

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

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

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

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

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

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

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

$$(\textsc{Lam})\dfrac{\Gamma(x : A) \vdash t : B}{\Gamma \vdash \lambda x : A.t : \Pi x : A.B} \qquad (\textsc{App})\dfrac{\Gamma \vdash t : \Pi x_\mathsf{s} : A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash (t\ u) : B[x \setminus u]}(\text{if } s(u) = \mathsf{s})$$

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

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

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

Figure 1: The ECC fragment

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

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

There is however a peculiarity 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 $(\mathsf{Eq\_rec}\ A\ P\ a\ b\ p\ e) \triangleright p$ 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, conditional rule

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

When being precise, this means that $=_{\varepsilon\beta}$ and $\rhd$ are actually two mutual inductive definitions.

An alternative would be the non-linear rule

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

but this allows an encoding of Klop's counter-example [20] and thus breaks the Church-Rosser property (for untyped terms). We thus develop the metatheory for the first version.

2.5. **Generalization.** In Coq, computational eliminations are provided for more inductive definitions than just propositional equality. The condition is that

(1) The definition has at most one constructor,
(2) the arguments of this constructor are all, themselves, non-computational.

It appears that it is reasonably straightforward to extend our type theory, by generalizing the $\mathsf{Eq\_rec}$ feature, in order to capture this Coq behavior in the case where the inductive definition is non-recursive. We briefly indicate how but without precise justification. The remainder of this paragraph is thus not considered in the meta-theoretical justifications; it is also not necessary for the rest of the article.

We write $\Pi\vec{x} : \vec{A}.T$ for $\Pi x_1 : A_1. \ldots \Pi x_n : A_n.T$ and $t\ \vec{u}$ for $(t\ u_1\ \ldots\ u_n)$.

Consider an inductive definition $I : \Pi\vec{x} : \vec{A}.\mathsf{Prop}$ with a unique constructor $c : \Pi\vec{y} : \vec{B}.(I\ \vec{u})$. The non-computational elimination scheme is

$$\mathrm{I\_ind} : \Pi P : (\Pi\vec{x} : \vec{A}.\mathsf{Prop}).(\Pi\vec{y} : \vec{B}.P\ \vec{u}) \to \Pi\vec{x} : \vec{A}.I\ \vec{x} \to P\ \vec{x}$$

with the reduction rule

$$(\mathrm{I\_ind}\ X\ p\ \vec{a}\ (c\ \vec{b})) \rhd (p\ \vec{b})$$

We can then provide a computational elimination

$$\mathrm{I\_rec} : \Pi X : (\Pi\vec{x} : \vec{A}.\mathsf{Type}).(\Pi\vec{y} : \vec{B}.X\ \vec{u}) \to \Pi\vec{x} : \vec{A}.I\ \vec{x} \to X\ \vec{x}$$

with the following reduction rule

$$(\mathrm{I\_rec}\ X\ p\ \vec{a}\ i) \rhd (p\ \vec{\varepsilon}) \qquad \text{if } \vec{u} =_{\beta\varepsilon} \vec{a}$$

To understand the last condition, one should note that although the variables $\vec{y}$ are free in $\vec{u}$, they do not interfere with the conversion since their types ensure they are all tagged by $*$.

2.6. **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 [11, 14]; for a possibly more modern presentation [8] is interesting.

Let us just mention that data types live in Type. That is, for instance, $\mathsf{nat} : \mathsf{Type}(0)$; thus, their elements are of tag $\diamond$.

3. Basic metatheory

We sketch the basic meta-theory of the calculus defined up to 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 [29]).

Lemma 3.1. *If $t \triangleright_\beta t'$, then $s(t) = s(t')$. Thus, the same is true if $t \triangleright^*_{\beta\varepsilon} t'$.*

Proof. By a straightforward case analysis of the form of t. □

Lemma 3.2 (β-postponement)**.** *If $t \triangleright^*_{\beta\varepsilon} t'$, then there exists t'' such that $t \triangleright^*_\varepsilon t''$ and $t'' \triangleright^*_\beta t'$.*

Proof. One first shows that if $t \triangleright_\beta t' \triangleright_\varepsilon t''$, then there exists t''' such that $t \triangleright^*_\varepsilon t'''$ and either $t''' = t''$ or $t''' \triangleright_\beta t''$. This is done by checking how the two redexes are located with respect to each other. The proof of the lemma then easily follows. □

Lemma 3.3 (Church-Rosser)**.** *For t a raw term, if $t \triangleright^*_{\beta\varepsilon} t_1$ and $t \triangleright^*_{\beta\varepsilon} t_2$, then there exists t_3 such that $t_2 \triangleright^*_{\beta\varepsilon} t_3$ and $t_2 \triangleright^*_{\beta\varepsilon} t_3$.*

Proof. By a quite straightforward adaptation of the usual Tait and Martin-Löf method. The delicate point was to choose the right formulation of the reduction rule specific to the elimination of propositional equality, as mentioned in section 2.4. □

An immediate but very important consequence is that

Corollary 3.4 (Uniqueness of product formation)**.** *If $\Pi x : A.B \leq \Pi x : A'.B'$, then $A =_{\beta\varepsilon} A'$ and $B \leq B'$.*

Corollary 3.5. *For any T,*

- $T \leq \mathsf{Prop} \iff \mathsf{Prop} \leq T \iff T =_{\beta\varepsilon} \mathsf{Prop} \iff T \triangleright^*_{\beta\varepsilon} \mathsf{Prop}$
- $T \leq \mathsf{Type}(i) \iff T \triangleright^*_{\beta\varepsilon} \mathsf{Type}(j)$ *with* $j \leq i$
- $\mathsf{Type}(i) \leq T \iff T \triangleright^*_{\beta\varepsilon} \mathsf{Type}(j)$ *with* $i \leq j$
- *if* $T \leq U$ *and* $U \leq T$ *then* $U =_{\beta\varepsilon} T$.

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

Lemma 3.6 (pre-cooking of terms)**.** *Let t_1 and t_2 be raw terms. Let t'_1 and t'_2 be their respective $\triangleright_\varepsilon$-normal forms. Then, $t_1 =_{\beta\varepsilon} t_2$ 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 3.7. *Let t_1 be any raw term. Suppose $t_1 \triangleright_\varepsilon t_2 \triangleright_\beta t_3$. Then there exists t_4 such that $t_1 \triangleright_\beta t_4 \triangleright^*_\varepsilon t_3$.*

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

Lemma 3.8. *If $t \triangleright_{\beta\varepsilon} t'$, for any term u and variable $x_{s(u)}$, one has $t[x_{s(u)} \setminus u] \triangleright_{\beta\varepsilon} t'[x_{s(u)} \setminus u]$. Thus, if $t =_{\beta\varepsilon} t'$ then $t[x_{s(u)} \setminus u] =_{\beta\varepsilon} t'[x_{s(u)} \setminus u]$.*

Proof. By straightforward induction over the structure of t. One uses the fact that, since $x_{s(u)}$ and u have the same syntactic sort, the terms t and $t[x_{s(u)} \setminus u]$ also have the same syntactic sort. □

Lemma 3.9 (Substitution)**.** *If $\Gamma(x : A)\Delta \vdash t : T$ and $\Gamma \vdash a : A$ are derivable, if a and x have the same (syntactic) sort, then $\Gamma\Delta[x \setminus a] \vdash t[x \setminus a] : T[x \setminus a]$ is derivable.*

Proof. By induction over the structure of the first derivation, like in the usual proof. The condition over the syntactic sorts is necessary for the case of the conversion rule, in order to apply the previous lemma. □

Lemma 3.10 (Inversion or Stripping)**.** *If $\Gamma \vdash t : T$ is derivable, then so are $\Gamma \vdash wf$ and $\Gamma \vdash T : s$ for some sort s. Furthermore, the following clauses hold.*

If $\Gamma \vdash x : T$ is derivable, then: • $(x, U) \in \Gamma$, • $\Gamma \vdash T : s$ *is derivable,* • $U \leq T$.	*If $\Gamma \vdash (t\ u) : V$ is derivable, then:* • $\Gamma \vdash t : \Pi x : U.W$, • $\Gamma \vdash u : U$, • $W[x \setminus u] \leq V$.
If $\Gamma \vdash \lambda x : U.t : W$ is derivable, then: • $\Gamma(x : U) \vdash t : T$, • $\Pi x : U.T \leq W$.	*If $\Gamma \vdash \Pi x : A.B : T$ is derivable, then* • $\Gamma \vdash A : s_1$, • $\Gamma(x : A) \vdash B : s_2$, • *either* $s_2 = \mathsf{Prop}$ *and* $\mathsf{Prop} \leq T$ *or* $max(s_1, s_2) \leq T$.
If $\Gamma \vdash \Sigma^* x_\diamond : A.B : T$ is derivable, then • $\Gamma \vdash A : \mathsf{Type}(i)$, • $\Gamma(x : A) \vdash B : \mathsf{Prop}$, • $\mathsf{Type}(i) \leq T$.	*If $\Gamma \vdash \Sigma^\diamond x_\diamond : A.B : T$ is derivable, then* • $\Gamma \vdash A : \mathsf{Type}(i)$, • $\Gamma(x : A) \vdash B : \mathsf{Type}(j)$, • $\mathsf{Type}(max(i, j)) \leq T$.
$\Gamma \vdash \Sigma x_* : A.B : T$ *is not derivable.*	*If $\Gamma \vdash < t, u >_{\Sigma x_\diamond : T.U} : V$ is derivable,* • $\Sigma x_\diamond : T.U \leq V$, • $\Gamma \vdash t : T$, • $\Gamma \vdash u : U[x_\diamond \setminus t]$, • $s(t) = \diamond$.
If $\Gamma \vdash \pi_1(t) : T$ is derivable, then • $\Gamma \vdash t : \Sigma x_\diamond : A.B$, • $A \leq T$.	*If $\Gamma \vdash \pi_2^{\mathsf{s}}(t) : T$ is derivable, then* • $\Gamma \vdash t : \Sigma^{\mathsf{s}} x_\diamond : A.B$, • $B[x_\diamond \setminus \pi_1(t)] \leq T$
If $\Gamma \vdash \mathsf{Prop} : T$ is derivable, then $\mathsf{Type}(1) \leq T$	*If $\Gamma \vdash \mathsf{Type}(i) : T$, then* $\mathsf{Type}(i + 1) \leq T$

Proof. Simultaneously by induction over the derivation. □

Corollary 3.11 (Principal type)**.** *If $\Gamma \vdash t : T$, then there exists U such that $\Gamma \vdash t : U$ and for all V, if $\Gamma \vdash t : V$, then $U \leq V$.*

Proof. By induction over the structure of t, using the previous lemma and corollaries 3.4 and 3.5. □

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

Lemma 3.12 (Subject reduction)**.** *If $\Gamma \vdash t : T$ is derivable, if $t \rhd_\beta t'$ (resp. $T \rhd_\beta T'$, $\Gamma \rhd_\beta \Gamma'$), then $\Gamma \vdash t' : T$ (resp. $\Gamma \vdash t : T', \Gamma' \vdash t : T$).*

Proof. By induction over the structure of t. Depending upon the position of the redex, one uses either the substitution or the stripping lemmas above. We only detail the case where a β-reduction occurs at the root of the term.

If $t = \lambda x^{\mathsf{s}} : U.v\ u$, $s(u) =_{\mathsf{s}}$ and $t' = v[x^{\mathsf{s}} \setminus u]$, we know that $\Gamma(x^{\mathsf{s}} : U) \vdash v : V$, $\Gamma \vdash u : U$ and $V[x^{\mathsf{s}} \setminus u] \leq T$. Thus we can apply lemma 3.9 to deduce

$$\Gamma \vdash v[x^{\mathsf{s}} \setminus u] : V[x^{\mathsf{s}} \setminus u]$$

and

$$\Gamma \vdash V[x^{\mathsf{s}} \setminus u] : s$$

where s is the sort such that $\Gamma(x^{\mathsf{s}} : U) \vdash V : s$. The result then follows through one application of the conversion rule. □

Lemma 3.13. *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 $*$.*

Proof. By induction over the structure of t. The Church-Rosser property ensures that Prop and $\mathsf{Type}(i)$ are not convertible. □

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 the simple set-theoretical model using an interpretation of types as saturated Λ-sets as first proposed by Altenkirch [2, 27].

Conjecture 3.14 (Strong Normalization)**.** If $\Gamma \vdash t : T$ is derivable, then t is strongly normalizing.

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

Corollary 3.15. *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.*

Proof. By induction over the structure of t, using the stripping lemma. Normalization ensures that the relation $\leq$ is decidable for well-formed types. □

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

Corollary 3.16. *If $[] \vdash t : \Sigma x : A.B$, then $t \rhd_\beta^* < a, b >_{\Sigma x:A.B}$ with $[] \vdash a : A$ and $[] \vdash b : B[x \setminus a]$.*

Proof. By case analysis over the normal form of t, using the stripping lemma. □

4. 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.

4.1. **Dependent 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. Suppose 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 $\mathsf{tab} : \mathsf{nat} \to \mathsf{Type}(i)$. Furthermore, let us assume we have a function modeling access to the array $\mathsf{acc} : \Pi n : \mathsf{nat}.\mathsf{tab}\ n \to \mathsf{nat} \to A$.

Commutativity of addition can be proved in the theory: $\mathsf{com} : \Pi m, p : \mathsf{nat}.(m + p) = (p + m)$. Yet $\mathsf{tab}\ (m + p)$ and $\mathsf{tab}\ (p + m)$ are two distinct types with distinct inhabitants. For instance, if we have an array $t : \mathsf{tab}\ (m + p)$, we can use the operator $\mathsf{Eq\_rec}$ described above to transform it into an array of size $p + m$

$$t' \equiv \mathsf{Eq\_rec}\ nat\ \mathsf{tab}\ (m + p)\ (p + m)\ t\ (\mathsf{com}\ (m + p)\ (p + m))\ \ :\ \ \mathsf{tab}(p + m)$$

Of course, t and t' should have the same inhabitants, and we would like to prove

$$\Pi i : \mathsf{nat}.\mathsf{acc}\ (m + p)\ t\ i =_A \mathsf{acc}\ (p + m)\ t'\ i$$

It is known [19, 25] that in order to do so, one needs the reduction rule for $\mathsf{Eq\_rec}$ 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 work like McBride's [25] in the framework of proof-irrelevant theories.

4.2. **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 intensional 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

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

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

$$< a, p > \ \simeq_{A,P}\ < a', p' > \quad \equiv \quad a =_A a'$$

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

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

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 $< a, p > =_{\beta\varepsilon} < a, p' >$, so the two terms are computationally identified which is stronger than Leibniz equality, avoids the use of the deductive level and makes proofs and developments more concise.

Array bounds. The same can be observed when partial functions are curryfied. Let us take again the example of arrays, but suppose this time the access function awaits a proof that the index is within the bounds of the array.

$$\begin{aligned} \mathsf{tab} &\ :\ \mathsf{nat} \to \mathsf{Type}(i) \\ \mathsf{acc} &\ :\ \Pi n : nat.\mathsf{tab}\ n \to \Pi i : nat.i < n \to A \end{aligned}$$

So given an array t of size n, its corresponding access function is

$$a \equiv \mathsf{acc}\ n\ t\ \ :\ \ \Pi i : nat.i < n \to A$$

In traditional type theory, this definition is cumbersome to use, since one has to state explicitly that the values $(a\ i\ p_i)$, 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

$$T_{irr} : \Pi i : nat.\Pi p_i, p_i' : i < n.(a\ i\ p_i) =_A (a\ i\ p_i')$$

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 $(a\ i\ p_i) =_{\beta\varepsilon} (a\ i\ p_i')$, 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.

4.3. **On-the-fly extraction.** An important point, which we 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 $\rhd_\varepsilon$-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.

5. 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, and objects of type A can be of type $\{x : A|P\}$. 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.

What is presented in this section is strongly related to the work of Sozeau [35], which describes a way to provide a PVS style input mode for Coq.

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

Lemma 5.1 (Singleton simplification)**.** *The typing relation of our theory remains unchanged if we extend the* $\triangleright_\varepsilon$ *reduction of our theory by*[2]*.*

$$\begin{array}{rcl} < a, p >_{\Sigma^* x:A.P} & \triangleright_\varepsilon & a \\ \pi_1(c) & \triangleright_\varepsilon & c \textit{ when } c : \Sigma^* x : A.B \end{array}$$

The following definition is directly transposed[3] from PVS [31]. We do not treat dependent types in full generality (see chapter 3 of [31]).

Definition 5.2 (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.

$$\begin{array}{rclcrcl} \mu(A) & \equiv & A \quad \text{if } A \text{ is a data-type} & & \mu(\{x : A|P\}) & \equiv & \mu(A) \\ \mu(A \to B) & \equiv & A \to \mu(B) & & \mu(A \times B) & \equiv & \mu(A) \times \mu(B). \end{array}$$

Definition 5.3 (η-reduction)**.** The generalized η-reduction, written $\triangleright_\eta$, is the contextual closure of

$$\begin{array}{rcl} \lambda x : A.(t\ x) & \triangleright_\eta & t \ \text{ if } x \text{ is not free in } t \\ < \pi_1(t), \pi_2(t) > & \triangleright_\eta & t \end{array}$$

We can now construct the coercion function from A to $\mu(A)$.

Lemma 5.4. *If* $\Gamma \vdash A : \mathsf{Type}(i)$ *and* $\mu(A)$ *is defined, then*

- $\Gamma \vdash \mu(A) : \mathsf{Type}(i)$,

[2] To make the second clause rigorous, a solution is to modify slightly the theory by adding a tag the first projection ($\pi_1^*(t)$ and $\pi_1^\diamond(t)$). This does not significantly change the metatheory.

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

- *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 $\overline{\mu}(A)$ one obtains an η-expansion of the identity function; to be precise, $\mathcal{S}(\overline{\mu}(A)) \rhd^*_{\varepsilon\beta\eta} \lambda x : A.x$.*

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 A is of the form $\{x : B|P\}$ with $\overline{\mu}(B) : B \to \mu(B)$, then
$$\overline{\mu}(A) \equiv \lambda x : \{x : B|P\}.(\overline{\mu}(B)\ \pi_1(x)) : \{x : B|P\} \to \mu(B)$$
Furthermore, since $P : \mathsf{Prop}$, $\pi_1(x)$ is here simplified to x, and by induction hypothesis we know that $\mathcal{S}(\overline{\mu}(B))\ x$ reduces to x. We can conclude that $\mathcal{S}(\overline{\mu}(A)) \rhd^*_{\varepsilon\beta\eta} \lambda x : \{x : B|P\}.x$.
- If A is of the form $C \to B$ with $\overline{\mu}(B) : B \to \mu(B)$, then
$$\overline{\mu}(A) \equiv \lambda h : C \to B.\lambda x : C.\overline{\mu}(B)\ (h\ x) : C \to \mu(B)$$
Since $(\mathcal{S}(\overline{\mu}(B))\ (h\ x)) \rhd^*_{\varepsilon\beta\eta} (h\ x)$, we have $\mathcal{S}(\overline{\mu}(A)) \rhd^*_{\varepsilon\beta\eta} \lambda h : A \to B.h$.
- If A is of the form $B \times C$, then
$$\overline{\mu}(A) \equiv \lambda x : B \times C. < (\overline{\mu}(B)\ \pi_1(x)), (\overline{\mu}(C)\ \pi_2(x)) >_{\mu(B)\times\mu(C)}$$
Again, the induction hypotheses assure that $\overline{\mu}(A) \rhd^*_{\varepsilon\beta\eta} \lambda x : B \times C.x$. □

The opposite operation, going from from $\mu(A)$ to A, can only be performed when some conditions are verified (*type-checking conditions*, or 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 being 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[4].

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[5].

Lemma 5.5 (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 $\rhd_\eta$, *one has*
$$\Gamma \vdash \pi(A) : \mu(A) \to \mathsf{Prop} \textit{ and } \Gamma \vdash \overline{\pi}(A) : \Pi x : \mu(A).(\pi(A)\ x) \to A$$
Furthermore, $\overline{\pi}(A)$ $\rhd_{\varepsilon\beta\eta}$-normalizes to $\lambda x : \mu(A).\lambda p : (\pi(A)\ x).x$.

Proof. By straightforward induction. We only provide detail for 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 $\overline{\pi}(A) \equiv \lambda f : A \to \mu(B).\lambda p : \forall x : A.(\pi(B)\ (f\ x)).\lambda x : A.(\overline{\pi}(B)\ (f\ x)\ (p\ x))$. □

[4] It should be mentionned that adding η-reduction to such a type system yields non-trivial technical difficulties, which are mostly independent of the question of proof-irrelevance.

[5] 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 by proving the corresponding equality on the deductive level.

6. A more extensional theory

Especially during the 1970s and 1980s, 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.

6.1. 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 choose 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 [36]).

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 type corresponding to the unordered pair

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

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

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

and from the axiom of choice we deduce

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

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 $a' =_{\{a,b\}} b'$ and $a =_A b$ is provable in the theory. Therefore, if $(f\ a')$ and $(f\ b')$ are different, so are a and b. On the other hand, if $(f\ a') =_{\mathsf{bool}} (f\ b') =_{\mathsf{bool}} \mathsf{true}$ then $\pi_1(b') =_A a$ and so $b =_A a$. In the same way, $(f\ a') =_{\mathsf{bool}} (f\ b') =_{\mathsf{bool}} \mathsf{false}$ entails $b =_A a$.

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

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

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.

6.2. **Other classical non-computational axioms.** At present, we have not been able to deduce the excluded middle (EM) from the statement above[6]. 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

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

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.

6.3. **Quotients and normalized types.** Quotient sets are a typically extensional concept whose adaptation to type theory has always been problematic. Again, one has to choose between "effective" quotients and decidability of type-checking. Searching for a possible compromise, Courtieu [13] ended up with an interesting notion of *normalized type*[7]. The idea is remarkably simple: given a function $f : A \to B$, we can define $\{(f\ x)|x : A\}$ which is the subtype of B corresponding to the range of f. His rules are straightforwardly translated into our theory by simply taking

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

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 $A \to A$.

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

$$x\ R\ y \iff f\ x =_A f\ y$$

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

[6] 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.

[7] A similar notion has been developed for NuPRL [30].

7. Simple semantics

When justifying the correctness of a program extraction mechanism, one can use either semantics or syntax. In the first case, one builds a model and verifies it validates extraction [7]. In the latter case, at least in the framework of type theories, this mainly means building a realizability interpretation on top of the strong normalization property [32]. This second approach is difficult here, since our theory is *itself* built using the erasure of non-computational terms. Furthermore, for complex theories, it appears easier to prove strong normalization using an already defined model [2, 27, 12].

For this reason alone, it is worth treating the topic of semantics here. Furthermore, we believe it is a good point for a theory meant to be used in a proof-system to bear simple semantics, in order to justify easily the validity of additional axioms like the ones mentioned in the previous section or extensions like the useful reduction rule for $\mathsf{Eq\_rec}$ (par. 2.4) which is difficult to treat by purely syntactic means.

Set-theoretical interpretations are the most straightforward way to provide semantics for typed λ-calculi. It consists, given an interpretation $\mathcal{I}$ of the free variables, of interpreting a type T by a set $|T|_{\mathcal{I}}$, and terms $t : T$ by elements $|t|_{\mathcal{I}}$ of $|T|_{\mathcal{I}}$. Furthermore, λ-abstractions are interpreted by their set-theoretical counterparts: $|\lambda x : A.t|_{\mathcal{I}}$ is the function mapping $\alpha \in |A|_{\mathcal{I}}$ to $|t|_{\mathcal{I};x\leftarrow\alpha}$. While these interpretations are not interesting for studying the dynamics of proof-normalization, they have the virtue of simplicity.

Since Reynolds [34], it is well-known that impredicative or polymorphic types, as the inhabitants of Prop, bear only a trivial set-theoretical interpretation: if $P : \mathsf{Prop}$, then $|P|_{\mathcal{I}}$ is either the empty set or a singleton. In other words, all proofs of proposition P have the same interpretation. Since our theory precisely identifies all the elements of P at the computational level, the set-theoretical setting is, for its simplicity the most appealing for our goal.

Although the set-theoretical model construction is not as simple as it might seem [29], the setting is not new; We try to give a reasonably precise description here.

7.1. Notations.

Peter Aczel's way to encode set-theoretic functions provides a tempting framework for a model construction, and a previous version of this section relied on it. However, because of technical difficulties appearing when proving the subject reduction property for the semantic interpretation we finally favor the traditional set theoretic vision of functions, where the application $f(x)$ is only defined when x belongs to the domain of the function f.

If $\mathcal{I}$ is a mapping from variables to sets and α is a set, we write $\mathcal{I}; x \leftarrow \alpha$ for the function mapping x to α and identical to $\mathcal{I}$ elsewhere.

The interpretation of the hierarchy $\mathsf{Type}(i)$ goes beyond ZFC set theory and relies on the existence of inaccessible cardinals. This means, we postulate, for every natural number n the existence of a set $\mathcal{U}_n$ such that

- $\mathcal{U}_n \in \mathcal{U}_{n+1}$,
- $\mathcal{U}_n$ is closed by all set-theoretical operations.

As usual, we write $\emptyset$ for the empty set. We write $\mathbb{I}$ for the canonical singleton $\{\emptyset\}$. If A is a set and $(B_a)_{a\in A}$ a family of sets indexed over A, we use the set-theoretical dependent

$$
\begin{array}{rcl}
|\Gamma \vdash t|_{\mathcal{I}} & \equiv & \emptyset \text{ if } t \text{ is of sort } * \\
\text{In the other cases:} & & \\
|\Gamma \vdash x_\diamond|_{\mathcal{I}} & \equiv & \mathcal{I}(x_\diamond) \\
|\Gamma \vdash \lambda x : A.t|_{\mathcal{I}} & \equiv & \alpha \in |\Gamma \vdash A|_{\mathcal{I}} \mapsto |\Gamma(x:A) \vdash t|_{\mathcal{I};x\leftarrow\alpha} \\
|\Gamma \vdash (t\ u)|_{\mathcal{I}} & \equiv & |\Gamma \vdash t|_{\mathcal{I}}(|\Gamma \vdash u|_{\mathcal{I}}) \\
|\Gamma \vdash \Sigma x : A.B|_{\mathcal{I}} & \equiv & \Sigma_{\alpha\in|\Gamma\vdash A|_{\mathcal{I}}}|\Gamma(x:A) \vdash B|_{\mathcal{I};x\leftarrow\alpha} \\
|\Gamma \vdash \Pi x : A.B|_{\mathcal{I}} & \equiv & \displaystyle\bigcap_{\alpha\in|\Gamma\vdash A|_{\mathcal{I}}} |\Gamma(x:A) \vdash B|_{\mathcal{I};x\leftarrow\alpha} \text{ if } \Gamma \vdash \Pi x : A.B : \mathsf{Prop} \\
|\Gamma \vdash \Pi x : A.B|_{\mathcal{I}} & \equiv & \Pi_{\alpha\in|\Gamma\vdash A|_{\mathcal{I}}}|\Gamma(x:A) \vdash B|_{\mathcal{I};x\leftarrow\alpha} \text{ in the other cases} \\
|\Gamma \vdash < t, u >_{\Sigma x:A.B}|_{\mathcal{I}} & \equiv & (|\Gamma \vdash t|_{\mathcal{I}}, |\Gamma \vdash u|_{\mathcal{I}}) \\
|\Gamma \vdash \pi_i(t)|_{\mathcal{I}} & \equiv & \alpha_i \text{ if } |\Gamma \vdash t|_{\mathcal{I}} \text{ is a pair } (\alpha_1, \alpha_2) \\
|\Gamma \vdash \mathsf{Prop}|_{\mathcal{I}} & \equiv & \{\emptyset; \mathbb{I}\} \\
|\Gamma \vdash \mathsf{Type}(i)|_{\mathcal{I}} & \equiv & \mathcal{U}_i \\
|\Gamma \vdash (\mathsf{Eq\_rec}\ A\ P\ a\ b\ p\ e)|_{\mathcal{I}} & \equiv & |\Gamma \vdash p|_{\mathcal{I}}
\end{array}
$$

Figure 2: Definition of the semantics

products and sums

$$
\begin{array}{rcl}
\Pi_{a\in A}B_a & \equiv & \{f \in A \to \displaystyle\bigcup_{a\in A} B_a \mid \forall a \in A.f(a) \in B_a\} \\
\Sigma_{a\in A}B_a & \equiv & \{(a,b) \in A \times \displaystyle\bigcup_{a\in A} B_a \mid a \in A \wedge b \in B_a\}
\end{array}
$$

Finally we write $x \in A \mapsto t$ for the set-theoretical function construction and, of course, $f(x)$ for set-theoretical function application.

7.2. **Construction.** Over the ECC fragment of the type theory, the interpretation is constructed straightforwardly. The fact that non-computational terms are syntacticly tagged makes the definition easier. We define

Definition 7.1. For any mapping $\mathcal{I}$ from variables to $\bigcup_{i\in\mathbb{N}}\mathcal{U}_i$, we define a mapping associating a set $|\Gamma \vdash t|_{\mathcal{I}}$ to a term t and a context Γ. This function is defined by induction over the size of t by the following equations of figure 2; we can restrict ourselves to the case where $\Gamma \vdash t : T$ for some T.

The following extension of interpretations to contexts is the usual.

Definition 7.2. We define the condition $\mathcal{I} \in |\Gamma|$ by the following clauses

- $\mathcal{I} \in |[]|$,
- $\mathcal{I} \in |\Gamma| \wedge \mathcal{I}(x) \in |\Gamma \vdash A|_{\mathcal{I}} \Rightarrow \mathcal{I} \in |\Gamma(x:A)|$.

This definition should not be surprising. It is a partial definition, because of two clauses

- The case of the application, since $|\Gamma \vdash (t\ u)|_{\mathcal{I}}$ is only defined when $|\Gamma \vdash t|_{\mathcal{I}}$ is a (set-theoretic) function and its domains contains $|\Gamma \vdash u|_{\mathcal{I}}$.

- The cases of the projections, since $|\Gamma \vdash \pi_i(t)|_{\mathcal{I}}$ is only defined when $|\Gamma \vdash t|_{\mathcal{I}}$ is a (set-theoretic) pair.

Note also that the definition depends upon Γ only to discriminate between the case where $\Pi x : A.B$ is impredicative and is not. A more interesting technical point is the last clause: by anticipating the reduction of $\mathsf{Eq\_rec}$ we have a total definition which is obviously invariant by reduction.

Lemma 7.3 (Substitutivity). *Suppose* $\Gamma(x_{\mathsf{s}} : U)\Delta \vdash t : T$, $\Gamma \vdash u : U$ *and* $\mathsf{s} = s(u)$. *Suppose furthermore*

(1) $|\Gamma(x : U)\Delta|$ *is defined,*
(2) *if* $\mathcal{I} \in |\Gamma(x : U)\Delta| \Rightarrow |\Gamma(x : U)\Delta \vdash t|_{\mathcal{I}} \in |\Gamma(x : U)\Delta \vdash T|_{\mathcal{I}}$,
(3) $\mathcal{I} \in |\Gamma| \Rightarrow |\Gamma \vdash u|_{\mathcal{I}} \in |\Gamma \vdash U|_{\mathcal{I}}$.

Let $\mathcal{I} \in |\Gamma\Delta[x \setminus u]|$*; we have*

- $|\Gamma\Delta[x \setminus u] \vdash t[x \setminus u]|_{\mathcal{I}}$ *is defined and equal to* $|\Gamma(x_{\mathsf{s}} : U)\Delta \vdash t|_{\mathcal{I};x \leftarrow |\Gamma \vdash u|_{\mathcal{I}}}$
- $(\mathcal{I}; x \leftarrow |\Gamma \vdash u|_{\mathcal{I}}) \in \||\Gamma(x : U)\Delta|$.

Proof. By a simple induction over the structure of the derivation.

Note that in the case where t is of the form $\Pi y : A.B$, one uses the fact that typing is preserved by substitution (lemma 3.9) in order to ensure that the applied clause remains the same ($\Pi x : A.B$ being of type Prop or $\mathsf{Type}(i)$). □

Lemma 7.4 (Correctness for reduction). *Let* $\Gamma \vdash t : T$ *be derivable; have* $\mathcal{I} \in |\Gamma|$ *such that* $|\Gamma \vdash t|_{\mathcal{I}}$ *is defined. If* $t \rhd_{\varepsilon} t'$, *then* $|\Gamma \vdash t'|_{\mathcal{I}} = |\Gamma \vdash t|_{\mathcal{I}}$.

Proof. By induction over the typing derivation. As pointed out in [29], the restriction on the β-reduction that ensures that the tag does not change is essential here. □

Corollary 7.5. *Let* $\Gamma \vdash t : T$ *and* $\Gamma \vdash t' : T$ *be derivable; have* $\mathcal{I} \in |\Gamma|$ *such that* $|\Gamma \vdash t|_{\mathcal{I}}$ *and* $|\Gamma \vdash t'|_{\mathcal{I}}$ *are defined.* $t =_{\beta\varepsilon} t'$, *then we have* $|t|_{\mathcal{I}} = |t'|_{\mathcal{I}}$.

Soundness is then proved without much difficulty.

Theorem 7.6. *If* $\Gamma \vdash wf$ *is derivable, then* $|\Gamma|$ *is defined. If* $\Gamma \vdash t : T$ *is derivable, and* $\mathcal{I} \in |\Gamma|$ *then* $|\Gamma \vdash t|_{\mathcal{I}} \in |\Gamma \vdash T|_{\mathcal{I}}$ *(and both objects are defined).*

Proof. By induction over the derivation. When checking the correctness of the interpretation of $\mathsf{Eq\_rec}$, one simply has to remark that propositional Leibniz equality is indeed interpreted by set-theoretical equality; that is, if $|\Gamma \vdash a|_{\mathcal{I}}$ and $|\Gamma \vdash b|_{\mathcal{I}}$ are both elements of $|\Gamma \vdash A|_{\mathcal{I}}$, then

- $|\Gamma \vdash a =_A b|_{\mathcal{I}} = \mathbb{I}$ if $|\Gamma \vdash a|_{\mathcal{I}} = |\Gamma \vdash b|_{\mathcal{I}}$,
- $|\Gamma \vdash a =_A b|_{\mathcal{I}} = \emptyset$ if $\Gamma \vdash |a|_{\mathcal{I}} \neq |\Gamma \vdash b|_{\mathcal{I}}$. □

It is easy to check that the axioms AC, EM and EXT of the previous section are valid in this model.

8. Conclusion and further work

We have tried to show that a relaxed conversion rule can make type theories more practical, without necessarily giving up normalization or decidable type checking. In particular, we have shown that this approach brings closer 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 Dependent ML or ATS [10, 37]. But this will only be assessed by practice; the first step is thus to implement such a theory.

9. Acknowledgements

Christine Paulin deserves special thanks, since she me gave the idea of a relaxed conversion rule long ago as well as the hint for the computation rule of paragraph 2.4. Bruno Barras was very helpful by pointing out some errors and Makoto Tatsuta by helping me to check that defining simultaneously reduction and conversion did not break Church-Rosser. A question of Russell O'Connor suggested the generalization 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] P. Aczel. "On relating type theories and set theories", in *Types for Proofs and Programs*, edited by Altenkirch, Naraschewski and Reus, Proceedings of Types '98, LNCS 1657 (1999).

[2] 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, Springer-Verlag, 1994.

[3] T. Altenkirch. "Extensional Equality in Intensional Type Theory". LICS'99, IEEE, 1999.

[4] 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.

[5] B. Barras and B. Bernardo. "The Implicit Calculus of Constructions as a Programming Language with Dependent Types". In R. Amadio, editor, Proceedings of FOSSACS'08, volume 4962 of LNCS, pages 365-379. Springer-Verlag, 2008.

[6] 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.

[7] S. Berardi. "An Application of PER Models to Program Extraction". *Mathematical Structures in Computer Science*, vol. 3(3), pages 309-331, 1993.

[8] F. Blanqui. "Definitions by rewriting in the Calculus of Constructions". *Mathematical Structures in Computer Science*, vol. 15(1), 2003.

[9] J. Caldwell. "Moving Proofs-as-Programs into Practice", In Proceedings of the 12th IEEE International Conference on Automated Software Engineering, IEEE, 1997.

[10] Chiyan Chen and Hongwei Xi, "Combining Programming with Theorem Proving". ICFP'05, IEEE, 2005.

[11] The Coq Development Team. *The Coq Proof-Assistant User's Manual*, INRIA. On `http://coq.inria.fr/`.

[12] T. Coquand and A. Spiwack. "A Proof of Strong Normalisation using Domain Theory". In Proceedings of LICS'06, IEEE, 2006.

[13] P. Courtieu. "Normalized Types". *Proceedings of CSL 2001*, L. Fribourg Ed., LNCS 2142, Springer, 2001.

[14] E. Gimenez. "A Tutorial on Recursive Types in Coq". INRIA Technical Report. 1999.

[15] G. Gonthier. A computer-checked proof of the Four Colour Theorem. Manuscript, `http://research.microsoft.com/~gonthier/4colproof.pdf`, 2005.

[16] B. Grégoire and X. Leroy. "A compiled implementation of strong reduction", proceedings of ICFP, IEEE, 2002.

[17] B. Grégoire. *Compilation des termes de preuves: un (nouveau) mariage entre Coq et Ocaml*. Thèse de doctorat, Université Paris 7, 2003.

[18] B. Grégoire, L. Théry and B. Werner. "A computational approach to Pocklington certificates in type theory". In proceedings of FLOPS 2006, M. Hagiya and P. Wadler (Eds), Volume 3945 of LNCS, Springer-Verlag, 2006.

[19] M. Hofmann and T. Streicher. "A groupoid model refutes uniqueness of identity proofs". Proceedings of LICS'94, Paris. IEEE, 1994.

[20] J. W. Klop. *Combinatory Reduction Systems.* Ph.D. Thesis, Utrecht University, 1980.

[21] P. Letouzey. *Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq.* Thèse de doctorat. Université Paris-sud, 2004.

[22] P. Letouzey. A New Extraction for Coq. In Geuvers and Wiedijk (Eds.), Proceedings of TYPES 2002, LNCS 2646, 2003.

[23] Z. Luo. "ECC: An Extended Calculus of Constructions." Proceedings of LICS'89, IEEE, 1989.

[24] P. Martin-Löf. *Intuitionistic Type Theory.* Studies in Proof Theory, Bibliopolis, 1984.

[25] C. McBride. "Elimination with a Motive". P. Callaghan, Z. Luo, J. McKinna and R. Pollack (Eds.), Proceedings of TYPES'00, LNCS 2277, Springer Verlag, 2002.

[26] 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.

[27] 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.

[28] A. Miquel. "The Implicit Calculus of Constructions". In proceedings of TLCA'01, LNCS 2044, Springer-Verlag, 2001.

[29] 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.

[30] A. Nogin and A. Kopilov. "Formalizing Type Operations Using the "Image" Type Constructor". To appear in WoLLIC 2006, ENTCS.

[31] S. Owre and N. Shankar. "The Formal Semantics of PVS". SRI Technical Report CSL-97-2R. Revised March 1999.

[32] C. Paulin-Mohring. *Extraction de Programmes dans le Calcul des Constructions.* Thèse de doctorat, Université Paris 7, 1989.

[33] F. Pfenning. "Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory". In Proceedings of LICS, IEEE, 2001.

[34] J. Reynolds. "Polymorphism is not Set-Theoretic". In *Semantics of Data Types*, G. Kahn, D. MacQueen and G. Plotkin (Eds.), LNCS 173, Springer-Verlag, 1984.

[35] M. Sozeau. "Subset Coercions in Coq". In TYPES'06, T. Altenkirch and C. McBride (Eds.), LNCS 4502, Springer-Verlag, 2007.

[36] B. Werner. "Sets in Types, Types in Set". In, M. Abadi and T. Itoh (Eds), Theoretical Aspects of Computer Science, TACS'97, LNCS 1281, Springer-Verlag, 1997.

[37] Hongwei Xi, *Dependent Types in Practical Programming*, Ph.D, CMU, 1998.