Towards a Proof-Irrelevant Calculus of Inductive Constructions

Philipp Haselwarter

Towards a Proof-Irrelevant Calculus of Inductive Constructions

Philipp Haselwarter
under the supervision of Matthieu Sozeau, PPS and πr^2

2nd September 2014

Summary

The general context

Through the Curry-Howard correspondence, dependent type theories are appealing to both the mathematical and the programming community. To the first, they provide an expressive logical framework, in which mathematics can be developed. To the second, they offer a functional programming language that allows to state precise invariants programs have to respect and to build certified proofs thereof.

Several dependent type systems have been investigated and implemented, with some early ones geared more towards the mathematical community [Con+86; Pol94; Coq12], called proof-assistants, and later putting a stronger accent on their viability as a programming environment [McB99; Nor07; Soz08]. The Calculus of Inductive Constructions (pCIC) is one such theory that attempts to stay faithful to the correspondence and bridge the two worlds of programming and proving. It is implemented in the Coq system [Coq12] and Matita [Asp+11].

The research problem

The utilisation of Coq as a programming language as advocated by Sozeau relies on extending pCIC with a principle known as proof-irrelevance, which means that any two proofs of a logical proposition are identified by the system. This principle does not hold in the current theory and implementation of Coq, but as proofs appear as parts of dependently typed programs, they get in the way during their verification. Furthermore, it corresponds to

the mathematical intuition that the existence of a proof of a theorem is more important than its exact wording. In fact, under this aspect, there have been demands for proof-irrelevance in the community of interactive theorem provers since the beginning of their development, for instance in AUTOMATH [Ned94, Sec. D.3]. We studied how to extend PCIC in order to incorporate this principle.

Although there was a demand by the users of (intensional) dependent type theories, the first occasions where proof-irrelevance has been studied were through semantic models [Hof97; MW03]. A presentation of a syntax of a theory accommodating for proof-irrelevance has been given by Pfenning [Pfe01] for a simply typed calculus, and by Werner for a fragment of PCIC [Wer06] and recently for a restricted form of irrelevance in Martin-Löf Type Theory with an implementation in AGDA [AS12].

Our approach has a more semantic inspiration, building on the insights from the treatment of propositions by Awodey and Bauer [AB04] and in homotopy type theory [Pro13].

My contribution

My solution relies on the distinction of the universe of propositions or specifications from that of computations. The former is represented in COQ as the sort Prop, the latter as Set. Currently, this distinction is mainly used for the extraction of programs, during which proofs get erased, but the only difference inside the system is that only Prop allows for impredicative definitions.

We make a sharper separation between the two, which allows us to treat propositions as definitionally equal *inside* the system. To start with, we give a more modular presentation of PCIC, following that of [HS13]. We then refine this calculus by adding specific rules for the inhabitants of the propositional universe, in particular implementing definitional proof-irrelevance. Finally, we clarify the rôle of the two new type-formers that allow the interaction between the propositions and the computationally relevant terms. This allows us to have both a computationally relevant and an irrelevant, but still substitutive, equality type.

The resulting system is an extension of PCIC that brings it closer to an extensional type theory, while retaining the good properties of an intensional type theory. Notably, type checking remains decidable, and the extension of the system we give should be conservative.

Arguments supporting its validity

During our investigation we verified that we can indeed still express the same reasoning principles on our new propositions as in pCIC, in particular with regards to the "singleton elimination" property of some of Coq's propositions. Using the aforementioned irrelevant equality, we were able to encode the inductive families we considered in our examples in a way that is faithful to their traditional presentation as schemata. We have not attempted any proofs for the system yet, but hope that the modular presentation will facilitate them and allow for a formalisation.

The construction for irrelevance we present should be portable to other intentional dependent type theories. It could for instance be added as a universe to Martin-Löf Type Theory as an alternative notion of proof-irrelevance.

Summary and future work

The system as it is presented now seems like a stable enough basis to start investigating its meta-theory. In particular, we want to show that subject reduction holds and that we can give an algorithmic version of convertibility, proving normalisation. We hope to continue the work of Abel on normalisation-by-evaluation for dependent type theories with impredicativity [Abe13] to construct such an algorithm, which should then be implemented to bring proof-irrelevance to the end-user. The notion of strict equality and further links with propositional truncation as it is presented in homotopy type theory (HoTT) should be explored.

Acknowledgements

Matthieu Sozeau deserves my fullest gratitude. He set me on the track of this question, which I highly enjoyed working on, and was a source of great insight in the vast world of type theory.

1 Introduction and Related Work

1.0.0.1 Syntax Conventions There are two different notions of equality in type theory, the definitional and the propositional one. Definitional equality is introduced by a judgement just like type-hood and defines when two objects are indistinguishable inside the theory. The propositional one on the other hand is a primitive type constructor that represents the logical connective of equality. As such, it can be hypothesised and can model arbitrarily complex provable equations.

- $Id\ A\ x\ y$: the equality type; also written as $x =_A y$
- $x \equiv y : A$: definitional equality, in our case always with respect to a type
- $x :\equiv t$: x can be unfolded transparently into its definition t
- $t[u/x]$: in t, u is substituted for x

Definition 1. A type is said to be definitionally (respectively propositionally) *(proof-) irrelevant* if all of its inhabitants are definitionally (resp. propositionally) equal.

In the following, unless stated otherwise, we will use proof-irrelevance to mean definitional proof-irrelevance. Conversion and definitional equality are used interchangeably.

1.1 Motivation

1.1.1 Algebraic Presentation: De Bruijn Criterion

The idea that a proof-assistant, or any piece of software that we want to have substantial confidence in, should be built around a small core is known as the de Bruijn criterion [BW05], after the late N.G. de Bruijn, who pioneered it for the AUTOMATH system [Ned94]. This ideal seems in closer reach for a *closed* theory, rather that an *open* theory, a style which has been advocated to accommodate for user-definable inductive families [Dyb94]. Instead of the traditional monolithic presentation of the Calculus of Inductive Construction, we thus follow Herbelin and Spiwack [HS13] and give a more modular, algebraic presentation of the Proof-Irrelevant Calculus of Constructions CIC. Instead of relying on the experienced readers intuition to extrapolate, we avoid the use of ellipses in our definitions. We feel that this facilitates understanding and hope that proof-assistants will share this reaction, that is to say this presentation should more easily lend itself to formalisation.

1.1.2 Proof-Irrelevance: Poincaré Principle

Poincaré is known for attacking logic and emphasising the rôle of intuition in mathematics: "Obvious" computations should not warrant explanations. In our setting, this can be understood as the fact that they should hold as *conversions* [Bar97]. There are, however, issues with this perspective:

- *Obvious* is not well-defined. For example, some systems allow $n + 0$ and $0 + n$ to be convertible thanks to rewriting but Coq's intentional theory cannot deal with this as a conversion. Indeed, it requires an induction to show this equality.
- A useful principle that seems reasonable to add is that of Unicity of Identity Proofs (UIP), which states that any two proofs of the equality type can be identified *definitionally*, as the inductive definition of the equality type has a single constructor with no computational content. While UIP is not provable in Type Theory, it can consistently be added as an axiom.
- The Poincaré principle is inherently in tension with the de Bruijn Criterion: Definitional equalities have to be verified by an algorithm which might be complex to implement and verify, thus potentially lowering the trust in the system.

With CIC we strengthen the conversion, by implementing definitional proof-irrelevance (which implies UIP for *propositional* equalities), thus extending the class of "obvious" identifications the system can handle automatically. Here we benefit from the algebraic presentation, as it confines the propositions, which have to be treated specially, to an orthogonal part of the formalism.

1.1.3 Benefits for programming and proving with dependent types

1.1.3.1 Subset types The addition of proof-irrelevance makes the system much more pleasant when working with dependently typed programs. In particular, the notion of subsets becomes closer to informal practice. Subsets are defined as pairs of a term and a proof of a proposition about it, hence two inhabitants of the same subset type become definitionally equal as soon as their underlying terms are convertible. In the original theory, such conversions would have to be witnessed by propositional equalities which most of the time have to appeal to proof-irrelevance as an axiom. This is the main obstacle to a computationally well-behaved and sound interpretation

of the PROGRAM extension of Sozeau [Soz07], which develops a language for strongly specified programs based on subset types.

1.1.3.2 Computational Behaviour As identified in [Wer06], who proposes a proof-irrelevant variant of CC based on annotations of binders, using a proof-erased version of programs can make conversion checking more efficient. While our proposal is not to erase proofs but to tag them and treat them as indistinguishable objects, we will get the same benefits in terms of efficient conversion.

1.2 Related Work

1.2.1 Curry-Howard (-de Bruijn-Lambek-BHK-...) Correspondence

The "main slogans" of the Curry-Howard correspondence are "propositions-as-types", "proofs-as-programs" and "formulas-as-types" [SU06]. This idea is manifested in PCIC which has a distinguished sort Prop for propositions, which morally is where logic should be done, as it gets erased by extraction and allows for impredicativity. In fact logic is usually concerned with the existence rather than the exact shape of proofs, and there is a consensus [Thi86; AB04] that there is a correspondence between propositions and types rather than an isomorphism. This will be reflected by our even sharper separation of Prop and Type.

1.2.2 Same same but different: Notions of Equality

While most constructions of type theory have evolved very little since their introduction, the treatment of equality is still an area of active research.

1.2.2.1 Definitional Equality There are two options for introducing definitional equality. It can be represented as an untyped, external relation on two terms u, v, denoted $conv\ u\ v$, usually governed by a set of rewriting rules deciding when two terms are convertible. This is the way it was introduced in a early version of Martin-Löf Type Theory (MLTT) [Mar98] and in Pure Type Systems, including the Calculus of Constructions. Alternatively, it can be made into a judgement $\Gamma \vdash u \equiv v : A$, defined on a type by type basis, which expresses that two terms u, v are equal with respect to the type A they inhabit. The latter definition allows to handle extensional rules such as -/uniqueness-principles more easily, thanks to the available type information. This approach is used in later presentations of MLTT [Mar82]. For

type-checking to remain decidable, this judgement needs to be decided by an algorithm, usually employing techniques of normalisation-by-evaluation.

1.2.2.2 Propositional Equality Type theories represent propositional equality as an inductive type introduced by the reflexivity constructor $\mathsf{refl}_A t : \mathit{Id}\ A\ t\ t$. In *extensional* theories, such as NuPrl [Con+86] there is a reflection rule that states that propositional equalities are included in the definitional equality: $\Gamma \vdash e : \mathit{Id}\ A\ u\ v$ implies $\Gamma \vdash u \equiv v : A$, giving up any hope for decidability of type checking.

In *intentional* type theories, this implication only holds in empty contexts $\cdot \vdash e : \mathit{Id}\ A\ u\ v \implies \cdot \vdash u \equiv v : A$ which follows from the canonicity property of type theory, which ensures that e must be an application of refl.

1.2.2.3 Streicher's Axiom K One might expect that a similar property would still hold under contexts, that is for any proof of equality $\Gamma \vdash e : \mathit{Id}\ A\ x\ x$, there is a proof p such that $\Gamma \vdash p : \mathit{Id}\ (\mathit{Id}\ A\ x\ x)\ e\ (\mathsf{refl}_A x)$. This is known as Streicher's axiom K and is equivalent to the Uniqueness of Identity Proofs principle (UIP). But Streicher and Hofmann showed [HS96] that there are models of type theory that do not validate UIP. On the other hand, there are variants of type theory which do model this principle, notably Observational Type Theory (OTT) [AMS07]. OTT was developed as a core type theory for a dependently typed programming language (Epigram), in which UIP is crucial to interpret dependent pattern matching.

While including K into the theory might seem like a reasonable choice for a dependently-typed programming language such as Epigram or Idris, we prefer for a proof assistant to keep the underlying logical system independent of this axiom, especially knowing that it is inconsistent with the homotopy interpretation of type theory [Pro13]. In fact, there has been some effort in the Agda community to modify their dependent pattern matching construct which made K provable [CDP14] to regain compatibility with this interpretation. In Coq, high-level pattern matching is reduced down to eliminators, encoded as primitive case-constructs, which do not allow proving K.

However, we would like to allow K on a specific equality type, which will live in the Prop sort. This way we can hope to get the benefits of pattern-matching in the style of OTT while retaining general compatibility with HoTT. In effect, we will derive this equality with the UIP principle from our general proof-irrelevance construction.

1.2.3 Proof-Irrelevance

1.2.3.1 Historical account The idea that equality of proofs of propositional statements should be trivial goes back as early as 1975 in AUTOMATH [Ned94, Sec. A.4] where Zucker's goal of formalising classical mathematics motivates the type/prop-distinction, breaking the full propositions/types symmetry of [Mar75]. The idea of proof-irrelevance is attributed to de Bruijn. He identifies that proof-irrelevance is incompatible with informative Σ-eliminations which hence have to be restricted. Indeed, in [Ned94, Sec. B.3] de Bruijn postulates definitional proof-irrelevance for propositions of definitionally equal types, using the logarithm as running example. In D.3, van Benthem Jutting discusses in subsection 4.0.3 the possibility to add proof-irrelevance either as an axiom or to include it in the definitional equality and concludes that only the latter would be sufficiently convenient to support his development.

In extensional theories, this principle is easily added, either by introducing a type constructor for squash-types as in NUPRL, or through subset-types as in PVS. For intentional theories however, integrating proof-irrelevance is delicate.

1.2.3.2 From Semantics to Syntax The first studies of proof-irrelevance in the intentional setting were semantic models where proofs are interpreted as truth-values. Hofmann constructs a categorical model of CC in [Hof97] where proof-irrelevance is valid, with the goal of defining subset types for specified programs as we have presented them earlier.

1.2.3.3 Observational Type Theory In [Alt99], Altenkirch proposes a setoid model, assuming proof irrelevance and η for Σ and Π types, where function extensionality holds while conserving canonicity of normal forms, decidability of type checking and allowing for large eliminations. This later gave rise to the development of Observational Type Theory, in which proof-irrelevance as well as functional extensionality are valid. This type theory departs significantly from traditional foundations by defining not only the definitional equality but also the equality *type* by recursion over the type-formers. This type equality proceeds by structural analysis of types, which goes drastically against and is incompatible with the idea of univalence, which says that type equality coincides with a much larger type equivalence relation.

1.2.3.4 Modal type theory Pfenning has considered proof-irrelevance as a modality [Pfe01] instead of prop/type-sorting, which is similar to our approach in that the theory is "non-extensional". On the other hand, it is very different syntactically, as Pfenning has two kinds of application and all binders appear in three versions, whereas we will reflect irrelevance on the level of types.

1.2.3.5 Bracket Types Building upon this work, Awodey and Bauer use a "bracket type" constructor $[A]$ which represents inhabitation of a type A while hiding its computational content. This gives a type-based criterion for propositionality. As their main concern is the study of the semantic properties of these bracket types, they work in an extensional type theory. In particular they give an undecidable elimination rule for bracket types.

1.2.3.6 Calculus of Constructions A set-theoretic model focusing on the pitfalls of impredicativity is presented in [MW03]. Like Pfenning's, this presentation also makes use of sort-, i.e. relevance-tagged binders. This line of work continues with the promised syntactic type theory in [Wer06], where conversion is defined using erasure of propositional content. The equality type is defined in Prop and the reduction of its eliminator is modified to rely on the convertibility of the extracted indices instead of matching on the refl constructor. Werner proves the Church-Rosser property and "very strongly conjectures" strong normalisation. An experimental implementation of this theory was developed for Coq by Sozeau. In [LW11], Lee and Werner build a set-theoretical model of a proof-irrelevant Calculus of Constructions, where the conversion is switched to a typed definitional equality. They do however not prove the equivalence with the untyped system and encounter problems related to the implicit inclusion of Prop into Type. The inclusion is rendered explicit in [HS13], which we will follow. It should simplify such a model construction for our theory.

1.2.3.7 Other Directions There are other works that focus on giving the user a more fine-grained control of what is considered as computationally relevant. They are based on syntactic annotations on the level of binders and on an extraction procedure [BB08; AS12]. They do however not provide the same kind of expressive power as proof-irrelevance, notably with regards to the treatment of irrelevant equality. An alternative notion was studied by Asperti and Guidi [AG12] where PTS are extended with a term constructor that sends a given term to an opaque, irrelevant one of the same type. This

has the advantage of being light on the syntax, but again does not capture full proof-irrelevance.

2 A Proof-Irrelevant Calculus of Constructions

We will now present a proof-irrelevant Calculus of Constructions that draws inspiration from the algebraic presentation of pCIC of Hugo Herbelin and Arnaud Spiwack [HS13] but extends the conversion relation to implement proof-irrelevance.

2.1 Grammar

$$
\begin{array}{lrl}
\text{contexts} & \Gamma, \Delta ::= & \cdot \mid \Gamma,\, x : A \\
\text{sorts} & s ::= & \mathsf{Prop} \mid \mathsf{Type}_i, \quad i \in \mathbb{N} \\
\text{variables} & v \in & \mathscr{V} \\
\text{terms} \quad t,\, u,\, v,\, A,\, B,\, C & ::= & s \mid v \mid \mathsf{False} \mid \mathsf{True} \mid \mathsf{I} \mid \\
 & & A + B \mid \mathsf{inl}\ A \mid \mathsf{inr}\ B \mid \mathsf{case}\, t\, \mathsf{as}\, v\, \mathsf{return}\, P\, \mathsf{of}\, \mathsf{inl}\ x \Rightarrow t \mid \mathsf{inr}\ y \Rightarrow u \mid \\
 & & \textstyle\sum_{(v:A)} B \mid (t, u) \mid \mathsf{pr}_1\, t \mid \mathsf{pr}_2\, t \mid \prod_{(v:A)} B \mid \lambda v : A.\, t \mid \\
 & & \mu\, v : A \to \mathsf{Type}_i.\, t \mid \mathsf{fix}\, v\, (\overrightarrow{v : A}) \Rightarrow t \mid t\ u \mid \\
 & & \|A\| \mid |t| \mid \mathsf{let}\ |v| := u\ \ \mathsf{in}\ \ t \mid \{A\} \mid \mathsf{prf} t \mid t.\mathsf{prf}
\end{array}
$$

2.2 Judgements

There are five different kinds of judgements:

$$
\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u \equiv v : A \qquad \mathsf{sp}_x A \qquad \Gamma \vdash \mathsf{guarded}\ f\ x_1 \ldots x_n \Rightarrow t
$$

Only the first three of these will be detailed in this work. The judgements of strict positivity $\mathsf{sp}_x F$ of a family F with respect to a variable x and guardedness will be left abstract (c.f. subsubsection 2.4.14). We do require them to be decidable.

2.3 Contexts

$$
\frac{}{\cdot\ \mathsf{ctx}}\ \mathsf{ctx\text{-}EMP} \qquad \frac{x_1 : A_1,\, \ldots,\, x_{n-1} : A_{n-1} \vdash A_n : \mathsf{Type}_i}{(x_1 : A_1,\, \ldots,\, x_{n-1} : A_{n-1},\, x_n : A_n)\ \mathsf{ctx}}\ \mathsf{ctx\text{-}EXT}
$$

Restricting the sort of the types of variables to be Type_i ensures that all propositional variables are tagged with $\{\cdot\}$. This bears resemblance with Werner's, Pfenning's and Abel's presentation of irrelevance, where all binders are annotated with the (ir-) relevance of the variables that are bound, except that ours is a type-based criterion.

2.4 Types

2.4.1 Generalities: Shape of Rules

For each type-former, we give its formation rule, specifying the requirements for a valid type, its introduction rule, describing the terms inhabiting it. The corresponding elimination rule or induction principle describes how a term of a type can be used, the computation rule gives definitional equalities, explaining the interaction of the elimination rules with the results of the introduction rules. Finally, there may be a uniqueness principle, which is a judgemental equality explaining how every element of the type is uniquely determined by the results of elimination rules applied to it.

2.4.2 Variables

$$\frac{(x_1 : A_1, \ldots, x_n : A_n)\ \mathsf{ctx} \qquad 1 \leq i \leq n}{x_1 : A_1, \ldots, x_n : A_n \vdash x_i\ :\ A_i}\ \mathsf{Vble}$$

2.4.3 Sorts

The types of types are called **sorts**. Compared to the traditional presentation of CC, the sort Set corresponds to Type_0, Type_i to Type_i and there is an impredicative sort Prop, at the bottom of the hierarchy but excluded from the implicit cumulativity.

$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathsf{Prop}\ :\ \mathsf{Type}_1}\ \mathsf{Prop\text{-}Ax} \qquad \frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathsf{Type}_i\ :\ \mathsf{Type}_{i+1}}\ \mathsf{Type\text{-}Ax}$$

$$\frac{\Gamma \vdash A\ :\ \mathsf{Type}_i}{\Gamma \vdash A\ :\ \mathsf{Type}_{i+1}}\ \mathsf{Type\text{-}CUMUL}$$

Note that the rule for cumulativity is derivable from the definition of conversion given in subsection 2.5.

2.4.4 The type of Absurdity False

$$\frac{\Gamma \ \mathsf{ctx}}{\Gamma \vdash \mathsf{False} : \mathsf{Prop}}\ \mathsf{False}\text{-}\textsc{form} \qquad \frac{\Gamma \vdash C : s \qquad \Gamma \vdash t : \mathsf{False}}{\Gamma \vdash !_C\, t : C}\ \mathsf{False}\text{-}\textsc{elim}$$

$$\frac{\Gamma \vdash u : \mathsf{False} \qquad \Gamma \vdash v : \mathsf{False}}{\Gamma \vdash u \equiv v : \mathsf{False}}\ \mathsf{False}\text{-}\textsc{irrel}$$

As inductive types are generated by their constructors, and we do not give any for False, it is an empty type. The second premise of the elimination rule assumes that we can construct a term t of type False, which is absurd. Hence the context Γ is contradictory and we can derive anything. In logic, this principle is known as ex falso quadlibet, in a program this corresponds to an unreachable point. For example, it can be used in a dead branch of a case construct. This illustrates why there is no computation rule corresponding to this elimination. Correspondingly, we make it a proof-irrelevant as it has no computational content.

We intend to give a type-based conversion algorithm to decide the judgement of definitional equality, in the spirit of [AMS07]. This enables us to give extensionally flavoured rules such as False-IRREL, which could not be implemented with an untyped conversion, which does not have access to the information that both u and v are inhabitants of False.

2.4.5 The Trivial Type True

$$\frac{\Gamma \ \mathsf{ctx}}{\Gamma \vdash \mathsf{True} : \mathsf{Prop}}\ \mathsf{True}\text{-}\textsc{form} \qquad \frac{\Gamma \ \mathsf{ctx}}{\Gamma \vdash \mathsf{I} : \mathsf{True}}\ \mathsf{True}\text{-}\textsc{intro}$$

$$\frac{\Gamma \vdash u : \mathsf{True}}{\Gamma \vdash u \equiv \mathsf{I} : \mathsf{True}}\ \mathsf{True}\text{-}\textsc{uniq}$$

Like for False, there is a uniqueness rule for True: terms of type True are all convertible to I and thus irrelevant. The absence of an elimination rule is easily explained in terms of irrelevance: Any type $A[x]$ constructed depending on an instance of $\mathsf{True}\ x{:}\mathsf{True}$ is convertible by True-IRREL to $A[\mathsf{I}/X]$. Therefore, any term implementing $A[u]$ for a particular $u{:}\mathsf{True}$ also implements $A[v]$ for any $v{:}\mathsf{True}$.

2.4.6 Disjoint Sums

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma \vdash B : \mathsf{Type}_j \qquad (\mathsf{Type}_i, \mathsf{Type}_j, s_3) \in \mathscr{R}}{\Gamma \vdash A + B : s_3}\ +\text{-FORM}$$

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma \vdash B : \mathsf{Type}_j \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{inl}\ a : A + B}\ +\text{-INTRO}_1$$

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma \vdash B : \mathsf{Type}_j \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{inr}\ b : A + B}\ +\text{-INTRO}_2$$

$$\frac{\begin{array}{c}\Gamma, z\!:\!A + B \vdash C : s \\ \Gamma, x\!:\!A \vdash u : C[\mathsf{inl}\ x/z] \qquad \Gamma, y\!:\!B \vdash v : C[\mathsf{inr}\ y/z] \qquad \Gamma \vdash t : A + B\end{array}}{\Gamma \vdash \mathsf{case}\, t\, \mathsf{as}\, z\, \mathsf{return}\, C\, \mathsf{of}\, \mathsf{inl}\ x \Rightarrow u \mid \mathsf{inr}\ y \Rightarrow v : C[t/z]}\ +\text{-ELIM}$$

$$\frac{\begin{array}{c}\Gamma, z\!:\!A + B \vdash C : s \\ \Gamma, x\!:\!A \vdash u : C[\mathsf{inl}\ x/z] \qquad \Gamma, y\!:\!B \vdash v : C[\mathsf{inr}\ y/z] \qquad \Gamma \vdash a : A\end{array}}{\Gamma \vdash \mathsf{case}\, \mathsf{inl}\ a\, \mathsf{as}\, z\, \mathsf{return}\, C\, \mathsf{of}\, \mathsf{inl}\ x \Rightarrow u \mid \mathsf{inr}\ y \Rightarrow v \equiv u[a/x] : C[\mathsf{inl}\ a/z]}\ +\text{-COMP}_1$$

$$\frac{\begin{array}{c}\Gamma, z\!:\!A + B \vdash C : s \\ \Gamma, x\!:\!A \vdash u : C[\mathsf{inl}\ x/z] \qquad \Gamma, y\!:\!B \vdash v : C[\mathsf{inr}\ y/z] \qquad \Gamma \vdash b : B\end{array}}{\Gamma \vdash \mathsf{case}\, \mathsf{inr}\ b\, \mathsf{as}\, z\, \mathsf{return}\, C\, \mathsf{of}\, \mathsf{inl}\ x \Rightarrow u \mid \mathsf{inr}\ y \Rightarrow v \equiv v[b/y] : C[\mathsf{inr}\ b/z]}\ +\text{-COMP}_2$$

Note that disjoint sums are always informative, even if they are constructed over two proof objects. Therefore, their sort has to be at least Type_0.

2.4.7 Π-Types: Dependent Functions

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma, x\!:\!A \vdash B : s_2 \qquad (\mathsf{Type}_i, s_2, s_3) \in \mathscr{R}_\Pi}{\Gamma \vdash \prod_{(x:A)} B : s_3}\ \Pi\text{-FORM}$$

$$\frac{\Gamma, x\!:\!A \vdash t : B}{\Gamma \vdash \lambda x\!:\!A.\, t : \prod_{(x:A)} B}\ \Pi\text{-INTRO} \qquad \frac{\Gamma \vdash u : \prod_{(x:A)} B \qquad \Gamma \vdash v : A}{\Gamma \vdash u\ v : B[v/x]}\ \Pi\text{-ELIM}$$

$$\frac{\Gamma, x\!:\!A \vdash t : B \qquad \Gamma \vdash v : A}{\Gamma \vdash (\lambda x\!:\!A.\, t)\ v \equiv t[v/x] : B[v/x]}\ \Pi\text{-COMP}$$

$$\frac{\Gamma \vdash t \,:\, \prod_{(x:A)} B}{\Gamma \vdash t \equiv (\lambda x{:}A.\, t\ x) \,:\, \prod_{(x:A)} B}\ \Pi\text{-UNIQ}$$

where

$$\begin{aligned}\mathscr{R} &= \left\{ (s_1, s_2, \max(s_1, s_2)) \right\} \\ \mathscr{R}_{impred} &= \{ (s, \mathsf{Prop}, \mathsf{Prop}) \} \\ \mathscr{R}_{\Pi} &= \mathscr{R}_{impred} \cup \mathscr{R}\end{aligned}$$

$\max \mathsf{Type}_i\, \mathsf{Type}_j = \mathsf{Type}_{\max i\, j} \quad \max s\, \mathsf{Prop} = \max \mathsf{Prop}\, s = s$

Our dependent function types that allow to quantify only over types of sort Type, as dictated by the ctx-EXT rule, but we let the codomain range over Prop as well to allow the construction of impredicative quantifications. It seems possible to get a more uniform framework where the codomain is restricted to Type as well and removing $\mathscr{R}_{impred}$ yet retaining impredicativity, using truncation of the respective lifted domain/codomain, but we did not explore this idea in depth yet. We have the usual $\beta-$rule and $\eta-$conversion.

In the special case of non-dependent functions, we will use the usual arrow-abbreviation.

$$A \to B :\equiv \prod_{(\_:A)} B$$

2.4.8 Σ-Types: Dependent Pairs

$$\frac{\Gamma \vdash A \,:\, \mathsf{Type}_i \qquad \Gamma, x{:}A \vdash B \,:\, s_2 \qquad (\mathsf{Type}_i, s_2, s_3) \in \mathscr{R}}{\Gamma \vdash \sum_{(x:A)} B \,:\, s_3}\ \Sigma\text{-FORM}$$

$$\frac{\Gamma, x{:}A \vdash B \,:\, s \qquad \Gamma \vdash a \,:\, A \qquad \Gamma \vdash b \,:\, B[a/x]}{\Gamma \vdash (a, b) \,:\, \sum_{(x:A)} B}\ \Sigma\text{-INTRO}$$

$$\frac{\Gamma \vdash t \,:\, \sum_{(x:A)} B}{\Gamma \vdash \mathsf{pr}_1\, t \,:\, A}\ \Sigma\text{-ELIM-1} \qquad \frac{\Gamma \vdash t \,:\, \sum_{(x:A)} B}{\Gamma \vdash \mathsf{pr}_2\, t \,:\, B[\mathsf{pr}_1\, t\,/x]}\ \Sigma\text{-ELIM-2}$$

$$\frac{\Gamma \vdash (a, b) \,:\, \sum_{(x:A)} B}{\Gamma \vdash \mathsf{pr}_1(a, b) \equiv a \,:\, A}\ \Sigma\text{-COMP-1} \qquad \frac{\Gamma \vdash (a, b) \,:\, \sum_{(x:A)} B}{\Gamma \vdash \mathsf{pr}_2(a, b) \equiv b \,:\, B[a/x]}\ \Sigma\text{-COMP-2}$$

$$\frac{\Gamma \vdash t \,:\, \sum_{(x:A)} B}{\Gamma \vdash (\mathsf{pr}_1\, t, \mathsf{pr}_2\, t) \equiv t \,:\, \sum_{(x:A)} B}\ \Sigma\text{-UNIQ}$$

As the name indicates, the second component of a dependent pair may be *dependent* on its first component. Because of the restriction of the sorts of types of variables added to the context in ctx-EXT of subsection 2.3, the first component of a pair has to be of sort Type. The elimination of a pair is defined using projections to its first and second components. This allows us to define a uniqueness rule, known as surjective pairing, the canonical form being the constructor applied to the projections. Like for dependent functions, we define a notation for the non-dependent case:

$$(A * B) :\equiv \sum_{(\_ : A)} B$$

2.4.9 Equality Types

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a =_A b : \mathsf{Type}_i} =\text{-FORM}$$

$$\frac{\Gamma \vdash A : \mathsf{Type}_i \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}_A a : a =_A a} =\text{-INTRO}$$

$$\frac{\begin{array}{c}\Gamma, x\!:\!A, y\!:\!A, p\!:\!x =_A y \vdash C : \mathsf{Type}_i \\ \Gamma, z\!:\!A \vdash c : C[z/x, z/y, \mathsf{refl}_A z/p] \end{array} \quad \Gamma \vdash a : A \qquad \Gamma \vdash b : A \qquad \Gamma \vdash e : a =_A b}{\Gamma \vdash \mathsf{ind}_{=_A}\, x.y.p.C\ z.c\ a\ b\ e : C[a/x, b/y, e/p]} =\text{-ELIM}$$

$$\frac{\begin{array}{c}\Gamma, x\!:\!A, y\!:\!A, p\!:\!x =_A y \vdash C : \mathsf{Type}_i \\ \Gamma, z\!:\!A \vdash c : C[z/x, z/y, \mathsf{refl}_A z/p] \qquad \Gamma \vdash a : A \end{array}}{\Gamma \vdash \mathsf{ind}_{=_A}\, x.y.p.C\ z.c\ a\ a\ \mathsf{refl}_A a \equiv c[a/z] : C[a/x, a/y, \mathsf{refl}_A a/p]} =\text{-COMP}$$

In $\mathsf{ind}_{=_A}$, x, y, and p are bound in C, and z is bound in c.

The inductive type family of equalities over a type A is defined as the smallest reflexive relation over A. This is witnessed by the fact that it is generated by a single constructor, refl. Its elimination principle is clearly justified by the fact that there is a single constructor. From this definition, we can derive the Leibnitz principle that equality is substitutive and show that it is an equivalence.

Henceforth we will avoid using the terminology "propositional equality" for the general equality *type*. Traditionally, it is thought that propositions and types should be identified through the so-called Curry-Howard

isomorphism. But in our setting we have a clear distinction between the types living in the sort of propositions Prop and the hierarchy of sorts Type, and Curry-Howard *correspondence* seems like a more appropriate name.

Note that two types whose equality is witnessed by a proof in a closed context are necessarily definitionally equal, but this does of course not hold for open terms. The reason for the distinction is that definitional equality should be a decidable property, while propositional equality captures logically much stronger statements: For instance, the judgement $\mathit{nat} : \mathsf{Set}, (+) : \mathit{nat} \to \mathit{nat} \to \mathit{nat} \vdash 0 + n \equiv n \ : \ \mathit{nat}$ holds definitionally, i.e. by computation of plus and is not witnessed by a term. On the other hand, $\mathit{nat} : \mathsf{Set}, (+) : \mathit{nat} \to \mathit{nat} \to \mathit{nat} \ \vdash n + 0 \equiv 0 + n \ : \ \mathit{nat}$ does not hold by computation, but we can give a term that inhabits the type $n + 0 =_{\mathit{nat}} 0 + n$, building on the induction principle for nat.

We do not postulate a uniqueness principle for equality types, known as UIP ("uniqueness of identity proofs"), so as to remain compatible with Coq and not to exclude possible models, such as the groupoid interpretation. This allows us to add axioms that would themselves be incompatible with UIP, such as univalence.

Nonetheless it is possible to define a proof-irrelevant notion of equality, by using the type of truncations. Naturally there is a price to pay for irrelevance, and the elimination rule for truncated equalities is weaker than for proof-relevant equalities, as detailed in subsubsection 2.4.11.

2.4.10 Lifting

$$\frac{\Gamma \vdash A \ : \ \mathsf{Prop}}{\Gamma \vdash \{A\} \ : \ \mathsf{Type}_0} \textsc{Lift-form} \qquad \frac{\Gamma \vdash A \ : \ \mathsf{Prop} \qquad \Gamma \vdash a \ : \ A}{\Gamma \vdash \mathsf{prf} a \ : \ \{A\}} \textsc{Lift-intro}$$

$$\frac{\Gamma \vdash a \ : \ \{A\}}{\Gamma \vdash a.\mathsf{prf} \ : \ A} \textsc{Lift-elim} \qquad \frac{\Gamma \ \mathsf{ctx}}{\Gamma \vdash (\mathsf{prf} a).\mathsf{prf} \equiv a \ : \ A} \textsc{Lift-comp}$$

$$\frac{\Gamma \vdash u \ : \ \{A\} \qquad \Gamma \vdash v \ : \ \{A\}}{\Gamma \vdash u \equiv v \ : \ \{A\}} \textsc{Lift-irrel}$$

Lifting can be seen as the channel of communication from Prop to Type, which allows propositions to be treated as data. In particular, we allow functions to abstract over lifted propositions. This guarantees us that any variable x or y of propositional type $P : \mathsf{Prop}$ can enter the context only after it is lifted to $\{P\}$. In turn, the irrelevance of the variable is preserved. This

has a practical advantage when checking the convertibility x and y, because their type is tagged as lifted, and they become definitionally equal trivially.

$$\Gamma,\ \boldsymbol{P}\!:\mathsf{Prop},\ x\,y\!:\{\boldsymbol{P}\} \vdash x \equiv y\ :\ \{\boldsymbol{P}\}$$

Let us compare this restriction to the situation where we allow to introduce variables whose type may not only be of sort Type but also variables with a type of sort Prop. Consider the situation where a type of sort Prop has been introduced in the context and we want to check the convertibility of two variables inhabiting it.

$$\Gamma,\ \boldsymbol{P}\!:\mathsf{Prop},\ x\,y\!:\boldsymbol{P} \vdash x \equiv y\ :\ \boldsymbol{P}$$

We do not have any criterion neither about the shape of the terms nor about their type that would allow us to conclude: There is no local evidence for this judgement. One could try to determine the sort of $\boldsymbol{P}$ and conclude that x and y should be convertible based on the fact that they are hypothetically propositional, but then the conversion would no longer be type-directed.

2.4.11 Truncation

$$\frac{\Gamma \vdash A\ :\ \mathsf{Type}_i}{\Gamma \vdash \|A\|\ :\ \mathsf{Prop}}\ \textsc{Trunc-form} \qquad \frac{\Gamma \vdash a\ :\ A \qquad \Gamma \vdash A\ :\ \mathsf{Type}_i}{\Gamma \vdash |a|\ :\ \|A\|}\ \textsc{Trunc-intro}$$

$$\frac{\Gamma \vdash a\ :\ \|A\| \qquad \Gamma \vdash \boldsymbol{P}\ :\ \mathsf{Prop} \qquad \Gamma, x\!:\!A \vdash p\ :\ \boldsymbol{P}}{\Gamma \vdash \mathsf{let}\ |x| \coloneqq a\ \ \mathsf{in}\ \ p\ :\ \boldsymbol{P}}\ \textsc{Trunc-elim}$$

$$\frac{\Gamma \vdash \mathsf{let}\ |x| \coloneqq a\ \ \mathsf{in}\ \ p\ :\ \boldsymbol{P}}{\Gamma \vdash \mathsf{let}\ |x| \coloneqq |a|\ \ \mathsf{in}\ \ p \equiv p[a/x]\ :\ \boldsymbol{P}}\ \textsc{Trunc-comp}$$

$$\frac{\Gamma \vdash x\ :\ \|A\| \qquad \Gamma \vdash y\ :\ \|A\|}{\Gamma \vdash x \equiv y\ :\ \|A\|}\ \textsc{Trunc-irrel}$$

Truncation is a type constructor turning an arbitrary informative type into a proposition. Conceptually, a term of a truncated type witnesses the fact that the type is *inhabited*, but does not provide any information about *how* it was constructed. Two terms in a truncated type are definitionally equal, so this precisely implements proof-irrelevance at this type. The rationale behind the elimination rule is that we allow introspection of the

term only to build an inhabitant of another proposition, which in turn will be uninformative itself.

In [AB04], the bracket type has the following elimination rule:

$$\frac{\Gamma \vdash a : \|A\| \qquad \Gamma \vdash P : s \qquad \Gamma, x{:}A \vdash p : P \qquad \Gamma, x{:}A, y{:}A \vdash p \equiv p[y/x] : P}{\Gamma \vdash \mathsf{let}\ |x| \coloneqq a\ \ \mathsf{in}\ \ p : P}\ \textsc{Awodey-Bauer}$$

This rule relies on an extensional equality to ensure that all usages of x are irrelevant, as expressed by the premise $p \equiv p[y/x]$, which can require arbitrarily complex reasoning using the propositional equality.

In contrast to their construction, the premises of our elimination rule are decidable, as we have a sorting condition on the type P we eliminate into. This ensures that p is itself irrelevant, even if it makes relevant use of x.

Indeed, in their system all propositionally irrelevant types are treated as definitionally irrelevant. In other words, irrelevance is reflective. In our case, the user has to be explicit about when she wants to use irrelevance.

In contrast to the propositions of homotopy type theory, the Prop sort captures only the "strict" propositions. Homotopy propositions are defined as those types A, such that $\mathsf{hProp}\,A :\equiv \prod_{(x\,y:A)} x =_A y$, while for strict propositions all of the inhabitants have to be equal definitionally.

For instance we can prove by induction that $Id\ nat$ is decidable, i.e. $\prod_{(n\,m:nat)} n =_{nat} m + \neg(n =_{nat} m)$, hence from Hedberg's theorem [Hed98], it follows that $\prod_{(m\,n:nat)} \mathsf{hProp}\ (x =_{nat} y)$. In our system, only the truncated equality over natural numbers is a proposition. But both h-propositions and our strict propositions can cohabit in the same system, thanks to the separation of the universe of strict propositions from that of general types.

2.4.12 Truncated Equality

$$\frac{\Gamma \vdash e : \{\|a =_A b\|\} \qquad \Gamma, x{:}A, y{:}A, p{:}\{\|x =_A y\|\} \vdash C : s \qquad \Gamma, z{:}A \vdash c : C\left[z/x, z/y, \mathsf{prf}|\mathsf{refl}_A z|/p\right]}{\Gamma \vdash \mathsf{ind}_{\|=\|}\ x.y.p.C\ z.c\ a\ b\ e : C[a/x, b/y, e/p]}\ \textsc{trunc-=-elim}$$

$$\frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash \mathsf{ind}_{\|=\|}\ x.y.p.C\ z.c\ a\ b\ e \equiv c[a/z] : C[a/x, a/y, e/p]}\ \textsc{trunc-=-comp}$$

In $\mathsf{ind}_{\|=\|}$, x, y, and p are bound in C, and z is bound in c.

Drawing inspiration from the presentation of the substitution principle of equality in [AMS07] and [Wer06], we can introduce a special elimination principle for truncated equalities. It differs from the principle derivable for any truncated type in that it allows to eliminate truncated equalities to Type. This allows to write programs that make informative use of the indices a, b, while still preventing them from looking at the original truncated proof. The computation rule correspondingly does not depend on the proof-term but relies on the definitional equality of the indices. This corresponds to Werner's reduction rule for the equality eliminator $\mathsf{Eq\_rec}$. Note that this truncated equality obviously enjoys Streicher's K axiom, as it follows from proof-irrelevance.

2.4.13 Inductive Fixpoint

$$\frac{\Gamma \vdash A : s \qquad \Gamma,\, X : A \to \mathsf{Type}_i \vdash F : A \to \mathsf{Type}_i \qquad \mathsf{sp}_X F}{\Gamma \vdash \mu\, X : A \to \mathsf{Type}_i.\, F : A \to \mathsf{Type}_i}\ \mu\text{-FORM}$$

$$\frac{\Gamma \vdash \mu\, X : A \to \mathsf{Type}_i.\, F : A \to \mathsf{Type}_i}{\Gamma \vdash \mu\, X : A \to \mathsf{Type}_i.\, F \equiv F[\mu\, X : A \to \mathsf{Type}_i.\, F\,/\,X] : A \to \mathsf{Type}_i}\ \mu\text{-COMP}$$

Directly following the presentation of [HS13], we give a least fixpoint operator over types, that allows to construct strictly positive inductive families and also allow free folding and unfolding of the fixpoint operator. In contrast, we do not allow to construct fixpoints directly in Prop, as it goes against our concept of irrelevance of propositions. A fixpoint over a propositional type would contradict the idea that any two proofs of a proposition should be indistinguishable inside the system. The computational behaviour of a fixpoint over a proof would clearly have to depend on its exact shape, as is explained in paragraph 2.6.3.2.

2.4.14 Fixpoint on Functions

$$\frac{\begin{array}{c}\Gamma \vdash \prod_{(x_1 : A_1)} \cdots \prod_{(x_n : A_n)} B : s \\ \Gamma,\, f : \prod_{(x_1 : A_1)} \cdots \prod_{(x_n : A_n)} B,\, x_1 : A_1,\, \ldots,\, x_n : A_n \vdash t : B \\ \Gamma \vdash \text{guarded } f\ x_1 \ldots x_n \Rightarrow t\end{array}}{\Gamma \vdash \text{fix } f\ (x_1 : A_1) \ldots (x_n : A_n) \Rightarrow t : \prod_{(x_1 : A_1)} \cdots \prod_{(x_n : A_n)} B}\ \text{FIX-INTRO}$$

Once we have defined inductive types, we can write recursive functions over them. Logically speaking, this allows us to realise their induction principles. To ensure their termination, we require that these definitions respect

the structural order of the data-type definitions we recurse over, which is verified by the guardedness-checker. The sophistication of this guardedness-checker is a design choice of the system. As it is part of the trusted code base, the Coq system implements a relatively concise structural criterion, while Agda uses a more elaborate type-based analysis. We plan to follow the tradition of Coq and use a simple guard-condition, but we can use the well-known method of wrapping a complex recursion into a structural recursion over the accessibility predicate Acc (c.f. paragraph 2.6.3.2).

2.5 Definitional Equality

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad \Gamma \vdash A \leq B : s}{\Gamma \vdash t : B}\ \text{-CONV}$$

$$\frac{\Gamma \vdash A_1 : s \qquad \Gamma \vdash A_1 \equiv A_2 : s \qquad \Gamma, x{:}A_1 \vdash B_1 \leq B_2 : s'}{\Gamma \vdash \prod_{(x:A_1)} B_1 \leq \prod_{(x:A_2)} B_2 : s'}\ \text{-}\Pi$$

$$\frac{i \leq j}{\Gamma \vdash \mathsf{Type}_i \leq \mathsf{Type}_j : s}\ \text{-Type} \qquad \frac{\Gamma \vdash A \leq B : s \qquad \Gamma \vdash B \leq C : s}{\Gamma \vdash A \leq C : s}\ \leq\text{-TRANS}$$

$$\frac{\Gamma \vdash A \equiv B : s}{\Gamma \vdash A \leq B : s}\ \equiv\text{-}$$

$$\frac{\Gamma \vdash u \equiv v : A \qquad \Gamma \vdash B : s \qquad \Gamma \vdash A \leq B : s}{\Gamma \vdash u \equiv v : B}\ \equiv\text{-COMPAT}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t \equiv t : A}\ \equiv\text{-REFL} \qquad \frac{\Gamma \vdash u \equiv v : A}{\Gamma \vdash v \equiv u : A}\ \equiv\text{-SYM}$$

$$\frac{\Gamma \vdash u \equiv v : A \qquad \Gamma \vdash v \equiv t : A}{\Gamma \vdash u \equiv t : A}\ \equiv\text{-TRANS}$$

The cumulativity relation $\leq$ used in the conversion rule formalises subtyping of dependent products with respect to the universe hierarchy. It includes the definitional equality relation $\equiv$. The definitional equality $\equiv$ contains all of the equations given in the presentation of the types and is a congruence for all term- and type-formers.

We will decide this judgement using a normalisation-by-evaluation algorithm, that compares terms in their canonical form according to their types. This is however work in progress and we do not have a definition of such an algorithm yet.

2.6 Derived Types

This terminates the presentation of the core type theory, but a day-to-day Coq user would hardly recognise his favourite proof-assistant here. Yet we argue that everything is right there and we shall show how we can derive the familiar concepts she is missing. But fear not, this is simply a different presentation and not a set of clumsy encodings, and we have good hope that the surface language of an implementation of Coq that will be presented to the user will be very close to the convenience of the current syntax, while allowing the power-user to look what is happening under the hood.

2.6.1 Finite Types

Somewhat contrary to common practice, we do not define falsehood and trivial truth as the truncations of the empty type $\mathbf{0}$ and respectively the unit $\mathbf{1}$, but rather take them as primitive and derive $\mathbf{0}$ and $\mathbf{1}$ from their corresponding propositions.

The reason for this has to do with the elimination behaviour of truth and falsehood, which can be eliminated into arbitrary sorts, even though they are propositions. In fact it has no importance whether $\mathbf{1}$ is defined in terms of True or the contrary, but we follow the choice made for False for homogeneity. Care has to be taken with regards to False, if instead of $\mathsf{False} : \mathsf{Prop}$ we take $\mathbf{0} : \mathsf{Type}_0$ as primitive and define falsehood as its truncation $\mathsf{False} :\equiv \|\mathbf{0}\| : \mathsf{Prop}$, we run into problems when we want to eliminate propositional contradictions into higher sorts. We would like to write the following elimination:

$$\dfrac{\dfrac{\Gamma \vdash H \,:\, P \to \mathsf{False} \qquad \Gamma \vdash p \,:\, P}{\Gamma \vdash H\ p \,:\, \|\mathbf{0}\|}\ \Pi\text{-ELIM} \qquad \dfrac{\Gamma \vdash A \,:\, \mathsf{Type}_i \qquad \dfrac{\Gamma,\, f : \mathbf{0} \vdash f \,:\, \mathbf{0}}{\Gamma,\, f : \mathbf{0} \vdash !_A\, f \,:\, A}\ \mathbf{0}\text{-ELIM}}{\Gamma \vdash \mathsf{let}\ |f| \coloneqq (H\ p)\ \mathsf{in}\ !_A\, f \,:\, A}\ \textsc{Trunc-elim}$$

But one of the premises of TRUNC-ELIM is violated: A is not of sort Prop. We could add such an elimination rule, but it would be quite ad-hoc because it makes the system less orthogonal.

On the other hand, the elimination of $\{\mathsf{False}\}$ to $A : \mathsf{Type}_i$ is well-behaved:

$$\dfrac{\dfrac{A : \mathsf{Type}_i,\, t : \{\mathsf{False}\} \vdash A \,:\, \mathsf{Type}_i \qquad A : \mathsf{Type}_i,\, t : \{\mathsf{False}\} \vdash t \,:\, \{\mathsf{False}\}}{A : \mathsf{Type}_i,\, t : \{\mathsf{False}\} \vdash t.\mathsf{prf} \,:\, \mathsf{False}}\,\textsc{Lift-elim}}{A : \mathsf{Type}_i,\, t : \{\mathsf{False}\} \vdash !_A\, t.\mathsf{prf} \,:\, A}\,\mathsf{False}\text{-}\textsc{elim}$$

Therefore, the empty type $\mathbf{0}$ is simply defined as the lifting of the absurdity, and the unit type $\mathbf{1}$ as the lifting of triviality.

$$\mathbf{0} :\equiv \{\mathsf{False}\}, \quad \mathbf{1} :\equiv \{\mathsf{True}\}, \quad \star :\equiv \mathsf{prfI}$$

With this design, the only propositions that allow for eliminations into Type are False, True and truncated equalities as presented in subsubsection 2.4.11. All of these are in fact irrelevant for the run-time behaviour of closed programs. In the case where a term of type False is eliminated, it can safely be erased, as we are at a point of the program that has a contradictory context and therefore should be unreachable. Terms of type True have no computational content whatsoever. The elimination of a truncated equality only allows to use the information on its indices inside the term, leaving the equality-proof opaque which can hence be erased.

2.6.2 Booleans

$$\mathbf{2} :\equiv \mathbf{1} + \mathbf{1}, \quad 0_{\mathbf{2}} :\equiv \mathsf{inl}\ \star, \quad 1_{\mathbf{2}} :\equiv \mathsf{inr}\ \star$$

Disjoint sums allow for large eliminations, and we can thus prove, as one would hope that $0_{\mathbf{2}} \neq 1_{\mathbf{2}}$.

2.6.3 Inductive Types and Families

We now have all the tools required to implement the usual inductive types.

Non-dependent inductive types like the natural numbers can conveniently be written as nullary fixed points:

$$\mu\, X : \mathsf{Type}_i.\, t :\equiv (\mu\, Y : \mathbf{1} \to \mathsf{Type}_i.\, \lambda\_ : \mathbf{1}.\, t[Y\ \star / X])\ \star$$

For indexed inductive types, we will use truncated equalities to constrain the indices of the recursive arguments.

2.6.3.1 Natural Numbers In Coq, we can give a definition of the natural numbers with two constructors, O and S:

```
Inductive nat : Set :=  O : nat | S : nat → nat
```

The natural numbers are defined as the following fixpoint, and the two constructors are defined such as to inhabit this fixpoint.

$$\begin{aligned} nat &:\equiv \mu\, X : \mathsf{Type}_0.\, \mathbf{1} + X \\ \mathrm{O} &:\equiv \mathsf{inl}\ \star : \mathbf{1} + nat \\ \mathrm{S} &:\equiv \lambda(n : nat).\ \mathsf{inr}\ n : \mathbf{1} + nat \end{aligned}$$

The usual recursion principle nat_rect over Type is now derivable and can be used to define the induction principle nat_ind over Prop.

$$\begin{aligned} \mathrm{nat\_rect} &:\equiv \lambda(P : nat \to \mathsf{Type})\ (f_0 : P\ 0)\ (f_S : \textstyle\prod_{(n:nat)} P\ n \to P\ (\mathrm{S}\ n)). \\ &\quad \mathrm{fix}\ F\ (n : nat) \Rightarrow \mathrm{case}\ n\ \mathrm{as}\ x\ \mathrm{return}\ P\ x\ \mathrm{of}\ \mathsf{inl}\ \_ \Rightarrow f_0 \mid \mathsf{inr}\ m \Rightarrow f_S\ m\ (F\ m) \\ \mathrm{nat\_ind} &:\equiv \lambda(P : nat \to \mathsf{Prop})\ (p_0 : \{P\ 0\})\ (p_S : \textstyle\prod_{(m:nat)} \{P\ m\} \to \{P\ (\mathrm{S}\ m)\}). \\ &\quad \mathrm{nat\_rect}\ (\lambda n : nat.\ \{Pn\})\ p_0\ p_S \end{aligned}$$

2.6.3.2 Acc: Accessibility Predicates Our decision to keep the guard-condition as simple as possible means that we can only do *structural* recursion. But many recursive functions one would naturally write in a functional programming language are based on *well-founded* recursion instead. Take for example the following implementation of euclidean division:

```
let rec div a b =
  if a < b then 0, a
  else let q, r = div (a - b) b in
       q + 1, r
```

The recursive call is made on $a - b$, which we know is smaller than a if $0 < b$. Combined with a proof that $<$ is a well-founded order, we can use the accessibility predicate `acc` to implement `div` as a structural recursion. We will not dwell into the details of how this encoding can be achieved [BC04]. However, let us analyse why Acc can not be a proposition:

$$\mathrm{Acc} :\equiv \lambda(A : \mathsf{Type}_i)\ (R : A \to A \to \mathsf{Prop}).\ \mu\, F : A \to \mathsf{Type}_i.\ \lambda x : A.\ \textstyle\prod_{(y:A)} R\ y\ x \to F y$$

From this definition, we can derive a fixpoint operator with the following type:

$$\prod_{(A:\mathsf{Type})} \prod_{(P:A\to\mathsf{Type})} \prod_{(R:A\to A\to s)} \prod_{(a:\prod_{(x:A)} \mathrm{Acc}\, R x)} \prod_{(f:\prod_{(x:A)}(\prod_{(y:A)} R\, y\, x \to P\, y)\to P\, x)} \prod_{(x:A)} P\, x$$

The meaning of this type is that for a type A and a relation R over it, assuming that we have an accessibility proof a for all x, computes $P\ x$ for any x, as long as it is provided with a function f that takes an x and a function that computes the recursive calls for any y lower than x. The computational content of this is a recursion over the accessibility proof. Assuming that we allow accessibility proofs to live in Prop, a closed proof term will be convertible to a hypothetical one. Now the normal forms of a definition applied to each of the two different proofs will generally not coincide, although we expect them to always be identified, as we require that convertibility is a congruence for application. The problem does not occur when Acc is defined in Type. We hope to be able to modify the extraction mechanism so as to eliminate the accessibility argument from the extracted code by an explicit annotation.

2.6.4 General Inductive Types

The algebraic presentation allows the definition of indexed inductive families. For more sophisticated inductive definitions, like mutual or nested inductive families, it is claimed that they can be encoded using techniques similar to those presented by Paulin-Mohring in [Pau96]. However, given the new treatment of propositions this conjecture needs to be studied thoroughly.

3 Logic

We can interpret standard first order predicate logic in our universe of propositions as follows:

$$
\begin{aligned}
[\![\bot]\!] &:\equiv \mathsf{False} \\
[\![\top]\!] &:\equiv \mathsf{True} \\
[\![A \wedge B]\!] &:\equiv \left\|\{[\![A]\!]\} * \{[\![B]\!]\}\right\| \\
[\![A \vee B]\!] &:\equiv \left\|\{[\![A]\!]\} + \{[\![B]\!]\}\right\| \\
[\![A \Rightarrow B]\!] &:\equiv \{[\![A]\!]\} \to [\![B]\!] \\
[\![\forall x : A, B]\!] &:\equiv \textstyle\prod_{(x:A)} [\![B]\!] \\
[\![\exists x : A, B]\!] &:\equiv \left\|\textstyle\sum_{(x:A)} [\![B]\!]\right\| \\
[\![a =_A b]\!] &:\equiv \left\|a =_A b\right\|
\end{aligned}
$$

Note that although the introduction rule of dependent pairs requires the first component of a pair to be of sort Type, singleton-elimination for conjunction is preserved. Indeed, the first and second projections out of a truncated pair of propositions can be defined as follows:

$$
\begin{aligned}
&\mathrm{pfst} \;:\; \textstyle\prod_{(A\ B:\mathsf{Prop})} \to \left\{\left\|\{A\} * \{B\}\right\|\right\} \to A \\
&\mathrm{pfst} :\equiv \lambda(A\ B : \mathsf{Prop}).\, \lambda t : \left\{\left\|\{A\} * \{B\}\right\|\right\}.\ \mathsf{let}\ |x| \coloneqq t.\mathsf{prf}\ \ \mathsf{in}\ \ (\mathsf{pr}_1\ x).\mathsf{prf} \\
&\mathrm{psnd} \;:\; \textstyle\prod_{(A\ B:\mathsf{Prop})} \to \left\{\left\|\{A\} * \{B\}\right\|\right\} \to B \\
&\mathrm{psnd} :\equiv \lambda(A\ B : \mathsf{Prop}).\, \lambda t : \left\{\left\|\{A\} * \{B\}\right\|\right\}.\ \mathsf{let}\ |x| \coloneqq t.\mathsf{prf}\ \ \mathsf{in}\ \ (\mathsf{pr}_2\ x).\mathsf{prf}
\end{aligned}
$$

4 Examples, Applications

We can define standard inductive types using the fixpoint operator. For example, the polymorphic list is encoded as follows:

$$
\begin{aligned}
\mathrm{list} &:\equiv \lambda A : \mathsf{Type}_i.\, \mu X : \mathsf{Type}_i.\, \mathbf{1} + A * X \\
\mathrm{nil} &:\equiv \mathsf{inl}\ \star \\
\mathrm{cons} &:\equiv \lambda(A : \mathsf{Type}_i)\,(h : A)\,(t : \mathrm{list}\ A).\ \mathsf{inr}\,(h, t)
\end{aligned}
$$

Its recursion scheme can be constructed in analogy of that of natural numbers 2.6.3.1.

For indexed data types, we propose a different encoding than that of [HS13], taking advantage of the truncated equality. This guarantees a certain canonicity property of the objects of indexed families: If we were to

use the untruncated equality to witness equalities between indices, as their encoding suggests, propositional equality of indexed objects would involve equalities on the proofs constraining the indices. But with primitive inductive types these equalities do not occur in the constructors, so they should not matter. Here we give the definition of the type of finite types with n elements and vectors in this encoding.

$$
\begin{aligned}
\mathtt{finite} &:\equiv \mu\, F : \mathit{nat} \to \mathsf{Type}_0.\ \lambda n : \mathit{nat}.\ \textstyle\sum_{(m:\mathit{nat})} \left\{ \left\| n = \mathrm{S}\, m \right\| \right\} + \sum_{(m:\mathit{nat})} \left\{ \left\| n = \mathrm{S}\, m \right\| \right\} * F\ m \\
\mathtt{finz} &:\equiv \lambda n : \mathit{nat}.\ \mathsf{inl}\,(n, \mathsf{prf}|\mathsf{refl}_{\mathit{nat}}\, \mathrm{S}\, n|) \\
\mathtt{fins} &:\equiv \lambda (m : \mathit{nat})\,(f : \mathtt{finite}\, m).\ \mathsf{inr}\,(m, (\mathsf{prf}|\mathsf{refl}_{\mathit{nat}}\, \mathrm{S}\, m|, f)) \\
\mathtt{vect} &:\equiv \lambda A : \mathsf{Type}_i.\ \mu\, V : \mathit{nat} \to \mathsf{Type}_i.\ \lambda n : \mathit{nat}.\ \left\{ \left\| n = \mathrm{O} \right\| \right\} + \textstyle\sum_{(m:\mathit{nat})} \left\{ \left\| n = \mathrm{S}\, m \right\| \right\} * A * (V\ m) \\
\mathtt{vnil} &:\equiv \mathsf{inl}\ \mathsf{prf}|\mathsf{refl}_{\mathit{nat}}\, \mathrm{O}| \\
\mathtt{vcons} &:\equiv \lambda (A : \mathsf{Type}_i)\,(n : \mathit{nat})\,(h : A)\,(t : \mathtt{vect}\ A\ n).\ \mathsf{inr}\,(n, (\mathsf{prf}|\mathsf{refl}_{\mathit{nat}}\, \mathrm{S}\, n|, (h, t)))
\end{aligned}
$$

In order to construct inductive types of sort Prop, we have to take the truncation of a definition in Type. For instance, the predicate *even* is represented as follows:

$$
\begin{aligned}
\mathtt{even} &:\equiv \left\| \mu\, F : \mathit{nat} \to \mathsf{Type}_0.\ \lambda n : \mathit{nat}.\ \left\{ \left\| n =_{\mathit{nat}} 0 \right\| \right\} + \textstyle\sum_{(m:\mathit{nat})} \left\{ \left\| n =_{\mathit{nat}} \mathrm{S}(\mathrm{S}\, m) \right\| \right\} * \left\{ \| F\ m \| \right\} \right\| \\
\mathtt{evenO} &:\equiv \left| \mathsf{inl}\ \mathsf{prf}|\mathsf{refl}_{\mathit{nat}}\, \mathrm{O}| \right| \\
\mathtt{evenSS} &:\equiv \lambda (m : \mathit{nat})\,(H_{rec} : \{\mathtt{even}\, m\}).\ \left| \mathsf{inr}\,(m, (\left\{ |\mathsf{refl}_{\mathit{nat}} \mathrm{S}(\mathrm{S}\, m)| \right\}, H_{rec})) \right|
\end{aligned}
$$

The standard definitions as one would write them in Coq of these types are recalled in Appendix B.

5 Conclusion

We have presented a type theory close to that of Coq with definitional proof-irrelevance. We have tested its validity on the treatment of the classic text-book inductive types. We have verified that we can recover the same reasoning principles for propositions as in Coq, including the singleton rules compatible with definitional proof-irrelevance. The next step will be to give an algorithm for definitional equality and to verify the meta-theory of the system. The verification of such an algorithm might be simplified by a modification to the system, where the only source of impredicativity is truncation.

A References

[Abe13] Andreas Abel. 'Normalization by Evaluation: Dependent Types and Impredicativity'. HDR. Munich: Institut für Informatik, Ludwig-Maximilians-Universität, 31st May 2013. URL: http://www.cse.chalmers.se/~abela/habil.pdf (cited on page 3).

[AS12] Andreas Abel and Gabriel Scherer. 'On irrelevance and algorithmic equality in predicative type theory'. In: *arXiv preprint arXiv:1203.4716* (2012). URL: http://arxiv.org/abs/1203.4716 (cited on pages 2, 9).

[Alt99] Thorsten Altenkirch. 'Extensional equality in intensional type theory'. In: *Logic in Computer Science, 1999. Proceedings. 14th Symposium on*. IEEE, 1999, pp. 412–420. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=782636 (cited on page 8).

[AMS07] Thorsten Altenkirch, Conor McBride and Wouter Swierstra. 'Observational equality, now!' In: *Proceedings of the 2007 workshop on Programming languages meets program verification*. ACM, 2007, pp. 57–68. URL: http://dl.acm.org/citation.cfm?id=1292608 (cited on pages 7, 12, 19).

[AG12] Andrea Asperti and Ferruccio Guidi. 'Type systems for dummies'. In: *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*. ACM, 2012, pp. 79–90. URL: http://dl.acm.org/citation.cfm?id=2103797 (cited on page 9).

[Asp+11] Andrea Asperti, Wilmer Ricciotti, Claudio Coen Sacerdoti and Enrico Tassi. 'The Matita interactive theorem prover'. In: *Automated Deduction–CADE-23*. Springer, 2011, pp. 64–69. URL: http://link.springer.com/chapter/10.1007/978-3-642-22438-6_7 (cited on page 1).

[AB04] Steven Awodey and Andrej Bauer. 'Propositions as [types]'. In: *Journal of Logic and Computation* 14.4 (2004), pp. 447–471. URL: http://logcom.oxfordjournals.org/content/14/4/447.short (cited on pages 2, 6, 9, 18).

[Bar97] Henk Barendregt. 'The Impact of the Lambda Calculus in Logic and Computer Science'. In: *Bulletin of Symbolic Logic* 3.02 (June 1997), pp. 181–215. ISSN: 1079-8986, 1943-5894. DOI: 10.2307/421013. URL: http://www.journals.cambridge.org/abstract_S1079898600007599 (cited on page 5).

[BW05] Henk Barendregt and Freek Wiedijk. 'The challenge of computer mathematics'. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 363.1835 (2005), pp. 2351–2375. URL: http://rsta.royalsocietypublishing.org/content/363/1835/2351.short (cited on page 4).

[BB08] Bruno Barras and Bruno Bernardo. 'The implicit calculus of constructions as a programming language with dependent types'. In: *Foundations of Software Science and Computational Structures*. Springer, 2008, pp. 365–379. URL: http://link.springer.com/chapter/10.1007/978-3-540-78499-9_26 (cited on page 9).

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004 (cited on page 23).

[CDP14] Jesper Cockx, Dominique Devriese and Frank Piessens. 'Pattern matching without K'. In: *International Conference on Functional Programming (ICFP 2014)*. ACM, Sept. 2014. URL: https://lirias.kuleuven.be/handle/123456789/452283 (cited on page 7).

[Con+86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. NJ: Prentice-Hall, 1986 (cited on pages 1, 7).

[Coq12] Coq Development Team. 'The Coq proof assistant reference manual, version 8.4'. 2012. URL: http://coq.inria.fr/refman/ (cited on page 1).

[Dyb94] Peter Dybjer. 'Inductive families'. In: *Formal aspects of computing* 6.4 (1994), pp. 440–465. URL: http://link.springer.com/article/10.1007/BF01211308 (cited on page 4).

[Hed98] Michael Hedberg. 'A coherence theorem for Martin-Löf's type theory'. In: *Journal of Functional Programming* 8.04 (1998), pp. 413–436. URL: http://journals.cambridge.org/abstract_S0956796898003153 (cited on page 18).

[HS13] Hugo Herbelin and Arnaud Spiwack. 'The Rooster and the Syntactic Bracket'. In: *arXiv preprint arXiv:1309.5767* (2013). URL: http://arxiv.org/abs/1309.5767 (cited on pages 2, 4, 9, 10, 19, 25).

[Hof97] Martin Hofmann. 'Proof irrelevance and subset types'. In: *Extensional Constructs in Intensional Type Theory*. Springer, 1997, pp. 89–113. URL: http://link.springer.com/chapter/10.1007/978-1-4471-0963-1_4 (cited on pages 2, 8).

[HS96] Martin Hofmann and Thomas Streicher. 'The Groupoid Interpretation of Type Theory'. In: *In Venice Festschrift*. Oxford University Press, 1996, pp. 83–111 (cited on page 7).

[LW11] Gyesik Lee and Benjamin Werner. 'Proof-irrelevant model of CC with predicative induction and judgmental equality'. In: *arXiv preprint arXiv:1111.0123* (2011). URL: http://arxiv.org/abs/1111.0123 (cited on page 9).

[Mar98] Per Martin-Löf. 'An intuitionistic theory of types'. In: *Twenty-five years of constructive type theory* 36 (1998), pp. 127–172 (cited on page 6).

[Mar75] Per Martin-Löf. 'An Intuitionistic Theory of Types: Predicative Part'. In: *Studies in Logic and the Foundations of Mathematics*. Ed. by H.E. Rose and J.C. Shepherdson. Vol. Volume 80. Elsevier, 1975, pp. 73–118. ISBN: 0049-237X. URL: http://www.sciencedirect.com/science/article/pii/S0049237X08719451 (cited on page 8).

[Mar82] Per Martin-Löf. 'Constructive Mathematics and Computer Programming'. In: *Studies in Logic and the Foundations of Mathematics*. Ed. by Jerzy Łoś L. Jonathan Cohen Helmut Pfeiffer and Klaus-Peter Podewski. Vol. Volume 104. Elsevier, 1982, pp. 153–175. ISBN: 0049-237X. URL: http://archive-pml.github.io/martin-lof/pdfs/Constructive-mathematics-and-computer-programming-1982.pdf (cited on page 6).

[McB99] Conor McBride. 'Dependently Typed Functional Programs and their Proofs'. PhD. University of Edinburgh, 1999. URL: http://www.lfcs.informatics.ed.ac.uk/reports/00/ECS-LFCS-00-419/ (cited on page 1).

[MW03] Alexandre Miquel and Benjamin Werner. 'The not so simple proof-irrelevant model of CC'. In: *Types for proofs and programs*. Springer, 2003, pp. 240–258. URL: http://link.springer.com/chapter/10.1007/3-540-39185-1_14 (cited on pages 2, 9).

[Ned94] R. P Nederpelt. *Selected papers on Automath*. Amsterdam; New York: Elsevier, 1994. ISBN: 0-444-89822-0 978-0-444-89822-7 (cited on pages 2, 4, 8).

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. Göteborg: Chalmers Univ. of Technology, 2007. ISBN: 978-91-7291-996-9 91-7291-996-5 (cited on page 1).

[Pau96] C. Paulin-Mohring. 'Définitions Inductives en Théorie des Types d'Ordre Supérieur'. HDR. Dec. 1996. URL: http://www.lri.fr/%20paulin/PUBLIS/habilitation.ps.gz (cited on page 24).

[Pfe01] Frank Pfenning. 'Intensionality, extensionality, and proof irrelevance in modal type theory'. In: *Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on*. IEEE, 2001, pp. 221–230. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=932499 (cited on pages 2, 9).

[Pol94] Robert Pollack. 'The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions'. Univ. of Edinburgh, 1994. URL: http://homepages.inf.ed.ac.uk/rpollack/export/thesis.ps.gz (cited on page 1).

[Pro13] The Univalent Foundations Program. 'Homotopy Type Theory: Univalent Foundations of Mathematics'. In: *arXiv preprint arXiv:1308.0729* (2013). URL: http://arxiv.org/abs/1308.0729 (cited on pages 2, 7).

[SU06] Morten Heine Sørensen and Paweł Urzyczyn. *Lectures on the Curry-Howard isomorphism*. Amsterdam; Boston [MA]: Elsevier, 2006. ISBN: 978-0-444-52077-7 0-444-52077-5 978-0-08-047892-0 0-08-047892-1. URL: http://www.sciencedirect.com/science/book/9780444520777 (cited on page 6).

[Soz07] Matthieu Sozeau. 'Subset coercions in Coq'. In: *Types for Proofs and Programs*. Springer, 2007, pp. 237–252. URL: http://link.springer.com/chapter/10.1007/978-3-540-74464-1_16 (cited on page 6).

[Soz08] Matthieu Sozeau. 'Un environnement pour la programmation avec types dépendants'. Orsay, France: Université Paris 11, Dec. 2008 (cited on page 1).

[Thi86] Thierry Coquand. 'An Analysis of Girard's Paradox'. In: *In Symposium on Logic in Computer Science*. IEEE Computer Society Press, 1986, pp. 227–236 (cited on page 6).

[Wer06] Benjamin Werner. 'On the strength of proof-irrelevant type theories'. In: *Automated Reasoning*. Springer, 2006, pp. 604–618. URL: http://link.springer.com/chapter/10.1007/11814771_49 (cited on pages 2, 6, 9, 19).

B Inductive Definitions in Coq

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A → list A → list A

Inductive finite: nat -> Set :=
| finz: ∀ n, finite (S n)
| fins: ∀ n, finite n -> finite (S n).

Inductive vect (A : Type) : nat -> Type :=
| vnil : vect A 0
| vcons : forall n, A -> vect A n -> vect A (S n).

Inductive even : nat -> Prop :=
| evenO : even 0
| evenSS n : even n -> even (S (S n)).
```