

# Uniform Logical Relations

Edwin Westbrook  
Department of Computer Science  
Rice University  
Houston, TX 77005  
Email: emw4@rice.edu

## Abstract

*Strong Normalization (SN) is an important property for intensional constructive type theories such as the Calculus of Inductive Constructions (CiC), the basis for the Coq theorem prover. Not only does SN imply consistency, but it also ensures that type-checking is decidable, and further, it provides a straightforward model, the term model, for a theory. Unfortunately, although SN has been proved for fragments of CiC, it is not known how to prove SN for CiC in its entirety, including eliminations for large inductive types as well as higher predicative universes. In this work, we show how to prove SN for full CiC. The key insight given here is that terms must be interpreted in a uniform manner, meaning that the form of the interpretation of a term must not depend on whether the term is a type. We introduce a new technique called Uniform Logical Relations, with uniformity as a guiding principle, and we show that this technique can then be used to prove SN for CiC. An important property of our technique is that it does not rely on Confluence, and thus it could potentially be used for extensions of CiC with added computation rules, such as Extensionality, for which Confluence relies on SN.*

## 1. Introduction

Intensional constructive type theory (ICTT), as first introduced by Martin-Löf [12], is compelling because it gives a combined programming language and mathematical theory. This allows users to write programs and proofs about those programs in the same language (see e.g. [15]). Further, theories based on ICTT are defined entirely syntactically. This makes them foundational, in the sense that they can be defined with little or no appeal to models or set theory.

An important consideration for such theories is Strong Normalization (SN), which states that no well-

typed term of the theory has an infinite reduction path. Stated differently, SN says that the programs of a theory have no infinite loops. SN is useful for a number of reasons. First, it ensures the consistency of a theory, as it implies that every proof has a cut-free version. Second, it ensures that type-checking is decidable. Finally, SN also gives a simple model of the theory, namely, the normal forms of that theory.

Unfortunately, it is not known how to prove SN for more powerful extensions of ICTT such as the Calculus of Inductive Constructions (CiC), the basis of the Coq theorem prover [17]. Although there have been numerous proofs of SN for fragments of CiC [7], [10], [2], [8], [11], [4], none of these approaches has been generalized to the entirety of CiC. This has led to research into other approaches to proving the consistency of CiC by providing set-theoretic models [3], [19], [14], [13], [18], [16]. The key difficulty is in combining two features: an impredicative universe  $\text{Prop}$ , and a cumulative type hierarchy of universes  $\text{Type}_i$  for  $i \geq 0$ . To handle impredicativity requires the Logical Relations technique introduced by Girard for System F [9], in which each type is interpreted as a logically defined set of terms, also called a logical relation. With a cumulative type hierarchy, however, we do not know whether a type is a sort or not unless we normalize it, and thus, under the standard approach to Logical Relations, it is unknown how to even interpret a type without normalizing it. This seems to require SN to prove SN! The problem is especially difficult in the presence of inductive types (as pointed out by e.g. [7]), since the form of a type can depend on whether a natural number term evaluates to 0 or to 1. Although Luo showed how to break this cycle in the case of ECC using a difficult and complex quasi-normalization theorem [11], ECC contains only products, not inductive types, and it is not clear that this approach generalizes to theories that do contain inductive types.

In this paper, we show how to solve this problem by using an interpretation that is *uniform*, meaning that the interpretation does not need to know whether a type is a sort.<sup>1</sup> We call this approach Uniform Logical Relations. The key technical development that ensures uniformity is to interpret types as sets of triples of context, terms, and interpretations, i.e., sets of the form  $\{\langle \Gamma, M, I \rangle \mid \phi(\Gamma, M, I)\}$  where  $M$  is a term that is well-typed with respect to  $\Gamma$ , and  $\phi$  is a logical formula which states that  $I$  is a valid interpretation of  $M$ . This is as opposed to the usual approach of interpreting types as sets of terms  $\{M \mid \phi(M)\}$ . Under the usual approach, the interpretation of the function type  $\Pi x : A. B$  depends on the form of  $A$ : this interpretation must quantify over terms of type  $A$  and reducibility candidates when  $A$  is a sort; over terms of type  $A$  and functions on reducibility candidates when  $A$  is a kind-level  $\Pi$ -type; and over just terms of type  $A$  when  $A$  is a type. This information about  $A$  requires the normal form of  $A$  in general. Under Uniform Logical Relations, in contrast, the interpretation of  $\Pi x : A. B$  always quantifies over triples  $\langle \Gamma, N, X \rangle \in \llbracket A \rrbracket$ . This uniformity means we do not require the normal form of  $A$ .

From a different viewpoint, uniformity means the interpretation captures “deep” information about every term in the theory, instead of just the types. For example, the interpretation of a constructor application such as  $c M_1 M_2$  is the tuple  $\langle c, I_1, I_2 \rangle$ , where each  $I_i$  is the interpretation of  $M_i$ . Intuitively, this deep information is needed because even “simple” objects, like natural numbers, can influence the interpretation of a type built using elimination forms, such as the type

$$\text{case } x \text{ of } z \rightarrow \text{nat} \mid s y \rightarrow \text{nat} \rightarrow \text{nat}$$

A central concern with deep interpretations inside ZF set theory is the Axiom of Regularity, which states that the set membership relation  $\in$  is well-founded. On the positive side, Regularity immediately justifies primitive recursion on inductive types, because recursive calls in CiC are guaranteed to follow the transitive closure of the  $\in$  relation. More subtly, Regularity explains why Uniform Logical Relations does not work for potentially non-terminating terms like  $\lambda x. x x$ , where a term is applied to itself, since no function can be in its own domain or range without creating a cycle in the  $\in$  relation.

1. This approach came out of the author’s dissertation, which gave an interpretation of a nominal extension of CiC into CiC [20]; the interpretation for CiC in set theory given here is actually much simpler because we do not have to worry things like intensional equality and type universes in set theory.

On the negative side, however, Regularity makes it difficult to interpret proofs because of impredicativity, as proofs can quantify over domains that contain themselves. To break the “vicious cycle” of impredicativity, all proofs are given a “dummy” interpretation, written  $\bullet$ , which intuitively contains no information. (The empty set  $\emptyset$  could be used here, but we use a separate notation  $\bullet$  for clarity.) This works because it is not possible in CiC to perform eliminations of proofs except when either the input type intuitively contains no information, like the empty type `False`, or the output is another proof, whose interpretation is also a dummy and does not need any information from the input. This technique also means that Uniform Logical Relations construction satisfies the Proof Irrelevance property (see, e.g., [14]), since all proofs have the same interpretation.

The remainder of this document is organized as follows. Section 2 reviews the syntax and semantics of the Calculus of Inductive Constructions (CiC). Section 3 then proves Strong Normalization for CiC using Uniform Logical Relations. Finally, Section 4 relates Uniform Logical Relations to previous approaches, and Section 5 concludes.

## 2. The Calculus of Inductive Constructions

In this section, we give a brief overview of the Calculus of Inductive Constructions. The version we give here has some small modifications relative to the theory as presented in, for example, the Coq manual [17]. These modifications are for increased conciseness, but do not change the theory in a material way.

**A Note on Syntax:** Here and in the below we use particular letters to stand for elements of particular syntactic classes; e.g.,  $M$  is always used for a term. These can appear with subscripts or primes, as in  $M'_2$  or  $M_{\text{fun}}$ . We also use an arrow over a syntactic construct to indicate a sequence of zero or more of those constructs. For example,  $\vec{x}$  denotes a sequence of variables. We use subscripts  $i$  and  $j$  to denote the  $i$ th and  $j$ th elements of this sequence, e.g.,  $x_i$ , and we use vertical bars  $|\vec{x}|$  to denote the length of this sequence. If the letter under the arrow has a prime, then so do the elements of the sequence, so for instance the elements of  $\vec{M}'$  are referred to as  $M'_i$ . We also use the following compound notations:  $M \vec{M}$  denotes the multi-arity application  $M M_1 \dots M_{|\vec{M}|}$ ;  $[\vec{M}/\Gamma]$  denotes the multi-arity substitution  $[M_1/x_1, \dots, M_{|\vec{M}|}/x_{|\vec{M}|}]$ , where  $\vec{x}$  are the variables on the left in  $\Gamma$ ; and  $\vec{c} \setminus \vec{\Gamma} \rightarrow \vec{M}$  denotes the sequence of pattern cases  $c_1 \setminus \Gamma_1 \rightarrow M_1 \mid \dots \mid c_n \setminus \Gamma_n \rightarrow M_n$

$$\begin{aligned}
\Sigma & ::= \cdot \mid \Sigma, c : M \mid \Sigma, a : M \\
\Gamma & ::= \cdot \mid \Gamma, x : M \mid \Gamma, u : {}^{n;\vec{x}}M \\
M & ::= \text{Prop} \mid \text{Type}_i \mid \Pi x : M . M \mid a(\vec{M}; \vec{N}) \\
& \quad \mid x \mid M N \mid \lambda x : M . M \mid c(\vec{M}; \vec{N}) \\
& \quad \mid u \mid \text{fun}^{ip} u (\Gamma) (c \setminus \Gamma \rightarrow M \mid \dots \mid c \setminus \Gamma \rightarrow M)
\end{aligned}$$

Figure 1. Syntax of CiC

We assume mutually disjoint sets of constructors  $c$ , type constructors  $a$ , variables  $x$ , and recursive variables  $u$ , where the latter are used for recursive calls in primitive recursive pattern-matching functions and the sets of variables and recursive variables are infinite. We also use  $y$  and  $z$  for variables.

Figure 1 gives the syntax of CiC. The signatures  $\Sigma$  associate types with constructors  $c$  and type constructors  $a$ , while the contexts  $\Gamma$  map variables  $x$  and recursive variables  $u$  to types. We also use  $\Delta$  for contexts below. Recursive variables  $u$  are also associated with a number  $n$  and a sequence of variables  $\vec{x}$ , which specify that all occurrences of  $u$  should have the form  $u \vec{M} x_i$  for some  $i$  where  $|\vec{M}| = n$ , to ensure termination of primitive recursion. In the below we write  $|\Gamma|$  for the length of  $\Gamma$ , i.e., the number of commas.

The remainder of Figure 1 defines the terms  $M$ . We also use  $N$ ,  $Q$ , and  $R$  for terms, as well as  $A$  and  $B$  for terms that are meant to denote types. The terms include the impredicative universe  $\text{Prop}$  and the predicative universes  $\text{Type}_i$  for all  $i \geq 0$ . These are the *sorts*, written  $s$ . We also sometimes use  $ip$  for meta-variable that varies over  $\{p, i\}$ . In addition,  $\text{Type}_{-1}$  is also used to denote  $\text{Prop}$ . Terms also include the function types  $\Pi x : A . B$  and the inductive types  $a(\vec{M}; \vec{N})$ , where the latter distinguishes between the *parameters*  $\vec{M}$  and the *arguments*  $\vec{N}$ . Next are the variables  $x$ , the applications  $M N$ , the  $\lambda$ -abstractions  $\lambda x : A . M$ , and the constructor applications  $c(\vec{M}; \vec{N})$ , which again distinguish between parameters and arguments.

The final line lists the recursive variables  $u$  and the pattern-matching functions, where the latter have the form  $\text{fun}^{ip} u (\Gamma_{\text{arg}}) (c \setminus \vec{\Gamma} \rightarrow \vec{M})$ . Pattern-matching functions combine the  $\text{Fix}$  and  $\text{case}$  terms of Coq. Each compound form  $c_i \setminus \Gamma_i \rightarrow M_i$  is called a *pattern case* with *pattern context*  $\Gamma_i$  and *body*  $M_i$ . Pattern-matching functions of this form take in  $|\Gamma_{\text{arg}}| + 1$  arguments and pattern-match on the last argument. The last argument is called the *scrutinee* and the earlier arguments are called the *parameters*. If the scrutinee has the form  $c_i(\vec{N}; \vec{Q})$  then the pattern-matching function returns  $M_i$ , substituting the parameters for the variables of  $\Gamma_{\text{arg}}$  and substituting the arguments  $\vec{Q}$  for the pattern variables listed in  $\Gamma_i$ . Pattern-

$$\begin{aligned}
(\lambda x : A . M) N & \longrightarrow [N/x]M \\
(M_{\text{fun}} = \text{fun } u (\Gamma_{\text{arg}}) (c \setminus \vec{\Gamma} \rightarrow \vec{M})) \vec{N} c_i(\vec{Q}; \vec{R}) & \longrightarrow [\vec{N}/\Gamma_{\text{arg}}, \vec{R}/\Gamma_i, M_{\text{fun}}/u]M_i
\end{aligned}$$

Figure 2. Operational Semantics of CiC

matching functions are also annotated with either  $p$  or  $i$ , indicating whether they eliminate a predicative or impredicative inductive type.

The operational semantics of CiC is given as a higher-order rewrite system  $\longrightarrow$  in Figure 2. This means that  $M_1 \longrightarrow M_2$  iff  $M_2$  can be obtained by replacing a subterm of  $M_1$  that matches the left-hand side of a rule listed in Figure 2 by the corresponding right-hand side of the rule. The first rule performs the usual  $\beta$ -reduction, while the second performs the pattern-matching reduction described in the previous paragraph. In the below, we write  $\longrightarrow^*$  and  $\longleftarrow^*$  respectively for the reflexive-transitive closure and the reflexive-symmetric-transitive closure of  $\longrightarrow$ .

In the below we also make use of call-by-name (CBN) reduction. Intuitively,  $M$  CBN reduces to  $N$ , written  $M \longrightarrow_n N$ , iff  $N$  can be reached from  $M$  by only reducing a redex that is either to the left of zero or more applications in  $M$  or is in the scrutinee of a pattern-matching function at the top level of  $M$ . When  $M \longrightarrow_n M'$  holds by contracting a redex  $R$  in  $M$ , we call the immediate subterms of  $R$  the *constituents* of the reduction, and we further say that  $M$  call-by-name reduces to  $M'$  *with normalizing constituents*, written  $M \longrightarrow_{n\text{-nc}} M'$ , if all these constituents are strongly normalizing. If there is no  $M'$  such that  $M \longrightarrow_n M'$  then  $M$  is called a *weak head normal form* (WHNF). These notions will be useful in Section 3.2 below.

We do not give the well-formedness rules for signatures  $\Sigma$ ; these may be found in the Coq manual [17]. At a high level, however, these require that  $\Sigma$  contain sequences that define a type constructor  $a$  and its constructors  $c_i$  of the following form:

$$a : (\text{III}\Gamma_p . \text{III}\Gamma_a . s), c_1 : (\text{III}\Gamma_p . \text{III}\Gamma_{c_1} . a(\Gamma_p; \vec{M}_c)), \dots$$

The context  $\Gamma_p$  lists the *parameters* of the inductive type  $a$ , which must be the same for all of the constructors. The contexts  $\Gamma_a$  and  $\Gamma_{c_i}$  list the *arguments* of  $a$  and  $c_i$ , respectively. All of the types must be well-typed for the prefix of  $\Sigma$  before  $a$ , except that the  $\Gamma_{c_i}$  may contain a *strictly positively*, meaning that types of the form  $a(\vec{N}; \vec{Q})$  can only occur as the return type of zero or more  $\Pi$ -abstractions directly to the right of a colon in  $\Gamma_{c_i}$ . The signature  $\Sigma$  and its well-formedness are left implicit in the below. We also implicitly assume that any contexts  $\Gamma$  are well-formed, meaning that all listed types are well-typed.

The typing rules are given in Figure 3. The first rule is the conversion rule, which states that equal types have the same elements. Note that this only allows a single step of conversion; although the rule may be used multiple times, each use requires the destination type  $A'$  to be well-typed, ensuring that only conversions that use well-typed types can be used. The second rule is the subtyping rule, which states that if  $M$  has type  $A$  for  $A$  a *subtype* of  $A'$ , written  $A <: A'$ , then  $M$  has type  $A'$ . Subtyping is the reflexive-transitive closure of the following three cases:  $\text{Prop} <: \text{Type}_0$ ;  $\text{Type}_i <: \text{Type}_{i+1}$ ; and if  $B_1 <: B_2$  then  $\Pi x : A. B_1 <: \Pi x : A. B_2$ . Note that subtyping is guaranteed to preserve well-typedness of types. The next two rules state that  $\text{Prop} : \text{Type}_0$  and  $\text{Type}_i : \text{Type}_{i+1}$ . The next two rules type  $\Pi$ -types  $\Pi x : A. B$  by requiring the types of both  $A$  and  $B$  to be sorts. Also, if  $B$  is impredicative, meaning it has type  $\text{Prop}$ , then the type of  $\Pi x : A. B$  is  $\text{Prop}$ , and otherwise the type of  $\Pi x : A. B$  is the maximum universe of that of both  $A$  and  $B$ .

For inductive types  $a(\vec{M}; \vec{N})$ , the parameters  $\vec{M}$  and the arguments  $\vec{N}$  must have the types given by the contexts  $\Gamma_p$  and  $\Gamma_a$  in  $\Sigma$ . This is stated with the judgment  $\Gamma \vdash (\vec{M}; \vec{N}) : \Gamma_p; \Gamma_a$  which states that  $|\vec{M}| = |\Gamma_p|$ ,  $|\vec{N}| = |\Gamma_a|$ , and that, for each term  $Q$  in  $\vec{M}, \vec{N}$  with type  $A$  occurring at the same position in  $\Gamma_p, \Gamma_a$ , we have  $\Gamma \vdash Q : [\vec{M}/\Gamma_p, \vec{N}/\Gamma_a]A$ .

For variables, we look the type up in the context. For applications  $M N$ , we require  $M$  to have function type  $\Pi x : A. B$  and  $N$  to have type  $A$ , returning type  $[N/x]B$ . For  $\lambda$ -abstractions, we require the domain type  $A$  to be well-formed, the body  $M$  to have type  $B$  under the context extended with variable  $x$ , and we return function type  $\Pi x : A. B$ . For constructor applications  $c(\vec{M}; \vec{N})$ , we again require the parameters  $\vec{M}$  and arguments  $\vec{N}$  to have the appropriate types given in  $\Gamma_p$  and  $\Gamma_c$ , yielding the return type of  $c$  with the parameters and arguments substituted in. For recursive calls to  $u$ , we require the term to be of the form  $u \vec{M} x_i$  where  $x_i$  is one of the variables to which  $u$  may be applied, where  $|\vec{M}|$  is the number of arguments before  $x_i$  to which  $u$  must be applied, and where all of  $\vec{M}$  and  $x_i$  have the types required by the type of  $u$ .

To type a pattern-matching function, we first require that its type  $A_{\text{fun}}$  (which includes the parameter types  $\Gamma_{\text{arg}}$ ) has type  $s$  for some  $s$ . We then require that  $a$  can be eliminated at sort  $s$ , written  $\text{elim-ok}(a \mapsto s)$ ; this states that, if  $a$  is impredicative then  $s = \text{Prop}$ , except for the special case that  $a$  has at most one constructor, all of whose arguments are proofs (i.e.,

have a type which has type  $\text{Prop}$ ), when  $s$  is allowed to be predicative. Next, we require that the patterns exactly match the constructors of  $a$  in  $\Sigma$ , written  $\text{ctors}_\Sigma(a)$ . Next, we require that the pattern contexts  $\Gamma_i$  are all well-formed. Finally, we require that each  $c_i$  applied to the given parameters and the variables listed in  $\Gamma_i$  has same type as the input type, and that each body  $M_i$  has as its type the result of substituting this application of  $c_i$  into the return type  $B$ .

*Lemma 1:* If  $\Gamma \vdash M : A$  then  $\exists s. \Gamma \vdash A : s$ .

### 3. Strong Normalization for CiC

In this section, we give a Uniform Logical Relations (ULR) interpretation  $\llbracket \cdot \rrbracket$  of the terms of CiC and use this to prove SN for the well-typed terms of CiC. Types  $A$  are interpreted as sets of triples  $\langle \Gamma, M, I \rangle$  with  $\Gamma \vdash M : A$  such that, intuitively,  $M$  satisfies the logical condition for  $A$  and  $I$  is a valid interpretation of  $M$  with respect to  $A$ . SN is proved via the Reducibility Theorem, which states that  $\langle \Gamma, M, \llbracket M \rrbracket \rangle$  is in  $\llbracket A \rrbracket$  whenever  $M$  has type  $A$  (relative to  $\Gamma$ ) and  $A$  itself satisfies  $\langle \Gamma, A, \llbracket A \rrbracket \rangle$  is in  $\llbracket s \rrbracket$  for some sort  $s$ . SN follows since all interpretations in  $\llbracket s \rrbracket$  must be *reducibility candidates*, which among other things require that they contain only strongly normalizing terms.

#### 3.1. Reducibility Candidates

To prove SN for the bodies of lambdas, which can quantify over potentially empty domain types, we need to be able to support free variables which may have no valid interpretation. To do this, we define the concept of *stuck terms*, which intuitively are the free variables closed under application and pattern-matching elimination forms. We then ensure (in the definition of Reducibility Candidates, below) that all stuck terms have the dummy interpretation  $\bullet$ .

*Definition 1 (Stuck Terms):* The *stuck terms* are defined inductively as follows. If  $M$  is a stuck term, then so are:

- $x$  and  $u$  for variable  $x$  or recursive variable  $u$ ;
- $M N$  for any  $N$ ; and
- $M_{\text{fun}} \vec{N} M$  for pattern-matching function  $M_{\text{fun}}$  that takes  $|\vec{N}|$  parameters.

Since types can also be stuck terms, we define the notation  $\llbracket I \rrbracket_{\Gamma \vdash A}$ , called the *set denoted by  $I$*  (relative to  $\Gamma$  and  $A$ ), as follows: if  $I = \bullet$ , then  $\llbracket I \rrbracket_{\Gamma \vdash A}$  is the set of SN terms of type  $A$ , i.e., the set

$$\{ \langle \Gamma', M, \bullet \rangle \mid \Gamma' \geq \Gamma \wedge \Gamma' \vdash M : A \wedge \text{SN}(M) \}$$

$$\begin{array}{c}
\frac{\Gamma \vdash M : A \quad A \longleftrightarrow^* A' \quad \Gamma \vdash A' : s}{\Gamma \vdash M : A'} \quad \frac{\Gamma \vdash M : A \quad A <: A'}{\Gamma \vdash M : A'} \quad \frac{}{\Gamma \vdash \text{Prop} : \text{Type}_0} \quad \frac{}{\Gamma \vdash \text{Type}_i : \text{Type}_{i+1}} \\
\frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, x : A \vdash B : \text{Type}_i}{\Gamma \vdash \Pi x : A. B : \text{Type}_i} \quad \frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash B : \text{Prop}}{\Gamma \vdash \Pi x : A. B : \text{Prop}} \quad \frac{\Gamma \vdash A : s \quad \Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A. M : \Pi x : A. B} \quad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \\
\frac{\Gamma \vdash M : \Pi x : A. B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : [N/x]B} \quad \frac{a : \Pi \Gamma_p. \Pi \Gamma_a. s \in \Sigma \quad \Gamma \vdash (\vec{M}; \vec{N}) : \Gamma_p; \Gamma_a}{\Gamma \vdash a(\vec{M}; \vec{N}) : s} \\
\frac{c : \Pi \Gamma_p. \Pi \Gamma_c. a(\Gamma_p; \vec{Q}) \in \Sigma \quad \Gamma \vdash (\vec{M}; \vec{N}) : \Gamma_p; \Gamma_c}{\Gamma \vdash c(\vec{M}; \vec{N}) : a(\vec{M}; [\vec{M}/\Gamma_p, \vec{N}/\Gamma_c]\vec{Q})} \quad \frac{(u : |\vec{M}|; \vec{x} \ \Pi \Gamma_u. B) \in \Gamma \quad \Gamma \vdash (\vec{M}, (x_i \vec{N})) : \Gamma_u}{\Gamma \vdash u \vec{M} (x_i \vec{N}) : [(\vec{M}, (x_i \vec{N})/\Gamma_u)B]} \\
\frac{\Gamma \vdash A_{\text{fun}} : s \quad a : \Pi \Gamma_p. \Pi \Gamma_a. s_a^{ip} \in \Sigma \quad \text{elim-ok}(a \mapsto s) \quad \vec{c} = \text{ctors}_\Sigma(a) \quad \forall i. (\Gamma, \Gamma_{\text{arg}} \vdash \Gamma_i)}{\Gamma \vdash \text{fun}^{ip} u (\Gamma_{\text{arg}}) (\vec{c} \setminus \vec{\Gamma} \rightarrow \vec{M}) : (A_{\text{fun}} = \Pi \Gamma_{\text{arg}}. \Pi x : a(\vec{N}; \vec{Q}). B)} \\
\forall i. (\Gamma, \Gamma_{\text{arg}}, \Gamma_i \vdash c_i(\vec{N}; \Gamma_i) : a(\vec{N}; \vec{Q})) \quad \forall i. (\Gamma, \Gamma_{\text{arg}}, \Gamma_i, u : |\Gamma_{\text{arg}}|; \Gamma_i \vdash M_i : [c_i(\vec{N}; \Gamma_i)/x]B)
\end{array}$$

Figure 3. Typing Rules of CiC

Otherwise  $[I]_{\Gamma \vdash A} = I$ . Intuitively, this turns the dummy interpretation into a dummy type. We often use  $[I]$  when  $\Gamma$  and  $A$  can be inferred from context.

We can now define the reducibility candidates, which constitute the well-formed sets denoted by interpretations of types. Note that these are very similar to the saturated sets of, e.g., Coquand and Gallier [4] and Geuvers [7]:

*Definition 2 (Reducibility Candidate):* We say that  $I$  is a *reducibility candidate* for  $\Gamma$  and  $A$ , written  $\text{CR}_{\Gamma \vdash A}(I)$  (or  $\text{CR}(I)$  when  $\Gamma$  and  $A$  can be inferred from context), iff  $S = [I]_{\Gamma \vdash A}$  is a set of triples such that, whenever  $\langle \Gamma', M, I' \rangle \in S$ , we have the following named properties:

- (G):  $\Gamma' \geq \Gamma$ ;
- (E): If  $\Gamma'' \geq \Gamma'$  then  $\langle \Gamma'', M, I' \rangle \in S$ ;
- (T):  $\Gamma' \vdash M : A$ ;
- (CR 1):  $\text{SN}(M)$ ;
- (CR 2): If  $M \longrightarrow M'$  then  $\langle \Gamma', M', I' \rangle \in S$ ;
- (CR 3): If  $M' \longrightarrow_{\text{n-nc}} M$  then  $\langle \Gamma', M', I' \rangle \in S$ ; and
- (CR 4): If  $M'$  is a stuck term such that  $\text{SN}(M')$  and if  $\Gamma'' \vdash M' : A$  for  $\Gamma'' \geq \Gamma'$  then  $\langle \Gamma'', M', \bullet \rangle \in S$ .

As an immediate application of this definition, it is straightforward to see that  $\bullet$  defines a reducibility candidate:

*Lemma 2:*  $\text{CR}_{\Gamma \vdash A}(\bullet)$  for all  $A$  with  $\Gamma \vdash A : s$ .

### 3.2. A ULR Interpretation for CiC

We now define the interpretation  $\llbracket M \rrbracket^{\Delta; \Xi}$  in Figure 4. This is defined relative to a context  $\Delta$  and an *interpretation valuation*  $\Xi$  of the following syntax:

$$\Xi ::= \cdot \mid \Xi, x \mapsto (N, I) \mid \Xi, u \mapsto (N, I) \mid \Xi, a \mapsto I$$

Intuitively, interpretation valuations map variables and recursive variables to a pair of a term and an interpretation of the term. Considering just the terms, interpretation valuations define a substitution, written  $\Xi^\sigma$ , while  $\Xi^I(\cdot)$  looks up just the interpretation associated with some  $x, u$ , or  $a$ . We also use the following abbreviated notations:  $\Gamma \mapsto (\vec{N}, \vec{I})$  represents the interpretation valuation  $x_1 \mapsto (N_1, I_1), \dots, x_n \mapsto (N_n, I_n)$ ; the notation  $\llbracket A \rrbracket^{\Delta; \Xi}$  represents  $\llbracket [A] \rrbracket^{\Delta; \Xi^\sigma}$ ; the notations  $\llbracket \vec{M} \rrbracket^{\Delta; \Xi}$  and  $\llbracket \Gamma' \rrbracket^{\Delta; \Xi}$  denote sequences of interpretations of each of the  $M_i$  or the types in  $\Gamma'$ , respectively; and finally  $\langle \vec{\Delta}, \vec{N}, \vec{I} \rangle \in \llbracket \Gamma \rrbracket^{\Delta; \Xi}$  denotes the fact that that  $\langle \Delta_i, N_i, I_i \rangle$  is in  $\llbracket A_i \rrbracket^{\Delta; \Xi}$  for each type  $A_i$  in  $\Gamma$ .

Let  $\text{Term}(i)$  denote the set of all terms that do not contain  $\text{Type}_j$  for  $j \geq i$ , denoting  $\text{Prop}$  by  $\text{Type}_{-1}$ . The interpretation  $\llbracket M \rrbracket^{\Delta; \Xi}$  is defined by primary induction the least  $i$  such that  $M \in \text{Term}(i)$ , by secondary induction on the number of inductive types  $a$  not in  $\text{Dom}(\Delta)$ , and by tertiary induction on the structure of  $M$ . The primary induction hypothesis used to define the interpretation of sorts  $\text{Type}_i$ , which works because normal forms of type  $A$  such that  $A$  has type  $\text{Type}_i$  do not contain  $\text{Type}_i$ . The secondary induction hypothesis is used to define the interpretations of inductive types, which works because the type of any  $a$  can only contain constructors and type constructors which come before  $a$  in  $\Sigma$ . Otherwise  $\llbracket \cdot \rrbracket^{\Delta; \Xi}$  is defined using the tertiary induction hypothesis, structural induction on  $M$ .

In order to handle the dummy interpretation  $\bullet$ , we define the following “smart constructors”:

$$\begin{aligned}
c^\bullet(\vec{I}) &\triangleq \text{if } c : A : \text{Prop} \text{ then } \bullet \text{ else } \langle c, \vec{I} \rangle \\
F @^\bullet I &\triangleq \text{if } F = \bullet \text{ then } \bullet \text{ else } F(I) \\
\lambda^\bullet(x \in S). F(x) &\triangleq \text{if } \forall x. F(x) = \bullet \text{ then } \bullet \text{ else } F
\end{aligned}$$

$$\begin{aligned}
\llbracket \text{Type}_i \rrbracket^{\Delta; \Xi} &= \{ \langle \Delta, A, I \subseteq (\text{Ctxt} \times \text{Term} \times \mathbb{I}_i) \rangle \mid \text{SN}(A) \wedge \Delta \vdash A : \text{Type}_{i-1} \wedge \mathbf{CR}_{\Delta \vdash A}(I) \} \\
\llbracket \Pi x : A . B \rrbracket^{\Delta; \Xi} &= \{ \langle \Delta \geq \Delta, M, (\lambda^\bullet p . F @^\bullet p) \rangle \\
&\quad \mid \Delta' \vdash M : \Pi x : A . B \wedge \\
&\quad \forall (\langle \Delta'' \geq \Delta', N, I \rangle \in \llbracket A \rrbracket^{\Delta'; \Xi}). \langle \Delta'', M N, F @^\bullet \langle \Delta'', N, I \rangle \rangle \in \llbracket B \rrbracket^{\Delta''; \Xi, x \mapsto (N, I)} \} \\
\llbracket a(\vec{M}; \vec{N}) \rrbracket^{\Delta; \Xi} &= (\mathbf{if } a \mapsto F \in \Xi \mathbf{ then } F \mathbf{ else } \llbracket a \rrbracket_{\text{ind}})(\Delta, \Xi^\sigma(\vec{M}), \Xi^\sigma(\vec{N}), \llbracket \vec{M} \rrbracket^{\Delta; \Xi}, \llbracket \vec{N} \rrbracket^{\Delta; \Xi}) \\
\llbracket x \rrbracket^{\Delta; \Xi} &= \Xi^I(x) \\
\llbracket u \rrbracket^{\Delta; \Xi} &= \Xi^I(u) \\
\llbracket M N \rrbracket^{\Delta; \Xi} &= \llbracket M \rrbracket^{\Delta; \Xi} @^\bullet \langle \Delta, N, \llbracket N \rrbracket^{\Delta; \Xi} \rangle \\
\llbracket \lambda x : A . M \rrbracket^{\Delta; \Xi} &= \lambda^\bullet (\langle \Delta', N, I \rangle \in \llbracket A \rrbracket^{\Delta'; \Xi}). \llbracket M \rrbracket^{\Delta'; \Xi, x \mapsto (N, I)} \\
\llbracket c(\vec{M}; \vec{N}) \rrbracket^{\Delta; \Xi} &= c^\bullet(\llbracket \vec{N} \rrbracket^{\Delta; \Xi}) \\
\llbracket M_{\text{fun}} = \mathbf{fun}^p u (\Gamma_{\text{arg}}) (\vec{c} \setminus \vec{I} \rightarrow \vec{M}) \rrbracket^{\Delta; \Xi} &= \mathbf{primrec}_F^\bullet (\langle \vec{\Delta}, \vec{N}, \vec{I} \rangle \in \{\Gamma_{\text{arg}}\}^{\Delta; \Xi}) \\
&\quad (\dots \mid \langle \Delta'', c_i(\vec{Q}; \vec{N}'), \langle c_i, \vec{I}' \rangle \rangle \rightarrow \llbracket M_i \rrbracket^{\Delta''; \Xi, \Gamma \mapsto (\vec{N}, \vec{I}), \Gamma_i \mapsto (\vec{N}', \vec{I}'), u \mapsto (M_{\text{fun}}, F)} \mid \dots \mid X \rightarrow \bullet) \\
\llbracket M_{\text{fun}} = \mathbf{fun}^i u (\Gamma_{\text{arg}}) (\vec{c} \setminus \vec{I} \rightarrow \vec{M}) \rrbracket^{\Delta; \Xi} &= \lambda^\bullet (\langle \vec{\Delta}, \vec{N}, \vec{I} \rangle \in \{\Gamma_{\text{arg}}\}^{\Delta; \Xi}). \lambda^\bullet (I \in \llbracket a(\vec{Q}; \vec{R}) \rrbracket^{\Delta_n; \Xi}) \\
&\quad \prod_{\langle \Delta'', \vec{N}', \vec{I}' \rangle \in \{\Gamma_i, A_{\text{fun}}\}^{\Delta; \Xi}} \llbracket M_i \rrbracket^{\Delta''; \Xi, \Gamma_{\text{arg}} \mapsto (\vec{N}, \vec{I}), (\Gamma_i, u) \mapsto (\vec{N}', \vec{I}')} \\
\llbracket a \rrbracket_{\text{ind}} = \mu F(\Delta, \vec{M}, \vec{N}, \vec{X}, \vec{Y}). & \\
\{ \langle \Delta', M, I \rangle \mid \Delta' \geq \Delta \wedge \text{SN}(M) \wedge \Delta' \vdash M : a(\vec{M}; \vec{N}) \wedge & \\
(\forall \vec{Q} \vec{R}. M \longrightarrow^* c(\vec{Q}; \vec{R}) \mathbf{ implies } c : \text{III} \Gamma_p . \text{II}(x_1 : A_1, \dots, x_n : A_n) . a(\Gamma_p; \vec{M}_c) \in \Sigma \wedge & \\
\exists \Delta'' . \exists \vec{I}' . \forall i. \langle \Delta'', R_i, I_i \rangle \in \llbracket A_i \rrbracket^{a \mapsto F, \Gamma_p \mapsto \vec{X}, \vec{x} \mapsto \vec{I}'} \wedge I = c^\bullet(\vec{I}') \wedge \llbracket \vec{M}_c \rrbracket^{\Gamma_p \mapsto \vec{X}, \vec{x} \mapsto \vec{I}'} = \vec{Y} \} \wedge & \\
(\neg(\exists c. \exists \vec{Q} \exists \vec{R}. M \longrightarrow^* c(\vec{Q}; \vec{R})) \mathbf{ implies } I = \bullet) \} &
\end{aligned}$$

Figure 4. A Uniform Logical Relations Interpretation for CiC

The first constructs the interpretation of  $c(\vec{M}; \vec{N})$  from the interpretations  $\vec{I}$  of  $\vec{N}$ . In order to satisfy Proof Irrelevance, the result is  $\bullet$  when  $c$  is impredicative, and otherwise is  $\langle c, \vec{I} \rangle$ . The second applies a function  $F$  to an argument  $I$ , returning  $\bullet$  if  $F$  is  $\bullet$ . The third constructs a set-theoretic function using  $\lambda$  notation, returning  $\bullet$  if the function is uniformly  $\bullet$ .

As discussed above, a type  $A$  is interpreted as a set of triples  $\langle \Delta, M, I \rangle$  where  $\Gamma \vdash M : A$ . Thus the interpretation of  $\text{Type}_i$ , which is a type of types, contains all possible triples  $\langle \Delta, A, I \rangle$  where  $A$  is a type and  $I$  itself is a set of triples that forms a reducibility candidate for  $A$ . In order to include all possible reducibility candidates  $I$  for  $A$ , we quantify  $I$  over the set of all triples  $\langle \Delta', M, I' \rangle$  where  $I'$  is in  $\mathbb{I}_i$ , defined as follows:

$$\begin{aligned}
\mathbb{I}_{-1} &= \{*\} \\
\mathbb{I}_{i \geq 0} &= \mu X . \{ \llbracket M \rrbracket^{\Delta; \Xi} \mid M \in \text{Term}(i) \wedge \\
&\quad \text{Ran}(\Xi) \subseteq \text{Term}(i) \times X \}
\end{aligned}$$

$\mathbb{I}_{-1}$  contains only  $*$  because of Proof Irrelevance, while all other  $\mathbb{I}_i$  contain the set of all interpretations of any  $M$  that does not itself contain  $\text{Type}_i$ .

The interpretation of function types  $\Pi x : A . B$  is the set of all triples  $\langle \Delta', M, F \rangle$  such that, for all  $\langle \Delta'', N, I \rangle$  in  $\llbracket A \rrbracket^{\Delta'; \Xi}$  we have that applying  $M$  to  $N$  and applying  $F$  to  $\langle \Delta'', N, I \rangle$  yields a triple (with first element  $\Delta''$ ) in the interpretation of  $B$ . Note that

we require  $F$  to have the form  $\lambda^\bullet p . F @^\bullet p$ , meaning that  $F = \bullet$  whenever all of the outputs of  $F$  are  $\bullet$ .

Inductive type constructors  $a$  are interpreted by applying either  $\Xi^I(a)$ , if it is defined, or the result of the helper definition  $\llbracket a \rrbracket_{\text{ind}}$ , to the parameters and type arguments, along with their interpretations, of  $a$ . The helper definition constructs a function  $F$  using the least fixed-point operator  $\mu$ , returning a set of triples  $\langle \Delta', M, I \rangle$  where  $M$  is strongly normalizing of the correct type and that satisfies the following: if  $M$  rewrites to a constructor term  $c(\vec{Q}; \vec{R})$ , then  $I$  must be  $c^\bullet(\vec{I})$ , where the  $\vec{I}$  are valid interpretations for the  $\vec{R}$ , i.e., the  $\vec{R}$  occur in triples with the  $\vec{I}$  in the interpretations of their types. Note that, if  $a$  is impredicative,  $c^\bullet(\vec{I}) = \bullet$ . Further, we require  $c$  to be a constructor of  $a$  and the interpretations of the arguments  $\vec{M}_c$  in the return type of  $c$  to equal the arguments  $\vec{Y}$ . Otherwise, if  $M$  rewrites to no such term,  $I = \bullet$ . We can see that this use of the least fixed-point operator  $\mu$  is guaranteed to be well-defined because  $a$  can only be used strictly positively in the argument types  $A_i$  of its constructors.

The interpretations of variables and recursive variables look those variables up in  $\Delta$ . The interpretation of an application applies (using  $I_1 @^\bullet I_2$ ) the interpretation of the function to that of the argument. The interpretation of a  $\lambda$ -abstraction forms a function (using the  $\lambda^\bullet(I \in S) . F(x)$  notation) that takes in an

interpretation for its argument and returns the interpretation of the body. The interpretation of a constructor term  $c(\vec{M}; \vec{N})$  applies the constructor (using the  $c^\bullet(\vec{I})$  notation) to the interpretations of the arguments  $\vec{N}$ .

The interpretation of a predicative pattern-matching function performs primitive recursion on the scrutinee. This is justified by the Axiom of Regularity, since any recursive calls are through interpretations of the recursive variable  $u$ , which is guaranteed to only be applied to variables that match subterms of the scrutinee. If the scrutinee is a triple of a context  $\Delta''$ , an application of  $c_i$ , and the interpretation  $\langle c_i, \vec{I} \rangle$  of an application of  $c_i$ , then the interpretation of  $M_i$  is returned, possibly with primitive recursive calls to the entire function; otherwise,  $\bullet$  is returned in the catch-all case, which holds when the scrutinee is a stuck term.

The final case is that for impredicative pattern-matching functions. Because of Proof Irrelevance, the scrutinee for such a function will always have interpretation  $\bullet$ , and thus we cannot perform primitive recursion on it. Instead, the interpretation of an impredicative pattern-matching function throws away its input, and instead considers *all possible* interpretations of the pattern cases  $M_i$ , for all possible interpretations that can be assigned to the pattern variables  $\Gamma_i$  and the recursive function variable  $u$ . If all possible interpretations of the  $M_i$  are all equal to some result  $I$ , then the interpretation of the impredicative pattern-matching function returns  $I$ . Otherwise the interpretation is undefined. This is written with the notation  $\prod S$ , where  $\prod \{I\} = I$ ,  $\prod \{\bullet\} = \bullet$ , and  $\prod S$  is undefined otherwise.

The reason this approach works is due to Proof Irrelevance. If the return type is a proposition, then all pattern cases are proofs and so must all have the same interpretation  $\bullet$ . Otherwise we either have that the scrutinee type must be an empty type, so there are no pattern cases (and  $\bullet$  is returned), or the scrutinee type must be a singleton type whose sole constructor takes only proof arguments, and there is only one possible interpretation, again  $\bullet$ , for those arguments.

The following structural properties can be proved by straightforward induction on the definition of  $\llbracket \cdot \rrbracket^{\Delta; \Xi}$ :

*Lemma 3 (Interpretation Weakening):*

$\llbracket M \rrbracket^{\Delta; \Xi_1, \Xi_2, \Xi_3} = \llbracket M \rrbracket^{\Delta; \Xi_1, \Xi_3}$  for any  $M$  and  $\Xi_i$  when both of these are defined.

*Lemma 4 (Interpretation Substitution):*

$\llbracket M \rrbracket^{\Delta; \Xi, x \mapsto (N, \llbracket N \rrbracket^{\Delta; \Xi'})}, \Xi' = \llbracket [N/x]M \rrbracket^{\Delta; \Xi, \Xi'}$  or both of these are undefined.

### 3.3. Proving Reducibility

In order to prove Reducibility, we first need to define the notion of well-formed interpretatio valuations. We

say that  $\Xi$  is *well-formed* with respect to input context  $\Gamma$  and output context  $\Delta$ , written  $\Gamma \vdash \Xi : \Delta$ , iff:

- 1)  $\Gamma \vdash \Xi^\sigma : \Delta$ , meaning that  $\text{Dom}(\Xi^\sigma) = \text{Dom}(\Gamma)$  and that  $\forall x \in \text{Dom}(\Xi^\sigma). \Delta \vdash \Xi^\sigma(x) : \Xi^\sigma(\Gamma(x))$ ;
- 2)  $\text{Dom}(\Xi)$  restricted to variables and recursive variables equals  $\text{Dom}(\Gamma)$ ;
- 3) For all  $x \in \text{Dom}(\Gamma)$  we have  $\langle \Delta, \Xi^\sigma(x), \Xi^I(x) \rangle \in \llbracket \Gamma(x) \rrbracket^{\Delta; \Xi}$ ;
- 4) For all  $u :^{n; \vec{x}} \prod \Gamma_u . B \in \Gamma$  and for all  $x_i$  we have  $\langle \Delta, \Xi^\sigma(u) \vec{N} \Xi^\sigma(x_i), \Xi^I(u) @^\bullet \vec{I} @^\bullet \Xi^I(x_i) \rangle \in \llbracket B \rrbracket^{\Delta; \Xi}$  whenever  $\forall j. \langle \Delta, N_j, I_j \rangle \in \llbracket A_j \rrbracket^{\Delta; \Xi}$  where  $A_1, \dots, A_m$  are the types in  $\Gamma_u$ ; and
- 5) For all  $a \in \text{Dom}(\Xi)$  we have that  $\Xi^I(a)$  is a function from  $(\Delta, \vec{M}, \vec{N}, \vec{I}, \vec{I}')$  for  $\langle \Delta, \vec{M}, \vec{N}, \vec{I}, \vec{I}' \rangle \in \llbracket \Gamma_p, \Gamma_a \rrbracket^{\Delta; \Xi}$  to reducibility candidates for  $a(\vec{M}; \vec{N})$ .

At a high level, the idea is that, whenever we form  $\llbracket M \rrbracket^{\Delta; \Xi}$ ,  $\Gamma$  lists the function arguments in scope, and  $\Xi$  lists terms and interpretations that have been passed in for these arguments. In order to state that an interpretation passed for  $x$  is valid, it must be associated with the term  $\Xi^\sigma(x)$  passed for  $x$ . The output context  $\Delta$  means that the range of  $\Xi^\sigma$  can contain open terms. For recursive variables  $u$ , the requirement is looser, only requiring that the interpretation of any fully applied occurrence of  $u$  is valid given the value of  $\Xi^I(u)$ . As an example, for any  $\Gamma$  we have  $\Gamma \vdash \Gamma \mapsto (\Gamma, \bullet) : \Gamma$  where  $\Gamma \mapsto (\Gamma, \bullet)$  assigns term  $x$  and interpretation  $\bullet$  to each variable  $x \in \text{Dom}(\Gamma)$ . Using  $\bullet$  here is valid by **CR 4** because the variables of  $\Gamma$  and fully applied recursive variables of  $\Gamma$  are stuck terms.

*Lemma 5 (Proof Irrelevance):* Let  $\Gamma \vdash \Xi : \Delta$  and  $\langle \Delta', \Xi^\sigma B, \llbracket B \rrbracket^{\Delta; \Xi} \rangle \in \llbracket \text{Prop} \rrbracket$ . If  $\langle \Delta'', M, I \rangle \in \llbracket B \rrbracket^{\Delta; \Xi}$  then  $I = \bullet$ .

Conversion is mostly straightforward by Interpretation Substitution. The interesting case is reduction of an impredicative pattern-matching function applied to a constructor application in the scrutinee. The result follows in this case from the requirement that the interpretation of the source term  $M$  is well-defined, as this ensures that the interpretation of the result, which has the form  $\sigma M_i$  for some  $\sigma$  and some pattern-case  $M_i$  of the impredicative pattern-matching function, is independent of the particular  $\sigma$  used, and so is equal to the interpretation returned by the pattern-matching function. Subtyping then follows directly.

*Lemma 6 (Conversion):* If  $M \longrightarrow N$  then we have that  $\llbracket M \rrbracket^{\Delta; \Xi} = \llbracket N \rrbracket^{\Delta; \Xi}$  whenever both of these are defined.

*Lemma 7 (Subtyping):* If  $A_1 <: A_2$  then we have that  $\llbracket A_1 \rrbracket^{\Delta; \Xi} \subseteq \llbracket A_2 \rrbracket^{\Delta; \Xi}$ .

We now prove the main result of this section, Reducibility, which states that if  $\Gamma \vdash M : A$  then  $\llbracket M \rrbracket^{\Delta; \Xi}$  is well-defined and is a valid interpretation of  $M$ . This assumes that  $\llbracket A \rrbracket^{\Delta; \Xi}$  itself is valid, meaning that it is a valid reducibility candidate. As discussed above, we must also assume  $\Gamma \vdash \Xi : \Delta$ . Since, at a high level,  $\Xi$  acts as an ‘‘interpretation substitution’’ for the variables in  $M$ , we have that  $\llbracket M \rrbracket^{\Delta; \Xi}$  is actually a valid interpretation for  $\Xi^\sigma(M)$ , with respect to the output context  $\Delta$ . Thus Reducibility yields  $\langle \Delta, \Xi^\sigma(M), \llbracket M \rrbracket^{\Delta; \Xi} \rangle \in \llbracket A \rrbracket^{\Delta; \Xi}$ . This is not a restriction, though, as we can always take  $\Delta = \Gamma$  and  $\Xi = \Gamma \mapsto (\Gamma, \bullet)$  as discussed above, which can be used to show SN for terms  $M$  with free variables.

*Theorem 1 (Reducibility):* If  $\Gamma \vdash M : A$  and  $\Gamma \vdash A : s$ , and if  $\langle \Delta, \Xi^\sigma(A), \llbracket A \rrbracket^{\Delta; \Xi} \rangle \in \llbracket s \rrbracket$  for  $\Gamma \vdash \Xi : \Delta$ , then  $\llbracket M \rrbracket^{\Delta; \Xi}$  is defined and  $\langle \Delta, \Xi^\sigma(M), \llbracket M \rrbracket^{\Delta; \Xi} \rangle \in \llbracket A \rrbracket^{\Delta; \Xi}$ .

*Proof:* Proof is by the same induction scheme as used to define  $\llbracket \cdot \rrbracket^{\Delta; \Xi}$ . For reasons of space, we consider only a few of the more interesting cases, at a high level; the remaining cases are either straightforward, such as subtyping (by Lemma 7), variables  $x$ , and recursive variables  $u$  (the latter two by well-formedness of  $\Xi$ ), and/or are similar to previous approaches, such as the original proof by Girard [9].

If  $M = a(\vec{M}; \vec{N})$ , we know  $A = s$  for some  $s$ , and we must prove  $\langle \Gamma', \sigma(M), \llbracket M \rrbracket^{\Delta; \Xi} \rangle$  is in  $\llbracket s \rrbracket^{\Delta; \Xi}$ , meaning that  $\text{SN}(M)$ ,  $\Gamma' \vdash \sigma(M) : s$ , and  $\text{CR}_{\Gamma \vdash A}(I)$ . The first is immediate by the tertiary IH and **CR 1**, the second follows by Substitution, and the third is straightforward for most of the conditions for reducibility candidates. For example, **CR 2** follows because, if  $N \longrightarrow N'$  and  $N' \longrightarrow^* c(\vec{Q}; \vec{R})$  then  $N \longrightarrow^* c(\vec{Q}; \vec{R})$ . **CR 4** follows because, if  $N$  is stuck, then  $N \longrightarrow^* c(\vec{Q}; \vec{R})$  is not possible. The most difficult condition to show is **CR 3**, which requires showing that, if  $N' \longrightarrow_{\text{n-nc}} N$  and  $N' \longrightarrow^* c(\vec{Q}; \vec{R})$  then  $N \longrightarrow^* c(\vec{Q}; \vec{R})$ . Note that this is a much weaker property than Confluence. It follows because any reduction to a constructor application must eventually perform a CBN reduction step, and any non-CBN reduction step can always be commuted (to zero or more steps) after a CBN step.

The other case we consider here is if  $M$  is a pattern-matching function with pattern cases  $M_i$  over type  $a(\vec{N}; \vec{Q})$  with return type  $B$ . Since the assumption  $\Gamma \vdash A_{\text{fun}} : s$  is part of the typing rule for  $M$ , we immediately have, by the tertiary IH, that the interpretations of  $B$  and of all the argument types are reducibility candidates. If  $a$  is impredicative, we first must show that  $\llbracket M \rrbracket^{\Delta; \Xi}$  is well-defined, meaning all  $M_i$  have the same interpretations for all valid

extensions  $\Xi, \Gamma_i \mapsto (\vec{N}, \vec{I})$  of  $\Xi$ . This follows by Proof Irrelevance, because either there is at most one pattern case  $M_i$  and at most one possible sequence of  $\vec{I}$ , namely  $I_j = \bullet$  for all  $j$ , or all possible interpretations of the  $M_i$  equal  $\bullet$ . We then must show that  $\langle \Delta', \Xi^\sigma(M) \vec{R} R_0, \llbracket M \rrbracket^{\Delta; \Xi} @ \bullet \vec{I} @ \bullet I_0 \rangle$  is in  $\llbracket B \rrbracket^{\Delta; \Xi}$  whenever  $\langle \Delta', R_j, I_j \rangle$  is in the interpretation of the appropriate argument type. This is shown by **CR 3**, using induction on the number of CBN steps it takes to either reduce  $R_0$  to a constructor application or a stuck term. This induction is possible since  $\text{SN}(R_0)$  follows by **CR 1** for  $a(\vec{N}; \vec{Q})$ .  $\square$

*Corollary 1:* (Strong Normalization) If  $\Gamma \vdash M : A$  then  $\text{SN}(M)$ .

## 4. Related Work

There have been numerous proofs of SN for various forms of both the Calculus of Inductive Constructions (CiC) and the Calculus of Constructions without inductive types (CC). As discussed above, none has been extended to full CiC. Many of these proofs are based on the *Candidates de Reducibilit e* of Girard [9], or on a modified version called *saturated sets*. Under these approaches, each type is interpreted as a set of terms that satisfies some closure properties (such as properties **CR 1** through **CR 4** above), which include strong normalization of the elements. The proof then proves the Reducibility Theorem, which states that each term is in the interpretation of its type, and SN follows.

Using this approach, Luo [11] proves SN for the Extended Calculus of Constructions (ECC), a version of CC with an impredicative universe (Prop), a hierarchy of infinitely many predicative universes ( $\text{Type}_i$ ), and product types. This seems to be the only SN proof for a calculus with infinitely many universes, probably due to the non-uniformity problem discussed in the Introduction. Luo’s proof defines an interpretation for each type construct, and then proves Reducibility. The interpretation only defined for types that are in weak head normal form, however. Thus Luo requires a complex and involved Quasi-Normalization theorem, which states that all terms can be reduced to a weak head normal form.

Coquand and Gallier [4] adapt the saturated sets approach to work on typed sets of terms, yielding a proof of SN for CC with one universe. They call their interpretation a Kripke structure, as each type is interpreted with respect to a particular world. The interpretation of a type contains pairs  $\langle \Delta', M \rangle$  of interpretation contexts  $\Delta'$  that extend  $\Delta$  and terms  $M$  such that  $M$  is well-typed with respect to (the context

associated with)  $\Delta'$ . This was the inspiration for including contexts  $\Gamma$  in our triples  $\langle \Gamma, M, I \rangle$ . Again, because of non-uniformity, however, it is unclear how to extend this approach to multiple universes. It is also interesting to consider how sets of typed terms would make our approach more difficult.

Geuvers [7] extends the saturated sets approach by defining two interpretations, one for terms and one for types. The interpretation for terms maps a term to a substitution instance of that term, and Reducibility then requires only that the instance be in the interpretation of a related substitution instance of the type of the term. Geuvers then shows that this approach can be used to support CC with strong products and inductive kinds, but comments that it will probably not work with small inductive types like the natural numbers.

Altenkirch [2] adapts the notion of saturated sets to be a semantic notion, using the saturated sets as the basis for a denotational idea called a *CC-structure* and thus building a model of CC. Soundness of this model then implies SN of CC. Altenkirch then shows that this argument can be extended to handle inductive types. In a similar manner, Goguen [10] proves SN by providing a typed operational semantics of terminating computations of a given type. He then shows that this semantics is sound for the syntactic type theory, i.e., that all well-typed terms have a corresponding terminating computation. This result is proved for CiC with two universes, one predicative and one impredicative.

Finally, Geuvers and Nederhof [8] prove SN for CC without universes by mapping terms in this language into System  $F_\omega$ . The proof for SN of System  $F_\omega$  uses saturated sets but is much simpler, because types are trivially strongly normalizing. It is not clear, however, that this approach could be adapted easily (without proving SN already) to extensions of CC.

## 5. Conclusion

In this work, we have shown how to adapt the Logical Relations proof technique to prove strong normalization (SN) for the full Calculus of Inductive Constructions (CiC) with a cumulative predicative type hierarchy. The guiding principle that has allowed us to handle full CiC is *uniformity*, meaning that our interpretation does not depend on whether a term is a type. One benefit of this is that our interpretation does not depend on confluence. The key technical development that enables uniformity is to interpret types not as sets of terms but as sets of triples  $\langle \Gamma, M, I \rangle$  of context  $\Gamma$ , terms  $M$  that are well-typed under  $\Gamma$ , and valid interpretations  $I$  of  $M$ . Closure conditions

on the sets associated with types, for example that the sets be saturated sets, are handled simply by including only such sets as valid interpretations of types in the interpretations  $\llbracket \text{Prop} \rrbracket$  and  $\llbracket \text{Type}_i \rrbracket$  of universes. Surprisingly, the proof is actually very similar to the original proof given by Girard, with the proof of the closure conditions on interpretations of types being folded into the proof of Reducibility, leading to a simple and elegant proof. In fact, the proof given by Girard can be seen as a special case of Uniform Logical Relations.

As a final consideration, the interpretation we give can easily be extended to support a transfinite hierarchy of universes. That is, we could support universes  $\text{Type}_\alpha$  for any ordinal  $\alpha$ . Of course, to define such a theory syntactically requires a notation of well-founded ordinals, such as the well-known notation based on  $\phi$ -functions for ordinals less than  $\Gamma_0$  (see e.g. [6]). Thus not only is Zermelo-Frankel set theory (ZF) proof-theoretically stronger than CiC — since we have proved consistency of CiC in ZF — but, if we view CiC as an ordinal operation that assigns to each  $\alpha$  the proof-theoretic ordinal of CiC with up to  $\alpha$  universes, then the ordinal of ZF is also larger than this function applied to  $\alpha$  (as well as infinitely many fixed points of this function). See e.g. Coquand et al. [5] for more on proof-theoretic ordinals of type theory.

It has been shown by Aczel [1], however, that *constructive* ZF (CZF), without the Axiom of the Excluded Middle, can be encoded in CiC, and so the proof-theoretic strength of CZF is no more than that of CiC. Note that our interpretation critically relies on Excluded Middle to define the interpretations of inductive types. If we add excluded middle for each sort  $s$  ( $EM_s$ ) to CiC, then the work of Werner [18] shows that full ZF can be encoded, where excluded middle for sort  $s$  has the following type:<sup>2</sup>

$$\Pi A : s. A +_s (A \rightarrow \text{False})$$

Here  $\text{False}$  is the empty inductive type and  $+_i$  denotes the inductive type for disjunction in sort  $s$ . In fact, Werner shows that ZF with  $i$  inaccessible cardinals ( $ZF_i$ ) can be encoded in CiC with  $EM_{\text{Type}_{i+2}}$ , where an *inaccessible cardinal* in ZF is essentially a set that is so big that it includes a whole model of ZF (and thus  $ZF_{i+1}$  is guaranteed to be proof-theoretically stronger than  $ZF_i$ ). Note that, although it is possible to interpret  $EM_{\text{Prop}}$  as  $\bullet$  in our construction (because of Proof Irrelevance), it is not possible to interpret  $EM_{\text{Type}_i}$

2. Werner actually uses  $EM_{\text{Prop}}$  along with the Type-Theoretical Description Axioms at each sort  $s$  ( $TTDA_s$ ), but the author has verified in the Coq proof assistant that  $EM_s$  implies  $TTDA_s$ .

since, for some types  $A$ , neither  $A$  nor its negation is inhabited. This is good, because, by the above and by Gödel's Second Incompleteness Theorem, defining an interpretation that did allow this in ZF would only be possible if ZF is inconsistent. In this light, we can see that (predicative) excluded middle is in fact a very powerful axiom.

## Acknowledgments

The author would like to thank Andreas Abel for spotting an important bug, Chris Casinghino and Peter Hancock for helpful discussions, and the past anonymous reviewers who caught a number of bugs and omissions in previous versions of this paper.

## References

- [1] Peter Aczel. The type theoretic interpretation of constructive set theory. In *Logic Colloquium '77*, 1977.
- [2] Thorsten Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, University of Edinburgh, 1993.
- [3] Bruno Barras. Sets in coq, coq in sets. *Journal of Formalized Reasoning*, 3(1):29–48, 2010.
- [4] Thierry Coquand and Jean Gallier. A proof of strong normalization for the theory of constructions using a kripke-like interpretation. Technical Report MS-CIS-90-44, University of Pennsylvania, 1990.
- [5] Thierry Coquand, Peter Hancock, and Anton Setzer. Ordinals in type theory, 1997. invited talk at Computer Science Logic (CSL '97).
- [6] Jean Gallier. What's so special about Kruskal's theorem and the ordinal  $\Gamma_0$ ? A survey of some results in proof theory. *Annals of Pure and Applied Logic*, 53(3):199–260, 1991.
- [7] Herman Geuvers. A short and flexible proof of strong normalization for the calculus of constructions. In *International Workshop on Types for Proofs and Programs*, pages 14–38, 1995.
- [8] Herman Geuvers and Mark-Jan Nederhof. Modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, 1991.
- [9] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.
- [10] Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994.
- [11] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.
- [12] Per Martin-Löf. An intuitionistic theory of types. In G Sambin and J Smith, editors, *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, 1998.
- [13] Alexandre Miquel. A model for impredicative type systems, universes, intersection types and subtyping. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2000.
- [14] Alexandre Miquel and Benjamin Werner. The not so simple proof-irrelevant model of cc. In *The 2002 international conference on Types for proofs and programs (TYPES)*, pages 240–258, 2002.
- [15] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.
- [16] M. Stefanova and H. Geuvers. A simple model construction for the calculus of constructions. In *The 1996 International Conference on Types for Proofs and Programs (TYPES)*, pages 5–8, 1996.
- [17] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version V8.3*, 2010. <http://coq.inria.fr/doc/>.
- [18] Benjamin Werner. Sets in types, types in sets. In *Third International Symposium on Theoretical Aspects of Computer Software (TACS)*, pages 530–346, 1997.
- [19] Benjamin Werner. On the strength of proof-irrelevant type theories. *Logical Methods in Computer Science*, 4, 2008.
- [20] Edwin Westbrook. *Higher-Order Encodings with Constructors*. PhD thesis, Washington University in Saint Louis, 2008.