Observational Equality Meets CIC

LOÏC PUJET, Department of Mathematics, Stockholm University, Stockholm, Sweden
YANN LERAY and NICOLAS TABAREAU, Inria, Nantes, France

The notion of equality is at the heart of dependent type theory, as it plays a fundamental role in program specifications and mathematical reasoning. In mainstream proof assistants such as AGDA, LEAN, and COQ, equality is usually defined using Martin-Löf's identity type, an elegant and simple approach that has stood the test of time since the 1970s. However, this definition also comes with serious downsides: the intensional nature of Martin-Löf's identity type means that it is impractical for reasoning about functions and predicates, and it is impossible to define quotient types. Recently, observational equality has garnered attention as an alternative method for encoding equality, particularly in proof assistants supporting definitionally proof-irrelevant propositions. However, it has yet to be integrated in any of the three proof assistants mentioned above, as it is not fully compatible with another important feature of type theory: indexed inductive types. In this article, we propose a systematic approach to reconcile observational equality with indexed inductive types, using a type coercion operator that computes on reflexive identity proofs. The second contribution of this article is a formal proof that this additional computation rule can be integrated to the system without compromising the decidability of conversion. Finally, we provide an implementation of our observational equality in an extension of COQ. This extension is based on the recently introduced rewrite rules and provides new extensionality principles while remaining fully backward-compatible.

CCS Concepts: • **Theory of computation** → **Type theory**;

Additional Key Words and Phrases: Type theory, Proof assistants, Observational equality, Inductive types, Rewriting rules

ACM Reference format:
Loïc Pujet, Yann Leray, and Nicolas Tabareau. 2025. Observational Equality Meets CIC. *ACM Trans. Program. Lang. Syst.* 47, 2, Article 6 (April 2025), 35 pages.
https://doi.org/10.1145/3719342

1 Introduction

Equality is a fundamental tool for mathematical reasoning and formal specification and thus plays a central role in proof assistants. In Martin-Löf's intensional type theory [19], equality is expressed using 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 (which is known as the *J rule*, or simply *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

Authors' Contact Information: Loïc Pujet (corresponding author), Department of Mathematics, Stockholm University, Stockholm, Sweden; e-mail: loic@pujet.fr; Yann Leray, Inria, Nantes, France; e-mail: yann.leray@inria.fr; Nicolas Tabareau, Inria, Nantes, France; e-mail: nicolas.tabareau@inria.fr.

ACM 1558-4593/2025/4-ART6
https://doi.org/10.1145/3719342

higher groupoid [30]. Martin-Löf's identity type serves as the basis for expressing equality in most proof assistants based on dependent type theory, in particular in Agda, Coq and Lean.

Unfortunately, this alluring formulation suffers from serious drawbacks: in intensional type theory, it is simply not possible to prove that Martin-Löf's identity type satisfies extensionality rules, and its type-agnostic definition makes it difficult to integrate types for which the equality relation is specified in an *ad hoc* manner, such as quotient types. Yet, the reality is that quotient types and extensionality rules are pervasive throughout 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 certainly possible to postulate these extensionality rules as axioms, doing so 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 to the root of the problem and replace Martin-Löf's identity type with a better-behaved alternative, such as the *observational equality* of Altenkirch et al. [5]. Unlike Martin-Löf's identity type, observational equality has a specific definition for every 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 leeway in the precise implementation of this idea; in this work we build upon the recently proposed system $\mathsf{CC}^{\mathrm{obs}}$ of Pujet and Tabareau [25]. 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 $\mathcal{J}$ 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 more than 15 years, none of the mainstream proof assistants support 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 so 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, we give a precise description of observational equality between two instances $I\ \vec{x}$ and $I\ \vec{y}$ of the same inductive type. The correct specification is slightly more subtle than the injectivity of type formers—in particular, when a parameter of I is not used in the type signatures 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 by Altenkirch and McBride [4], 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 Pujet and Tabareau [25]: 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

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

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 Pujet and Tabareau [24] 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 definition of identity type in cubical type theories [28]. 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 Allais et al. [3] 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 for $\mathsf{CC}^{\mathrm{obs}}$ [24]. 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 conversion are still poorly understood as already noticed by Abel and Coquand [1], Werner [32].

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 article 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 9), following previous work on logical relations by Abel et al. [2], Pujet and Tabareau [24, 25].

Finally, our last contribution is an implementation of $\mathsf{CIC}^{\mathrm{obs}}$ inside Coq, available at https://github.com/loic-p/coq. This implementation supports the computation of `cast` on reflexivity proofs and observational inductive types, while remaining compatible with developments written in plain CIC.

Related Work. The first proof assistant with an observational equality was the now-defunct Epigram 2, implemented by McBride [21]. 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, in particular in the work of Cavallo and Harper [9]. There, the situation is complicated by the fact that transport for the cubical equality does not support definitional computation on reflexivity as of today (this is known as the *regularity* problem), and thus the Fordism encoding cannot be used directly with the cubical equality. Instead, Cavallo and Harper add an *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. This solution is roughly equivalent to combining the Fordism technique with the identity type of Swan [28]. The resulting cubical inductive types have been implemented in Cubical Agda by Vezzosi et al. [31] and have been used to develop a sizeable standard library.

Plan of the article. Section 2 studies the interaction of the observational equality with indexed inductive type on increasingly complex examples to help the reader build an intuition for the subject. Section 3 presents the syntax, typing rules, and definition of conversion of $\mathsf{CIC}^{\mathrm{obs}}$ without inductive types, while Section 4 presents the scheme for inductive definitions. We then present our formal proof that conversion is decidable in $\mathsf{CIC}^{\mathrm{obs}}$ (Section 5.4) and build a model to show the consistency of the theory in Section 6. Finally, we go over our implementation of $\mathsf{CIC}^{\mathrm{obs}}$ in Coq using rewrite rules in Section 7.

2 Observational Equality Meets Calculus of Inductive Constructions (CIC) at Work

The CIC, which is the theoretical foundation of the proof assistants Coq and Lean, includes a powerful scheme for inductive definitions, originally introduced by Paulin-Mohring [23]. 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 want 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 the rules are self-evident, others turn out to be more delicate. In this section, we look at three concrete examples of inductive types to help the reader build their intuition: lists, vectors and Martin-Löf's identity type.

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.
```

In CIC, every type former comes with introduction rules, elimination rules, and computation rules. In our example, the constructors `nil` and `cons` provide the introduction rules, while the elimination rules are given by the induction principle for lists—in the language of Coq, this induction principle is decomposed into a pattern-matching operator (`match`) and a guarded fixpoint operator (`fix`). Lastly, the computation rules correspond to the usual ι-reduction. In an observational type theory however, 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 of the type, a definition of the observational equality between two instances of the type former, and computation rules for `cast`.

Note that there is a bit of leeway in what is meant by a *definition* of the observational equality: in the original version of Altenkirch et al. [5] and most of the subsequent literature, the observational equality type itself evaluates to a domain-specific equality type, meaning that a proof of equality between two functions is judgmentally the same as a proof of pointwise equality. Alternatively, it is possible to implement a version of 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 a proper equality, as done by Atkey [6]. In this article, we go with the second approach, as it turns out to be better suited for our implementation in Coq.

Now, let us get back to 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 eliminator for the observational equality—just like we would prove it in plain intensional Martin-Löf Type Theory. 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 types, on the other hand, does not benefit from such an induction principle and must be specified further. 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:

```
eq-list : list A ~ list B ⟶ A ~ B.
```

This principle is necessary, because a proof of equality between `list` A and `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 $\mathcal{J}$ eliminator), it does indeed fully determine the observational equality between list types. Finally, we need to explain how `cast` computes on lists. Unlike the computation rules for the observational equality type, these rules are very much necessary, unless we are fine with having stuck computations in an empty context. Here, there is only one natural choice: typecasting a constructor of `list A` should produce the corresponding constructor of `list B`.

$$\begin{array}{lcl}
\texttt{cast}\ (\texttt{list}\,A)\ (\texttt{list}\,B)\ e\ \texttt{nil} & \equiv & \texttt{nil} \\
\texttt{cast}\ (\texttt{list}\,A)\ (\texttt{list}\,B)\ e\ (\texttt{cons}\,a\,l) & \equiv & \texttt{cons}\ \ (\texttt{cast}\ A\ B\ (\texttt{eq-list}\ e)\ a) \\
 & & \qquad\ \ (\texttt{cast}\ (\texttt{list}\,A)\ (\texttt{list}\,B)\ e\ l)
\end{array}$$

Remark that in the case of a non-empty list, we need to use the `eq−list` operator 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 inductive definitions of Coq is indices. Here, the story will get more complicated, as indexed inductive types gain new inhabitants in the presence of the observational equality. To see this, consider Martin-Löf's identity type, which has two parameters and one index[1]:

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

In intensional type theory, it is well-known that Martin-Löf's equality type does not satisfy the principle of function extensionality. But in our observational type theory, it turns out we can 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

[1]Parameters are placed on the left side of the colon in the inductive definition and must remain unchanged in the output type of every constructor, while indices are placed on the right side of the colon, and constructors may specify their value arbitrarily.

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 non-canonical inhabitants, we can turn our attention to *Fordism*. This technique was invented by Coquand for his work on the proof assistant ALF 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 the Ph.D. dissertation of McBride [20], 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 bare-bones 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 non-uniform[2] 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 functions $\mathtt{vnil}_{ap}$ and $\mathtt{vcons}_{ap}$ which can be defined using the fact 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 eliminator that computes on reflexivity.

Additionally, this transformation sheds some light on the non-canonical elements of indexed inductive types: in CIC, the only closed proof of equality is a proof by reflexivity, thus the inhabitants of `vec𝔽 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

[2]A parameter is said to be *uniform* when it stays the same across all recursive calls and *non-uniform* otherwise.

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 Id𝔽 (A : Type) (x y : A) : Type :=
| id_refl𝔽 : x ~A y ⟶ Id𝔽 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 $\mathtt{Id}_{\mathbb{F}}$, a definition of the observational equality between two instances of $\mathtt{Id}_{\mathbb{F}}$, and computation rules for the `cast` operator. The first one is easy, as we can prove that any two inhabitants of $\mathtt{Id}_{\mathbb{F}}\ \mathsf{A}\ \mathsf{x}\ \mathsf{y}$ are equal: by induction, we only need to prove it for elements of the form $\mathtt{id\_refl}_{\mathbb{F}}\ \mathsf{e}$, with e being a proof of $\mathsf{x} \sim_A \mathsf{y}$. But the observational equality is definitionally proof-irrelevant, so this is true by reflexivity. In other words, the principle of UIP is provable for the inductive identity type in observational type theory, in stark contrast to MLTT or CIC. Thus, we do not need any further characterization of the observational equality between inhabitants of $\mathtt{Id}_{\mathbb{F}}$.

On the other hand, the definition of the observational equality between two instances of the identity type $\mathtt{Id}_{\mathbb{F}}\ \mathsf{A}\ \mathsf{x}\ \mathsf{y}$ and $\mathtt{Id}_{\mathbb{F}}\ \mathsf{A'}\ \mathsf{x'}\ \mathsf{y'}$ makes for a more interesting story. From our study of lists, it might be tempting to extrapolate that an observational equality between two instances of a parametrized inductive datatype should imply an equality between the parameters. In the case of $\mathtt{Id}_{\mathbb{F}}$, this translates to the following principle:

$$\mathtt{Id}_{\mathbb{F}}\ \mathsf{A}\ \mathsf{x}\ \mathsf{y} \sim \mathtt{Id}_{\mathbb{F}}\ \mathsf{B}\ \mathsf{z}\ \mathsf{w} \longrightarrow \exists\,(\mathsf{e} : \mathsf{A} \sim \mathsf{B}),\ (\mathtt{cast}\ \mathsf{A}\ \mathsf{B}\ \mathsf{e}\ \mathsf{x} \sim \mathsf{z}) \land (\mathtt{cast}\ \mathsf{A}\ \mathsf{B}\ \mathsf{e}\ \mathsf{y} \sim \mathsf{w}).$$

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:

```
inj (X : Type ⟶ Type) := Id𝔽 (Type ⟶ Type) X 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 as shown by Miquel [22]. 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 definition for the observational equality between inductive types. After all, what do we need this definition for?

[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 do not consider this option.

The answer is simple: we need some observational equalities to put in the computation rules for the `cast` operator.

```
cast (Id𝔽 A x y) (Id𝔽 B z w) e (id_refl𝔽 e') ≡ ...
```

For inductive types without indices, 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 x' $\sim_{A'}$ y' from an inhabitant of x $\sim_A$ y. This is a job for the `cast` operator:

```
cast (Id𝔽 A x y) (Id𝔽 B z w) e (id_refl𝔽 h) ≡ id_refl𝔽 (cast (x ∼ y) (z ∼ w) ? h).
```

In order to fill the question mark hole, we need a proof of observational equality between the two types x' $\sim_{A'}$ y' and x $\sim_A$ y. Since all we have is a proof of equality between `Id𝔽 A x y` and `Id𝔽 B z w`, we need a way to extract the desired equality out of it. The injectivity of 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 the types of their constructor arguments, and nothing more. In the case of the inductive `Id𝔽`, it means that we get the following projection:

```
eq-Id𝔽 : Id𝔽 A x y ∼ Id𝔽 B z w ⟶ (x ∼ y) ∼ (z ∼ w).
```

This is enough to fill the question mark hole in the computation rule for `cast`. Furthermore, as we prove in Section 6, this form of injectivity is sufficiently weak to allow the identity type to live in the lowest universe without endangering the consistency of the theory.

3 An Observational Type Theory with Martin-Löf's Computation Rule

At this stage, we have a good idea of the ingredients that are required for our observational type theory with inductive types: first, we need a system with a `cast` operator that computes on proofs by reflexivity. Next, we add parametrized inductive types with projection operators for the observational equality and computation rules for `cast`. Finally, we use the Fordism transformation to take care of indexed inductive types.

We thus turn ourselves to the definition of a formal system that incorporates all of these ingredients, which we call $\mathsf{CIC}^{\mathrm{obs}}$. It is based on the system $\mathsf{CC}^{\mathrm{obs}}$ of Pujet and Tabareau [25], 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 provide a brief presentation of the syntax, typing rules and declarative conversion for the core of $\mathsf{CIC}^{\mathrm{obs}}$, with an emphasis on the points that differ from $\mathsf{CC}^{\mathrm{obs}}$, before defining the scheme for inductive types in Section 4. All the definitions in the figures closely follow our Agda formalization. We refer to files in the formalization as *[myFile.agda]*.

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

The syntax of the sorts, contexts, terms, and types of $\mathsf{CIC}^{\mathrm{obs}}$ is specified in Figure 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 $A \longrightarrow B$ 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 4).

Compared to the previous system $\mathsf{CC}^{\mathrm{obs}}$, 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 equality which were previously given

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

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

as computation rules. For instance, in CC$^{\text{obs}}$, 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, and results in a more elegant system that does not need a primitive Σ-type. This way of handling the properties of the observational equality is 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 CIC$^{\text{obs}}$

The typing rules of 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 CC$^{\text{obs}}$, so we only give a brief overview here (Figure 2).. The complete set of rules is available in *[Typed.agda]*. We use PTS-style notations of Barendregt [7] to factorize the rules that involve universes: the formation rule for universes states that both $\mathcal{U}_i$ and Ω are inhabitants of a higher universe, as described by the relations

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

As for dependent products, we allow their formation 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

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

Equality and Type Casts. Every proof-relevant type is equipped with its observational equality type, which takes the form of a proof-irrelevant binary relation $t \sim_A u$. Of course, proof-irrelevant types have no use for an observational equality type, since any two inhabitants would always be

$$\text{Ctx-Nil}\ \frac{}{\vdash \bullet} \qquad \text{Ctx-Cons}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A : s}{\vdash \Gamma, x : A : s} \qquad \text{Var}\ \frac{\vdash \Gamma \qquad x : A : s \in \Gamma}{\Gamma \vdash x : A : s} \qquad \text{conv}\ \frac{\Gamma \vdash t : A : s \qquad \Gamma \vdash A \equiv B : s}{\Gamma \vdash t : B : s}$$

$$\text{Univ}\ \frac{\vdash \Gamma}{\Gamma \vdash s : s'}\mathcal{A}(s,s') \qquad \Pi\text{-Form}\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A : s \vdash B : s'}{\Gamma \vdash \Pi^{s,s'}(x : A).\, B : \mathcal{R}(s,s')} \qquad \text{Fun}\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A : s \vdash t : B : s'}{\Gamma \vdash \lambda(x : A).\, t : \Pi^{s,s'}(x : A).\, B : \mathcal{R}(s,s')}$$

$$\text{App}\ \frac{\Gamma \vdash A : s \qquad \Gamma, x : A : s \vdash B : s' \qquad \Gamma \vdash t : \Pi^{s,s'}(x : A).\, B : \mathcal{R}(s,s') \qquad \Gamma \vdash u : A : s}{\Gamma \vdash t\ u : B[x := u] : s'}$$

$$\bot\text{-Form}\ \frac{\vdash \Gamma}{\Gamma \vdash \bot : \Omega} \qquad \bot\text{-Elim}\ \frac{\Gamma \vdash A : s \qquad \Gamma \vdash t : \bot : \Omega}{\Gamma \vdash \mathtt{\bot\text{-}elim}\ A\ t : A : s} \qquad \text{Eq-Form}\ \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash t : A : \mathcal{U}_i \qquad \Gamma \vdash u : A : \mathcal{U}_i}{\Gamma \vdash t \sim_A u : \Omega} \qquad \text{Refl}\ \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash t : A : \mathcal{U}_i}{\Gamma \vdash \mathtt{refl}\ t : t \sim_A t : \Omega}$$

$$\text{Cast}\ \frac{\Gamma \vdash A, B : s \quad \Gamma \vdash e : A \sim_s B : \Omega \qquad \Gamma \vdash t : A : s}{\Gamma \vdash \mathtt{cast}\ A\ B\ e\ t : B : s}$$

$$\text{Transport-}\Omega\ \frac{\Gamma \vdash A : \mathcal{U}_i \quad \Gamma \vdash t, t' : A : \mathcal{U}_i \quad \Gamma, y : A : \mathcal{U}_i \vdash B : \Omega \quad \Gamma \vdash u : B[y := t] : \Omega \quad \Gamma \vdash e : t \sim_A t' : \Omega}{\Gamma \vdash \mathtt{transport}\ A\ t\ B\ u\ t'\ e : B[y := t'] : \Omega}$$

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

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

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

Fig. 2. CIC$^{\text{obs}}$ Typing rules *[Typed.agda]*.

in relation by reflexivity. The observational equality is equipped with two eliminators, which are called `transp` and `cast`. The former is similar to the $\mathcal{J}$ eliminator from MLTT, except that it is restricted to proof-irrelevant predicates. Elimination into the proof-relevant layer is thus handled by the `cast` operator, which provides coercions between two observationally equal types. Its type seems less general compared to the standard $\mathcal{J}$ eliminator at first glance, but since equality proofs are definitionally irrelevant, it turns out that a $\mathcal{J}$ eliminator for proof-relevant predicates can be derived from the `cast` operator [24].

As we already mentioned, the extensional properties of the observational equality are given by the primitives Π^1_ϵ, Π^2_ϵ, Ω_{ext}, and Π_{ext}: rules Eq-Π_1 and Eq-Π_2 allow us to deduce the equality of domains[4] and codomains from an equality between two dependent functions types, rule Eq-Ω provides propositional extensionality, and rule Eq-Fun provides function extensionality.

3.3 Conversion

Conversion, which is also called *definitional* or *judgmental* equality, is a judgment that relates terms that are interchangeable in typing derivations. The rules that define the conversion judgment are

[4]Because of contravariance, the equality between the domains is swapped with respect to the original equality.

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

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

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

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

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

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

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

reproduced in Figure 3. Conversion is defined as a reflexive, symmetric, and transitive relation; it is also closed under congruence (e.g., if $A \equiv A'$ and $B \equiv B'$ then $\Pi(x : A).B \equiv \Pi(x : A').B'$), although we did not reproduce all the corresponding rules in Figure 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[5] (rule η-Eq). The rule Proof-Irr reflects the proof-irrelevant nature of propositions: any two proofs 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 define in Section 4.

Next, we have the rules that describe 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 codomains. Furthermore, every declaration of an inductive type adds a handful of computation rules for the `cast` operator, described in Section 4. Last but not least, the rule Cast-Refl is the main innovation of CIC$^{\text{obs}}$. 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 $\mathcal{J}$ 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.

[5]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 it as a conversion rule, to get a more flexible system.

4 Inductive Definitions

On top of the rules from Section 3, $\mathsf{CIC}^{\mathsf{obs}}$ includes a scheme for proof-relevant inductive definitions that is based on the scheme of CIC (as defined by Timany and Sozeau [29]). Inductive definitions are not manipulated as first class objects: rather, the user declares all the necessary inductive types using a standard syntax, before starting their proof. After each declaration, the theory is extended with the new type former, inductive constructors, and so on, just like in Coq.

The syntax for the inductive definitions of $\mathsf{CIC}^{\mathsf{obs}}$ is exactly the same as in CIC; the difference lies in the fact that inductive definitions 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 inductives without indices, and then we extend it to general indexed inductive definitions by using the Fordism transformation and some syntactic sugar. We spare the reader the added complexity of mutually defined families and nested inductive types, which is mathematically straightforward but heavy on notation.

4.1 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 an non-indexed inductive type looks like this:

$$\begin{array}{l}\texttt{Inductive}\ \mathtt{Ind}\ (\vec{a} : \vec{A}) : \mathcal{U}_\ell := \\ \mid c_0 : \forall\, (\vec{b} : \vec{B}_0),\ \mathtt{Ind}\ \vec{a} \\ \mid \dots \\ \mid c_n : \forall\, (\vec{b} : \vec{B}_n),\ \mathtt{Ind}\ \vec{a}\end{array}$$

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 $a_1 : A_1, \dots, a_m : A_m$ 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* [11] with possibly non-uniform parameters—but 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 large enough to fit all the types that appear in the constructor arguments $\vec{B}_i$ and there is no size constraint regarding parameters. To simplify the presentation, inductive definitions are not allowed in the sort of propositions Ω, but they can be faithfully simulated using impredicative encodings and proof irrelevance.

After the user makes such a definition, the system is extended with the new type former `Ind` and the inductive constructors $c_0, \dots c_n$ with their prescribed types. Additionally, $\mathsf{CIC}^{\mathsf{obs}}$ 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 Coq Development Team [11]. As we explained in Section 2, this elimination principle is sufficient to fully characterize 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:

$$\mathtt{eq\_}c_i : \forall\, (\vec{a} : \vec{A})\, (\vec{a}' : \vec{A}),\ \mathtt{Ind}\ \vec{a} \sim \mathtt{Ind}\ \vec{a}' \longrightarrow \vec{B}_i[\vec{a}] \sim \vec{B}_i[\vec{a}']. \qquad (\forall\ \mathtt{i})$$

The projections $\mathtt{eq\_}c_i$ 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 $\mathtt{eq\_}c_i$ is implemented as a family of projections $(\mathtt{eq\_}c_{i,j})$, 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:

$$\texttt{cast}\ (\texttt{Ind}\ \vec{a})\ (\texttt{Ind}\ \vec{a}')\ \texttt{e}\ (c_i\ \vec{b})\ \equiv\ c_i\ (\texttt{cast}\ (\vec{B}_i[\vec{a}])\ (\vec{B}_i[\vec{a}'])\ (\texttt{eq\_}c_i\ \vec{a}\ \vec{a}'\ \texttt{e})\ \vec{b}). \qquad (\forall\ \texttt{i})$$

4.2 Deriving a Scheme for Indexed Inductive Types

In order for CIC$^{\text{obs}}$ 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 inductives with parametrized inductives. Consequently, we 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 CIC$^{\text{obs}}$, 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:

$$\begin{array}{lcl}
\texttt{Inductive Ind}\ (\vec{a} : \vec{A}) : \forall\ (\vec{x} : \vec{X}), \mathcal{U}_\ell := & & \texttt{Inductive Ind}_{\mathbb{F}}\ (\vec{a} : \vec{A})\ (\vec{x} : \vec{X}) : \mathcal{U}_\ell := \\
\mid c_0 : \forall\ (\vec{b} : \vec{B}_0),\ \texttt{Ind}\ \vec{a}\ \vec{y}_0 & & \mid c_{0\mathbb{F}} : \forall\ (\vec{b} : \vec{B}_0),\ \vec{y}_0 \sim \vec{x} \longrightarrow\ \texttt{Ind}_{\mathbb{F}}\ \vec{a}\ \vec{x} \\
\mid \ldots & \Rightarrow & \mid \ldots \\
\mid c_n : \forall\ (\vec{b} : \vec{B}_n),\ \texttt{Ind}\ \vec{a}\ \vec{y}_n & & \mid c_{n\mathbb{F}} : \forall\ (\vec{b} : \vec{B}_n),\ \vec{y}_n \sim \vec{x} \longrightarrow\ \texttt{Ind}_{\mathbb{F}}\ \vec{a}\ \vec{x}
\end{array}$$

This 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. After this step, Ind and its constructors are transparently elaborated in terms of their forded counterparts:

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

Likewise, 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 $\texttt{i} : \texttt{Ind}\ \vec{a}\ \vec{x}$:

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

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 $\texttt{P}\ \vec{y}_0\ (c_i\ \vec{b})$. After the elaboration, this branch matches a forded pattern $c_{i\mathbb{F}}\ \vec{b}\ e$ and should now return a result of type $\texttt{P}\ \vec{x}_0\ (c_{i\mathbb{F}}\ \vec{b}\ e)$. We can obtain this result by type casting the user-supplied term t_i along the equality proof e to obtain

$$\texttt{cast}\ (\texttt{P}\ \vec{y}_0\ (c_i\ \vec{b}))\ (\texttt{P}\ \vec{x}_0\ (c_{i\mathbb{F}}\ \vec{b}\ e))\ (\texttt{ap2 P}\ (c_{i\mathbb{F}}\ \vec{b})\ \texttt{e})\ t_i,$$

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 CIC$^{\text{obs}}$.

ℕ-Form

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} : \mathcal{U}_0}$$

Zero

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathtt{0} : \mathbb{N} : \mathcal{U}_0}$$

Suc

$$\frac{\Gamma \vdash n : \mathbb{N} : \mathcal{U}_0}{\Gamma \vdash \mathtt{S}\, n : \mathbb{N} : \mathcal{U}_0}$$

ℕ-Elim

$$\frac{\Gamma, m : \mathbb{N} \vdash A : s \qquad \Gamma \vdash t_0 : A[m := \mathtt{0}] : s \qquad \Gamma \vdash t_S : \Pi(n : \mathbb{N}).\, A[m := n] \to A[m := \mathtt{S}\, n] : s \qquad \Gamma \vdash n : \mathbb{N} : \mathcal{U}_0}{\Gamma \vdash \mathbb{N}\mathtt{-elim}\, A\, t_0\, t_S\, n : A[m := n] : s}$$

ℕ-Elim-Zero

$$\frac{\Gamma, m : \mathbb{N} \vdash A : s \qquad \Gamma \vdash t_0 : A[m := \mathtt{0}] : s \qquad \Gamma \vdash t_S : \Pi(n : \mathbb{N}).\, A[m := n] \to A[m := \mathtt{S}\, n] : s}{\Gamma \vdash \mathbb{N}\mathtt{-elim}\, A\, t_0\, t_S\, \mathtt{0} \equiv t_0 : A[m := \mathtt{0}] : s}$$

ℕ-Elim-Suc

$$\frac{\Gamma, m : \mathbb{N} \vdash A : s \qquad \Gamma \vdash t_0 : A[m := \mathtt{0}] : s \qquad \Gamma \vdash t_S : \Pi(n : \mathbb{N}).\, A[m := n] \to A[m := \mathtt{S}\, n] : s \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathbb{N}\mathtt{-elim}\, A\, t_0\, t_S\, \mathtt{S}\, n \equiv_S \mathbb{N}\mathtt{-elim}\, A\, t_0\, t_S\, n : A[m := \mathtt{S}\, n] : s}$$

Cast-Zero

$$\frac{\Gamma \vdash e : \mathbb{N} \sim_{\mathcal{U}_0} \mathbb{N} : \Omega}{\Gamma \vdash \mathtt{cast}\, \mathbb{N}\, \mathbb{N}\, e\, \mathtt{0} \equiv \mathtt{0} : \mathbb{N} : \mathcal{U}_0}$$

Cast-Suc

$$\frac{\Gamma \vdash e : \mathbb{N} \sim_{\mathcal{U}_0} \mathbb{N} : \Omega \qquad \Gamma \vdash n : \mathbb{N} : \mathcal{U}_0}{\Gamma \vdash \mathtt{cast}\, \mathbb{N}\, \mathbb{N}\, e\, (\mathtt{S}\, n) \equiv \mathtt{S}\, (\mathtt{cast}\, \mathbb{N}\, \mathbb{N}\, e\, n) : \mathbb{N} : \mathcal{U}_0}$$

Fig. 4. Typing and conversion rules for natural numbers *[Typed.agda]*.

5 Decidability of Conversion

In this section, we show that conversion is decidable in presence of the rule Cast-Refl for a simplified version of CIC$^{\text{obs}}$ in which the induction scheme is reduced to the type of integers. For completeness, the typing and conversion rules for natural numbers are given in Figure 4. 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 5.1) and a notion of conversion on neutral terms and **weak-head normal forms (whnf)** (Section 5.2). 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 whnf 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 whnf. Then we can show that transitivity is admissible for conversion on neutral terms and whnf. Naturally, this definition requires a proof of normalization of well-typed terms.

In the case of CIC$^{\text{obs}}$ however, the decision procedure for conversion of neutral terms and whnf 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 5.4) 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

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

Fig. 5. Weak-head normal and neutral forms *[Untyped.agda]*.

of Abel et al. [2] to guarantee that every term can be put in whnf and that algorithmic conversion is complete with respect to conversion (Section 5.5).

Note that our formalized version of CIC$^{\text{obs}}$ only supports the inductive type of natural numbers, and not the full scheme from Section 4. 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. In practice, we expect that the additional inductive types would not pose any additional difficulty with respect to the presence of Rule Cast-Refl.

5.1 Reduction to whnf

A notion that plays a central role in our normalization procedure is that of a whnf, which corresponds to a relevant term that cannot be head-reduced further (Figure 5). Whnf are either terms with a constructor in head position or *neutral terms* stuck on a variable or an elimination of a proof of $\bot$. In other words, neutral terms are whnf that should not exist in an empty context. In CIC$^{\text{obs}}$, 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 type 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 $\mathtt{\bot\text{-}elim}\ A\ e$ because they corresponds to casts based on an inconsistent proof of equality, thus similar to elimination of a proof of $\bot$.

Figure 6 introduces a notion of typed reduction, noted $\Gamma \vdash t \Rightarrow u : A$ that is at the heart of the decision procedure for conversion. Intuitively, reduction corresponds to an orientation of conversion rule in order to provide a rewrite system for which we can compute normal form. However, not every conversion rule gives rise to a reduction rule, and in particular Rule Cast-Refl which would enter in conflict with many other reduction rules of the system and more importantly would require a test of conversion for the rule to be triggered. This means that there would be a circularity between the definition of reduction and the notion of conversion. We are not aware of any framework to handle such a circularity. To avoid it, we go for a treatment of Cast-Refl only for neutral terms and whnf. However, for efficiency purpose, we will see in Section 7 that we can implement the directing version of Cast-Refl, once we know that the theory is decidable.

Actually, the purpose of reduction is to compute whnf so that conversion rules that are not part of the reduction have only to be checked on whnf.

Rules β-red, $\mathbb{N}$-Elim-Zero-red, $\mathbb{N}$-Elim-Suc-red, and Cast-Π-red are direct orientations of their corresponding conversion rule. Rules suffixed with subst corresponds to oriented congruence rules. Note that in the case of `cast`, we need to be careful to reduce one argument after the other

β-red

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

$\mathbb{N}$-Elim-Zero-red

$$\frac{\Gamma, m : \mathbb{N} \vdash A : s \qquad \Gamma \vdash t_0 : A[m := 0] : s \qquad \Gamma \vdash t_S : \Pi(n : \mathbb{N}).\, A[m := n] \to A[m := \mathsf{S}\ n] : s}{\Gamma \vdash \mathtt{\mathbb{N}{-}elim}\ A\ t_0\ t_S\ 0 \Rightarrow t_0 : A[m := 0]}$$

$\mathbb{N}$-Elim-Suc-red

$$\frac{\Gamma, m : \mathbb{N} \vdash A : s \qquad \Gamma \vdash t_0 : A[m := 0] : s \qquad \Gamma \vdash t_S : \Pi(n : \mathbb{N}).\, A[m := n] \to A[m := \mathsf{S}\ n] : s \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathtt{\mathbb{N}{-}elim}\ A\ t_0\ t_S\ \mathsf{S}\ n \Rightarrow t_S\ n\ \mathtt{\mathbb{N}{-}elim}\ A\ t_0\ t_S\ n : A[m := \mathsf{S}\ n]}$$

Cast-Π-red

$$\frac{\begin{array}{c}\Gamma \vdash A, A' : s \qquad \Gamma, x : A \vdash B : s' \\ \Gamma, x : A' \vdash B' : s' \qquad \Gamma \vdash e : \Pi(x : A).\, B \sim_{\mathcal{U}_i} \Pi(x : A').\, B' : \Omega \qquad \Gamma \vdash f : \Pi(x : A).\, B \qquad a := \mathtt{cast}\ A'\ A\ \Pi^1_\epsilon\ e\ a'\end{array}}{\Gamma \vdash \begin{array}{l}\mathtt{cast}\ \Pi(x : A).\, B\ \Pi(x : A').\, B'\ e\ f \Rightarrow \\ \lambda(a' : A').\, \mathtt{cast}\ B[x := a]\ B'[x := a']\ \Pi^2_\epsilon\ e\ a'\ f\ a\end{array} : \Pi(x : A').\, B'}$$

Cast-Zero

$$\frac{\Gamma \vdash e : \mathbb{N} \sim_{\mathcal{U}_0} \mathbb{N} : \Omega}{\Gamma \vdash \mathtt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ 0 \Rightarrow 0 : \mathbb{N}}$$

Cast-Suc

$$\frac{\Gamma \vdash e : \mathbb{N} \sim_{\mathcal{U}_0} \mathbb{N} : \Omega \qquad \Gamma \vdash n : \mathbb{N} : \mathcal{U}_0}{\Gamma \vdash \mathtt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ (\mathsf{S}\ n) \Rightarrow \mathsf{S}\ \mathtt{cast}\ \mathbb{N}\ \mathbb{N}\ e\ n : \mathbb{N}}$$

Cast-Univ

$$\frac{\Gamma \vdash e : s \sim_{s'} s \qquad \Gamma \vdash A : s}{\Gamma \vdash \mathtt{cast}\ s\ s\ e\ A \Rightarrow A : s}$$

Conv-Red

$$\frac{\Gamma \vdash t \Rightarrow u : A \qquad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash t \Rightarrow u : B}$$

app-subst

$$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A : s \vdash B : \mathcal{U}_i \qquad \Gamma \vdash t \Rightarrow u : \Pi^{s, \mathcal{U}_i}(x : A).\, B \qquad \Gamma \vdash a : A : s}{\Gamma \vdash t\ a \Rightarrow u\ a : B[x := a]}$$

$\mathbb{N}$-Elim-subst

$$\frac{\Gamma, m : \mathbb{N} \vdash A : s \quad \Gamma \vdash t_0 : A[m := 0] : s \quad \Gamma \vdash t_S : \Pi(n : \mathbb{N}).\, A[m := n] \to A[m := \mathsf{S}\ n] : s \quad \Gamma \vdash n \Rightarrow n' : \mathbb{N}}{\Gamma \vdash \mathtt{\mathbb{N}{-}elim}\ A\ t_0\ t_S\ n \Rightarrow \mathtt{\mathbb{N}{-}elim}\ A\ t_0\ t_S\ n' : A[m := n]}$$

Cast-subst

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

Cast-subst-nf

$$\frac{\Gamma \vdash A : s \qquad \mathrm{whnf}\ A \qquad \Gamma \vdash B \Rightarrow B' : s \qquad \Gamma \vdash e : A \sim_s B : \Omega \qquad \Gamma \vdash t : A : s}{\Gamma \vdash \mathtt{cast}\ A\ B\ e\ t \Rightarrow \mathtt{cast}\ A\ B'\ e\ t : B}$$

Cast-subst-nf-nf

$$\frac{\Gamma \vdash A, B : s \qquad \mathrm{whnf}\ A \qquad \mathrm{whnf}\ B \qquad \Gamma \vdash e : A \sim_s B : \Omega \qquad \Gamma \vdash t \Rightarrow u : A}{\Gamma \vdash \mathtt{cast}\ A\ B\ e\ t \Rightarrow \mathtt{cast}\ A\ B\ e\ u : B}$$

Id

$$\frac{\Gamma \vdash t : A : \mathcal{U}_i}{\Gamma \vdash t \Rightarrow^* t : A}$$

Step

$$\frac{\Gamma \vdash t \Rightarrow u : A \qquad \Gamma \vdash u \Rightarrow^* v : A}{\Gamma \vdash t \Rightarrow^* v : A}$$

Fig. 6. CIC$^{\text{obs}}$ reduction rules *[Typed.agda]*.

in order, so that weak-head reduction remains deterministic, which is crucial in the logical relation setting of Section 5.5.

The reduction rules Cast-Zero Cast-Suc, and Cast-Univ correspond to the rule Cast-Refl where the arguments are instantiated by whnf that are not neutral. Indeed, in that case `cast` must reduce. Conversion for `cast` when one of the scrutinees is neutral is differed to algorithmic conversion.

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

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

$$\textsc{$\mathbb{N}$-Elim-cong}\ \frac{\begin{array}{c}\Gamma, m : \mathbb{N} \vdash A \cong^{\downarrow} B : \mathcal{U}_i \qquad \Gamma \vdash t_0 \cong^{\downarrow} u_0 : A[m := \mathtt{0}] \\ \Gamma \vdash t_{\mathtt{S}} \cong^{\downarrow} u_{\mathtt{S}} : \Pi(n : \mathbb{N}).\, A[m := n] \to A[m := \mathtt{S}\ n] \qquad \Gamma \vdash n \cong^{\downarrow}_{ne} n' : \mathbb{N}\end{array}}{\Gamma \vdash \mathbb{N}\mathtt{-elim}\ A\ t_0\ t_{\mathtt{S}}\ n \cong_{ne} \mathbb{N}\mathtt{-elim}\ B\ u_0\ u_{\mathtt{S}}\ n' : A[m := n]}$$

$$\textsc{$\bot$-Elim-cong}\ \frac{\Gamma \vdash A \cong^{\downarrow} B : \mathcal{U}_i \qquad \Gamma \vdash t \cong_{ne} u : \bot}{\Gamma \vdash \bot\mathtt{-elim}\ A\ t \cong_{ne} \bot\mathtt{-elim}\ B\ u : A}$$

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

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

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

$$\textsc{U-cong}\ \frac{\vdash \Gamma \qquad \mathcal{A}(s, s')}{\Gamma \vdash s \cong s : s'} \qquad \textsc{$\mathbb{N}$-cong}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} \cong \mathbb{N} : \mathcal{U}_0} \qquad \textsc{$\bot$-cong}\ \frac{\vdash \Gamma}{\Gamma \vdash \bot \cong \bot : \Omega} \qquad \textsc{Eq-cong}\ \frac{\Gamma \vdash A \cong^{\downarrow} A' : \mathcal{U}_i \qquad \Gamma \vdash t \cong^{\downarrow} t' : A \qquad \Gamma \vdash u \cong^{\downarrow} u' : A}{\Gamma \vdash t \sim_A u \cong t' \sim_{A'} u' : \Omega}$$

$$\textsc{Π-cong}\ \frac{\Gamma \vdash A \cong^{\downarrow} A' : \mathcal{U}_i \qquad \Gamma, x : A : s \vdash B \cong^{\downarrow} B' : s'}{\Gamma \vdash \Pi^{s,s'}(x : A).\, B \cong \Pi^{s,s'}(x : A').\, B' : \mathcal{R}(s, s')}$$

$$\textsc{Zero-cong}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathtt{0} \cong \mathtt{0} : \mathbb{N}} \qquad \textsc{Suc-cong}\ \frac{\Gamma \vdash n \cong^{\downarrow} n' : \mathbb{N}}{\Gamma \vdash \mathtt{S}\ n \cong \mathtt{S}\ n' : \mathbb{N}} \qquad \textsc{η-dec}\ \frac{\Gamma \vdash t, u : \Pi^{s,\mathcal{U}_i}(x : A).\, B : \mathcal{R}(s, \mathcal{U}_i) \qquad \Gamma, x : A : s \vdash t\ x \cong^{\downarrow} u\ x : B}{\Gamma \vdash t \cong u : \Pi^{s,\mathcal{U}_i}(x : A).\, B}$$

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

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

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

Fig. 7. CIC$^{\text{obs}}$ algorithmic conversion rules *[ConversionGen.agda]*.

5.2 Algorithmic Conversion

Algorithmic conversion (Figure 7) 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 CIC$^{\text{obs}}$ 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 5.4.

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 whnf, the inferred type is also in whnf. Bidirectional typing as recently popularized by Lennon-Bertrand [17, 18] is traditionally used in type theory to provide a canonical typing derivation by splitting the typing judgment into two: one judgment that infers the type of a term and another 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 infer arbitrary types (for instance, the application rule infers the type of the codomain of the function with an additional substitution) whereas other whnf always infer types which are also in whnf. This means that we need to reflect this important distinction in the algorithmic conversion because the structural conversion rules for neutral terms ($\Gamma \vdash t \cong_{ne} u : A$) is naturally performed at an arbitrary type A whereas $\Gamma \vdash t \cong u : A$ is always done at a type A in whnf.

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 conversely $\Gamma \vdash t \cong^{\downarrow} u : A$ means that $\Gamma \vdash t' \cong u' : B$ 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. Rule η-dec is a direct implementation of η-conversion, where the terms $t\ x$ and $u\ x$ are first put into whnf before being compared. 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 whose type is 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. For Rules app-cong, $\mathbb{N}$-Elim-cong, and $\bot$-Elim-cong, there is only one scrutinee to which we inductively apply conversion of neutral terms. For the rest of the arguments, general conversion of weak-head normal form is asked. 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 cost 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 whnf 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) yet 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 key aspect of this algorithmic conversion is that it does not contain any rule for symmetry or transitivity, which are primary contributors to undecidability for conversion.

5.3 Symmetry and Transitivity of Algorithmic Conversion

The correctness of algorithmic conversion is immediate as the rules used are subsumed by the declarative conversion judgment *[Soundness.agda]*.

Showing that the algorithmic conversion is also complete is however much more complex, and the proof is deferred to Section 5.5. We here focus on two important ingredients, the facts that symmetry and transitivity are admissible for algorithmic conversion. The main issue for the statement of symmetry is that actually the comparison of neutral terms infers the type, but this inference is biased toward the left argument. Indeed, in the rule for congruence of application for instance (Rule app-cong), the term that is used to perform the substitution in the codomain of the function is the argument of the left-hand side. Thus, when considering the symmetric judgment, the inferred type may be different. Similarly, in the rule of congruence of Πs (Rule Π-cong), the context is extended with a variable in the type of the domain of the left-hand side, thus when considering the symmetric version, the context needs to be changed.

Hopefully, it is still possible to show actually all those differences are actually up to convertible types and contexts, convertible in the sense of declarative conversion. The fact that we rely on declarative conversion here is not a problem as the type annotation is used for correctness of the algorithmic conversion, but it does not play any role in the decidability procedure that we derive.

So the symmetry of algorithmic conversion that can be proven can be stated as follows.

Lemma 5.1 (symmetry of algorithmic conversion *[Symmetry.agda]*). *Given a proof of neutral comparison $\Gamma \vdash t \cong_{ne} u : A$ and a context Δ such that $\vdash \Gamma \equiv \Delta$, there exists a type B such that $\Gamma \vdash A \equiv B : s$ and $\Delta \vdash u \cong_{ne} t : B$.*

Proof. This lemma is proven mutually with similar statement for the three other forms of algorithmic conversion, by induction on $\Gamma \vdash t \cong_{ne} u : A$. □

The situation for transitivity introduces a different complication: because of the presence of the rules Cast-refl-L and Cast-refl-R, it is not possible to prove transitivity by a structural induction on the two derivations of algorithmic conversion simultaneously. Indeed, when proving transitivity for $\Gamma \vdash \texttt{cast}\ A\ B\ e\ t \cong_{ne} u : B$ proven by Cast-refl-L against any conversion $\Gamma \vdash u \cong_{ne} v : A$, we need to use transitivity between the sub-proof of $\Gamma \vdash t \cong_{ne} u : B$ and the original proof of $\Gamma \vdash u \cong_{ne} v : A$. Thus the recursive call only decreases on the left argument. But in the dual situation of $\Gamma \vdash t \cong_{ne} u : A$ against $\Gamma \vdash u \cong_{ne} \texttt{cast}\ A\ B'\ e'\ v : A$ proven by Cast-refl-R, we need to do a recursive call that only decreases on the right argument. This is illegal in general, and we need to justify termination some other way. Fortunately, it is enough to consider a notion of size of a derivation, noted `size`, that basically corresponds to the size of the underlying tree of the derivation.

Using this notion of size, we can state the following generalized transitivity lemma.

Lemma 5.2 (transitivity of algorithmic conversion *[Transitivity.agda]*). *For any natural number n, given two proofs of neutral comparison $\pi : \Gamma \vdash t \cong_{ne} u : A$ and $\pi' : \Delta \vdash u \cong_{ne} v : B$ such that $\vdash \Gamma \equiv \Delta$ and $\texttt{size}(\pi) + \texttt{size}(\pi') < n$, there exists a type C such that $\Gamma \vdash C \equiv A : s$, $\Gamma \vdash C \equiv B : s$, $\pi'' : \Gamma \vdash t \cong_{ne} v : C$ and $\texttt{size}(\pi'') \leq \texttt{size}(\pi) + \texttt{size}(\pi')$.*

Proof. This lemma is proven by induction on n. The case for $n = 0$ is trivial because the size condition cannot be met. In the successor case $n = S\ n'$, we do a case analysis on the proofs of algorithmic conversion and do recursive call to transitivity on derivations for which the additional size is smaller than n'. Because of commutativity of addition, the size only needs to decrease on one of the two derivations. Note that it is necessary to control the size of the resulting proof of transitivity because proving transitivity in the case of Cast-cong with Cast-refl-L involves some commutative squares that requires nested calls to transitivity. Indeed, on the one side, we have that $\texttt{cast}\ A\ B\ e\ t \cong \texttt{cast}\ A'\ B'\ e'\ u$ because $A \cong A'$ and $B' \cong B$, and $\texttt{cast}\ A'\ B'\ e'\ u \cong v$ because $A' \cong B'$. We have to show that $\texttt{cast}\ A\ B\ e\ t \cong v$, which requires in particular to show that $A \cong B$. This follows from the transitivity chain $A \cong A' \cong B' \cong B$. □

5.4 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 done by Abel et al. [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 $\Gamma \vdash t \cong_{ne} t' : A$ can be used as termination witness in the same way as $\Gamma \vdash t \cong_{ne} t : A$.

The main difficulty however is that in this new setting, if derivations don't share the exact same structure, it does not necessarily follow that the terms are not convertible. Consider for instance $\texttt{cast}\ A\ B\ e\ t$ against t which cannot have the same derivation proving they are in the diagonal of the algorithmic conversion, yet are convertible by Rule Cast-refl-L. And 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 5.3 (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.*

Proof. Again, the proof uses arguments of sizes, because it is not structurally decreasing on conversion derivations. The main difficulty in the proof is that when the derivations π and π' do not start with the same rule, it is not guaranteed that t and u are not convertible. Indeed, as mentioned above, π may for instance start with the rule Cast-cong that can be eliminated away because we could use Cast-refl-L instead. This means that there are many more potential successful cases

$\Gamma \Vdash_\ell A : s$	A is a reducible type of sort s in context Γ
$\Gamma \Vdash_\ell A \equiv B : s$	A and B are reducibly equal types of sort s in context Γ
$\Gamma \Vdash_\ell t : A : s$	t is a reducible term of type A, which has sort s in context Γ
$\Gamma \Vdash_\ell t \equiv u : A : s$	t and u are reducibly equal terms of type A, which has sort s in context Γ

Fig. 8. The four judgments of the logical relation *[LogicalRelation.agda]*.

that are needed to be tested before being sure that the two terms are not convertible. This also induces a blow up in the size of the proof term which has been challenging to make bearable to Agda's checker. This has been achieved by using some form of open recursion and doing each case as a separate `abstract` lemma to avoid extremely demanding computations in the type checking of the decidability procedure. □

5.5 Tying the Knot: The Logical Relation and the Fundamental Lemma

So far, we have proven decidability of algorithmic conversion of two well-typed terms, assuming reflexivity of algorithmic conversion. But actually, because reflexivity of algorithmic conversion entails strong normalization of the system, proving it requires very involved reasoning on the type system.

We use the logical relation setting already used by Abel et al. [2], Pujet and Tabareau [24, 25] to prove strong normalization and decidability of conversion in various type theories. The logical relation setting is used in two instances: first with the declarative conversion to show the normalization theorem (all well-typed terms with proof-relevant types can be reduced down to whnf by repeatedly applying weak-head reduction), a necessary lemma for the second instance with the algorithmic conversion to show completeness of algorithmic conversion with respect to declarative conversion (and thus reflexivity of algorithmic conversion).

Normalization of type theories cannot be proven by a naive induction on the typing derivations, because of the case of the application rule, where it is not possible to deduce normalization of $t\ u$ from normalization of t and u. Using logical relations is a standard technique to generalize the induction hypothesis and collect several invariants on the system so that the proof by induction on the typing derivation goes through. Concretely, the logical relation is a sophisticated inductive-recursive family consisting of four predicates which mirror the four typing judgments of $\mathsf{CIC}^{\mathsf{obs}}$, presented in Figure 8, defined on a type-by-type basis. Types and terms that satisfy the logical relation are called *reducible*. Note that those predicates are indexed by a level ℓ which reflects the predicative nature of the universe hierarchy on $\mathcal{U}$: reducibility is first defined at level 0 to characterize terms that inhabit the smallest universe $\mathcal{U}_0$, then this relation is used to define reducibility at level 1 for terms that live in $\mathcal{U}_1$ at most, and so on. Therefore, the whole definition is done by induction on ℓ.

We do not recall the definition of the logical relation here as it closely follows the one defined given by Pujet and Tabareau [25]. The only difference is that now observational equality does not compute anymore on its type arguments, so it gets a proper status in the logical relation. Indeed, in $\mathsf{CC}^{\mathsf{obs}}$, the observational equality is seen as an eliminator but here it is considered as a standard type constructor.

Completeness of the logical relation with respect to the type system is called the *fundamental lemma*, which states that every well-typed term is reducible.

Lemma 5.4 (Fundamental lemma (for terms) *[Fundamental.agda]*). *If $\Gamma \vdash t : A : s$, then there is a level ℓ such $\Gamma \Vdash_\ell t : A : s$.*

Proof. The theorem is proven by induction on the typing derivation. It involves many auxiliary lemmas and notions that can be found in the Agda formalization but are not relevant for the presentation of this article. □

A direct consequence of the fundamental lemma for declarative conversion is that any well-typed term has a whnf. This is proven by analyzing the definition of reducibility of a term and seeing that every definition starts by asking that the term reduces to a whnf such that a given property holds.

Once we know weak-head normalization of well-typed terms, and thanks to admissibility of symmetry and transitivity proven in Section 5.3, we have enough properties to replay the fundamental lemma with a definition of the logical relation that uses algorithmic conversion instead of the general conversion. Note that in our formal proof, we follow Abel et al. [2] in factoring the two instances of the fundamental lemma by defining a generic interface for both algorithmic conversion and typed conversion, and using this interface in the definition of the logical relation. In particular, this interface does not contains a generic reflexivity rule, which becomes a consequence of the fundamental lemma.

Theorem 5.5 (Completeness of algorithmic conversion *[Completeness.agda]*). *Given two terms t and u such that $\Gamma \vdash t : A : s$ and $\Gamma \vdash u : A : s$, we have that*

$$\Gamma \vdash t \equiv u : A \Longrightarrow \Gamma \vdash t \cong^{\downarrow} u : A.$$

Given two terms t and u well-typed at $A : s$ in context Γ, we can apply Theorem 5.6 plus the reflexivity of declarative conversion to get that $\Gamma \vdash t \cong^{\downarrow} t : A$ (and similarly for u) so that this provides decidability of declarative conversion.

Theorem 5.6 (Completeness of declarative conversion *[Decidability.agda]*). *Given two terms t and u such that $\Gamma \vdash t : A : s$ and $\Gamma \vdash u : A : s$, knowing whether $\Gamma \vdash t \equiv u : A$ holds is decidable.*

6 Consistency of the Theory

In Section 2.3, we encountered a cautionary tale: combining the inductive scheme of CIC with an observational equality can lead to inconsistencies if we boldly ask for the injectivity of inductive type formers. Drawing lessons from this story, we then devised a more cautious definition of equality, which provides weaker injectivity principles (see Section 4). Now, we would like to make sure that this adjusted definition will not fall victim to another paradox. To this end, we build a model of $\mathsf{CIC}^{\mathsf{obs}}$ 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 by Pujet and Tabareau [25] to general inductive definitions, using the interpretation of inductive definitions that was developed by Timany and Sozeau [29].

6.1 Observational Type Theory in Sets

We work in ZFC set theory with a countable hierarchy of Grothendieck universes $\mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_2$, and so on. 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 need set-theoretic dependent products and dependent sums. We write the former as $(a \in A) \longrightarrow (B\ a)$, and the latter as $(a \in A) \times (B\ a)$ to distinguish them from their type-theoretic counterparts.

Our model is based on the *types-as-sets* interpretation of dependent type theory developed by Dybjer [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 Timany and Sozeau [29] 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 $\mathsf{CIC}^{\mathsf{obs}}$, were it not for two small issues.

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

Fig. 9. Codes for universes, dependent products, and inductive types.

The first issue is the absence of observational equality and of the `cast` operator in the model of Timany and Sozeau. We can easily fix this by interpreting observational equality as the set-theoretic equality and `cast` as the identity function. That way, `cast` satisfies 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. Timany and Sozeau [29] directly interpret the type-theoretic universes as the corresponding Grothendieck universes, which is perfectly fine for CIC. But this does not work for $\mathsf{CIC}^{\mathsf{obs}}$, as we would lose the injectivity of dependent products: consider for instance the two types `Empty` $\to \mathbb{N}$ and `Empty` $\to \mathbb{B}$. Both are interpreted as a singleton set in ZFC, but we can prove that they are different in $\mathsf{CIC}^{\mathsf{obs}}$. To recover this injectivity, we will modify the model and *label* the sets in the universe with additional information that indicates how they were built. This way, the type `Empty` $\to \mathbb{N}$ is interpreted as a singleton set *and* an indication that it is a function type from `Empty` to $\mathbb{N}$, while `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 labeled 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 Gratzer [15]. However, his construction seems difficult to extend with parametrized inductive types—the use of induction-recursion seems to force us to have injectivity on inductive 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.

Figure 9 shows the interpretation for the proof-relevant type formers of $\mathsf{CIC}^{\mathsf{obs}}$. The interpretation that transforms a syntactic object into a semantic object is written $\llbracket \Gamma \vdash \_ \rrbracket_\rho$, 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 Ω are interpreted as their semantic counterparts, with the default label (the empty set). Proof-relevant 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. Lastly, the interpretation of inductive types is a bit more involved. Thankfully, we only need to explain the interpretation for non-indexed inductive types, as we decided to handle indices using Fordism (cf Section 4.2). Thus, consider a non-indexed inductive definition `Ind`, with a vector of parameters $\vec{A}$:

`Inductive` `Ind` $(\vec{a} : \vec{A}) : \mathcal{U}_\ell :=$
| $c_0 : \forall\, (\vec{b} : \vec{B}_0)$, `Ind` $\vec{a}$
| ...
| $c_n : \forall\, (\vec{b} : \vec{B}_n)$, `Ind` $\vec{a}$

In order to interpret this inductive family in our model, we need to associate a pair of sets $\langle \mathsf{IndElem}\,\vec{X}\,,\ \mathsf{IndLabel}\,\vec{X}\,\rangle$ to any vector $\vec{X}$ of elements of the family of sets $\mathsf{fst}(\llbracket\,\vec{A}\,\rrbracket_\rho)$. The definition of the first set follows the interpretation of inductive types given by Timany and Sozeau [29]. Reproducing their construction in full detail would take us too far from the scope of this article, so we simply mention that $\mathsf{IndElem}\,\vec{X}$ is the initial algebra for the endofunctor on $(\mathsf{fst}\,\llbracket\,\vec{A}\,\rrbracket_\rho)$-indexed families corresponding to $\mathtt{Ind}$, evaluated in $\vec{X}$. This initial algebra is well-defined as soon as the definition of $\mathtt{Ind}$ is strictly positive and all the interpretations of the $\vec{B}_i$ are well-defined. This gives us the first projection of $\llbracket\,\Gamma \vdash \mathtt{Ind}\,\vec{X}\,\rrbracket_\rho$, and now we need to define the second projection, which will play the role of the label—i.e., it will only play a part in the definition of equalities between inductive types. Recall from Section 4 that we would like the equality of two instances of $\mathtt{Ind}$ to satisfy:

$$\mathtt{Ind}\,\vec{X} \sim \mathtt{Ind}\,\vec{Y} \quad \longleftrightarrow \quad (\vec{B}_0(\vec{X}), \dots, \vec{B}_n(\vec{X})) \sim (\vec{B}_0(\vec{Y}), \dots, \vec{B}_n(\vec{Y})).$$

In other words, $\mathtt{Ind}$ should be determined up to equality by the types of its constructor arguments. Therefore, a natural option would be to define its label directly as the list of these types:

$$\mathsf{IndLabel}\,\vec{X} \quad = \quad (\llbracket\Gamma, \vec{A} \vdash \vec{B}_0\rrbracket_{(\rho,\vec{X})}, \dots, \llbracket\Gamma, \vec{A} \vdash \vec{B}_n\rrbracket_{(\rho,\vec{X})}). \tag{1}$$

However, remark that $\vec{B}_i$ may contain a recursive call to $\mathtt{Ind}$ (possibly with a different parameter), whose interpretation is defined using $\mathsf{IndLabel}$, so this "definition" is actually an equation that we need to solve. Admittedly, the equation is a bit notation-heavy and abstract because of its generality, so let us first try solving it for the special case of lists. The datatype of lists has two constructors: one without arguments for the empty list, and one with two arguments for appending an element to a list. The equation thus becomes

$$\mathsf{ListLabel}\,X \quad = \quad (\{*\}, (X, \mathsf{ListLabel}\,X)). \tag{2}$$

A simple look at the shape of that equation reveals that it is in fact the definition of a generalized stream, i.e., an infinite tree whose nodes are labeled with sets. It is not too difficult to encode such recursive streams in set theory (e.g., as families of sets indexed by natural numbers) provided they are *productive*, as is the case here. Thus, we define $\mathsf{ListLabel}$ to be the set-theoretic encoding of the stream defined by Equation (2). This solution for the case of lists can be readily generalized to a solution of Equation (1). The resulting stream may have more projections, and recursive calls may change the argument, but it will remain productive nonetheless. We can thus take its set-theoretic encoding as our definition of $\mathsf{IndLabel}$. Remark that the resulting label 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 6.1. *If the inductive definition* $\mathtt{Ind}$ *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* $\llbracket\Gamma \vdash \mathtt{Ind}\,\vec{X}\rrbracket_\rho$ *is well-defined. Furthermore,* $\llbracket\Gamma \vdash \mathtt{Ind}\,\vec{X}\rrbracket_\rho = \llbracket\Gamma \vdash \mathtt{Ind}\,\vec{Y}\rrbracket_\rho$ *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 by Timany and Sozeau [29]. For the sake of completeness, we give an outline of the definition and of the proof of soundness in this section.

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

$$
\begin{array}{rcl}
[\![\Gamma \vdash x]\!]_\rho & := & \rho(x) \\
[\![\Gamma \vdash \lambda(x : F).\, t]\!]_\rho & := & (x \in \mathsf{fst}\, [\![\Gamma \vdash F]\!]_\rho) \mapsto ([\![\Gamma, F \vdash t]\!]_{\rho,x}) \\
[\![\Gamma \vdash t\, u]\!]_\rho & := & [\![\Gamma \vdash t]\!]_\rho([\![\Gamma \vdash u]\!]_\rho) \\
[\![\Gamma \vdash c_i\, \vec{b}]\!]_\rho & := & \\
[\![\Gamma \vdash \mathtt{match}\ \mathsf{t}\ \mathtt{return}\ \mathsf{P}\ \mathtt{with}\ \{c_i\, \vec{b} \Rightarrow t_i\}]\!]_\rho & := & \left.\right\} \text{(as in Lee } \textit{et al.}\text{)} \\
[\![\Gamma \vdash \mathtt{fix}\, f\, \vec{x} := t]\!]_\rho & := & \\
[\![\Gamma \vdash \bot]\!]_\rho & := & \bot \\
[\![\Gamma \vdash \mathtt{\bot\text{-}elim}\, A\, t]\!]_\rho & := & \text{undefined} \\
[\![\Gamma \vdash t \sim_A u]\!]_\rho & := & \top \text{ if } [\![\Gamma \vdash t]\!]_\rho = [\![\Gamma \vdash u]\!]_\rho \\
 & & \bot \text{ otherwise} \\
[\![\Gamma \vdash \mathtt{cast}\, A\, B\, e\, t]\!]_\rho & := & [\![\Gamma \vdash t]\!]_\rho \\
[\![\Gamma \vdash \Pi^{\mathcal{U}_j, \Omega}(y : A).\, B]\!]_\rho & := & \forall x \in (\mathsf{fst}\, [\![\Gamma \vdash A]\!]_\rho),\ [\![\Gamma, A \vdash B]\!]_{\rho,x} \\
[\![\Gamma \vdash \Pi^{\Omega, \Omega}(y : A).\, B]\!]_\rho & := & \forall x \in (\mathsf{val}\, [\![\Gamma \vdash A]\!]_\rho),\ [\![\Gamma, A \vdash B]\!]_{\rho,x}
\end{array}
$$

Fig. 10. Interpretation of contexts and proof-relevant terms of $\mathsf{CIC}^{\mathsf{obs}}$.

Ultimately, our model is defined in terms of *partial* functions from the syntax to the semantics. We use a function $[\![\_]\!]$ that interprets contexts as sets and a function $[\![\Gamma \vdash \_]\!]_\rho$ that interprets terms and types in context Γ as sets indexed by $\rho \in [\![\Gamma]\!]$ (Figure 10). Both functions are mutually defined by recursion on the raw syntax, and we 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. Timany and Sozeau [29] use induction principles instead of `match` and `fix`, but argue that the two are equivalent. A model directly based on the latter has been defined by Lee and Werner [16]. 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 Π^1_ϵ 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:

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

Lemma 6.2 (Weakening). *π_A is the semantic counterpart to the weakening of A: for all terms u, when both sides are well defined, we have:*

$$[\![\Gamma, x : A : s, \Delta \vdash u]\!]_\rho = [\![\Gamma, \Delta \vdash u]\!]_{\pi_A(\rho)}$$

Lemma 6.3 (Substitution). *σ_t is the semantic counterpart to the substitution by t: for all terms u, when both sides are well defined, we have:*

$$[\![\Gamma, \Delta[x := t] \vdash u[x := t]]\!]_\rho = [\![\Gamma, x : A : s, \Delta \vdash u]\!]_{\sigma_t(\rho)}$$

Theorem 6.4 (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 6.5 (Consistency). *There are no proofs of $\bot$ in the empty context.*

We can use our model to go a bit further and prove a canonicity theorem for the inductive type of natural numbers. A natural number is said to be *canonical* if it can be obtained only using zero and successor—for instance, the natural numbers `0` and `S (S 0)` are canonical, while the natural number `cast`($\mathbb{B}$, $\mathbb{N}$, `e`, `true`) is not. In order to prove that every natural number in the empty context is canonical, we will need the following lemma:

Lemma 6.6. *There are no neutral terms in the empty context.*

Proof. Looking at the definition of neutral terms from Figure 5, we see that they must contain either a variable, a proof of $\bot$ or a proof of equality between two incompatible types. Since there are no variables in the empty context and our theory is consistent, it suffices to show that it is impossible to prove an equality between two types with different heads. We can achieve this by adding unique identifiers to the labels in our model, so that two incompatible types are always interpreted as different sets. □

Theorem 6.7 (Canonicity). *Any inhabitant of $\mathbb{N}$ in the empty context is convertible to a canonical natural number.*

Proof. By inspecting the normal forms provided by the normalization theorem, we see that reducible inhabitants of $\mathbb{N}$ are convertible to either zero, a neutral term or the successor of a reducible inhabitant of $\mathbb{N}$. But there are no neutral terms in the empty context, so every natural number is convertible to a canonical natural number by induction on the reducibility proof. □

Similar canonicity theorems can be obtained for the type of Booleans, lists, or any inductive type without indices. However, as soon as indices are involved, the notion of canonical term must be expanded to include `cast` applied to constructors.

7 Implementation in Coq

In this section, we present our implementation of CIC$^{\text{obs}}$ on top of the Coq proof assistant. Starting with version 8.10 (2019), Coq features an impredicative universe `SProp` for definitionally proof-irrelevant types [13], which we use for our universe of propositions Ω. Thus, it remains to define our observational equality in `SProp`, along with its associated `cast` operator and their various rules. Our main tool for this task is the recent extension of Coq with rewrite rules, as implemented by Gilbert et al. [14]. However, these rules are not quite expressive enough for our needs, and therefore we will have to extend them in the process.

7.1 Rewrite rules for CIC$^{\text{obs}}$

We start by defining observational equality as an inductive type family in `SProp`, so that all the infrastructure and the tactics that apply to the usual inductive equality (such as `rewrite`, `symmetry`...) may still work with the observational equality:

```
Inductive obseq (A : Type) (a : A) : A ⟶ SProp :=
| obseq_refl : obseq A a a.
Notation "a ∼ b" := obseq _ a b.
```

This definition already provides us with a transport operator for proof-irrelevant predicates. Indeed, Coq automatically generates eliminators *à la* Martin-Löf for each inductive definition, which are defined in terms of the primitive `match` and `fix` operators. Here, our inductive family is defined in `SProp`, which means that using `match` on a prof of `obseq` is only allowed when the return predicate is itself defined in `SProp`. As a consequence, the Martin-Löf style eliminator generated by Coq is restricted to propositional predicates, which corresponds precisely to our transport operator. Thus, it remains to define the typecasting operator to handle elimination in `Type`.

In order to define typecasting along with its various reduction rules, we use *rewrite rules*, which were introduced by Cockx et al. [10] and recently implemented in Coq by Gilbert et al. [14]. This extension of Coq allows us to extend the language with a new constant using the keyword `Symbol`, and then add reduction rules for the symbol by using the `Rewrite Rule` command. Thus, we start by defining the `cast` operator and its accompanying notation:

```
Symbol cast : ∀ (A B : Type), A ∼ B ⟶ A ⟶ B.
Notation "e # a" := cast _ _ e a.
```

Then, we need to equip every type former of Coq with the logical rules that characterize the observational equality on this type and the rewrite rules for `cast`. Let us have a look at the case of dependent function types first: on the logical side, we add the two projections out of an equality between dependent function types Eq-Π_1 and Eq-Π_2, as well as the extensionality of functions:

```
Parameter seq_∀₁ : ∀ {A A' B B'}, (∀ (x : A), B x) ∼ (∀ (x : A'), B' x) ⟶ A' ∼ A.
Parameter seq_∀₂ : ∀ {A A' B B'} e (x : A'), B (seq_∀₁ e # x) ∼ B' x.
Parameter funext : ∀ {A B} (f g : ∀ (x : A), B x), (∀ x, f x ∼ g x) ⟶ f ∼ g.
```

Since these three terms are irrelevant, we postulate them as parameters and not symbols. After all, irrelevant terms cannot block computations, so there is no need to equip them with rewrite rules. Finally, the reduction rule for casting a dependent function (Rule Cast-Π-red) is added as follows:

```
Rewrite Rule cast∀ :=
 cast (∀ (x : ?A), ?B) (∀ (x : ?A'), ?B') ?e ?f   ⤖   fun x ⇒ seq_∀₂ ?e x # ?f (seq_∀₁ ?e # x).
```

The term on the left of the ⤖ symbol is the *pattern* of the rewrite rule. Whenever the reduction engine encounters a term that matches this pattern in head position, it reduces to the right-hand side, substituting the existential variables (which are marked with "?") with corresponding subterms of the matched term. We can implement the observational rules for the sort `SProp` in a similar manner: we add one parameter for the principle of proposition extensionality, as well as a rewrite rule which states that using `cast` from `SProp` to `SProp` amounts to the identity.

```
Parameter propext : ∀ {P Q : SProp}, (P ↔ Q) ⟶ P ∼ Q
Rewrite Rule cast_sprop := cast SProp SProp ?e ?P   ⤖   ?P.
```

7.2 Rewriting with Equations

Rewrite rules are a convenient tool for implementing our system without having to modify the kernel of Coq, thereby minimizing the risk of introducing critical bugs. However, the rules as implemented by Gilbert et al. [14] are not quite powerful enough for our purposes, in particular because they do not provide a way to implement the rule Cast-Refl.

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

In order to implement this rule according to the decidability proof of Theorem 5.6, we need to modify the conversion checking algorithm of Coq to deal with casts on reflexivity proofs. However, we only have a hook to the reduction algorithm. To solve this tension, we remark that there is an alternative way to treat Rules Cast-refl-L and Cast-refl-R by using rewriting. Indeed, in those rules, only one side of the equality is modified (by removing a cast) and the other is unchanged. This means that the alternative neutral reduction rule

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

together with rules that say that neutral reduction preserves neutral conversion (on both arguments) performs the same check:

$$\frac{\Gamma \vdash t \Rightarrow_{ne} t' : A \qquad \Gamma \vdash t' \cong_{ne} u : A}{\Gamma \vdash t \cong_{ne} u : A}\ \textsc{Cast-red-L} \qquad \frac{\Gamma \vdash u \Rightarrow_{ne} u' : A \qquad \Gamma \vdash t \cong_{ne} u' : A}{\Gamma \vdash t \cong_{ne} u : A}\ \textsc{Cast-red-R}$$

The only difficulty with this presentation is that the reduction rule Cast-refl-red requires conversion checking as a precondition. To fulfill this requirement, we extend the syntax of rewrite rules with *equations*:

```
Rewrite Rule cast_refl := [?A = ?B] ⊢ cast ?A ?B ?e ?t  ⤔  ?t.
```

Now, whenever the reduction machine encounters a term that matches the pattern, it will perform a convertibility check between the two sides of the equations before performing the rewrite. In order to add this call to the conversion checker during reduction, we reused part of the implementation of UIP in `SProp` done by Gilbert et al. [13, §4.4].

Actually, equations also play a role in the computation of `cast` on the universe of types. In Coq, the universe hierarchy `Type` is indexed by universe levels which are traditionally hidden to the user, but which play an important part in keeping the theory consistent. If we were to add the naive reduction rule `cast Type Type e t ⤔ t` without making sure that the universe levels match, then we would allow coercions from large universes to small universes in an inconsistent context, which would break the decidability of typechecking. The correct rule must therefore check an equation between levels before firing, which one might write as follows:

```
Rewrite Rule cast_type@{u v} := [u = v] ⊢ cast Type@{u} Type@{v} ?e ?t  ⤔  ?t,
```

where the @{u} annotations make the hidden universe levels explicit. But this rule is in fact subsumed by `cast-refl`, and thus there is no need to add it. In fact, our syntax does not really support equations between universe levels, they can only be compared indirectly through an equation between terms.

7.3 Generating Equality and Cast for Parametrized Inductive Types

We now turn our attention to inductive types. As with all the other type formers, they should be equipped with rules for observational equality and for the `cast` operator. However, unlike the type formers that we have covered so far, the set of inductive types is not fixed in advance. Instead, Coq provides the user with a scheme for inductive definitions, that they may use to extend the language with a new type family whenever they need it. As a result, we cannot add the rules for inductives types once and for all; we are forced to generate them dynamically whenever the user defines a new inductive type. To this end, we modified the behavior of the `Inductive` command. Traditionally, this command adds the new inductive type to the global environment and generates eliminators from the `match` and `fix` primitives. Now, once the user sets the new flag `Set Observational Inductives`, this command additionally generates all the observational data that we described in Section 4.

Let us first look at how this works for inductive definitions that have *parameters* but no *indices*. In that case, we automatically generate a family of projections that turn equalities between two instances of the inductive type into equalities between the types of the constructor arguments, as well as a family of rewrite rules for `cast` applied to a constructor. For instance, consider the following definition of the type of lists:

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

In that case, the constructor `nil` has no arguments, and the constructor `cons` has two, namely `A` and `list A`. Thus, our implementation automatically generates two projections (one for each constructor argument), as well as two rewrite rules (one for each constructor):

```
Parameter obseq_cons₀ : ∀ A A₀, list A ∼ list A₀ ⟶ A ∼ A₀.
Parameter obseq_cons₁ : ∀ A A₀, list A ∼ list A₀ ⟶ list A ∼ list A₀.
Rewrite Rule cast_nil :=
  cast (list _) (list ?A₀) ?e (nil ?A)  ↣  nil ?A₀.
Rewrite Rule cast_cons :=
  cast (list _) (list ?A₀) ?e (cons ?A ?a ?l)  ↣
  let a₀ := (obseq_cons₀ ?A ?A₀ ?e) # ?a in
  let l₀ := (obseq_cons₁ ?A ?A₀ ?e) # ?l in
  cons ?A₀ a₀ l₀
```

Remark that the projection `obseq_cons₁` is not particularly useful, and it could be interesting to try and refine our scheme to simplify away trivial projections. But we leave this for future work, and for now we turn ourselves to the more pressing issue of inductive types with indices.

7.4 Automating the Fording Translation

As we explained in Section 2.2, indexed inductive definitions gain new inhabitants in presence of the observational equality, because casts on indices cannot reduce to a constructor in general. This phenomenon is unavoidable, and in fact it also appears in the slightly different setting of cubical type theories, where Cavallo and Harper [9] handle these new terms with an operator called *fcoe* that can be applied to a constructor in order to build a new canonical inhabitant of the inductive, but with a different index. Our approach is based on the Fordism encoding, which is a somewhat more principled way to arrive at the same result. Let us look at it in action on the type of length-indexed vectors:

```
Inductive vect (A : Type) : ℕ ⟶ Type :=
| vnil : vect A 0
| vcons : ∀ (a : A) (m : ℕ) (v : vect A m), vect A (S m).
```

Our implementation starts by generating *forded* constructors which explicitly state the equality between integers that is implied by the indexing. These forded constructors are equivalent to *fcoe* (`vnil`) and *fcoe* (`vcons`) with the notation of Cavallo and Harper [9].

```
Symbol vnil_cast : ∀ (A : Type) (n : ℕ), n ∼ 0 ⟶ vect A n.
Symbol vcons_cast : ∀ (A : Type) (n : ℕ) (a : A) (m : ℕ) (v : vect A m), n ∼ S m ⟶ vect A n.
```

Now the "standard" constructors `vnil` and `vcons` are supposed to represent the same vector as a forded constructor whose last argument is equal to a proof by reflexivity. At that point, we could decide to remove the standard constructors and work exclusively with the better-behaved forded constructors. However, we want our observational Coq to remain as compatible as possible with existing developments, and thus we go the extra mile and keep both kinds of constructors, with rewrite rules that ensure the forded constructors reduce to the standard ones whenever their indices match those of the corresponding standard constructor:

```
Rewrite Rule vnil_cast_refl := [ ?n = 0 ] ⊢ vnil_cast ?A ?n ?e  ↣  vnil ?A.
Rewrite Rule vcons_cast_refl := [ ?n = S ?m ] ⊢ vcons_cast ?A ?n ?a ?m ?v ?e  ↣  vcons ?A ?a ?m ?v.
```

Finally, in order to complete the Fordism translation, it remains to add rules for the interaction of `match` and `fix` with the forded constructors. For the `fix` operator, the interaction is quite simple: we only need to make sure that fixpoints unfold on the forded constructors as if they were regular constructors. To do this, we equip every symbol with a flag that states whether the symbol causes unfolding of fixpoints in the reduction machine, and we set this flag to `true` for the forded constructors. The interaction with `match`, on the other hand, is a bit more complicated. Basically, doing pattern-matching on a forded constructor should reduce to a cast of the corresponding branch:

```
Rewrite Rule match_vnil_cast :=
  match vnil_cast ?A ?n ?e as v in vect _ m return ?P with
  | vnil _ ⇒ ?t
  | vcons _ a m v ⇒ _
  end  ↣
  let e := sym (ap_ty2 ?P ?e obseq_refl) in
  cast ?P@{m = 0 ; v = vnil ?A} ?P@{m = ?n ; v = vnil_cast ?A ?n ?e} e ?t.
```

Here, we use the extended syntax for `match` to signify that the return predicate `?P` depends on both the index `m` (of type ℕ) and the vector `v` (of type `vect ?A m`). In order to use the branch `?t` which corresponds to `vnil`, we must provide a proof of equality between the return predicate `?P` instantiated with `0` and `vnil` on the one hand, and `?P` instantiated with `?n` and `vnil_cast` on the other hand. We can get this equality from the argument `?e` of the forded constructor (which has type `?n ∼ 0`) but in practice, performing this sort of dependent transport is quite tedious. Therefore, we define in advance a family of operators $\{\text{ap\_ty}\}_i$ that perform these dependent transport along a telescope of size i. All things considered, this difficulty would be most elegantly resolved by switching to a *heterogeneous* equality *à la* Altenkirch et al. [5], but we stick to the homogeneous equality out of compatibility concerns.

Now that we have the forded constructors, we can apply the recipe for inductive types without indices from the previous section. We generate a projection for every argument of the *forded* constructors:

```
Parameter obseq_vnil₀ : ∀ A A₀ n n₀, vect A n ∼ vect A₀ n₀ ⟶ (n ∼ 0) ∼ (n₀ ∼ 0).
Parameter obseq_vcons₀ : ∀ A A₀ n n₀, vect A n ∼ vect A₀ n₀ ⟶ A ∼ A₀.
Parameter obseq_vcons₁ : ∀ A A₀ n n₀ (e : vect A n ∼ vect A₀ n₀),
  ∀ (a : A), let a₀ := obseq_vcons₀ A A₀ n n₀ e # a in ℕ ∼ ℕ.
Parameter obseq_vcons₂ : ∀ A A₀ n n₀ (e : vect A n ∼ vect A₀ n₀),
  ∀ (a : A), let a₀ := obseq_vcons₀ A A₀ n n₀ e # a in
  ∀ (m : ℕ), let m₀ := obseq_vcons₁ A A₀ n n₀ e a # m in vect A n ∼ vect A₀ n₀.
```

```
Parameter obseq_vcons₃ : ∀ A A₀ n n₀ (e : vect A n ~ vect A₀ n₀),
  ∀ (a : A), let a₀ := obseq_vcons₀ A A₀ n n₀ e # a in
  ∀ (m : ℕ), let m₀ := obseq_vcons₁ A A₀ n n₀ e a # m in
  ∀ (v : vect A n), let v₀ := obseq_vcons₂ A A₀ n n₀ e a m # v in (n ~ S m) ~ (n₀ ~ S m₀).
```

And on top of that, we get rewrite rules for cast on forded constructors. Since we kept the regular constructors around, we must also equip them with rewrite rules for the cast operator. In that case, we simply re-use the rule for the forded constructor, replacing the last argument with a proof by reflexivity:

```
Rewrite Rule cast_vnil_cast :=
  cast (vect _ _) (vect ?A0 ?n0) ?e (vnil_cast ?A ?n ?o) ↣
   let o0 := castℙ (?n ~ 0) (?n0 ~ 0) (obseq_cnil₀ ?A ?A0 ?n ?n0 ?e) ?o in
   vnil_cast ?A0 ?n0 o0.
Rewrite Rule cast_vnil :=
  cast (vect _ _) (vect ?A0 ?n0) ?e (vnil ?A) ↣
   let o0 := castℙ (0 ~ 0) (?n0 ~ 0) (obseq_cnil₀ ?A ?A0 0 ?n0 ?e) obseq_refl in
   vnil_cast ?A0 ?n0 o0.
```

And likewise for cast of vcons. Note that because cast does not apply between two inhabitants of SProp, we use a computationally irrelevant replacement castℙ to transport equality proofs. This operator has a straightforward definition in terms of the eliminator for obseq.

7.5 Subtleties around Universe Levels

Following the Coq tradition, we have hidden all the universe levels in our presentation of the implementation. However, the management of universe levels with rewrite rules is all but trivial. In particular, we have to make sure that all the universe levels that appear on the right-hand side of a rule also appear on the left-hand side, otherwise we would need to synthesize them during reduction. Let us have a brief look at the problems this might cause.

First, remark that in the right-hand side of the rule match_nil_vcast, we are using an observational equality between two instances of ?P, where ?P is the return predicate of an arbitrary match statement. But in order to do this, it is in fact necessary to know the type of ?P, as this type is an implicit argument of the observational equality. And since ?P is itself a type, its type should be Type@{u} for some universe level u. Unfortunately, this universe level appears nowhere on the left-hand side of the rewrite rule, which breaks our golden rule. We fixed this by modifying the kernel of Coq to annotate all match statements with the universe level of the return predicate. This way, the level u appears on the left-hand side of the rewrite rule, as part of the match statement. Unfortunately, this is not enough to solve our universe issues. The observational equality type requires a *second* universe level! Indeed, once we explicitly display all the universe levels, the definition of the observational equality looks like this:

```
Inductive obseq@{v} (A : Type@{v}) (a : A) : A ⟶ SProp :=
| obseq_refl : obseq A a a.
```

Thus, if we want to set A := Type@{u}, we must find another universe level v which is strictly larger than u. But no such level appears on the left-hand side of rule match_nil_vcast, and we are stuck once again. Thankfully, Sozeau [27] recently implemented *algebraic universes* for the Coq proof assistant using the algorithm recently proposed by Bezem and Coquand [8]. This new feature allows algebraic expressions such as u+1 to be used in universe instances, which does solve our problem: now all the universe variables that appear on the right-hand side of match_nil_vcast also appear on the left-hand side.

7.6 Quotient Types

Observational type theories provide an ideal framework for quotient types, since we can use the observational equality to specify the equality of a type to be any desired relation—as long as said relation is preserved by all the constructs of type theory. Thus, we extend our implementation of CIC^{obs} in Coq with basic quotient types: given a type `A` and a binary relation `R` on `A`, we define `Quotient A R` as having the same elements as `A`, but with the additional requirement that any two elements related by `R` become equal in `Quotient A R`.

```
Symbol Quotient : ∀ (A : Type) (R : A ⟶ A ⟶ SProp), Type.
Symbol quo : ∀ (A : Type) (R : A ⟶ A ⟶ SProp) (a : A), Quotient A R.
Parameter quo_eq : ∀ (A : Type) (R : A ⟶ A ⟶ SProp) (a b : A), R a b ⟶ (quo A R a) ~ (quo A R b).
```

Then, the elimination principle for `Quotient A R` encodes the universal property of quotients: in order to define a function of type `Quotient A R ⟶ X`, it is sufficient to define a function `f : A ⟶ X` that sends any two elements in `R` to equal images. Furthermore, the resulting function should reduce to `f` when supplied with elements of the form `quo a`. The dependent version of this principle can be stated as follows:

```
Symbol Quotient_rect : ∀ (A : Type) (R : A ⟶ A ⟶ SProp) (P : Quotient A R ⟶ Type)
                        (Pquo : ∀ (a : A), P (quo A R a))
                        (Prel : ∀ (a b : A) (H : R a b), cast _ _ (ap P (quo_eq A R a b H)) (Pquo a) ~ Pquo b)
                        (x : Quotient A R), P x.
Rewrite Rule quo_rew := Quotient_rect _ _ ?P ?Pquo ?Prel (quo ?A ?R ?a) ⤖ ?Pquo ?a.
```

We also add an eliminator for `SProp` valued predicates. This one is a bit simpler, since it does not need to make sure that `Pquo` sends elements in `R` to equal images thanks to proof irrelevance, and it does not need a computation rule either.

```
Parameter Quotient_sind : ∀ (A : Type) (R : A ⟶ A ⟶ SProp) (P : Quotient A R ⟶ SProp)
                           (Pquo : ∀ (a : A), P (quo A R a))
                           (x : Quotient A R), P x.
```

These operators provide the introduction, elimination, and computation rules for our quotient types. Now, it remains to add rules that describe their interactions with the observational equality and the `cast` operator. Using `Quotient_rect` and `Quotient_sind`, we can already show that the observational equality between two elements of `Quotient A R` is equivalent to the reflexive, symmetric and transitive closure of the relation `R`. Therefore, the observational equality on quotient types is fully characterized by their eliminator, just like for the inductive types, and thus we will only need a rule that helps us describe the equality between `Quotient A R` and `Quotient A' R'`:

```
Parameter obseq_Quotient : ∀ A A' R R', Quotient A R ~ Quotient A' R' ⟶ A ~ A'.
```

This rule provides injectivity for the support `A`, which is strongly anti-univalent, but expected for a set-truncated type theory such as CIC^{obs}. Note that we have chosen not to have injectivity for the relation `R` (or for its transitive closure). We don't need it, so we might as well let the user decide if they want to postulate such a principle. Lastly, we add the computation rule for the `cast` operator on quo:

```
Rewrite Rule cast_quo :=
  cast (Quotient _ _) (Quotient ?A' ?R') ?e (quo ?A ?R ?a)
  ⤖ quo ?A' ?R' (cast ?A ?A' (obseq_Quotient ?e) ?a).
```

This concludes our overview of quotient types. Readers who are familiar with the Lean proof assistant may have noticed that our approach is very similar to the one used in Lean's quotient types and may thus wonder whether it also introduces non-canonical terms. Thankfully, this is not the case: the reason why quotients break the canonicity property in Lean is because they can be used to show function extensionality, and the $\mathcal{J}$ eliminator will get stuck when asked to coerce between two types that are extensionally equal but not convertible.[6] In $\mathsf{CIC}^{\mathrm{obs}}$ however, the $\mathcal{J}$ eliminator is defined in terms of the `cast` operator, which is compatible with function extensionality. In fact, the reader may convince themselves that it is not too difficult to adapt our canonicity proof to support quotient types.

8 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. Adaptation 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 implemented 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 by Vezzosi et al. [31] for Cubical Agda.

9 Data Availability Statement

The Agda companion formalization is available both on GitHub and as a long-term archived artifact [26].
The prototype implementation on top of the Coq proof assistant is available at https://github.com/loic-p/coq.

References

[1] Andreas Abel and Thierry Coquand. 2020. Failure of normalization in impredicative type theory with proof-irrelevant propositional equality. *Logical Methods in Computer Science* 16, 2 (Jun. 2020). DOI: https://doi.org/10.23638/LMCS-16(2:14)2020

[2] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of conversion for type theory in type theory. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 23 (Jan. 2018), 29 pages. DOI: https://doi.org/10.1145/3158111

[3] Guillaume Allais, Conor McBride, and Pierre Boutillier. 2013. 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)*. ACM, New York, NY, 13–24. DOI: https://doi.org/10.1145/2502409.2502411

[4] Thorsten Altenkirch and Conor McBride. 2006. Towards observational type theory. Retrieved from http://www.strictlypositive.org/ott.pdf

[5] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. 2007. Observational equality, now! In *Proceedings of the Workshop on Programming Languages meets Program Verification (PLPV '07)*. 57–68. DOI: https://doi.org/10.1145/1292597.1292608

[6] Bob Atkey. 2017. Simplified observational type theory. Retrieved from https://github.com/bobatkey/sott

[6]For instance, one can define a noncanonical integer in Lean by coercing an inhabitant of $\forall\ (n : \mathbb{N})$, `Finset` $(1 + n)$ to the type $\forall\ (n : \mathbb{N})$, `Finset` $(n + 1)$, and then applying the resulting function to zero.

[7] Henk P. Barendregt. 1993. Lambda calculi with types. In *Handbook of Logic in Computer Science (Vol. 2) Background: Computational Structures,* Oxford University Press, Inc., 117–309.

[8] Marc Bezem and Thierry Coquand. 2022. Loop-checking and the uniform word problem for join-semilattices with an inflationary endomorphism. *Theoretical Computer Science* 913 (2022), 1–7. DOI: https://doi.org/10.1016/j.tcs.2022.01.017

[9] Evan Cavallo and Robert Harper. 2019. Higher inductive types in cubical computational type theory. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 1 (Jan. 2019), 27 pages. DOI: https://doi.org/10.1145/3290314

[10] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. 2021. The taming of the tew: A type theory with computational assumptions. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 60 (Jan. 2021), 29 pages. DOI: https://doi.org/10.1145/3434341

[11] The Coq Development Team. 2024. *The COQ proof assistant reference manual.* Zenodo, DOI: https://doi.org/10.5281/zenodo.14542673

[12] Peter Dybjer. 1991. *Inductive Sets and Families in Martin-Löf's Type Theory and Their Set-Theoretic Semantics.* Cambridge University Press, 280–306.

[13] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional proof-irrelevance without K. *Proceedings of the ACM on Programming Languages* 3 (Jan. 2019), 1–28. DOI: https://doi.org/10.1145/3290316

[14] Gaëtan Gilbert, Yann Leray, Nicolas Tabareau, and Théo Winterhalter. 2023. The Rewster: The Coq proof assistant with rewrite rules. In *29th International Conference on Types for Proofs and Programs.* Retrieved from https://github.com/Yann-Leray/coq#readme

[15] Daniel Gratzer. 2022. An inductive-recursive universe generic for small families. arXiv: 2202.05529. Retrieved from https://doi.org/10.48550/arXiv.2202.05529

[16] Gyesik Lee and Benjamin Werner. 2011. Proof-irrelevant model of CC with predicative induction and judgmental equality. *Logical Methods in Computer Science* 7, 4 (Nov. 2011). DOI: https://doi.org/10.2168/lmcs-7(4:5)2011

[17] Meven Lennon-Bertrand. 2021. Complete bidirectional typing for the calculus of inductive constructions. In *Proceedings of the 12th International Conference on Interactive Theorem Proving (ITP '21).* Liron Cohen and Cezary Kaliszyk (Eds.), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193, Schloss Dagstuhl – Leibniz-Zentrum für Informatik. DOI: https://doi.org/10.4230/LIPIcs.ITP.2021.24

[18] Meven Lennon-Bertrand. 2022. *Bidirectional Typing for the Calculus of Inductive Constructions.* Ph. D. Dissertation. Nantes Université. Retrieved from https://theses.hal.science/tel-03848595

[19] Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73*, H. E. Rose and J. C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 73 – 118. DOI: https://doi.org/10.1016/S0049-237X(08)71945-1

[20] Conor McBride. 2000. *Dependently Typed Functional Programs and their Proofs.* Ph. D. Dissertation. University of Edinburgh.

[21] Conor McBride. 2011. Hier Soir, an OTT hierarchy. Blog post. Retrieved from https://mazzo.li/epilogue/index.html%3Fp=1098.html

[22] Alexandre Miquel. 2001. *Le Calcul des Constructions Implicites.* Ph. D. Dissertation. Université Paris Diderot. Retrieved from https://github.com/coq-contribs/paradoxes/blob/master/Russell.v

[23] Christine Paulin-Mohring. 1993. Inductive definitions in the system Coq rules and properties. In *Typed Lambda Calculi and Applications*, Marc Bezem and Jan Friso Groote (Eds.), Springer Berlin, Heidelberg, 328–345.

[24] Loïc Pujet and Nicolas Tabareau. 2022. Observational Equality: Now For Good. *Proceedings of the ACM on Programming Languages* 6, POPL (Jan. 2022), 1–29. DOI: https://doi.org/10.1145/3498693

[25] Loïc Pujet and Nicolas Tabareau. 2023. Impredicative observational equality. *Proceedings of the ACM on Programming Languages* 7, POPL, Article 74 (Jan. 2023), 26 pages. DOI: https://doi.org/10.1145/3571739

[26] Loïc Pujet and Nicolas Tabareau. 2024. *A logical relation for observational equality meets CIC.* Retrieved from https://doi.org/10.5281/zenodo.10499152

[27] Matthieu Sozeau. 2024. Algebraic universes and new solving algorithm. Retrieved from https://github.com/coq/coq/pull/18903

[28] Andrew Swan. 2016. An algebraic weak factorisation system on 01-substitution sets: A constructive proof. *Journal of Logic and Analysis* (2016). DOI: https://doi.org/10.4115/jla.2016.8.1

[29] Amin Timany and Matthieu Sozeau. 2017. *Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC).* Research Report RR-9105. KU Leuven, Belgium ; Inria Paris, 32 pages. Retrieved from https://inria.hal.science/hal-01615123

[30] Benno van den Berg and Richard Garner. 2010. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society* 102, 2 (Oct. 2010), 370–394. DOI: https://doi.org/10.1112/plms/pdq026.

[31] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical agda: A dependently typed programming language with univalence and higher inductive types. *Proceedings of the ACM on Programming Languages* 3, ICFP, Article 87 (Jul. 2019), 29 pages. DOI: https://doi.org/10.1145/3341691

[32] Benjamin Werner. 2006. On the strength of proof-irrelevant type theories. In *Automated Reasoning*. Ulrich Furbach and Natarajan Shankar (Eds.), Springer Berlin, Heidelberg, 604–618.

Received 26 April 2024; revised 1 December 2024; accepted 4 February 2025