Observational Equality Meets CIC

Loïc Pujet, Nicolas Tabareau

Observational Equality Meets CIC

Loïc Pujet[1] and Nicolas Tabareau[2]

[1] University of Stockholm, Sweden
[2] Inria, France

Abstract. Equality is at the heart of dependent type theory, as it plays a fundamental role in specifications and mathematical reasoning. The standard way to handle it in mainstream proof assistants such as Agda, Lean or Coq is based on Martin-Löf's identity type, which comes straight out of the '70s—its elegance and simplicity have earned it a long-standing use, despite a major discrepancy with traditional mathematical formulations: it does not satisfy any extensionality principles. Recently, the work on observational equality has regained interest as a new way to encode equality in proof assistants that support a universe of definitionally proof-irrelevant propositions; however it has yet to be integrated in any major proof assistant, because it is not fully compatible with another important feature of type theory: indexed inductive types. In this paper, we propose a systematic integration of indexed inductive types with an observational equality, and show that this integration can only be completely satisfactory if the observational equality satisfies the computational rule of Martin-Löf's identity type. The second contribution of this paper is a formal proof that this additional computation rule, although not present in previous works on observational equality, can be integrated to the system without compromising the decidability of conversion.

1 Introduction

Equality is a fundamental part of mathematical reasoning and formal specification, and it is therefore at the heart of any proof assistant. In Martin-Löf Type Theory [17], it is expressed with the *identity type*, which is characterized by two elegantly simple principles: equality is reflexive, and an equality proof cannot be told apart from a proof by reflexivity from inside the theory (this is known as the *J rule*, or *transport*). From these two principles, it is possible to show that the identity type is symmetric, transitive, and even that it satisfies all the laws of a higher groupoid [9]. This Martin-Löf identity type serves as the base for the interpretation of equality in the proof assistants Agda, Coq and Lean.

Unfortunately, this alluring formulation suffers from serious drawbacks: it is impossible to prove extensionality principles for the identity type, and the uniform definition makes it difficult to integrate types for which the equality relation is specified *ad hoc*, such as quotient types. In practice however, quotient types and extensionality principles are pervasive in mathematics; in particular the principle of function extensionality—which says that two functions are equal

when they are equal at every point—is taken for granted by most mathematicians and computer scientists. While it is possible to postulate those extensionality principles as axioms, this comes at the price of blocking computation for the transport operator.

In order to improve this sorry state of affairs, the most natural solution is to go back at the root of the problem and replace the dysfunctional identity type with a better-behaved alternative, for instance with the *observational equality* of [6]. Unlike Martin-Löf's identity type, the observational equality has a specific definition for each type former, so that the definition of quotient types becomes straightforward and extensionality principles can be added without too much trouble. There is some amount of freedom in the precise implementation of this idea; in this work we will build upon the recently proposed system $\mathsf{CC}^{\mathrm{obs}}$ [23]. Thus in $\mathsf{CC}^{\mathrm{obs}}$, every type A is equipped with an observational equality $t \sim_A u$, defined as a proof-irrelevant proposition with a reflexivity proof written `refl`. The system also provides a primitive type-casting operator `cast` $A\ B\ e\ t$ that can be used to coerce a term t of type A to the type B, given a proof e that these two types are observationally equal. This type-casting operator can then be used to derive the J rule for the observational equality, which ensures that it is a reasonable notion of equality and thus a good candidate for an implementation in a proof assistant.

But even though the idea has been around for almost two decades, none of the mainstream proof assistants supports the observational equality as of 2023. One possible reason is that it is not so easy to integrate it with the sophisticated type systems of modern proof assistants such as Agda, Coq and Lean, and in particular with their system of inductive definitions. Thus, the first contribution of this work is to extend $\mathsf{CC}^{\mathrm{obs}}$ with the indexed inductive types of Coq and their computation rules, resulting in a system that we call $\mathsf{CIC}^{\mathrm{obs}}$. We do this by exhibiting a general mechanism that distinguishes casts on parameters which can be propagated in the arguments of constructors, and casts on indices which are blocked and create new normal forms. Therefore, the indexed inductive types of $\mathsf{CIC}^{\mathrm{obs}}$ can contain more inhabitants than their counterparts in CIC; they only coincide when indices are taken in a type with decidable equality (*e.g.,* natural numbers in the case of vectors). Additionally, in order to properly handle the propagation of the casts, we give a general account of which equalities can be deduced from an observational equality between two instances $I\ \vec{x}$ and $I\ \vec{y}$ of the same inductive type. The correct rule is slightly more subtle than the injectivity of type formers—in particular, when a parameter of I is not used in the definition of the constructors of the inductive type, the equality of the two instances does not imply the equality of the parameter.

Our treatment of indices is based on *Fordism*, a technique that makes use of the equality type to reduce indexed inductive definitions to parametrized definitions. Its usefulness in an observational context has already been noted in [5], but it should be emphasized that the computational faithfulness of Fordism crucially relies on the computation rule for transport, which is weakened in the system of [23]: the encoding of transport *via* the `cast` operator does not compute

on reflexivity proofs as well as the eliminator of Martin-Löf's identity type. More precisely, in $\mathsf{CC}^{\mathrm{obs}}$ it is possible to prove that the propositional equality

$$\texttt{cast}\ A\ A\ (\texttt{refl}\ A)\ t \quad \sim_A \quad t$$

is inhabited for any type A, but the equality does not hold definitionally. This seemingly harmless difference implies that the observational equality of $\mathsf{CC}^{\mathrm{obs}}$ cannot be used to encode the indexed definitions of CIC. This issue is well-known, and previous work [22] introduced an auxiliary equality defined as a quotient type to recover this computation rule at the cost of the definitional uniqueness of identity proofs (UIP), in a way that is reminiscent of Swan's identity type in cubical type theories [25]. In our new system $\mathsf{CIC}^{\mathrm{obs}}$, we go a step further and show that the tension can be fully resolved by using the idea of [4] that under certain conditions, definitional equalities that hold on closed terms can be extended to open terms by adding *new definitional equations on neutral terms*. Indeed, the failure of the computation rule for transport only occurs on open terms, since `cast` computes on types and terms instead of the equality proof. For instance, in the case of the identity cast on natural numbers it is already true in $\mathsf{CC}^{\mathrm{obs}}$ that $\texttt{cast}\ \mathbb{N}\ \mathbb{N}\ (\texttt{refl}\ \mathbb{N})\ 32 \equiv 32$, and similarly for any closed natural number—this is a direct consequence of the canonicity theorem proved in [22]. What is missing is the equation $\texttt{cast}\ \mathbb{N}\ \mathbb{N}\ (\texttt{refl}\ \mathbb{N})\ n \equiv n$ when n is a neutral term, in particular a variable. Thus the problem to be addressed is:

"*Can we add those new definitional equations while keeping conversion and type checking decidable?*"

In the case of the type of natural numbers, it is very tempting to transform this equation into a new reduction rule $\texttt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ n \Rightarrow n$. However the case of two neutral types `A` and `B` seems more delicate, since the corresponding rule $\texttt{cast}\ A\ B\ e\ t \Rightarrow t$ should fire only when `A` and `B` are convertible, and reduction rules that rely on a conversion premise are still poorly understood [28,1].

Fortunately, this is not the only way to support the desired definitional equality. Coming back to the case of natural numbers, if n is neutral then neither n nor $\texttt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ n$ will trigger the reduction of an eliminator; therefore the decision that $\texttt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ n \equiv n$ can be deferred to equality checking after reduction, in the same way that one usually decides η-equality for functions. The second contribution of this paper is a formal proof that this algorithm does indeed lead to a sound and complete decision procedure for conversion. The argument is formalized in Agda, (see Section 8), following previous work on logical relations [2,22,23].

Related work The first proof assistant to implement an observational equality was the now-defunct Epigram 2 [19]. Although it did not have a primitive scheme for inductive definitions *à la* Coq, Epigram 2 had support for indexed W-types based on a fancy notion of containers, and its equality type did implement the computation rule on reflexivity, meaning that the user could use it to encode indexed definitions using Fordism. The normalization and consistency of Epigram 2 is justified with an inductive-recursive embedding into Agda, but this

embedding does not account for the computation rule on reflexivity, which is only conjectured not to break normalization and decidability.

In the world of cubical type theories, more attention has been paid to the definition of general (higher) inductive types [10]. There, the situation is complicated by the fact that transport for the cubical equality does not supports definitional computation on reflexivity as of today (this is known as the *regularity* problem), thus the Fordism encoding cannot be used straightforwardly. Instead, Cavallo and Harper add a *fcoe* constructor to their indexed inductive types in order to keep track of the coercions on indices, and they obtain that an inhabitant of an inductive type in normal form is a chain of *fcoe* applied to a canonical constructor. These inductive definitions have been implemented in Cubical Agda [27] and have been used to develop a sizeable standard library.

2 Observational Equality Meets CIC at Work

The Calculus of Inductive Constructions (CIC), which is the theoretical foundation of the proof assistants Coq and Lean, includes a powerful scheme for inductive definitions [21]. It supports parameters, indices and recursive definitions, but also more exotic features such as mutually defined or nested families. The high level of generality of this scheme allows it to subsume types as diverse as the natural numbers, Σ-types, W-types, and Martin-Löf's identity type. If we are to extend Coq with an observational equality, then we need to understand how it interacts with these inductive definitions, and to devise suitable computation rules. While some of these rules are self-evident, others will turn out to be more delicate. In order to help the reader build their intuition, we study the observational version of three common inductive types: lists, Martin-Löf's identity type and vectors.

2.1 Lists

We start with a brief look at the datatype of lists parametrized by an arbitrary type. Its definition in Coq might look something like this:

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

The basic rules of the CIC already provide us with an eliminator and computation rules for this inductive type. In the language of Coq, these are implemented *via* a pattern-matching construction (`match`) and a guarded fixpoint operator (`fix`) [11]. But in an observational type theory, we need more than just the rules for introduction, elimination and computation—every type former should come with three additional ingredients: a definition of the observational equality between inhabitants, a definition of the observational equality between two instances of the type, and computation rules for `cast`.

There is some leeway for the definition of the observational equality on any given type. In its original version and most of the subsequent literature, the observational equality type itself evaluates to a domain-specific equality, meaning

that a proof of equality between two functions is *definitionally* the same as a proof of pointwise equality [6,23]. On the other hand, it is possible to implement an observational type theory in which the equality type does not reduce, but is instead equipped with primitive operators that can be used to convert (for instance) a pointwise equality of functions into an equality [7]. In this paper, we will go with the second approach, as it turns out to be better suited for an implementation in Coq.

Now, what operators should we add in the case of lists? Obviously, two lists should be observationally equal if and only if they are either both empty, or have equal heads and recursively equal tails. But as it turns out, this logical equivalence is already derivable from the induction scheme for lists and the J eliminator for the observational equality—just like we would prove it in plain intensional Martin-Löf Type Theory (MLTT). Therefore, we do not need to characterize the equality between lists any further. This stems from the fact that inductive types are free algebras, and do not need any sort of quotienting in their construction. The observational equality between inhabitants of the universe, on the other hand, does not profit from such an induction principle. Thus we add a new operator to our theory, which takes an equality between two list types and "projects" out an equality between the underlying types:

$$\texttt{eq\_list} : \texttt{list A} \sim \texttt{list B} \to \texttt{A} \sim \texttt{B}.$$

This principle is necessary, because a proof of equality between $\texttt{list}\,A$ and $\texttt{list}\,B$ should allow us to coerce a list of elements of A into a list of elements of B, and thus in particular it should allow us to coerce from A to B. Since this implication is in fact a logical equivalence (the converse direction is provable from the J eliminator), it does indeed fully determine the observational equality between list types. Finally, we need to add rules that explain how `cast` computes on lists. Unlike the computation rules for the observational equality types, these are very much necessary, unless we are fine with having stuck computations in an empty context. Here, there is only one natural choice: casting a constructor of `list A` should evaluate to the corresponding constructor of `list B`.

$$\begin{aligned}
&\texttt{cast}\ (\texttt{list}\,A)\ (\texttt{list}\,B)\ e\ \texttt{nil} && \equiv \texttt{nil}\\
&\texttt{cast}\ (\texttt{list}\,A)\ (\texttt{list}\,B)\ e\ (\texttt{cons}\,a\,l) && \equiv \texttt{cons}\,(\texttt{cast}\ A\ B\ (\texttt{eq_list}\ e)\ a)\\
&&&\qquad (\texttt{cast}\ (\texttt{list}\,A)\ (\texttt{list}\,B)\ e\ l)
\end{aligned}$$

Remark that in the case of a non-empty list, we need the `eq_list` axiom in order to apply `cast` to the head of the list. *Voilà*, this is all it takes for an observational type theory with lists. With this example under our belt, we now move on to a more sophisticated example.

2.2 Indices and Fordism

The next layer of complexity offered by the scheme of Coq is indices. Here, the story gets more complicated, as indexed definitions gain new inhabitants in the presence of the observational equality. To see this, consider Martin-Löf's identity type, which is the prototypical example of an indexed inductive definition:

```
Inductive Id (A : Type) (x : A) : A → Type := id_refl : Id A x x.
```

In intensional type theory, it is well-known that this equality type does not satisfy the principle of function extensionality. But in our observational type theory, it turns out we can to prove that Martin-Löf's identity type is *logically equivalent* to the observational equality (we can use the `cast` operator in one direction, and the induction principle for `Id` in the other direction). In particular, the principle of function extensionality is now provable for `Id`! As convenient as it might sound, it also implies that we can get an inhabitant of the type `Id` $(\mathbb{N} \to \mathbb{N})$ $(\lambda n.1 + n)$ $(\lambda n.n + 1)$ in the empty context, since the two functions are extensionally equal. But this inhabitant cannot be definitionally equal to `id_refl`, as the two functions are not convertible. From this, we deduce that the closed inhabitants of an indexed inductive type may include more than the *canonical* ones, *i.e.*, those that can be built out of the constructors of the inductive type.

In order to get a better grasp on these noncanonical inhabitants, we can turn our attention to *Fordism*. This technique was invented by Coquand for his work on the proof assistant half in the 1990s, as a way to reduce indexed inductive types to parametrized inductive types and an equality type. The name Fordism first appeared in [18], in reference to a famous quote by Henry Ford: "A customer can have a car painted any color he wants as long as it's black". Let us look at the construction at work on the inductive definition of vectors, which is a little less barebones than the inductive identity type:

```
Inductive vec (A:Type) : ℕ → Type :=
| vnil : vec A 0
| vcons : ∀ m, A → vec A m → vec A (S m).
```

Vectors are basically lists with an additional index that makes their length available in the type, ensuring that a vector of type `vec A n` contains `n` elements. In order to get the *forded* version of vectors, we modify their definition so that the index becomes a parameter, and the two constructors gain a new argument:

```
Inductive vec𝔽 (A:Type) (n : ℕ) : Type :=
| vnil𝔽 : n ∼ℕ 0 → vec𝔽 A n
| vcons𝔽 : ∀ m, A → vec𝔽 A m → n ∼ℕ S m → vec𝔽 A n.
```

Remark that a forded empty vector `vnil𝔽 e` can have *a priori* the type `vec A n` for any n, except that `e` is a witness that `n` is equal to 0. An empty vector can have any size you want, as long as it's zero! The point of Fordism is that the induction principle of `vec` can be derived for `vec𝔽`, by combining the induction principle provided by the CIC for `vec𝔽` and the eliminator of the equality:

```
vec_elim (A : Type) (P : ∀ n : ℕ, vec𝔽 A n → Type) :
   P 0 (vnil𝔽 0 refl) →
   (∀ (m : ℕ) (a : A) (v : vec𝔽 A m), P m v → P (S m) (vcons𝔽 (S m) m a v refl)) →
   ∀ (n : ℕ) (v : vec𝔽 A n), P n v.
vec_elim A P Pnil Pcons n (vnil𝔽 n e) ≡
  cast (P 0 (vnil𝔽 0 refl)) (P n (vnil𝔽 n e)) (vnil_ap A e) Pnil.
vec_elim A P Pnil Pcons n (vcons𝔽 n m a v e) ≡
  cast (P (S m) (vcons𝔽 (S m) m a v refl)) (P n (vcons𝔽 n m a v e))
```

(vcons$_{ap}$ A m a e v) (Pcons m a v (vec_elim A P Pnil Pcons m v)).

Here, we used implicit arguments for refl and we used two auxiliary definitions vnil$_{ap}$ and vcons$_{ap}$ showing that functions preserve equalities. Furthermore, *if the* cast *operator satisfies the computation rule on reflexivity*, then the induction principle provided by the Fordism transformation satisfies the same computation rules as the standard induction principle for indexed inductive types. Thus, Fordism can serve as a recipe for the implementation of indexed inductive types, as long as we know how to handle parametrized inductive types and have an equality that computes on reflexivity.

Additionally, this transformation sheds some light on the noncanonical elements of indexed inductive types: in CIC, the only closed proof of equality is a proof by reflexivity, thus the inhabitants of vec$_{\mathbb{F}}$ A n in the empty context behave exactly like the canonical inhabitants of vec A n. But in an observational type theory, there are many proofs of equality in the empty context (think for example of a proof of equality between two functions that are not convertible, but extensionally equal) which give rise to new elements. These elements can be obtained by casting a canonical inhabitant to a type with a different (but observationally equal) index, and they cannot be eliminated away in general.[3]

2.3 Parameters and Equalities

Now that we know how to handle indexed types, we can revisit Martin-Löf's identity type, which plays an important role in CIC. After the Fordism transformation, its definition looks like this:

Inductive Id$_{\mathbb{F}}$ (A : Type) (x y : A) : Type := id_refl$_{\mathbb{F}}$: x $\sim_A$ y $\to$ Id$_{\mathbb{F}}$ A x y.

As we want to incorporate this type into our observational theory, we apply the standard recipe: we need a definition of the observational equality between inhabitants of Id$_{\mathbb{F}}$, a definition of the observational equality between two instances of Id$_{\mathbb{F}}$, and computation rules for the cast operator. The first one is easy, as we can prove that any two inhabitants of Id$_{\mathbb{F}}$ A x y are equal: by induction, we only need to prove it for elements of the form id_refl$_{\mathbb{F}}$ e, with e being a proof of x $\sim_A$ y. But the observational equality is definitionally proof-irrelevant, so this is true by reflexivity. In other words, the principle of *uniqueness of identity proofs* (UIP) is provable for the inductive identity type in observational type theory, in stark contrast with MLTT or CIC. Thus, we do not need any further characterization of the observational equality between inhabitants of Id$_{\mathbb{F}}$.

On the other hand, the definition of the observational equality between two instances of the identity type Id$_{\mathbb{F}}$ A x y and Id$_{\mathbb{F}}$ A' x' y' makes for another interesting story. From our study of lists, it might be tempting to extrapolate that an observational equality between two instances of a parametrized inductive

[3] In the case of vectors, it is possible to find alternative encodings that do not have these new canonical elements, because the equality between indices is decidable in the empty context. However, we aim at a systematic and uniform treatment of indexed inductive types, so we will not consider this option.

datatype should imply an equality between the parameters, or in the special case of $\mathtt{Id}_{\mathbb{F}}$, that we get the following principle:

$$\mathtt{Id}_{\mathbb{F}}\ \mathtt{A}\ \mathtt{x}\ \mathtt{y} \sim \mathtt{Id}_{\mathbb{F}}\ \mathtt{B}\ \mathtt{z}\ \mathtt{w} \to \exists\ (\mathtt{e} : \mathtt{A} \sim \mathtt{B}),\ (\mathtt{cast}\ \mathtt{A}\ \mathtt{B}\ \mathtt{e}\ \mathtt{x} \sim \mathtt{z}) \wedge (\mathtt{cast}\ \mathtt{A}\ \mathtt{B}\ \mathtt{e}\ \mathtt{y} \sim \mathtt{w})$$

This means that parametrized inductive definitions are *injective* functions from the type of parameters to the universe. Unfortunately, this idea turns out to be incompatible with the rules of CIC. Indeed, according to these rules the inductive equality `Id A x y` should live in the lowest universe, since it has only one constructor with no arguments. But then if `A` is a large type, we get an injective function from `A` into the lowest universe, which is potentially inconsistent—for instance, consider the following function:

$$\mathtt{inj}\ (\mathtt{X} : \mathtt{Type} \to \mathtt{Type}) := \mathtt{Id}_{\mathbb{F}}\ (\mathtt{Type} \to \mathtt{Type})\ \mathtt{X}\ \mathtt{X}$$

If the $\mathtt{Id}_{\mathbb{F}}$ type former is injective, then `inj` is an injection of `Type` → `Type` into `Type`, from which we can encode Russell's paradox and derive an inconsistency for CIC [20]. Thus, if we really want to have this injectivity of parameters, we need to modify the rules of our theory so that inductive definitions are only allowed in a universe that is sufficiently large to accommodate their parameters. But this is not exactly reasonable: this would mean that we cannot abstract over the definition of an inductive type using Coq's sections mechanism, since section variables are translated to inductive parameters. In other words, inductive definitions would only make sense in closed contexts.

In order to avoid such a serious drawback, we will use a completely different characterization for the observational equality between inductive types. After all, what do we need these axioms for? The answer is simple: we need some observational equalities to put in the computation rules for the `cast` operator.

$$\mathtt{cast}\ (\mathtt{Id}_{\mathbb{F}}\ \mathtt{A}\ \mathtt{x}\ \mathtt{y})\ (\mathtt{Id}_{\mathbb{F}}\ \mathtt{B}\ \mathtt{z}\ \mathtt{w})\ \mathtt{e}\ (\mathtt{id\_refl}_{\mathbb{F}}\ \mathtt{e'}) \equiv \ldots$$

For inductive types, these computation rules are very systematic: when `cast` is applied to a constructor, then it should naturally reduce to the corresponding constructor of the target inductive. Thus, we need to produce an inhabitant of $\mathtt{x'} \sim_{A'} \mathtt{y'}$ from an inhabitant of $\mathtt{x} \sim_{A} \mathtt{y}$. This is a job for the `cast` operator:

$$\mathtt{cast}\ (\mathtt{Id}_{\mathbb{F}}\ \mathtt{A}\ \mathtt{x}\ \mathtt{y})\ (\mathtt{Id}_{\mathbb{F}}\ \mathtt{B}\ \mathtt{z}\ \mathtt{w})\ \mathtt{e}\ (\mathtt{id\_refl}_{\mathbb{F}}\ \mathtt{h}) \equiv \mathtt{id\_refl}_{\mathbb{F}}\ (\mathtt{cast}\ (\mathtt{x} \sim \mathtt{y})\ (\mathtt{z} \sim \mathtt{w})\ ?\ \mathtt{h}).$$

In order to fill the question mark hole, we need a proof of observational equality between the two observational equality types. Since all we have is a proof of equality between $\mathtt{Id}_{\mathbb{F}}\ \mathtt{A}\ \mathtt{x}\ \mathtt{y}$ and $\mathtt{Id}_{\mathbb{F}}\ \mathtt{B}\ \mathtt{z}\ \mathtt{w}$, we need something to extract the desired equality. The injectivity of the inductive types is *sufficient* for this purpose, but it is not *necessary*. Instead, we can go for the bare minimum: an observational equality between two instances of the same inductive definition should imply the equality of all their argument contexts, and nothing more. In the case of the inductive $\mathtt{Id}_{\mathbb{F}}$, it means that we get the following projection:

$$\mathtt{eq\_Id}_{\mathbb{F}} : \mathtt{Id}_{\mathbb{F}}\ \mathtt{A}\ \mathtt{x}\ \mathtt{y} \sim \mathtt{Id}_{\mathbb{F}}\ \mathtt{B}\ \mathtt{z}\ \mathtt{w} \to (\mathtt{x} \sim \mathtt{y}) \sim (\mathtt{z} \sim \mathtt{w}).$$

As we will prove in Section 6, this is enough to get an identity type that lives in the lowest universe without endangering the consistency of the theory.

$$
\begin{array}{lll}
i, j & \in \mathbb{N} & \text{Universe levels} \\
s & ::= \mathcal{U}_i \mid \Omega & \text{Universes} \\
\Gamma, \Delta & ::= \bullet \mid \Gamma, x : A : s & \text{Contexts} \\
t, u, m, n, e, A, B & ::= x \mid s & \text{Variables and Universes} \\
& \mid \lambda(x : A).\, t \mid t\ u \mid \Pi^{s,s'}(x : A).\, B & \text{Dependent products} \\
& \mid \bot\texttt{-elim}\ A\ t \mid \bot & \text{Empty type} \\
& \mid t \sim_A u \mid \texttt{refl}\ t \mid \texttt{transport}\ A\ t\ B\ u\ t'\ e & \text{Observational equality} \\
& \mid \texttt{cast}\ A\ B\ e\ t & \text{Type cast} \\
& \mid \Pi^1_\epsilon \mid \Pi^2_\epsilon \mid \Omega_{\text{ext}} \mid \Pi_{\text{ext}} & \text{Properties of Equality}
\end{array}
$$

Fig. 1: Syntax for the negative fragment of $\mathsf{CIC}^{\text{obs}}$ *[Untyped.agda]*

3 $\mathsf{CIC}^{\text{obs}}$ with Martin-Löf's Computation Rule

At this stage, we have a clear roadmap for our observational type theory with inductive types: first, we need a system with a `cast` operator that computes on proofs by reflexivity. Then, we handle parametrized inductive types with projection functions for the equality types and computation rules for `cast`, and finally, we can take care of indexed inductive types with some syntactic sugar around the Fordism transformation.

We are now in position to define $\mathsf{CIC}^{\text{obs}}$, the observational type theory that will serve as our theoretical framework. It is based on the system CC^{obs} of [23], but with a few tweaks; the most important one being the additional computation rule for the `cast` operator on reflexivity proofs. In this section, we give a brief presentation of the syntax, typing rules and declarative conversion for the core of the type theory, with an emphasis on the points that differ from CC^{obs}, before defining the scheme for inductive definitions in Section 5. All the definitions in the figures follows closely our Agda formalization. We refer to files in the formalization as *[myFile.agda]*.

3.1 The Syntax of $\mathsf{CIC}^{\text{obs}}$

The syntax of the sorts, contexts, terms and types of $\mathsf{CIC}^{\text{obs}}$ is specified in Fig. 1. The sorts of our system are divided into a predicative hierarchy $(\mathcal{U}_i)_{i\in\mathbb{N}}$ which mirrors the `Type` hierarchy of Coq, and an impredicative sort Ω of proof-irrelevant propositions which corresponds to Coq's `SProp`. The base types are the false proposition $\bot$, the observational equality $t \sim_A u$ and the dependent function type $\Pi^{s,s'}(x : A).\, B$. For the sake of readability, we will frequently drop the sort annotations on dependent products when they can be inferred from the context, and when B does not depend on A, we write $A \to B$ instead of $\Pi(x : A).\, B$. In addition to these basic types, our theory also includes a definition scheme for indexed inductive types, that can be used to extend the syntax with new types and terms (cf. Section 5).

Compared to the system CC^{obs} of [23], we add four new primitives Π^1_ϵ , Π^2_ϵ , Ω_{ext} and Π_{ext}, whose role is to provide the properties of the observational

$$\textsc{Eq-}\Omega\ \frac{\Gamma \vdash A : \Omega \qquad \Gamma \vdash B : \Omega}{\Gamma \vdash \Omega_{\text{ext}} : (A \to B) \to (B \to A) \to A \sim_\Omega B}$$

$$\textsc{Eq-Fun}\ \frac{\Gamma \vdash A : s \quad \Gamma, x : A : s \vdash B : \mathcal{U}_i \quad \Gamma \vdash f, g : \Pi(x : A).\, B : \mathcal{R}(\mathcal{U}_i, s)}{\Gamma \vdash \Pi_{\text{ext}} : \Pi(x : A).\, f\ x \sim_B g\ x \to f \sim_{\Pi AB} g}$$

$$\textsc{Eq-}\Pi_1\ \frac{\Gamma \vdash A, A' : s \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma, x : A' \vdash B' : s'}{\Gamma \vdash \Pi^1_\epsilon\ : \Pi(x : A).\, B \sim_{\mathcal{R}(s,s')} \Pi(x : A').\, B' \to A' \sim_s A}$$

$$\textsc{Eq-}\Pi_2\ \frac{\Gamma \vdash A, A' : s \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma, x : A' \vdash B' : s'}{\Gamma \vdash \Pi^2_\epsilon\ : \Pi(e : \_).\, \Pi(a' : A').\, B[x := \texttt{cast}\ A'\ A\ (\Pi^1_\epsilon\ e)\ a'] \sim_{s'} B'[x := a']}$$

Fig. 2: $\mathsf{CIC}^{\text{obs}}$ rules for characterizing the observational equality *[Typed.agda]*

equality which were previously given as computation rules. For instance, in the system of [23] an equality between two function types evaluates to a Σ-type that contains equalities of the domain and codomain, while in our new system these two equalities are obtained by applying Π^1_ϵ and Π^2_ϵ to the proof of equality between function types. Replacing computations with these new primitives does *not* endanger the computational properties of our theory, since they only ever produce computationally irrelevant equality proofs. Plus, it results in a more elegant system that does not need a primitive Σ-type; this way of handling the properties of the observational equality will be especially convenient when dealing with inductive definitions, where equalities between types imply complex telescopes of equalities which would be cumbersome to express with nested Σ-types.

3.2 The Typing Rules of $\mathsf{CIC}^{\text{obs}}$

The typing rules of $\mathsf{CIC}^{\text{obs}}$ are based on five judgments:

$\vdash \Gamma$	Γ is a well-formed context,
$\Gamma \vdash A : s$	A is a well-formed type of sort s in Γ,
$\Gamma \vdash t : A : s$	t is a term of type A in sort s in Γ,
$\Gamma \vdash A \equiv B : s$	A and B are convertible types of sort s in Γ, and
$\Gamma \vdash t \equiv u : A : s$	t and u are convertible terms of type A in Γ.

In all the judgments, s denotes either $\mathcal{U}_i$ or Ω. Note that since every universe has a type, the well-formedness judgments for types $\Gamma \vdash A : s$ (and convertibility judgments of types) can be seen as special cases of typing judgments for terms $\Gamma \vdash A : s : s'$ for a suitable s', but we keep the type-level judgments to avoid writing unnecessarily many sort variables.

The rules for universes, dependent function types, and the empty type are taken directly from [23], so we only give a brief overview here. The complete set

of rules is available in *[Typed.agda]*. We use PTS-style notations [8] to factorize the impredicative and predicative rules for universes and dependent products: the formation rule for universes states that both $\mathcal{U}_i$ and Ω are inhabitants of a higher universe, as described by the relations

$$\mathcal{A}(\mathcal{U}_i, \mathcal{U}_j) := j = i + 1 \qquad \mathcal{A}(\Omega, \mathcal{U}_i) := i = 0.$$

We allow the formation of dependent products with a domain and a codomain that have different sorts. If the codomain is a proof-relevant type, then the dependent product should have a universe level that is the maximum between the level of the domain and that of the codomain. On the other hand, if the codomain is a proposition then the result is a proposition regardless of the size of the domain. This is made formal by using the function $\mathcal{R}(\_, \_)$ defined as

$$\mathcal{R}(s, \Omega) := \Omega \qquad \mathcal{R}(\Omega, \mathcal{U}_i) := \mathcal{U}_i \qquad \mathcal{R}(\mathcal{U}_i, \mathcal{U}_j) := \mathcal{U}_{max(i,j)}.$$

Equality and Type Casts Every proof-relevant type comes equipped with a propositional binary relation, noted $t \sim_A u$ and called the *observational equality*. This type has one introduction rule that turns it into a reflexive relation. Of course, proof-irrelevant types have no use for an observational equality, since any two inhabitants would always be in relation by reflexivity. The observational equality is equipped with two elimination principles, which are called `transp` and `cast`. The former is similar to the J eliminator from MLTT, except that it is restricted to propositional predicates. Elimination into the proof-relevant layer is thus handled by the `cast` operator, which provides coercions between two observationally equal types. It might seem less general than the standard J eliminator, but since equality proofs are definitionally irrelevant, it turns out that a J eliminator for proof-relevant predicates can be derived from the `cast` operator.

As we already mentioned, the extensional properties of the observational equality are given by the primitives $\mathbf{\Pi}^1_\epsilon$, $\mathbf{\Pi}^2_\epsilon$, $\mathbf{\Omega}_{\text{ext}}$ and $\mathbf{\Pi}_{\text{ext}}$: rules Eq-$\mathbf{\Pi}_1$ and Eq-$\mathbf{\Pi}_2$ allow us to deduce the equality of domains and codomains from an equality between two dependent functions types, rule Eq-$\mathbf{\Omega}$ provides propositional extensionality, and rule Eq-Fun provides function extensionality.

3.3 Conversion

The conversion, also called *definitional* equality, is a judgment that relates the terms that are interchangeable in typing derivations. The rules that define the conversion judgment are reproduced in Fig. 3. By definition, conversion is a reflexive, symmetric and transitive relation. It is also closed under congruence (e.g. if $A \equiv A'$ and $B \equiv B'$ then $\mathbf{\Pi}(x : A).B \equiv \mathbf{\Pi}(x : A').B'$), although we did not reproduce all the corresponding rules in Fig. 3 for the sake of brevity. The conversion judgment is itself subject to the conversion rule (rule Conv-Conv).

As usual, the conversion relation contains the β-equality for proof-relevant applications (rule β-conv), and the η-equality of functions[4] (rule η-Eq). The

[4] The propositional η-equality is actually provable in observational type theory, since it is a special case of the extensionality of functions. Nevertheless, it is still convenient to have as a conversion rule, to get a more flexible system.

$$\textsc{Refl}\ \frac{\Gamma \vdash t : A : \mathcal{U}_i}{\Gamma \vdash t \equiv t : A : \mathcal{U}_i} \qquad \textsc{Sym}\ \frac{\Gamma \vdash t \equiv u : A : \mathcal{U}_i}{\Gamma \vdash u \equiv t : A : \mathcal{U}_i} \qquad \textsc{Trans}\ \frac{\Gamma \vdash t \equiv t' : A : \mathcal{U}_i \qquad \Gamma \vdash t' \equiv u : A : \mathcal{U}_i}{\Gamma \vdash t \equiv u : A : \mathcal{U}_i}$$

$$\eta\text{-Eq}\ \frac{\Gamma \vdash A : s \qquad \Gamma \vdash t, u : \Pi^{s,\mathcal{U}_i}(x : A).\, B : \mathcal{R}(s, \mathcal{U}_i) \qquad \Gamma, x : A : s \vdash t\ x \equiv u\ x : B : \mathcal{U}_i}{\Gamma \vdash t \equiv u : \Pi^{s,\mathcal{U}_i}(x : A).\, B : \mathcal{R}(s, \mathcal{U}_i)}$$

$$\textsc{Proof-Irr}\ \frac{\Gamma \vdash t : A : \Omega \qquad \Gamma \vdash u : A : \Omega}{\Gamma \vdash t \equiv u : A : \Omega} \qquad \textsc{Conv-Conv}\ \frac{\Gamma \vdash t \equiv u : A : \mathcal{U}_i \qquad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash t \equiv u : B : \mathcal{U}_i}$$

$$\beta\text{-conv}\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A : s \vdash B : \mathcal{U}_i \qquad \Gamma, x : A \vdash t : B : \mathcal{U}_i \qquad \Gamma \vdash u : A : s}{\Gamma \vdash (\lambda(x : A).\, t)\ u \equiv t[x := u] : B[x := u] : \mathcal{U}_i}$$

$$\textsc{Cast-}\Pi\ \frac{\begin{array}{c}\Gamma \vdash A : s \qquad \Gamma \vdash A' : s \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma, x : A' \vdash B' : s' \\ \Gamma \vdash e : \Pi(x : A).\, B \sim \Pi(x : A').\, B' : \Omega \qquad \Gamma \vdash f : \Pi(x : A).\, B \qquad a := \texttt{cast}\ A'\ A\ (\Pi^1_\epsilon\ e)\ a'\end{array}}{\Gamma \vdash \begin{array}{l}\texttt{cast}\ (\Pi(x : A).\, B)\ (\Pi(x : A').\, B')\ e\ f \equiv \\ \lambda(a' : A').\, \texttt{cast}\ (B[x := a])\ (B'[x := a'])\ (\Pi^2_\epsilon\ e\ a')\ (f\ a)\end{array} : \Pi(x : A').\, B' : \mathcal{R}(s, s')}$$

$$\textsc{Cast-Refl}\ \frac{\Gamma \vdash A \equiv B : s \qquad \Gamma \vdash e : A \sim_s B \qquad \Gamma \vdash t : A : s}{\Gamma \vdash \texttt{cast}\ A\ B\ e\ t \equiv t : B : s}$$

Fig. 3: $\mathsf{CIC}^{\mathrm{obs}}$ Conversion Rules (except congruence rules) *[Typed.agda]*

rule Proof-Irr reflects the computational irrelevance of the propositions: any two inhabitants of the same proposition are deemed convertible. Additionally, the conversion relation also includes the computation rules for the pattern-matching of inductive constructors that we will define in Section 5.

Then, we have the rules describing the behaviour of the `cast` operator on each type. The rule Cast-Π is standard; it says that a cast function evaluates to a function that casts its argument, applies the original function, and then casts back the result. Note that this rule needs the two projections Π^1_ϵ and Π^2_ϵ to get equality between the domains and the co-domains. Likewise, every declaration of an inductive type will add a handful of computation rules for the `cast` operator. Last but not least, the rule Cast-Refl is the main innovation of $\mathsf{CIC}^{\mathrm{obs}}$. It states that `cast` between convertible types can be simplified away, regardless of the proof of equality. This rule plays an important role in ensuring compatibility with the CIC: recall that `cast` can be used to derive a J eliminator for the observational equality—then rule Cast-Refl implies that this eliminator computes on reflexivity proofs, just like the usual eliminator of Martin-Löf's inductive equality.

4 Decidability of Conversion

In this section, we show that conversion is decidable in presence of the rule Cast-Refl for a simplified version of $\mathsf{CIC}^{\mathrm{obs}}$ in which the induction scheme is reduced to the type of natural numbers. Generally speaking, the main source

of difficulty for the decidability of conversion in dependent type theory is the transitivity rule—because of it, we have no guarantee that comparing two terms structurally is a complete strategy, since transitivity may be used with an arbitrary intermediate term at any point. If we want a decision procedure, we need to replace this transitivity rule with something more algorithmic.

Our aim is thus to define an equivalent presentation of the conversion for which transitivity is an admissible rule, but is not primitive. This is traditionally achieved by separating the conversion into a notion of weak-head reduction (Section 4.1) and a notion of conversion on neutral terms and weak-head normal forms (Section 4.2). In standard CIC, this strategy is sufficient to get *canonical* derivations of conversion, for which we have a decision procedure: we check the existence of a canonical derivation by first reducing terms to their weak-head normal form, and then comparing their head constructors and making recursive calls on their arguments. The point of this algorithmic definition of conversion is to replace the arbitrary transitivity rules with deterministic computations of weak-head normal forms. Then we can show that transitivity is admissible for conversion on neutral terms and weak-head normal forms. Naturally, this definition requires a proof of normalization of well-typed terms.

In the case of $\mathsf{CIC}^{\mathrm{obs}}$ however, the decision procedure for conversion of neutral terms and weak-head normal forms cannot be defined as a straightforward structural comparison. When the two terms start with `cast`, there are three rules that may apply: either congruence of `cast`, rule Cast-Refl on the left-hand side, or rule Cast-Refl on the right-hand side. This means that the decision procedure (Section 4.3) will have to do some backtracking to explore all possible combinations of congruence of `cast` and Rule Cast-Refl. Fortunately, the search space is bounded as every recursive call is done on a smaller argument.

Finally, to conclude on the decidability of conversion, we need to show that the declarative conversion is equivalent to our algorithmic conversion. For that, we use the logical relation setting of [2] to guarantee that every term can be put in weak-head normal form and that algorithmic conversion is complete with respect to conversion.

Note that our formalized version of $\mathsf{CIC}^{\mathrm{obs}}$ only supports the inductive type of natural numbers, and *not* the full scheme from Section 5. This is due to the setting of the formal proof, which requires the added inductive types to be explicit because Agda's check that the logical relation is well-defined makes use of the strict positivity criterion, which is syntactic and cannot be abstracted away for a generic definition. Nevertheless, we expect that our formal proof can be extended to specific inductive types such as lists or Martin-Löf's identity type, with methods similar to the ones from [3].

4.1 Reduction to Weak-Head Normal Forms

A notion that plays a central role in our normalization procedure is that of a *weak-head normal form* (whnf), which corresponds to a relevant term that cannot be reduced further (Fig. 4). Weak-head normal forms are either terms with a constructor in head position, or *neutral terms* stuck on a variable or

$$
\begin{array}{lll}
\text{whnf} & w & ::= N \mid \Pi(x : A).\, B \mid s \mid \mathbb{N} \mid \bot \mid t \sim_A u \mid \lambda(x : A).\, t \mid 0 \mid \mathtt{S}\ n \\
\text{neutral} & N & ::= x \mid N\ t \mid \bot\mathtt{-elim}\ A\ e \mid \mathbb{N}\mathtt{-elim}\ P\ t\ u\ N \\
& & \quad \mid \mathtt{cast}\ N\ B\ e\ t \mid \mathtt{cast}\ \mathbb{N}\ N\ e\ t \mid \mathtt{cast}\ \Pi^{s,s'}(x : A).\, B\ N\ e\ t \\
& & \quad \mid \mathtt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ N \mid \mathtt{cast}\ w\ w'\ e\ t \\
& & \qquad (\text{where } w, w' \in \{\mathbb{N}, \Pi^{s,s'}(x : A).\, B, s\},\ \mathrm{hd}\, w \neq \mathrm{hd}\, w')
\end{array}
$$

Fig. 4: Weak-head normal and neutral forms *[Untyped.agda]*

an elimination of a proof of $\bot$. In other words, neutral terms are weak-head normal forms that should not exist in an empty context. In $\mathsf{CIC}^{\mathrm{obs}}$, inhabitants of a proof-irrelevant type are never considered as whnf, as there is no notion of reduction of proof-irrelevant terms.

This notion of neutral terms is standard, but we need to pay a particular attention to neutral terms for `cast`. They correspond to all forms of cast for which there is no attached reduction rule. Because we assume that `cast` first evaluates its left type argument, then the second and finally its term argument, neutral terms of cast occur either when the first type is neutral, or when the first type is a type constructor and second type is neutral, or when the two types are the same type constructor, but the argument is neutral. Note that the reduction rule for casting a function always fires, so there is no associated neutral term in that case. Finally, casts between two different type constructors are always considered as stuck terms and should be seen as variant of $\bot\mathtt{-elim}\ A\ e$ because they correspond to casts based on an inconsistent proof of equality, thus similar to elimination of a proof of $\bot$.

At the heart of the decision procedure for conversion, there is a notion of typed reduction, noted $\Gamma \vdash t \Rightarrow u : A$. Intuitively, reduction corresponds to an orientation of the conversion rule in order to provide a rewrite system for which we can compute normal forms. However, not every conversion corresponds to a reduction rule: turning Rule Cast-Refl into a reduction rule would spawn several critical pairs, and even more annoyingly, its convertibility premise would force us to define reduction mutually with conversion checking. As are not aware of any framework that properly handles this type of circularity, we will sidestep the issue by deferring Cast-Refl to conversion checking, where we only have to deal with neutral terms and weak-head normal forms.

Actually, the purpose of reduction is to compute weak-head normal forms so that conversion rules that are not part of the reduction have only to be checked on weak-head normal forms. We do not detail the standard rules for CIC and focus on the one for `cast` (Fig. 5). The congruence rule for `cast` corresponds to several reduction rules, because we need to be careful to reduce one argument after the other in order, so that weak-head reduction remains deterministic. The reduction rules Cast-Zero, Cast-Suc and Cast-Univ correspond to the rule Cast-Refl where the arguments are instantiated by weak-head normal forms that are not neutral. Indeed, in that case `cast` must reduce. Conversion for `cast` when one of the scrutinees is neutral is deferred to algorithmic conversion.

$$\textsc{Cast-}\Pi\textsc{-red}\quad \frac{\Gamma \vdash A, A' : s \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma, x : A' \vdash B' : s' \qquad \Gamma \vdash e : \Pi(x : A).\, B \sim \Pi(x : A').\, B' : \Omega \qquad \Gamma \vdash f : \Pi(x : A).\, B \qquad a := \mathtt{cast}\ A'\ A\ (\Pi^1_\epsilon\ e)\ a'}{\Gamma \vdash \begin{array}{l}\mathtt{cast}\ (\Pi(x : A).\, B)\ (\Pi(x : A').\, B')\ e\ f \Rightarrow \\ \lambda(a' : A').\, \mathtt{cast}\ B[x := a]\ B'[x := a']\ (\Pi^2_\epsilon\ e\ a')\ f\ a\end{array} : \Pi(x : A').\, B'}$$

$$\textsc{Cast-Zero}\quad \frac{\Gamma \vdash e : \mathbb{N} \sim_{\mathcal{U}_0} \mathbb{N} : \Omega}{\Gamma \vdash \mathtt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ 0 \Rightarrow 0 : \mathbb{N}} \qquad \textsc{Cast-Suc}\quad \frac{\Gamma \vdash e : \mathbb{N} \sim_{\mathcal{U}_0} \mathbb{N} : \Omega \qquad \Gamma \vdash n : \mathbb{N} : \mathcal{U}_0}{\Gamma \vdash \mathtt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ (\mathtt{S}\ n) \Rightarrow \mathtt{S}\ (\mathtt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ n) : \mathbb{N}}$$

$$\textsc{Cast-Univ}\quad \frac{\Gamma \vdash e : s \sim_{s'} s \qquad \Gamma \vdash A : s \qquad \mathcal{A}(s, s')}{\Gamma \vdash \mathtt{cast}\ s\ s\ e\ A \Rightarrow A : s} \qquad \textsc{Conv-Red}\quad \frac{\Gamma \vdash t \Rightarrow u : A \qquad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash t \Rightarrow u : B}$$

$$\textsc{Cast-subst}\quad \frac{\Gamma \vdash A \Rightarrow A' : s \qquad \Gamma \vdash B : s \qquad \Gamma \vdash e : A \sim_s B : \Omega \qquad \Gamma \vdash t : A : s}{\Gamma \vdash \mathtt{cast}\ A\ B\ e\ t \Rightarrow \mathtt{cast}\ A'\ B\ e\ t : B}$$

$$\textsc{Cast-subst-nf}\quad \frac{\Gamma \vdash A : s \qquad \mathrm{whnf}\ A \qquad \Gamma \vdash B \Rightarrow B' : s \qquad \Gamma \vdash e : A \sim_s B : \Omega \qquad \Gamma \vdash t : A : s}{\Gamma \vdash \mathtt{cast}\ A\ B\ e\ t \Rightarrow \mathtt{cast}\ A\ B'\ e\ t : B}$$

$$\textsc{Cast-subst-nf-nf}\quad \frac{\Gamma \vdash A, B : s \qquad \mathrm{whnf}\ A \qquad \mathrm{whnf}\ B \qquad \Gamma \vdash e : A \sim_s B : \Omega \qquad \Gamma \vdash t \Rightarrow u : A}{\Gamma \vdash \mathtt{cast}\ A\ B\ e\ t \Rightarrow \mathtt{cast}\ A\ B\ e\ u : B}$$

Fig. 5: $\mathsf{CIC}^{\mathrm{obs}}$ Reduction Rules (rules for `cast`) *[Typed.agda]*

Note that because reduction is typed, we need to be able to change the type to any convertible one (Rule Conv-Red). Finally, we consider the reflexive transitive closure of reduction, noted $\Gamma \vdash t \Rightarrow^* u : A$.

4.2 Algorithmic Conversion

Algorithmic conversion (Fig. 6) is defined by comparing weak-normal forms and interleaving it with reduction. This way, an algorithmic conversion derivation can be seen as a canonical derivation of declarative conversion, where "transitive cuts" have been eliminated. It is called algorithmic, because it becomes directed by the shape of the terms, and the premises of each rule are on smaller terms. In CIC, it is even the case that at most one rule can be applied, so decidability of algorithmic conversion is pretty direct. In $\mathsf{CIC}^{\mathrm{obs}}$ however, decidability of algorithmic conversion is less direct because there are three rules that can be applied when the head is `cast` on both side. We come back to this difficulty in Section 4.3.

The judgment $\Gamma \vdash t \cong_{ne} u : A$ corresponds to a canonical conversion derivation between two neutral terms t and u at an arbitrary type A while the judgment $\Gamma \vdash t \cong u : A$ corresponds to a canonical derivation of conversion for terms in whnf when the type is also in whnf. This can be understood from a bidirectional perspective because comparison of neutral terms infers an arbitrary type, whereas for other weak-head normal forms, the inferred type is in weak-head normal form. Bidirectional typing [15,16] is traditionally used in type theory to

$$\textsc{Proof-irr}\ \frac{\Gamma \vdash t, u : A : \Omega}{\Gamma \vdash t \cong_{ne} u : A} \qquad \textsc{Var-refl}\ \frac{\Gamma \vdash x : A : \mathcal{U}_i}{\Gamma \vdash x \cong_{ne} x : A} \qquad \textsc{app-cong}\ \frac{\Gamma \vdash t \cong^{\downarrow}_{ne} u : \Pi^{s,\mathcal{U}_i}(x : A).\, B \qquad \Gamma \vdash a \cong^{\downarrow} b : A}{\Gamma \vdash t\ a \cong_{ne} u\ b : B[x := a]}$$

$$\textsc{Cast-cong}\ \frac{\begin{array}{c}\Gamma \vdash A \cong A' : s \qquad \Gamma \vdash B' \cong B : s \qquad \Gamma \vdash t \cong^{\downarrow} t' : A \qquad \Gamma \vdash e : A \sim_s B : \Omega \\ \Gamma \vdash e' : A' \sim_s B' : \Omega \qquad \text{neutral } (\texttt{cast}\ A\ B\ e\ t) \qquad \text{neutral } (\texttt{cast}\ A'\ B'\ e'\ t')\end{array}}{\Gamma \vdash \texttt{cast}\ A\ B\ e\ t \cong_{ne} \texttt{cast}\ A'\ B'\ e'\ t' : B}$$

$$\textsc{Cast-refl-L}\ \frac{\Gamma \vdash A \cong B : s \qquad \Gamma \vdash t \cong u : A \qquad \Gamma \vdash e : A \sim_s B : \Omega \qquad \text{neutral } (\texttt{cast}\ A\ B\ e\ t) \qquad \text{neutral } u}{\Gamma \vdash \texttt{cast}\ A\ B\ e\ t \cong_{ne} u : B}$$

$$\textsc{Cast-refl-R}\ \frac{\Gamma \vdash B \cong A : s \qquad \Gamma \vdash t \cong u : A \qquad \Gamma \vdash e : A \sim_s B : \Omega \qquad \text{neutral } t \qquad \text{neutral } (\texttt{cast}\ A\ B\ e\ u)}{\Gamma \vdash t \cong_{ne} \texttt{cast}\ A\ B\ e\ u : A}$$

$$\textsc{Ne-whnf}\ \frac{\Gamma \vdash t, u : A : \mathcal{U}_i \quad \text{whnf } A \quad \Gamma \vdash t \cong^{\downarrow}_{ne} u : A}{\Gamma \vdash t \cong u : A}$$

$$\textsc{Ne-Red}\ \frac{\Gamma \vdash A \Rightarrow^{*} B : \mathcal{U}_i \quad \text{whnf } B \quad \Gamma \vdash t \cong_{ne} u : A}{\Gamma \vdash t \cong^{\downarrow}_{ne} u : B}$$

$$\textsc{Whnf-Red}\ \frac{\begin{array}{c}\Gamma \vdash A \Rightarrow^{*} B : \mathcal{U}_i \qquad \Gamma \vdash t \Rightarrow^{*} t' : A \qquad \Gamma \vdash u \Rightarrow^{*} u' : A \\ \text{whnf } B, \text{whnf } t', \text{whnf } u' \qquad \Gamma \vdash t' \cong u' : B\end{array}}{\Gamma \vdash t \cong^{\downarrow} u : A}$$

Fig. 6: $\mathsf{CIC}^{\mathrm{obs}}$ Algorithmic Conversion Rules (except congruence rules) *[ConversionGen.agda]*

provide a canonical typing derivation by splitting the typing judgment into two: one judgment that infers the type of a term and an other one that checks that the inferred type of a term is convertible to the type given as input. This allows bidirectional typing to restrict the use of the conversion rule only to well-controlled places, and thus to provide only canonical derivations. In this presentation, it should be noticed that neutral terms infers an arbitrary terms (for instance, the application rule infers the type of the codomain of the function with an additional substitution) whereas other weak-head normal forms always infer a type also in weak-head normal form. But the structural rules for conversion correspond to a relational version of the type judgments, so that in some sense conversion subsumes typing. This means that we need to reflect this important distinction in the algorithmic conversion because the structural conversion rules for neutral terms ($\Gamma \vdash t \cong_{ne} u : A$) will naturally be performed at an arbitrary type A whereas $\Gamma \vdash t \cong u : A$ is always done at a type A in weak-head normal form.

Because conversion of whnf must contain conversion of neutrals as a particular case, we need those two notions to be compatible. To that end, we introduce two other judgments: $\Gamma \vdash t \cong^{\downarrow}_{ne} u : B$ means that $\Gamma \vdash t \cong_{ne} u : A$ and B is the

whnf of A (Rule Ne-Red) and $\Gamma \vdash t \cong^{\downarrow} u : A$ means that $\Gamma \vdash t' \cong^{\downarrow} u' : A'$ and t', u' and B are the whnf of t, u and A respectively (Rule Whnf-Red).

Let us now turn to the description of the relation $\Gamma \vdash t \cong u : A$ which mainly contains congruence rules for weak-head constructors, that are used in particular to show that reflexivity is admissible. Those congruence rules just ask for convertibility of each sub-argument, with some sanity conditions on the leaves, to ensure that only well-typed terms are considered in the conversion relation. Then, the rule Ne-whnf says that two neutral terms are comparable as whnf when they are comparable as neutral terms.

The relation $\Gamma \vdash t \cong_{ne} u : A$ contains a first rule to deal with proof-irrelevance in Ω (Rule Proof-irr). As any term in Ω is neutral, this rule only checks that the two terms are proofs of the same proposition. The rule for variables (Rule Var-refl) applies when there is the same variable x on the left and on the right, and this variable is declared in the local context Γ.

Then, there are four congruence rules to deal with eliminators. An eliminator is neutral when one of its scrutinees is neutral.The situation for `cast` is more complex as there are three different scrutinees (the two types and the term to be cast) and the whole term is neutral as soon as one of them is neutral. There is also a last kind of neutrals for `cast` which corresponds to impossible casts, that is casts between types with different head constructors. We can actually factorize all those cases and present only one rule (Cast-cong) that simply asks both casts to be neutral terms, at the price of a seemingly less accurate system. Indeed, because we are oblivious to the reason why the casts are neutral, all preconditions are asking for conversion as weak-head normal form instead of specializing in the case of neutral terms. However, by inversion on the rule, it is possible to show that two neutral terms are convertible as whnf if and only if they are convertible as neutral terms, so in the end this factorized rule is equivalent to a system with one rule per kind of neutral terms as defined in *[Conversion.agda]*.

To deal with Cast-Refl, we need to introduce two rules, one for simplification of cast on the left, and one on the right. This is because we have no rule for symmetry (to keep the system algorithmic) and symmetry must be an admissible rule. So the conversion rule is split into the two rules Cast-refl-L and Cast-refl-R. Again, we use a factorization to get only two rules, not specializing on the reason why a cast is neutral.

The main point of this algorithmic conversion is that it does not contain any rule for symmetry or transitivity. This is because they make it very difficult to prove decidability of conversion. However, we can show that symmetry (*[Symmetry.agda]*) and transitivity are admissible (*[Transitivity.agda]*).

4.3 Decidability of Algorithmic Conversion

We now turn to the definition of a decision procedure for the algorithmic conversion *[Decidable.agda]*. Actually, what we first prove is the decidability of algorithmic conversion for two terms t and u, assuming that we know that $\Gamma \vdash t \cong_{ne} t : A$ and $\Gamma \vdash u \cong_{ne} u : A$. The fact that algorithmic conversion is reflexive is actually a consequence of the completeness of algorithmic conversion with respect

to declarative conversion that will be shown in the next section. The hypothesis that t and u are in diagonal of the algorithmic conversion contains a lot of information, because by inversion on the derivations, we can actually recover the fact that t and u can be reduced to a whnf whose subterms can also be reduced in whnf, and this again and again up-to getting a deep normal form.

The decidability proof of conversion for MLTT in [2] coarsely amounts to zipping the two reflexivity proofs together, showing that when the two derivations do not share the exact same structure, then the two terms are not convertible. This is not the case anymore in presence of the rules Cast-refl-L and Cast-refl-R and the reasoning cannot stay on the "diagonal" of the algorithmic conversion. This is not an issue as actually from the fact that $\Gamma \vdash t \cong_{ne} t' : A$, we can deduce that both t and t' can be put in deep normal form and so somehow, $\Gamma \vdash t \cong_{ne} t' : A$ can be used as termination witness in the same way as $\Gamma \vdash t \cong_{ne} t : A$.

However, the main difficulty in this new setting is that it is not true anymore that when the two derivations do not share the exact same structure, then the two terms are not convertible. Consider for instance $\texttt{cast}\ A\ B\ e\ t$ against t: the reflexivity proofs for these two terms cannot share the same structure, yet they are convertible by Rule Cast-refl-L. In addition, in the more complex case of $\texttt{cast}\ A\ B\ e\ t$ against $\texttt{cast}\ A'\ B'\ e'\ t'$, there are three cases to consider, because the last rule to show that they are convertible can be either Cast-cong, Cast-refl-L or Cast-refl-R. This means in particular that the proof that two terms are algorithmically convertible is not unique anymore, and the decidability procedure has to do an arbitrary choice, depending on which order it tests the three different possibility and backtracks.

The statement of decidability needs to be generalized in the following way.

Theorem 1 (Decidability of algorithmic conversion *[Decidable.agda]*). *For any natural number n, given two proofs of neutral comparison $\pi : \Gamma \vdash t \cong_{ne} t' : A$ and $\pi' : \Delta \vdash u \cong_{ne} u' : B$ such that $\vdash \Gamma \equiv \Delta$ and $\texttt{size}(\pi) + \texttt{size}(\pi') < n$, knowing whether there exists a type C such that $\Gamma \vdash t \cong_{ne} u : C$ is decidable.*

Note that the statement is based on a notion of size of a derivation, noted `size`, because the algorithm does recursive calls that are not structurally decreasing. To conclude on the completeness of algorithmic conversion *[Completeness.agda]*, we reuse the logical relation setting described in [2] for proving strong normalization and decidability of conversion in various type theories, later extended in [22,23]. We do not detail the definition of the logical relation here as there is not specific to our system, what is important is it provides the following consequence.

5 Inductive Definitions

On top of the rules from Section 3, $\mathsf{CIC}^{\mathrm{obs}}$ includes a scheme to define proof-relevant inductive types that is based on the scheme of CIC (as defined in [26]). Inductive definitions are not manipulated as first class objects: instead, the user declares all the necessary inductive types using a standard syntax, before starting

their proof. After each declaration, the theory is automatically extended with the new type former, inductive constructors, etc.

The syntax for the inductive scheme of $\mathsf{CIC}^{\mathrm{obs}}$ is exactly the same as the scheme of CIC; the difference lies in the fact that inductive definitions will additionally have to generate projections for the observational equality types and computation rules for the `cast` operator. We start by explaining how it works for inductive types without indices, and then we extend it to general indexed inductive definitions by using the Fordism transformation and some syntactic sugar. We will spare the reader the added complexity of mutually defined families, which is mathematically direct but heavy on notation.

5.1 Inductive Definitions Without Indices

We use a syntax based on the one used by the Coq proof assistant for inductive definitions. The general form of a non-indexed type looks like this:

$$\begin{array}{l}\texttt{Inductive Ind } (\vec{a} : \vec{A}) : \mathcal{U}_\ell := \\ \mid c_0 : \forall\ (\vec{b} : \vec{B_0}),\ \texttt{Ind } \vec{a} \\ \mid \ldots \\ \mid c_n : \forall\ (\vec{b} : \vec{B_n}),\ \texttt{Ind } \vec{a}\end{array}$$

In order to represent arbitrary contexts of parameters more compactly, we used a vector notation. The parameter $(\vec{a} : \vec{A})$ represents a context of the form $a_1 : A_1, ..., a_m : A_m$ where each type may depend on the previous ones. Similarly, every constructor of the inductive type has a context of arguments, that may include recursive calls to `Ind` in *strictly positive positions*—however we will not be paying special attention to recursive calls, as their treatment is not affected by the observational equality. The universe $\mathcal{U}_\ell$ must be larger than all the types that appear in the constructor arguments $\vec{B_i}$. Inductive definitions in the sort of propositions $\mathbf{\Omega}$ are not allowed.

After the user makes such a definition, the system is extended with the new type former `Ind` and the inductive constructors c_0, ... c_n with their prescribed types. Additionally, $\mathsf{CIC}^{\mathrm{obs}}$ provides two operators `match` and `fix` that are used to define functions out of an inductive definition, following the typing and computation rules described by the [11]. As we explain in Section 2, this elimination principle is enough to completely determine the observational equality between any two inhabitants of `Ind`, thus our system does not provide any additional rule for this. However, the observational equality between two instances of `Ind` does not benefit from any such principle, so we add "projection" operators to characterize equalities between inductive types:

$$\texttt{eq\_}c_i : \forall\ (\vec{a} : \vec{A})\ (\vec{a}' : \vec{A}),\ \texttt{Ind } \vec{a} \sim \texttt{Ind } \vec{a}' \to \vec{B_i}[\vec{a}] \sim \vec{B_i}[\vec{a}'] \qquad (\forall\ \texttt{i})$$

The projections $\texttt{eq\_}c_i$ are generated when the user makes the definition of `Ind`, just like the constructors c_i. Remark that the codomains of these projections are equalities between two vectors, which is a notational shorthand for a vector of equalities. In practice, this means that each $\texttt{eq\_}c_i$ will be implemented as a family of projections $(\texttt{eq\_}c_{i,j})$, where each projection depends on the previous

ones. Thus, we get as many projections as there are constructor arguments in the inductive definition. Finally, we add computation rules for `cast`:

$$\texttt{cast}\ (\texttt{Ind}\ \vec{a})\ (\texttt{Ind}\ \vec{a'})\ \texttt{e}\ (c_i\ \vec{b})\ \equiv\ c_i\ (\texttt{cast}\ (\vec{B_i}[\vec{a}])\ (\vec{B_i}[\vec{a'}])\ (\texttt{eq\_}c_i\ \vec{a}\ \vec{a'}\ \texttt{e})\ \vec{b})\quad (\forall\ \texttt{i})$$

5.2 Deriving a Scheme for Indexed Inductive Types

In order for $\mathsf{CIC}^{\mathrm{obs}}$ to be a proper extension of CIC, we need to extend our scheme to indexed inductive definitions. These get a bit messier than non-indexed definitions, but in fact we already have all the pieces we need: as we saw in Section 2.2, the rule Cast-Refl allows us to use the Fordism transformation and *faithfully* encode indexed inductive types with parametrized inductive types. Consequently, we will define the scheme for indexed definitions in terms of the scheme for non-indexed definitions, using syntactic sugar and elaboration. That way, the typing and computation rules of CIC that involve indexed inductive types remain valid in $\mathsf{CIC}^{\mathrm{obs}}$, but the inductive types and constructors are elaborated to their non-indexed counterpart under the hood.

We now explain in detail how this elaboration process works. When the user defines an indexed inductive type `Ind`, they are actually defining the forded version $\texttt{Ind}_{\mathbb{F}}$ *via* the scheme for non-indexed definitions:

$$\begin{array}{l}\texttt{Inductive Ind}\ (\vec{a} : \vec{A}) : \forall\ (\vec{x} : \vec{X}),\ \mathcal{U}_\ell :=\\ \mid c_0 : \forall\ (\vec{b} : \vec{B_0}),\ \texttt{Ind}\ \vec{a}\ \vec{y_0}\\ \mid \ldots\\ \mid c_n : \forall\ (\vec{b} : \vec{B_n}),\ \texttt{Ind}\ \vec{a}\ \vec{y_n}\end{array}\qquad \begin{array}{l}\texttt{Inductive Ind}_{\mathbb{F}}\ (\vec{a} : \vec{A})\ (\vec{x} : \vec{X}) : \mathcal{U}_\ell :=\\ \mid c_{0\mathbb{F}} : \forall\ (\vec{b} : \vec{B_0}),\ \vec{y_0} \sim \vec{x} \to\ \texttt{Ind}_{\mathbb{F}}\ \vec{a}\ \vec{x}\\ \mid \ldots\\ \mid c_{n\mathbb{F}} : \forall\ (\vec{b} : \vec{B_n}),\ \vec{y_n} \sim \vec{x} \to\ \texttt{Ind}_{\mathbb{F}}\ \vec{a}\ \vec{x}\end{array}$$

The scheme generates projections for observational equalities between the constructor arguments, *including* the index equalities $\vec{y_i} \sim \vec{x}$ that are hidden in the user definition. Then, our system defines `Ind` and its constructors in terms of their forded counterparts:

$$\texttt{Ind}\ \vec{a}\ \vec{x}\ \equiv\ \texttt{Ind}_{\mathbb{F}}\ \vec{a}\ \vec{x} \qquad\qquad c_i\ \vec{b}\ \equiv c_{i\mathbb{F}}\ \vec{b}\ \texttt{refl}$$

The pattern matching on inhabitants of the indexed inductive type is elaborated to a pattern matching on the forded version, by inserting a `cast` in each branch. Concretely, consider the following pattern matching on $\texttt{i} : \texttt{Ind}\ \vec{a}\ \vec{x}$:

$$\texttt{match i return P with} \mid c_0\ \vec{b} \Rightarrow t_0 \quad \mid \ldots. \mid c_n\ \vec{b} \Rightarrow t_n\ \texttt{end}$$

The return type is $\texttt{P}\ \vec{x}\ \texttt{i}$, and thus in the branch for $c_i\ \vec{b}$, the term t_i provided by the user has type $\texttt{P}\ \vec{y_i}\ (c_i\ \vec{b})$. After the elaboration, this branch matches a forded pattern $c_{i\mathbb{F}}\ \vec{b}\ e$, and should now return a result of type $\texttt{P}\ \vec{x_i}\ (c_{i\mathbb{F}}\ \vec{b}\ e)$. We can obtain this result by type-casting the user-supplied term t_i along the equality proof e to obtain

$$\texttt{cast}\ (\texttt{P}\ \vec{y_i}\ (c_i\ \vec{b}))\ (\texttt{P}\ \vec{x_i}\ (c_{i\mathbb{F}}\ \vec{b}\ e))\ (\texttt{ap2}\ \texttt{P}\ (c_{i\mathbb{F}}\ \vec{b}\ e))\ t_i$$

where `ap2` is a slight generalization of the proof that function applications preserve equalities. Thanks to the rule Cast-Refl, this elaboration preserves the computation rule of the pattern-matching for indexed inductive types. Note that

there is nothing special to do for fixpoints, they work out of the box. This concludes the description of our formal system $\mathsf{CIC}^{\mathrm{obs}}$.

6 Consistency of the Theory

In Section 2 we saw that combining the inductive scheme of CIC with the observational equality can endanger the consistency of the theory if we are not careful. In the end, it is possible to fix the issue by picking a better definition for the observational equality of inductive types, but now we want to make sure that this new definition will not lead to another inconsistency. To do this, we build a model of $\mathsf{CIC}^{\mathrm{obs}}$ in set theory, thereby reducing the consistency of our system to the consistency of ZFC set theory with Grothendieck universes. Our model is mostly an extension of the one that was presented in [23] to general inductive definitions, using the interpretation of inductive definitions that was developed in [26].

6.1 Observational Type Theory in Sets

We work in ZFC set theory with a countable hierarchy of Grothendieck universes $\mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_2$, *etc.* We write $\Omega := \{\bot, \top\}$ for the lattice of truth values, and given $p \in \Omega$ we write $\mathsf{val}\ p$ for the associated set $\{x \in \{*\} \mid p\}$. Since our goal is to interpret a dependent type theory, we will need set-theoretic dependent products and dependent sums. We write the former as $(a \in A) \to (B\ a)$, and the latter as $(a \in A) \times (B\ a)$ to distinguish them from their type-theoretic counterparts.

Our model will be based on the *types-as-sets* interpretation of dependent type theory [12], according to which contexts are interpreted as sets, types and terms over a context Γ become sets indexed over the interpretation of Γ, the typing relation corresponds to set membership, and conversion is interpreted as the set-theoretic equality. Such models have already been defined for a wide variety of type theories; of particular interest to us is the model of [26] which supports an impredicative sort of propositions (interpreted as the lattice of truth values) and the full scheme of inductive definitions of CIC. Since ZFC set theory is extensional by nature, this model also validates the principles of function extensionality and proposition extensionality, which would *almost* make it a model of $\mathsf{CIC}^{\mathrm{obs}}$, were it not for two small issues.

The first issue is the absence of the observational equality and the `cast` operator in the model of [26]. We can easily fix this by interpreting the observational equality as the set-theoretic equality, and `cast` as the identity function. That way, `cast` verifies all the desired equations for trivial reasons, including the rule Cast-Refl. After all, the model does not differentiate between conversion and propositional equality!

The second issue is a bit more serious, and deals with the universes. In [26], the authors directly interpret the type-theoretic universes as the corresponding Grothendieck universes, which is perfectly fine for CIC. But this does not work for $\mathsf{CIC}^{\mathrm{obs}}$, as we would lose the injectivity of dependent products: consider for

$$
\begin{array}{ll}
[\![\, \Gamma \vdash \mathcal{U}_j \,]\!]_\rho & := \langle\, \mathbf{V}_j \times \mathbf{V}_j \,,\, \emptyset \,\rangle \\
[\![\, \Gamma \vdash \mathbf{\Omega} \,]\!]_\rho & := \langle\, \mathbf{\Omega} \,,\, \emptyset \,\rangle \\
[\![\, \Gamma \vdash \Pi^{\mathcal{U}_j,\mathcal{U}_k}(x : A).\, B \,]\!]_\rho & := \langle\, (x \in \mathsf{fst}\ [\![\, \Gamma \vdash A \,]\!]_\rho) \to \mathsf{fst}\ [\![\, \Gamma, A \vdash B \,]\!]_{\rho,x} \,, \\
 & \qquad ([\![\, \Gamma \vdash A \,]\!]_\rho \ ,\ \lambda x \ .\ [\![\, \Gamma, A \vdash B \,]\!]_{\rho,x}) \,\rangle \\
[\![\, \Gamma \vdash \Pi^{\Omega,\mathcal{U}_j}(x : A).\, B \,]\!]_\rho & := \langle\, (x \in \mathsf{val}\ [\![\, \Gamma \vdash A \,]\!]_\rho) \to \mathsf{fst}\ [\![\, \Gamma, A \vdash B \,]\!]_{\rho,x} \,, \\
 & \qquad (\mathsf{val}\ [\![\, \Gamma \vdash A \,]\!]_\rho \ ,\ \lambda x \ .\ [\![\, \Gamma, A \vdash B \,]\!]_{\rho,x}) \,\rangle \\
[\![\, \Gamma \vdash \texttt{Ind}\ \vec{X} \,]\!]_\rho & := \langle\, \mathsf{IndElem}\ [\![\, \Gamma \vdash \vec{X} \,]\!]_\rho \,,\, \mathsf{IndLabel}\ [\![\, \Gamma \vdash \vec{X} \,]\!]_\rho \,\rangle
\end{array}
$$

Fig. 7: Codes for universes, dependent products and inductive types

instance the two types $\texttt{Empty} \to \mathbb{N}$ and $\texttt{Empty} \to \mathbb{B}$. Both are interpreted as a singleton set, but in $\mathsf{CIC}^{\mathrm{obs}}$ we can prove that they cannot be equal. To recover this injectivity, we *label* the sets in the universe with additional information that indicates how they were built. This way, the type $\texttt{Empty} \to \mathbb{N}$ is interpreted as a singleton set *and* an indication that it is a function type from $\texttt{Empty}$ to $\mathbb{N}$, while $\texttt{Empty} \to \mathbb{B}$ has a different label.

6.2 Coinductive Labels for Inductive Types

In this section, we give a proper definition for our labelled universe. The technique of using labels to build a universe that is generic for sets *and* ensures the injectivity of dependent products is a re-reading of the technique of [13]. However, his construction seems difficult to extend with parametrized inductive types—the use of induction-recursion seems to force us to have the injectivity of parameters, which we do not want (cf Section 2.3). Therefore we ditch induction-recursion for a definition that is somewhat more set-theoretic: our interpretation of the universe $\mathcal{U}_i$ is simply $\mathbf{V}_i \times \mathbf{V}_i$, meaning that a code in the universe is a pair of sets. The first set of the pair is the (semantic) type, and the second set is the label. The "El" function that transforms a code into a type is thus simply the first projection.

Fig. 7 shows the interpretation for the proof-relevant type formers of $\mathsf{CIC}^{\mathrm{obs}}$. The interpretation function that transforms a syntactic object into a semantic object is written $[\![\, \Gamma \vdash \_ \,]\!]_\rho$, where ρ is a set-theoretic function that assigns a set to every variable of the context Γ. Unsurprisingly, the syntactic universes $\mathcal{U}_i$ and $\mathbf{\Omega}$ are interpreted as their semantic counterparts, with the default label (the empty set). Dependent products also are interpreted as their set-theoretic counterparts, but in that case the label contains the domain and the codomain, ensuring that two dependent products are not identified unless their domain and codomain are themselves equal.

The case of the inductive definitions is a bit more involved. Thankfully we do not need to treat indices, as they are encoded using Fordism (cf Section 5.2). Thus, we consider a non-indexed inductive definition $\texttt{Ind}$ as in Section 5, with a vector of parameters $\vec{A}$:

$\texttt{Inductive Ind}\ (\vec{a} : \vec{A}) : \mathcal{U}_\ell :=$
$\mid c_0 : \forall\ (\vec{b} : \vec{B_0}),\ \texttt{Ind}\ \vec{a}$
$\mid \ldots$
$\mid c_n : \forall\ (\vec{b} : \vec{B_n}),\ \texttt{Ind}\ \vec{a}$

Given any vector $\vec{X}$ of elements of the family of sets $\mathsf{fst}(\llbracket\ \vec{A}\ \rrbracket_\rho)$, we define $\mathsf{IndElem}\ \vec{X}$ to be the set constructed in [26], which is well-defined if the definition of `Ind` is strictly positive and all the the interpretations of the $\vec{B_i}$ are well-defined. Reproducing their construction in full detail would take us too far from the scope of this paper, so we will simply mention that it is the initial algebra of the set-theoretic endofunctor corresponding to `Ind` evaluated in $\vec{X}$. This gives us the first projection of $\llbracket\ \Gamma \vdash \texttt{Ind}\ \vec{X}\ \rrbracket_\rho$, and now we need to define the second projection $\mathsf{IndLabel}\ \vec{X}$. Recall from Section 2.3 that we would like the equality of two instances of `Ind` to satisfy:

$$\texttt{Ind}\ \vec{X} \sim \texttt{Ind}\ \vec{Y} \quad \longleftrightarrow \quad (\vec{B_0}(\vec{X}), ..., \vec{B_n}(\vec{X})) \sim (\vec{B_0}(\vec{Y}), ..., \vec{B_n}(\vec{Y})).$$

In other words, `Ind` should be determined up to equality by the types of its constructor arguments. Therefore, it is natural to define its label directly as the list of these types:

$$\mathsf{IndLabel}\ \vec{X} \quad = \quad (\llbracket \Gamma, \vec{A} \vdash \vec{B_0} \rrbracket_{(\rho, \vec{X})}, ..., \llbracket \Gamma, \vec{A} \vdash \vec{B_n} \rrbracket_{(\rho, \vec{X})}).$$

However, remark that $\vec{B_i}$ may contain a recursive call to `Ind`, whose interpretation is defined using $\mathsf{IndLabel}$, so this definition is really an equation that we need to solve. Fortunately, a simple look at the shape of that equation reveals that it is in fact a definition for an infinite tree whose nodes are labeled with sets, which we take as our solution. Note that the result is indeed an inhabitant of $\mathbf{V}_\ell$, since the sets that intervene in its construction (the interpretation of the types of the constructor arguments and their labels) are all in $\mathbf{V}_\ell$. With this definition of $\mathsf{IndLabel}$, we get the following property:

Lemma 1. *If the inductive definition* `Ind` *is strictly positive,* $\llbracket \Gamma \vdash \vec{X} \rrbracket_\rho$ *is well-defined, and all the* $\llbracket \Gamma, \vec{A} \vdash \vec{B_i} \rrbracket_{(\rho, \vec{X})}$ *are well-defined, then* $\llbracket \Gamma \vdash \texttt{Ind}\ \vec{X} \rrbracket_\rho$ *is well-defined. Furthermore,* $\llbracket \Gamma \vdash \texttt{Ind}\ \vec{X} \rrbracket_\rho = \llbracket \Gamma \vdash \texttt{Ind}\ \vec{Y} \rrbracket_\rho$ *is equivalent to*

$$\forall i, \quad \llbracket \Gamma, \vec{A} \vdash \vec{B_i} \rrbracket_{(\rho, \llbracket \Gamma \vdash \vec{X} \rrbracket_\rho)} = \llbracket \Gamma, \vec{A} \vdash \vec{B_i} \rrbracket_{(\rho, \llbracket \Gamma \vdash \vec{Y} \rrbracket_\rho)}.$$

6.3 Soundness of the Model

The definition of the observational universe is the only new insight of our construction; the rest follows the strategy laid out in [26]. For the sake of completeness, we give an outline of the definition and of the proof of soundness in this section.

Ultimately, our model is defined in terms of *partial* functions from the syntax to the semantics. We use a function $\llbracket \_ \rrbracket$ that interprets contexts as sets and a function $\llbracket \Gamma \vdash \_ \rrbracket_\rho$ that interprets terms and types in context Γ as sets indexed by $\rho \in \llbracket \Gamma \rrbracket$ (Fig. 8). Both functions are mutually defined by recursion on the

$$
\begin{aligned}
[\![\bullet]\!] &:= \{\emptyset\} \\
[\![\Gamma, x : A : \mathcal{U}_i]\!] &:= \{(\rho, a) \mid \rho \in [\![\Gamma]\!] \ \wedge \ a \in \mathsf{fst}\ [\![\Gamma \vdash A]\!]_\rho\} \\
[\![\Gamma, x : A : \Omega]\!] &:= \{(\rho, a) \mid \rho \in [\![\Gamma]\!] \ \wedge \ a \in \mathsf{val}\ [\![\Gamma \vdash A]\!]_\rho\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash x]\!]_\rho &:= \rho(x) \\
[\![\Gamma \vdash \lambda(x : F).\, t]\!]_\rho &:= (x \in \mathsf{fst}\ [\![\Gamma \vdash F]\!]_\rho) \mapsto ([\![\Gamma, F \vdash t]\!]_{\rho,x}) \\
[\![\Gamma \vdash t\ u]\!]_\rho &:= [\![\Gamma \vdash t]\!]_\rho([\![\Gamma \vdash u]\!]_\rho)
\end{aligned}
$$

$$
\left.
\begin{aligned}
[\![\Gamma \vdash c_i\ \vec{b}]\!]_\rho &:= \\
[\![\Gamma \vdash \texttt{match t return P with } \{c_i\ \vec{b} \Rightarrow t_i\}]\!]_\rho &:= \\
[\![\Gamma \vdash \texttt{fix}\ f\ \vec{x} := t]\!]_\rho &:=
\end{aligned}
\right\} \text{(as in Lee } \textit{et al.}\text{)}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \bot]\!]_\rho &:= \bot \\
[\![\Gamma \vdash \bot\texttt{-elim}\ A\ t]\!]_\rho &:= \text{undefined} \\
[\![\Gamma \vdash t \sim_A u]\!]_\rho &:= \top \text{ if } [\![\Gamma \vdash t]\!]_\rho = [\![\Gamma \vdash u]\!]_\rho \\
&\quad\ \bot \text{ otherwise} \\
[\![\Gamma \vdash \texttt{cast}\ A\ B\ e\ t]\!]_\rho &:= [\![\Gamma \vdash t]\!]_\rho
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash \Pi^{\mathcal{U}_j,\Omega}(y : A).\, B]\!]_\rho &:= \forall x \in (\mathsf{fst}\ [\![\Gamma \vdash A]\!]_\rho),\ [\![\Gamma, A \vdash B]\!]_{\rho,x} \\
[\![\Gamma \vdash \Pi^{\Omega,\Omega}(y : A).\, B]\!]_\rho &:= \forall x \in (\mathsf{val}\ [\![\Gamma \vdash A]\!]_\rho),\ [\![\Gamma, A \vdash B]\!]_{\rho,x}
\end{aligned}
$$

Fig. 8: Interpretation of contexts and proof-relevant terms of $\mathsf{CIC}^{\mathrm{obs}}$

raw syntax, and we will then prove that they are total on well-typed terms by induction on the typing derivations. Variables, lambda-abstractions and applications are interpreted respectively as projections from the context, set-theoretic functions and applications. In order to interpret the inductive constructors and the `match` and `fix` operators, we need to develop a proper theory of set-theoretic induction. Since this part is completely orthogonal to the observational primitives, we deem it out of the scope of this work and we refer the interested reader to the literature instead. In [26], the authors use induction principles insted of `match` and `fix`, but argue that the two are equivalent. A model directly based on the latter can be found in [14]. The $\bot$ proposition is interpreted as the false proposition of ZFC, the observational equality as the equality of ZFC, and the cast operator as the identity function. Finally, the proof-irrelevant dependent products are interpreted as set-theoretic quantifications. The proofs of propositions such as `transport` or $\mathbf{\Pi}^1_\epsilon$ do not need to be interpreted—after all, the model is proof-irrelevant.

In order to prove the soundness of our interpretation, we need to extend it to weakenings and substitutions between contexts. Assume Γ and Δ are syntactical contexts, and A and t are syntactical terms. In case $[\![\Gamma, x : A : s, \Delta]\!]$ and $[\![\Gamma, \Delta]\!]$ are well-defined, let π_A be the projection:

$$\pi_A : [\![\Gamma, x : A : s, \Delta]\!] \to [\![\Gamma, \Delta]\!] \qquad (\vec{x_\Gamma}, x_A, \vec{x_\Delta}) \mapsto (\vec{x_\Gamma}, \vec{x_\Delta}).$$

In case $[\![\Gamma, \Delta[x := t]]\!]$ and $[\![\Gamma, x : A : s, \Delta]\!]$ are well-defined, we define the function σ_t by:

$$\sigma_t : [\![\Gamma, \Delta[x := t]]\!] \to [\![\Gamma, x : A : s, \Delta]\!] \qquad (\vec{x_\Gamma}, \vec{x_\Delta}) \mapsto (\vec{x_\Gamma}, [\![\Gamma \vdash t]\!]_{\vec{x_\Gamma}}, \vec{x_\Delta}).$$

Theorem 2 (Soundness of the Standard Model).

1. *If* $\vdash \Gamma$ *then* $[\![\Gamma]\!]$ *is defined.*
2. *If* $\Gamma \vdash A : \Omega$ *then* $[\![\Gamma \vdash A]\!]_\rho$ *is a semantic proposition for all* $\rho \in [\![\Gamma]\!]$.
3. *If* $\Gamma \vdash A : \mathcal{U}_i$ *then* $[\![\Gamma \vdash A]\!]_\rho$ *is in* $\mathbf{V}_i$ *for all* $\rho \in [\![\Gamma]\!]$.
4. *If* $\Gamma \vdash t : A : \Omega$ *then* $[\![\Gamma \vdash t]\!]_\rho \in \mathsf{val}([\![\Gamma \vdash A]\!]_\rho)$ *for all* $\rho \in [\![\Gamma]\!]$.
5. *If* $\Gamma \vdash t : A : \mathcal{U}_i$ *then* $[\![\Gamma \vdash t]\!]_\rho \in \mathsf{fst}([\![\Gamma \vdash A]\!]_\rho)$ *for all* $\rho \in [\![\Gamma]\!]$.
6. *If* $\Gamma \vdash t \equiv u : A$ *then* $[\![\Gamma \vdash t]\!]_\rho = [\![\Gamma \vdash u]\!]_\rho$ *for all* $\rho \in [\![\Gamma]\!]$.

Since our model interprets the false proposition $\bot$ as the empty set, we get a proof of consistency:

Theorem 3 (Consistency). *There are no proofs of* $\bot$ *in the empty context.*

Furthermore, by inspecting the normal forms provided by the normalization theorem, we note that the only neutral terms in the empty context are stuck casts. But having a stuck `cast` requires an equality proof between two incompatible types, which cannot exist from our definition of the universe. From there, we derive a canonicity theorem for inductive types: all elements of an inductive type without indices reduce to canonical elements in the empty context.

7 Conclusion and Future Work

We proposed a systematic integration of indexed inductive types with an observational equality, by defining a notion of observational equality that satisfies the computational rule of Martin-Löf's identity type and by using Fordism, a general technique to faithfully encode indexed inductive types with non-indexed types and equality. We developed a formal proof that this additional computation rule, although not present in previous works on observational equality, can be integrated to the system without compromising the decidability of conversion. This extension of CIC with an observational equality has been implemented at the top of the Coq proof assistant by using the recently introduced rewrite rules.

Although the technique has been developed in the setting of CIC and Coq specifically, there is no obstacle to adapt it to other settings such as Lean or Agda. Adaption to Lean should be pretty straightforward as it is sharing most of its metatheory with Coq. A partial version of $\mathsf{CIC}^{\mathrm{obs}}$ could be provided in Agda with rewrite rules. However, the management of elimination of inductive types in Agda is not done using an explicit pattern-matching syntax à la Coq, for which we can define new reduction rules. Instead, functions on inductive types are defined using case splitting trees and an exhaustivity checker. Therefore, a proper treatment of $\mathsf{CIC}^{\mathrm{obs}}$ in Agda would require modifications of the case splitting engine, similarly to what has been done for Cubical Agda [27].

8 Data-Availability Statement

The Agda companion formalization is available both on GitHub and as a long-term archived artifact [24].

References

1. Abel, A., Coquand, T.: Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality. Logical Methods in Computer Science **Volume 16, Issue 2** (2020). https://doi.org/10.23638/LMCS-16(2:14)2020. URL https://lmcs.episciences.org/6606
2. Abel, A., Öhman, J., Vezzosi, A.: Decidability of conversion for type theory in type theory. Proceedings of the ACM on Programming Languages **2**(POPL), 23:1–23:29 (2018). https://doi.org/10.1145/3158111. URL http://doi.acm.org/10.1145/3158111
3. Adjedj, A., Lennon-Bertrand, M., Maillard, K., Pédrot, P.M., Pujet, L.: Martin-löf à la coq (2023)
4. Allais, G., McBride, C., Boutillier, P.: New equations for neutral terms: A sound and complete decision procedure, formalized. In: Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming, DTP '13, pp. 13–24. ACM, New York, NY, USA (2013). https://doi.org/10.1145/2502409.2502411. URL http://doi.acm.org/10.1145/2502409.2502411
5. Altenkirch, T., McBride, C.: Towards Observational Type Theory (2006). URL http://www.strictlypositive.org/ott.pdf
6. Altenkirch, T., McBride, C., Swierstra, W.: Observational equality, now! In: Proceedings of the Workshop on Programming Languages meets Program Verification (PLPV 2007), pp. 57–68 (2007). https://doi.org/10.1145/1292597.1292608
7. Atkey, B.: Simplified Observational Type Theory (2017). URL https://github.com/bobatkey/sott
8. Barendregt, H.P.: Lambda calculi with types. In: Handbook of logic in computer science (vol. 2) background: computational structures, pp. 117–309 (1993)
9. Berg, B.v.d., Garner, R.: Types are weak ω-groupoids. Proceedings of the London Mathematical Society **102** (2008)
10. Cavallo, E., Harper, R.: Higher inductive types in cubical computational type theory. Proc. ACM Program. Lang. **3**(POPL) (2019). https://doi.org/10.1145/3290314. URL https://doi.org/10.1145/3290314
11. Coq Development Team, T.: The Coq proof assistant reference manual (2016). URL http://coq.inria.fr. Version 8.6
12. Dybjer, P.: Inductive Sets and Families in Martin-Löf's Type Theory and Their Set-Theoretic Semantics, p. 280–306. Cambridge University Press, USA (1991)
13. Gratzer, D.: An inductive-recursive universe generic for small families (2022). https://doi.org/10.48550/ARXIV.2202.05529. URL https://arxiv.org/abs/2202.05529
14. Lee, G., Werner, B.: Proof-irrelevant model of CC with predicative induction and judgmental equality. Logical Methods in Computer Science **Volume 7, Issue 4** (2011). https://doi.org/10.2168/lmcs-7(4:5)2011
15. Lennon-Bertrand, M.: Complete Bidirectional Typing for the Calculus of Inductive Constructions. In: L. Cohen, C. Kaliszyk (eds.) 12th International Conference on Interactive Theorem Proving (ITP 2021), *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 193. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.24. URL https://drops.dagstuhl.de/opus/volltexte/2021/13919
16. Lennon-Bertrand, M.: Bidirectional Typing for the Calculus of Inductive Constructions. Theses, Nantes Université (2022). URL https://theses.hal.science/tel-03848595

17. Martin-Löf, P.: An intuitionistic theory of types: Predicative part. In: H. Rose, J. Shepherdson (eds.) Logic Colloquium '73, *Studies in Logic and the Foundations of Mathematics*, vol. 80, pp. 73 – 118. Elsevier (1975). https://doi.org/https://doi.org/10.1016/S0049-237X(08)71945-1. URL http://www.sciencedirect.com/science/article/pii/S0049237X08719451
18. McBride, C.: Dependently typed functional programs and their proofs. Ph.D. thesis, University of Edinburgh (2000)
19. McBride, C.: Hier soir, an ott hierarchy. Blog post (2011). URL https://mazzo.li/epilogue/index.html%3Fp=1098.html
20. Miquel, A.: Le calcul des constructions implicites. Ph.D. thesis, Université Paris Diderot (2001). URL https://github.com/coq-contribs/paradoxes/blob/master/Russell.v
21. Paulin-Mohring, C.: Inductive definitions in the system coq rules and properties. In: M. Bezem, J.F. Groote (eds.) Typed Lambda Calculi and Applications, pp. 328–345. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)
22. Pujet, L., Tabareau, N.: Observational Equality: Now For Good. Proceedings of the ACM on Programming Languages **6**(POPL), 1–29 (2022). https://doi.org/10.1145/3498693. URL https://hal.inria.fr/hal-03367052
23. Pujet, L., Tabareau, N.: Impredicative observational equality. Proc. ACM Program. Lang. **7**(POPL) (2023). https://doi.org/10.1145/3571739. URL https://doi.org/10.1145/3571739
24. Pujet, L., Tabareau, N.: A Logical Relation for Observational Equality Meets CIC (2024). https://doi.org/10.5281/zenodo.10499152. URL https://doi.org/10.5281/zenodo.10499152
25. Swan, A.: An algebraic weak factorisation system on 01-substitution sets: a constructive proof. Journal of Logic and Analysis (2016). https://doi.org/10.4115/jla.2016.8.1. URL http://dx.doi.org/10.4115/jla.2016.8.1
26. Timany, A., Sozeau, M.: Consistency of the predicative calculus of cumulative inductive constructions (pcuic) (2020)
27. Vezzosi, A., Mörtberg, A., Abel, A.: Cubical agda: A dependently typed programming language with univalence and higher inductive types. Proc. ACM Program. Lang. **3**(ICFP) (2019). https://doi.org/10.1145/3341691. URL https://doi.org/10.1145/3341691
28. Werner, B.: On the strength of proof-irrelevant type theories **4**, 1–20 (2008)