Definitional Proof-Irrelevance without K*

GAËTAN GILBERT, Inria, France
JESPER COCKX, Chalmers / Gothenburg University, Sweden
MATTHIEU SOZEAU, Inria and IRIF, France
NICOLAS TABAREAU, Inria, France

Definitional equality—or conversion—for a type theory with a decidable type checking is the simplest tool to prove that two objects are the same, letting the system decide just using computation. Therefore, the more things are equal by conversion, the simpler it is to use a language based on type theory. Proof-irrelevance, stating that any two proofs of the same proposition are equal, is a possible way to extend conversion to make a type theory more powerful. However, this new power comes at a price if we integrate it naively, either by making type checking undecidable or by realizing new axioms—such as uniqueness of identity proofs (UIP)—that are incompatible with other extensions, such as univalence. In this paper, taking inspiration from homotopy type theory, we propose a general way to extend a type theory with definitional proof irrelevance, in a way that keeps type checking decidable and is compatible with univalence. We provide a new criterion to decide whether a proposition can be eliminated over a type (correcting and improving the so-called singleton elimination of Coq) by using techniques coming from recent development on dependent pattern matching without UIP. We show the generality of our approach by providing implementations for both Coq and Agda, both of which are planned to be integrated in future versions of those proof assistants.

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

Additional Key Words and Phrases: type theory, proof assistants, proof irrelevance

ACM Reference Format:
Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. 2019. Definitional Proof-Irrelevance without K. *Proc. ACM Program. Lang.* 3, POPL, Article 3 (January 2019), 28 pages. https://doi.org/10.1145/3290316

1 INTRODUCTION

Proof-irrelevance, the principle that any two proofs of the same proposition are equal, is at the heart of the Coq extraction mechanism [Letouzey 2004]. In the Calculus of Inductive Constructions (CIC), the underlying theory of the Coq proof assistant, there is an (impredicative) sort Prop that is used to characterize types that can be seen as propositions—as opposed to types whose inhabitants have a computational meaning, which live in the predicative sort Type. This static piece of information is used to extract a Coq formalization to a purely functional program, erasing safely all the parts involving terms that inhabit propositions, because a program can not use those terms in computationally relevant ways.

*This work is supported by the CoqHoTT ERC Grant 64399.

Authors' addresses: Gaëtan Gilbert, Inria, Gallinette Project-Team, Nantes, France; Jesper Cockx, Chalmers / Gothenburg University, Gothenburg, Sweden; Matthieu Sozeau, Inria and IRIF, Pi.R2 Project-Team, Paris, France; Nicolas Tabareau, Inria, Gallinette Project-Team, Nantes, France.

2475-1421/2019/1-ART3
https://doi.org/10.1145/3290316

In order to concretely guarantee that no computation can leak from propositions to types, Coq uses a restriction of the dependent elimination from inductive types in Prop into predicates in Type. This restriction, called the *singleton elimination* condition, checks that an inductive proposition can be eliminated into a type only when:

(1) the inductive proposition has at most one constructor,
(2) all the arguments of the constructor are themselves non-computational, *i.e.* are in Prop.

However, in the current version of Coq, singleton elimination is the price to pay to be compatible with proof irrelevance, but there is no payback. This means that although two proofs of the same proposition can not be relevantly distinguished in the system, one can not use the fact that they are equal in the logic.

Consider for instance a working mathematician or computer scientist who defines bounded integers in the following way:

```
Definition boundedℕ (k : ℕ) : Type := { n : ℕ & n ≤ k }.
```

Here boundedℕ k is the dependent sum of an integer n together with a proof (in Prop) that n is below k, using the inductive definition

```
Inductive ≤ : ℕ → ℕ → Prop :=
  ≤0 : ∀ n, 0 ≤ n
| ≤S : ∀ m n, m ≤ n → S m ≤ S n.
```

Then, our user defines an implicit coercion from boundedℕ to ℕ so that she can work with bounded integers almost as if they were integers, apart from additional proofs of boundedness.

```
Coercion boundedℕ_to_ℕ : boundedℕ >-> ℕ.
```

For instance, she can define the addition of bounded integers by simply relying on the addition of integers, modulo a proof that the result is still bounded:

```
Definition add {k} (n m : boundedℕ k) (e : n + m ≤ k) : boundedℕ k := ( n + m ; e).
```

Unfortunately, when it comes to reasoning on bounded integers, the situation becomes more difficult. For instance, the fact that addition of bounded natural numbers is associative

```
Definition bounded_add_associativity k (n m p: boundedℕ k) e₁ e₂ e'₁ e'₂ :
  add (add n m e₁) p e₂ = add n (add m p e'₁) e'₂.
```

does not directly follow from associativity of addition on integers, as it additionally requires a proof that e_2 equals e'_2 (modulo the proof of associativity of addition of integers). This should be true because they are two proofs of the same proposition, but it does not follow automatically. Instead, the user has to prove proof-irrelevance for ≤ *manually*. This can be proven (using Agda style pattern matching of the Equations plugin[1]) by induction on the proof of m ≤ n:

```
Equations ≤_hprop {m n}  (e e' : m ≤ n) : e = e' :=
  ≤_hprop (≤0 _) (≤0 _) := eq_refl;
  ≤_hprop (≤S _ _ e) (≤S n m e') := ap (≤S n m) (≤_hprop e e').
```

Note the use of functoriality of equality ap : ∀ f, x = y → f x = f y which requires some explicit reasoning on equality. Even if proving associativity of addition was more complicated than expected, our user is still quite lucky to deal with an inductive type that is actually a mere proposition, in the sense of Homotopy Type Theory (HoTT) [Univalent Foundations Program 2013]. Indeed, ≤ satisfies the propositional (as opposed to definitional, which holds by computation) version of proof

[1] http://mattam82.github.io/Coq-Equations/

irrelevance, as expressed by the ≤_hprop lemma. For an arbitrary inductive type in Prop, there is no reason anymore why it would be a mere proposition, and thus proof irrelevance, even in its propositional form, can not be proven and must be stated *as an axiom*.

In a setting where proof-irrelevance for Prop is built in, it becomes possible to define an operation on types which turns any type into a definitionally proof irrelevant one and thus makes explicit in the system the fact that a value in Squash T will never be used for computation.

```
Definition Squash (T : Type) : Prop := ∀ P : Prop, (T → P) → P.
```

This operator satisfies the following elimination principle (reduced to proposition) given by

```
∀ (T : Type) (P : Prop), (T → P) → Squash T → P.
```

which means that it corresponds to the propositional truncation [Univalent Foundations Program 2013] of HoTT, originally introduced as the bracket type by Awodey and Bauer [Awodey and Bauer 2004].

The importance of (definitional) proof irrelevance to simplify reasoning has been noticed for a long time, and various works have tried to promote its implementation in a proof assistant based on type theory [Pfenning 2001; Werner 2008]. However, this has never been achieved, mainly because of the fundamental misunderstanding that *singleton elimination was the right constraint on which propositions can be eliminated into a type*. Indeed, one can think of singleton elimination as a *syntactic* approximation of which types are naturally mere propositions, and thus can be eliminated into an arbitrary type without leaking a piece of computation or without implying new axioms. But, singleton elimination does not work for indexed datatypes, as for instance it applies to the equality type of Coq

```
Inductive eq (A : Type) (x : A) : A → Prop := eq_refl : eq A x x
```

If proof irrelevance holds for the equality type, every equality has at most one proof, which is known as Uniqueness of Identity Proofs (UIP). Therefore, assuming proof irrelevance together with the singleton elimination enforces a new axiom in the theory, which is for instance incompatible with the univalence axiom from HoTT. This may not seem too problematic to some, but another consequence of singleton elimination in presence of definitional proof irrelevance is that it breaks decidability of conversion. For instance, the accessibility predicate:

```
Inductive Acc (A : Type) (R : A → A → Prop) (x : A) : Prop :=
  Acc_intro : (∀ y : A, R y x → Acc R y) → Acc R x
```

satisfies the singleton elimination criterion but implementing definitional proof irrelevance for it leads to an undecidable conversion and thus an undecidable type checker (we come back to this point in more detail in Section 2).

An alternative approach is to do as in Lean, where they do have proof irrelevance with singleton elimination, but they only implement a partial version of proof irrelevance for recursive inductive types satisfying the singleton elimination, which is restricted to closed terms.[2] But this partial implementation of the conversion algorithm breaks in particular subject reduction, which seems a desirable property for a proof assistant (see a concrete example in Appendix A). Finally, singleton elimination fails to capture inductive types with multiple constructors such as ≤ which are perfectly valid mere propositions and could be eliminated into types.

Looking back, Coq and its impredicative sort Prop may not be the only way to implement proof irrelevance in a proof assistant. Agda, which only has a predicative hierarchy of universes and no Prop, instead uses a notion of irrelevant arguments [Abel and Scherer 2012]. The idea there is to

[2] See a description of this issue in https://github.com/leanprover/lean/issues/654.

mark in the function type which arguments can be considered as irrelevant. For instance, our user can encode bounded natural numbers in this setting, by specifying that the second argument of the dependent pair is irrelevant (as marked by the . in the definition of $.(n \leq k)$):

```
data boundedNat (k : ℕ) : Set where
  pair : (n : ℕ) → .(n ≤ k) → boundedNat k
```

The fact that equality of the underlying natural numbers implies equality of the bounded natural numbers comes for free from irrelevance of the second component:

```
piBoundedNat : {k : ℕ}(n m : ℕ)(e : n ≤ k)(e' : m ≤ k) → n ≡ m → pair n e ≡ pair m e'
piBoundedNat n m _ _ refl = refl
```

However, in this approach proof irrelevance is not a property of the type being considered but rather a way to use the argument of a given function. To get closer to a real management of proof irrelevance, irrelevant fields were added to Agda.[3] It becomes thus possible to define a variant of propositional truncation by defining a record with only one field which is proof irrelevant

```
record Squash (A : Set) : Set where
  constructor sq
  field .unsq : A
```

Squash A is proof-irrelevant, as shown by the following lemma:

```
piSquash : {A : Set}(x y : Squash A) → x ≡ y
piSquash x y = refl
```

However, together with irrelevant fields, there is a notion of irrelevant projections[4], which, as observed recently by the Agda community[5], gives rise to an inconsistency in the theory. Indeed, in Agda 2.5.3 it is possible to define a function from Squash Bool to Bool that can be shown to be definitionally the inverse of sq in an irrelevant context.

```
bizarre : Squash (Σ (Bool → Squash Bool)
                    (λ i → Σ (Squash Bool → Bool) (λ u → (a : Bool) → u (i a) ≡ a)))
bizarre = sq (sq, unsq, (λ a → refl {x = a}))
```

This can be used to prove that Squash (true ≡ false) which directly leads to an inconsistency. We have worked with Agda developers since then and they have been able to correct this issue based on our explanation using a definitionally proof irrelevant universe. Thus, using a universe SProp, of definitionally proof irrelevant propositions, is necessary to give a good theoretical justification of irrelevant fields, something that hasn't been done so far. Moreover, being an inhabitant of SProp is a property of a *type* rather than the property of a particular *argument of a function*, which makes it much more flexible and easier to justify.

Contributions. In this paper, we propose the first general treatment of a dependent type theory with a proof irrelevant sort, with intuitions coming from the notion of homotopy levels of HoTT and propositional truncation. This extension of type theory does not add any additional axioms (apart from the existence of a proof irrelevant universe) and is in particular compatible with univalence. We prove both consistency and decidability of type checking of the resulting type theory. Then we show how to define an almost complete criterion to detect which proof-irrelevant inductive definitions can be eliminated into types, correcting and extending singleton elimination.

[3] Agda 2.2.8, see https://github.com/agda/agda/blob/master/CHANGELOG.md#release-notes-for-agda-2-version-228.
[4] Added in Agda 2.2.10.
[5] Issue #2170 at https://github.com/agda/agda/issues/2170.

This criterion uses the general methodology of proof-relevant unification and dependent pattern matching [Cockx and Devriese 2018; Cockx et al. 2014]. Our presentation is not specific to any particular type theory, which is shown by an implementation both in Coq, using an impredicative proof-irrelevant sort, and in Agda, using a hierarchy of predicative proof-irrelevant sorts.

Links to the source code of the Coq and Agda implementations together with examples can be foudn in https://github.com/CoqHoTT/sProp.

Plan of the paper. In Section 2, we give an overview of the main techniques and results developed in this paper. Section 3 presents the general dependent type theory with proof-irrelevant sorts, called sMLTT. We prove the main metatheoretical properties of sMLTT in Section 4. We then describe our criterion correcting and extending singleton elimination in Section 5. Finally, in Section 6 we discuss our implementation of definitional proof irrelevance in Coq and in Agda, and show its usefulness on various examples (available as anonymous supplemental material), including a formalization of the setoid model in Coq.

2 LESSONS FROM HOMOTOPY TYPE THEORY

Before diving into the precise definition of a type theory with definitional proof irrelevance, let us explore what makes it difficult to introduce while keeping decidable type checking and avoiding to induce additional axioms such as UIP or functional extensionality.

2.1 sProp as a Syntactical Approximation of Mere Propositions

The first lesson from HoTT is that each type in sProp must be a mere proposition, i.e. have homotopy level -1. Formally, the universe of mere propositions hProp is defined as

```
Definition hProp := { A : Type & ∀ x y : A , x = y }.
```

and thus it corresponds exactly to the universe of types satisfying *propositional* proof irrelevance. As we have mentioned in the introduction, the operator Squash which transforms any type into an inhabitant of sProp, corresponds to the propositional truncation.

The existence of sProp becomes interesting only when we can eliminate some inductive definitions in sProp to arbitrary types. If not, sProp constitutes an isolated logical layer corresponding to propositional logic, without any interaction with the rest of the type theory. The question is thus:

> " *Which inductive types of a universe of definitionally proof irrelevant types can be eliminated over arbitrary types?* "

Of course, to preserve consistency, this should be restricted to inductive types that can be proven to be mere propositions, but this is not enough to preserve decidability of type checking and independence from UIP.

2.2 Flirting with Extensionality

Let us first look at an apparently simple class of mere propositions, contractible types. They constitute the lowest level in the hierarchy of types, for which not only there is at most one inhabitant up to equality, but there is exactly one. Of course, the unit type in sProp which corresponds to the true proposition

```
Inductive sUnit : sProp := tt : sUnit.
```

is contractible. We will say that the unit type can be eliminated to any type, and thus we get a unit type with a definitional η-law such that $u \equiv tt$ for any $u : sUnit$. But in general, it should not be

expected that any contractible type in sProp can be eliminated to an arbitrary type, as we can see below.

Singleton types. A prototypical example of a non-trivial contractible type is the so-called singleton type. For any type A and a:A, it is defined as the subset of points in A equal to a:

```
Definition Sing (A : Type) (a : A) := { b : A & a = b }.
```

If we include singleton types in sProp, and hence permit its elimination over arbitrary types, we are led to an extensional type theory, in which propositional equality implies definitional equality. This would thus add UIP and undecidability of type checking to the theory.[6] Indeed, assume that singleton types can be eliminated to arbitrary types, then there exists a projection $\pi_1 : \mathrm{Sing}\ A\ a \to A$ which recovers the point from the singleton. But then, for any proof of equality $e : a = b$ between a and b in A, using congruence of definitional equality, there is the following chain of implications

$$(a\ ;\ \mathrm{refl}) \equiv (b\ ;\ e) : \mathrm{Sing}\ A\ a \ \Rightarrow\ \pi_1\ (a\ ;\ \mathrm{refl}) \equiv \pi_1\ (b\ ;\ e) : A \ \Rightarrow\ a \equiv b : A$$

and hence $e : a = b$ implies $a \equiv b$. From this analysis, it is clear that sProp cannot include all contractible types.

2.3 Flirting with Undecidability

Let us now look at other inductive types that are mere propositions without being contractible. The first canonical example is the empty type

```
Inductive sEmpty : sProp := .
```

that has no inhabitant, together with an elimination principle which states that anything can be deduced from the empty type

```
sEmpty_rect : ∀ T : Type, sEmpty → T.
```

We will see that this type can be eliminated to any type, and this is actually the main way to make sProp communicate with Type. In particular, it allows the construction of computational values by pattern matching and use a contraction in sProp to deal with absurd branches.

The other kind of inductive definition in sProp that can be eliminated into any type is a dependent sum of a type A in sProp and a dependent family over A in sProp, the nullary case being the unit type sUnit.

Let us now have a look at more complex inductive definitions.

Accessibility predicate. As mentioned in the introduction, the accessibility predicate is a mere proposition but it cannot be eliminated into any type while keeping conversion—and thus type-checking—decidable. Intuitively, this is because the accessibility predicate allows to define fixpoints with a *semantic* guard (the fact that every recursive call is on terms y such that R y x) rather than a *syntactic* guard (the fact that every recursive call is on a syntactic subterm). This is problematic in a definitionally proof irrelevant setting because a function that is defined by induction on an accessibility predicate could be unfolded infinitely many times. To understand why, consider the inversion lemma that we can define as soon as Acc can be eliminated into any type

```
Definition Acc_inv A R (x : A) (X : Acc x) : ∀ y:A, R y x → Acc y :=
  Acc_rect (fun x ⇒ ∀ y, R y x → Acc y) (fun x X _ ⇒ X) x X.
```

This inversion lemma makes use of the general eliminator on Acc:

[6] This remark is originally due to Peter LeFanu Lumsdaine.

$$\begin{aligned}&\mathrm{Acc\_rect} : \forall\ \mathrm{P} : \mathrm{A} \to \mathrm{Type}, (\forall\ \mathrm{x} : \mathrm{A}, (\forall\ \mathrm{y} : \mathrm{A}, \mathrm{R\ y\ x} \to \mathrm{Acc\ y}) \to (\forall\ \mathrm{y} : \mathrm{A}, \mathrm{R\ y\ x} \to \mathrm{P\ y}) \to \mathrm{P}\\ &\quad \mathrm{x})\\ &\quad \to \forall\ \mathrm{x} : \mathrm{A}, \mathrm{Acc\ x} \to \mathrm{P\ x}\end{aligned}$$

But then, from this inversion and using definitional proof irrelevance, the following definitional equality is derivable, for any predicate $\mathrm{P} : \mathrm{A} \to \mathrm{Type}$ and function $\mathrm{F} : \forall\ \mathrm{x}, (\forall\ \mathrm{y}, \mathrm{r\ y\ x} \to \mathrm{P\ y}) \to \mathrm{P\ x}$ and $\mathrm{X} : \mathrm{Acc\ x}$

$$\mathrm{Acc\_rect\ P\ F\ x\ X} \equiv \mathrm{F\ x\ (fun\ y\ r} \Rightarrow \mathrm{Acc\_rect\ P\ F\ y\ (Acc\_inv\ A\ R\ x\ X\ y\ r))}$$

In an open context, it is undecidable to know how many time this unfolding must be done. Even the strategy that there is at most one unfolding may not terminate. Indeed, suppose that we are in a (possibly inconsistent) context where there is a proof $\mathrm{R\_refl}$ showing that R is reflexive. Then, applying the unfolding above once to $\mathrm{F} := \mathrm{fun\ x\ f} \Rightarrow \mathrm{f\ x\ (R\_refl\ x)}$ computes to

$$\mathrm{Acc\_rect\ P\ F\ x\ X} \equiv \mathrm{Acc\_rect\ P\ F\ x\ (Acc\_intro\ x\ (Acc\_inv\ A\ R\ x\ X))}$$

and the unfolding can start again for ever.

As mentioned above, if we analyze the source of this infinite unfolding, it is due to the recursive call to Acc in the argument of $\mathrm{Acc\_intro}$ on an arbitrary variable y that is not *syntactically* smaller than the initial x variable, but *semantically* guarded by the $\mathrm{R\ y\ x}$ condition. This example shows that singleton elimination is not a sufficient criterion for when an inductive type in sProp can be eliminated into any type, as one needs to introduce something similar to the syntactic guard condition on fixpoints.

Let us now see why this is *not a necessary condition* either.

The Good and the Bad Less Than or Equal. The definition of less than or equal given in introduction does not satisfy the singleton elimination criterion because it has two constructors. However, $\mathrm{m} \leq \mathrm{n}$ can easily be shown to be a mere proposition for any natural numbers m and n. Thus, it is a good candidate for an sProp being eliminable into any type. The reason why it is a mere proposition is however more subtle than what singleton elimination usually requires, as not every argument of the constructors of $\leq$ is in sProp. To see why it is a mere proposition, one needs to distinguish between forced and non-forced constructor arguments[7]. A forced argument is an argument that can be deduced from the indices of the return type of the constructor, and that are not computationally relevant. Consider for instance the constructor $\leq\mathrm{S} : \forall\ \mathrm{m\ n},\ \mathrm{m} \leq\ \mathrm{n} \to\ \mathrm{m} \leq \mathrm{S\ n}$, its two first arguments m and n can be computed from the return type $\mathrm{S\ m} \leq \mathrm{S\ n}$ and are thus forced. In contrast, the argument of type $\mathrm{m} \leq \mathrm{n}$ cannot be deduced from the type and thus must be in sProp.

However, being a mere proposition is not sufficient as we have seen with singleton types and the accessibility predicate. Here the situation is even more subtle. Consider the (propositionally) equivalent definition of $\mathrm{n} \leq \mathrm{m}$ that is actually the one used in the Coq standard library:

$$\begin{aligned}&\mathrm{Inductive}\ \leq_{bad} : \mathbb{N} \to \mathbb{N} \to \mathrm{sProp} :=\\ &\mid \leq_{bad}\mathrm{\_refl} : \forall\ \mathrm{n}, \mathrm{n} \leq_{bad} \mathrm{n}\\ &\mid \leq_{bad}\mathrm{S} : \forall\ \mathrm{m\ n},\ \mathrm{m} \leq_{bad}\ \mathrm{n} \to\ \mathrm{m} \leq_{bad} \mathrm{S\ n}.\end{aligned}$$

It can also be shown that $\mathrm{m} \leq_{bad} \mathrm{n}$ is a a mere proposition, but $\leq$ and $\leq_{bad}$ do not share the same inversion principle. Indeed, in the (absurd) context that $\mathrm{e} :\ \mathrm{S\ n} \leq_{bad} \mathrm{n}$, there are two ways to form a term of type $\mathrm{S\ n} \leq_{bad} \mathrm{S\ n}$, either by using $\leq_{bad}\mathrm{\_refl\ (S\ n)}$ or by using $\leq_{bad}\mathrm{S\ (S\ n)\ n\ e}$. This means that allowing $\mathrm{m} \leq_{bad} \mathrm{n}$ to be eliminated into any type would require to decide whether the context is absurd or not, which is obviously not a decidable property of type theory. For $\mathrm{m} \leq \mathrm{n}$

[7] This terminology has been introduced by [Brady et al. 2004].

the situation is different, because the return type of the two constructors ≤0 and ≤S are orthogonal, in the sense they cannot be unified.

2.4 Dependent Pattern Matching to the Rescue

We propose a new criterion for when a type in sProp can be eliminated to an arbitrary type, fixing and generalizing the singleton elimination criterion. This criterion is general enough to distinguish between the definitions of ≤ and $\leq_{bad}$. In general, an inductive type in sProp may be eliminated if it satisfies three properties:

(1) Every non-forced argument must be in sProp.
(2) The return types of constructors must be pairwise orthogonal.
(3) Every recursive call must satisfy a syntactic guard condition.

To justify this criterion, we provide a general translation from any inductive type satisfying this criterion to an equivalent type defined as a fixpoint, using ideas coming from dependent pattern matching [Cockx and Devriese 2018; Cockx et al. 2014]. Indeed, looking at the inductive definition from right (its conclusion) to left (its arguments) allows us to construct a *case tree* similarly to what is done with a definition by pattern matching. Providing this translation also means we avoid the need to extend our core language, as all inductive types can be encoded using the existing primitives.

Rejecting constructor arguments not in sProp. The first property is the most straightforward to understand: if a constructor of an inductive type can store some information that is computationally relevant, then it should not be in sProp (or at least, we should never eliminate it into Type).

Rejecting non-orthogonal definitions. The idea of the second property is that the indices of the return type of each constructor should fix in which constructor we are, by using disjoint indices for the different constructors. This is a syntactical approximation of the orthogonality criterion. This is the property that fails to hold for $\leq_{bad}$.

Rejecting non terminating fixpoints. In addition to the first two properties, we also require a syntactic guard condition on the recursive constructor arguments. This guard condition enforces that the resulting fixpoint definition is well-founded. We may thus use the exact same syntactic condition already used for fixpoints already implemented in the type theory (no matter which one it is, as long as it guarantees termination).

For instance, in the case of Acc, the case tree induces the following definition, which is automatically rejected by the termination checker because of the unguarded recursive call Acc' y

```
Fail Equations Acc' (x: A) : sProp :=
  Acc' x := (∀ y:A, R y x → Acc' y).
```

Deriving fixpoints and eliminators automatically. If all three properties are satisfied, we can automatically derive a fixpoint in sProp that is equivalent to the inductive definition. Each constructor corresponds to a unique branch of a case tree, and the return type in each branch is the dependent sum of the non-forced arguments of the corresponding constructor (the zero case being sUnit). For instance, for ≤, this is given by

```
Equations ≤fix (n m : ℕ) : sProp :=
 0 ≤fix n := sUnit;
 S m ≤fix S n := m ≤fix n;
 S _ ≤fix 0 := sEmpty.
```

$$A, B, M, N \quad ::= \quad \textsc{Type}_i \mid x \mid M\ N \mid \lambda\, x : A.\ M \mid \Pi\, x : A.\ B \mid \Sigma x : A.\ B \mid \pi_1\ M \mid \pi_2\ M \mid (M, N)$$

$$\Gamma, \Delta \quad ::= \quad . \mid \Gamma, x : A$$

$$\frac{}{\vdash .} \qquad \frac{\Gamma \vdash A : \textsc{Type}}{\vdash \Gamma, x : A} \qquad \frac{\Gamma \vdash A : \textsc{Type}}{\Gamma, x : A \vdash x : A} \qquad \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \textsc{Type}}{\Gamma, x : A \vdash M : B}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \textsc{Type}_i : \textsc{Type}_{i+1}} \qquad \frac{\Gamma \vdash A : \textsc{Type}_i \qquad \Gamma, x : A \vdash B : \textsc{Type}_j}{\Gamma \vdash \Pi\, x : A.\ B : \textsc{Type}_{\max(i,j)}}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi\, x : A.\ B : \textsc{Type}}{\Gamma \vdash \lambda\, x : A.\ B : \Pi\, x : A.\ B} \qquad \frac{\Gamma \vdash M : \Pi\, x : A.\ B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B\,\{x := N\}}$$

$$\frac{\Gamma \vdash A : \textsc{Type}_i \qquad \Gamma, x : A \vdash B : \textsc{Type}_j}{\Gamma \vdash \Sigma(x : A), B : \textsc{Type}_{\max(i,j)}} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash M : B}{\Gamma \vdash (M, N) : \Sigma(x : A).B}$$

$$\frac{\Gamma \vdash p : \Sigma(x : A).B}{\Gamma \vdash \pi_1 p : A} \qquad \frac{\Gamma \vdash p : \Sigma(x : A).B}{\Gamma \vdash \pi_2 p : B[x := \pi_1 p]} \qquad \frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : \textsc{Type} \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash M : A}$$

$$\Gamma \vdash (\lambda\, x : A.\ M)\ N \equiv M\,\{x := N\} \qquad (\pi_1\ M, \pi_2\ M) \equiv M \qquad \Gamma \vdash \pi_1\ (M, N) \equiv M$$

$$\Gamma \vdash \pi_2\ (M, N) \equiv N \qquad \text{(congruence rules omitted)}$$

Fig. 1. Syntax and typing rules of MLTT

Note that branches corresponding to no constructor are given the value sEmpty. One can then also define functions corresponding to the constructors and the elimination principle (to Type) of the inductive type. Details of the algorithm and its correctness are given in Section 5.

3 ADDING DEFINITIONAL PROOF IRRELEVANCE TO MLTT

Most type theories describable by Pure Type Systems (PTS) can be extended with a notion of predicative or impredicative hierarchy of sorts $\textsc{sProp}_i$ which satisfies definitional proof irrelevance. This is illustrated in this paper by the fact that this extension can be applied to both Coq and Agda, which correspond to slightly different PTSs. However, to keep the theoretical presentation simple, we present this extension for a prototypical PTS-style type theory, namely Martin-Löf Type Theory [Martin-Löf 1975] (MLTT), and only go into the difference between the Coq and Agda implementations in Section 6. In this section, we first introduce MLTT and then explain how to add a proof irrelevant hierarchy of sorts, and various specific types and type constructors in it.

3.1 MLTT

MLTT is the PTS-style type theory described in Figure 1, using, as usual, three statements mutually recursively defined. The statement $\vdash \Gamma$ means that the environment Γ is well formed, while $\Gamma \vdash M : A$ means that the term M has type A in environment Γ and the statement $\Gamma \vdash A \equiv B$ means A is convertible to B in context Γ.

MLTT features dependent products, a negative presentation of dependent sums (using projection instead of pattern-matching) and a predicative hierarchy of universes $\textsc{Type}_i$. The conversion judgment features β-reduction, traditional rules of computation for dependent sums (surjective pairing and projections), together with congruence rules for every constructor. As usual, we note $A \to B$ for $\Pi\, x : A.\ B$ when B does not depend on x.

MLTT satisfies many good properties among which consistency and decidability of type-checking (see [Abel et al. 2018] for the first mechanized proof of those properties). In the sequel, we present an extension of MLTT with definitional proof irrelevance that preserves those two properties.

3.2 Adding sProp to MLTT

MLTT can be extended with a predicative hierarchy of sorts $\textsc{sProp}_i$ by adding the rule

$$\frac{\vdash \Gamma}{\Gamma \vdash \textsc{sProp}_i : \textsc{Type}_{i+1}}$$

and extending the rule of creation of a dependent product to deal with sProp

$$\frac{\Gamma \vdash A : \textsc{sProp}_i \qquad \Gamma,\, x : A \vdash B : \textsc{Type}_j}{\Gamma \vdash \Pi\, x : A.\ B : \textsc{Type}_{\max(i,j)}} \qquad \frac{\Gamma \vdash A : \textsc{Univ}_i \qquad \Gamma,\, x : A \vdash B : \textsc{sProp}_j}{\Gamma \vdash \Pi\, x : A.\ B : \textsc{sProp}_{\max(i,j)}}$$

where Univ is either sProp or Type.

An impredicative variant. We will see in section 4 that we can also allow an impredicative version of sProp, which amounts to just ignoring the indices on sProp throughout.

The main feature of sProp is that the types inhabiting it are definitionally proof-irrelevant, which is enforced by the following rule of conversion:

$$\frac{\Gamma \vdash A : \textsc{sProp}_i \qquad \Gamma \vdash x : A \qquad \Gamma \vdash y : A}{\Gamma \vdash x \equiv y : A}$$

So far the universe hierarchy of sProp is mostly unconnected from the universe hierarchy of Type's. Therefore, we add several extensions to improve on this. Namely, we add the empty type, a squash operator, a box operator and dependent sums in sProp. We call the resulting system sMLTT.

3.3 Empty Type

We add an empty type in sProp, eliminable to any type (including those in Type which are proof relevant):

$$\frac{}{\Gamma \vdash \mathsf{sEmpty} : \textsc{sProp}_i} \qquad \frac{\Gamma \vdash A : \textsc{Univ}_i \qquad \Gamma \vdash e : \mathsf{sEmpty}}{\Gamma \vdash \mathsf{sEmpty\_rect}\ A\ e : A}$$

This is enough to define many other types in sProp. For instance sUnit is just $\mathsf{sEmpty} \to \mathsf{sEmpty}$ (whose inhabitant $\lambda\, x : \mathsf{sEmpty}.\ x$ is unique up to conversion due to proof irrelevance).

3.4 Squash

We can add a squash operator (a.k.a. bracket type) by the following rules:

$$\frac{\Gamma \vdash A : \textsc{Type}_i}{\Gamma \vdash \|A\| : \textsc{sProp}_i} \qquad \frac{\Gamma \vdash x : A}{\Gamma \vdash \mathrm{sq}\ x : \|A\|}$$

$$\frac{\Gamma \vdash A : \textsc{Type}_i \qquad \Gamma \vdash P : \textsc{sProp}_j \qquad \Gamma \vdash f : A \to P \qquad \Gamma \vdash x : \|A\|}{\Gamma \vdash \mathrm{unsq}\ P\ f\ x : P}$$

Note that thanks to definitional proof irrelevance, the non-dependent eliminator $\mathrm{unsq}\ P\ f\ x$ is enough to define the dependent one.

In an impredicative setting, $\|A\|$, $\mathrm{sq}\ x$ and $\mathrm{unsq}\ P\ f\ x$ can be defined using a standard impredicative encoding:

$$\begin{array}{lcl} \|A\| & \coloneqq & \Pi(P : \textsc{sProp}), (A \to P) \to P \\ \mathrm{sq}\ x & \coloneqq & \lambda\, P : \textsc{sProp}.\ \lambda\, f : A \to P.\ f\ x \\ \mathrm{unsq}\ P\ f\ x & \coloneqq & x\ P\ f \end{array}$$

In a predicative setting, the situation is similar but the encoding only allows one to define the eliminator to lower universes as captured by the following rules

$$\frac{\Gamma \vdash A : \textsc{Type}_i}{\Gamma \vdash \|A\|_{i,j} : \textsc{sProp}_{max(i,j+1)}}$$

$$\frac{\Gamma \vdash A : \textsc{Type}_i \qquad \Gamma \vdash P : \textsc{sProp}_j \qquad \Gamma \vdash f : A \to P \qquad \Gamma \vdash x : \|A\|_{i,j} \qquad j < i}{\Gamma \vdash \mathrm{unsq}\ P\ f\ x : P}$$

If we want to eliminate the squash type to arbitrary types in a predicative setting, then we have to add it as a primitive.

3.5 Box

If we see sProp as a sub-universe of Type there should be an inclusion from sProp to Type, converse to the squash operator.

$$\frac{\Gamma \vdash A : \textsc{Prop}_i}{\Gamma \vdash \Box A : \textsc{Type}_i} \qquad \frac{\Gamma \vdash x : A}{\Gamma \vdash \mathrm{box}\ x : \Box A}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : \textsc{sProp}_i \\ \Gamma \vdash P : \Box A \to \textsc{Type}_j \textit{ or } \Gamma \vdash P : \Box A \to \textsc{sProp}_j \end{array} \qquad \Gamma, x : A \vdash f : P\ (\mathrm{box}\ x) \qquad \Gamma \vdash x : \Box A}{\Gamma \vdash \mathrm{unbox}\ P\ f\ x : P\ x}$$

together with the conversion rule

$$\Gamma \vdash \mathrm{unbox}\ P\ f\ (\mathrm{box}\ a) \equiv f\ a$$

This time, the dependent elimination can not be deduced from the non-dependent one, and the conversion rule is not automatic, because $\Box A$ lives in $\textsc{Type}_i$, not in $\textsc{sProp}_i$.

Having the box operator is equivalent to allowing one of the sides of a pair to be in sProp. Indeed, $\Box A$ is the same as $\Sigma(x : A).\mathrm{Unit}$ or $\Sigma(x : \mathrm{Unit}).A$, and instead of $\Sigma(x : A).B$ with $A : \textsc{sProp}$ we could use $\Sigma(y : \Box A).B[x \coloneqq \mathrm{unbox}\ A\ (\lambda\, z : A.\ z)\ y]$.

This is what we need to be able to talk about properties with both relevant and irrelevant parts. For instance the type of symmetric strict relations on some type A is

$$\Sigma(R : A \to A \to \text{sProp}).\Pi\, x\ y : A.\ R\ x\ y \to R\ y\ x$$

$A \to A \to \text{sProp}$ is a large type, and the proof of symmetry is an sProp.

Making the box a primitive construct and deriving from it the concept of relevant pairs with an irrelevant component allows for a better separation of concerns. For instance, if we allowed irrelevant components in pairs with eta we would have to take care that if all components were irrelevant, then the resulting type would be definitionally irrelevant and so should be in sProp.

In the implementations for Coq and Agda the box is subsumed by inductive types, see Section 6.

3.6 Dependent Sums

If both components of a pair are in sProp, we can allow the pair itself to be in sProp too:

$$\frac{\Gamma \vdash A : \text{sProp} \qquad \Gamma, x : A \vdash B : \text{sProp}_j}{\Gamma \vdash \Sigma_s(x : A).B : \text{sProp}_{\max(i,j)}} \qquad \frac{\Gamma \vdash p : \Sigma_s(x : A).B}{\Gamma \vdash \pi_{s1} p : A} \qquad \frac{\Gamma \vdash p : \Sigma_s(x : A).B}{\Gamma \vdash \pi_{s2} p : B[x \coloneqq \pi_{s1} p]}$$

Since we have definitional proof irrelevance, the expected computation rules for the projections and the surjective pairing conversion rule already hold and do not have to be added explicitly to the conversion.

In an impredicative setting, dependent sums can be encoded as follows:

$$\begin{array}{lcl}
\Sigma_s(x : A).B & := & \Pi\, P : \text{sProp}.\ (\Pi\, x : A.\ B\ x \to P) \to P \\
(a, b) : \Sigma_s(x : A).B & := & \lambda\, P : \text{sProp}.\ \lambda\, f : _.\ f\ a\ b \\
\pi_{s1}(p : \Sigma_s(x : A).B) : A & := & p\ A\ (\lambda\, x : A.\ \lambda\, y : B\ x.\ x) \\
\pi_{s2}(p : \Sigma_s(x : A).B) : B(\pi_{s1}\ p) & := & p\ (B(\pi_{s1}\ p))\ (\lambda\, x : A.\ \lambda\, y : B\ x.\ y)
\end{array}$$

Note that π_{s2} is well typed due to proof irrelevance of A.

Dependent sums can also be encoded in a predicative setting extended with primitive squash and box:

$$\begin{array}{lcl}
\Sigma_s(x : A).B & := & \|(x : \Box A).\Box B(\text{unbox}\ A\ id\ x)\| \\
(a, b) : \Sigma_s(x : A).B & := & \text{sq}\ (\text{box}\ a, \text{box}\ b) \\
\pi_{s1}(p : \Sigma_s(x : A).B) : A & := & \text{unsq}\ A\ (\lambda\, x : _.\ \text{unbox}\ _\ id\ (\pi_{s1} x))\ p \\
\pi_{s2}(p : \Sigma_s(x : A).B) : B(\pi_{s1} p) & := & \text{unsq}\ (B(\pi_{s1} p))\ (\lambda\, x : _.\ \text{unbox}\ _\ id\ (\pi_{s2} x))\ p
\end{array}$$

where we have omitted some obvious type annotations and where *id* is the identity function.

3.7 Other Inductive Types

There is no need for other primitive inductive types in sProp (assuming the corresponding proof relevant inductive types are available in Type).

Indeed, either an inductive should not be permitted in sProp because it would introduce inconsistency (booleans) or undecidability (accessibility) and in that case, squashing the corresponding proof relevant inductive type is enough.

In the other case, if the inductive type satisfies the criteria for being in sProp, we will see in section 5 that it can actually be encoded as a recursive definition using sEmpty, sUnit and proof irrelevant pairs. This translation comes with preconditions on the shape of the inductive type being translated, preventing it from working in cases where the inductive cannot be regarded as a strict proposition such as booleans (which would make the theory inconsistent) or the accessibility predicate (which would make conversion undecidable).

4 METATHEORETICAL PROPERTIES OF sMLTT

In this section, we prove the consistency of the predicative version of sMLTT by reducing it to the consistency of Extensional Type Theory (ETT). We then remark that the impredicative version is justified by the propositional resizing rule proposed by Vladimir Voevodsky [Voevodsky 2011]. We further show that sMLTT is either compatible with the univalence axiom or supports a computational version of UIP, depending on whether we authorize elimination of equality in sProp or not. Finally, we show that type checking of sMLTT is decidable by modifying the proof by logical relations of Abel *et al* [Abel et al. 2018; Abel and Scherer 2012].

4.1 ETT

$$A,\ B,\ M,\ N \quad ::= \quad \ldots \mid M =_A N \mid \mathrm{refl}_M \mid J(y.q.M,\ e\ ,u)$$

$$\frac{\Gamma \vdash A : \mathrm{Type}_i \qquad \Gamma \vdash x : A \qquad \Gamma \vdash y : A}{\Gamma \vdash x =_A y : \mathrm{Type}_i} \qquad \frac{\Gamma \vdash e : x =_A y}{\Gamma \vdash x \equiv y} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathrm{refl}_M \ : M =_A M}$$

$$\frac{\Gamma \vdash e : M =_A N \qquad \Gamma, y : A, q : M =_A y \vdash P : \mathrm{Type}_i \qquad \Gamma \vdash u : P\{y := M\}\{q := \mathrm{refl}_M\ \}}{\Gamma \vdash J(y.q.P, e, u) : P\{y := N\}\{q := e\}}$$

$$\Gamma \vdash J(y.q.P, \mathrm{refl}_M\ , u) \equiv u \qquad\qquad (+ \text{ other rules of MLTT})$$

Fig. 2. Syntax and typing rules of ETT

ETT is an extension of MLTT, presented in Figure 2, with a propositional equality $x =_A y$ that satisfies the *reflection rule*, that is every propositionally equal terms are convertible. We also assume that there is a type Unit with only one inhabitant tt.

It is well known that ETT has an undecidable type checking as deciding conversion requires to decide propositional equality, since it is possible to encode the halting problem with an equality in ETT. But ETT still satisfies some good properties: it has a decidable "derivation" checking and it is consistent as it is conservative over MLTT with an identity type (plus UIP and functional extensionality), as shown by Martin Hofmann [Hofmann 1995].

We should notice that using the reflection rule and η-law for functions, one can derive functional extensionality, that is given two functions $f\ , g : \Pi\, x : A.\ B$ there is a term

$$\mathsf{funext}_{f,g} : (\Pi\, a : A.\ f\ a =_{B\{x:=a\}} g\ a) \to f =_{\Pi\, x:A.\ B} g$$

Also, the reflection rule allows deriving UIP. So adding the reflection rule to MLTT has additional consequences. We show in this section that for sMLTT, UIP can be safely added, but is independent from the theory, as we show in Section 4.3 that sMLTT is compatible with univalence.

4.2 Consistency of sMLTT

Following the notion of syntactical translations advocated in [Boulier et al. 2017], we prove consistency of sMLTT by a type preserving translation from sMLTT into ETT. The idea of the translation is to see inhabitant of sProp as a pair of type in ETT together with a proof that it is a mere proposition. Therefore, sProp is translated as the following dependent sum:

$$[\mathrm{sProp}_i] := \Sigma\, A : \mathrm{Type}_i.\ \Pi\, x\ y : A.\ x = y.$$

$$
\begin{array}{ll}
[\textsc{Type}_i] & := \Sigma\, A : \textsc{Type}_i.\ \mathsf{Unit} \\
[\textsc{sProp}_i] & := \Sigma\, A : \textsc{Type}_i.\ \Pi\, x\ y : A.\ x = y \\
[x] & := x \\
[\Pi\, x : A.\ B : \textsc{Type}] & := (\Pi\, x : [\![A]\!].\ [\![B]\!]; \mathsf{tt}) \\
[\Pi\, x : A.\ B : \textsc{sProp}] & := (\Pi\, x : [\![A]\!].\ [\![B]\!]; \\
 & \quad \lambda\, f\ g : \Pi\, x : [\![A]\!].\ [\![B]\!].\ \pi_\Pi\ [\![A]\!]\ [\![B]\!]_{\mathsf{pf}}\ f\ g) \\
[\lambda\, x : A.\ M] & := \lambda\, x : [\![A]\!].\ [M] \\
[MN] & := [M]\ [N] \\
[\mathsf{sEmpty}] & := (\mathsf{Empty}; \lambda\, x\ y : \mathsf{Empty}.\ \pi_{\mathsf{Empty}}\ x\ y) \\
[\mathsf{sEmpty\_rect}\ P\ t] & := \mathsf{Empty\_rect}\ [P]\ [t] \\
[\Box A] & := (\pi_1 [A], \mathsf{tt}) \\
[\mathrm{box}\ x] & := [x] \\
[\mathrm{unbox}\ P\ f\ a] & := [f]\ [a] \\
[\Sigma\, x : A.\ B : \textsc{Type}] & := (\Sigma\, x : [\![A]\!].\ [\![B]\!]; \mathsf{tt}) \\
[\Sigma\, x : A.\ B : \textsc{sProp}] & := (\Sigma\, x : [\![A]\!].\ [\![B]\!]; \\
 & \quad \lambda\, X\ Y : \Sigma\, x : [\![A]\!].\ [\![B]\!].\ \pi_\Sigma\ [\![A]\!]_{\mathsf{pf}}\ [\![B]\!]_{\mathsf{pf}}\ X\ Y) \\
[\lambda\, x : A.\ M] & := \lambda\, x : [\![A]\!].\ [M] \\
[MN] & := [M]\ [N] \\
[\![A]\!] & := \pi_1 [A] \\
[\![A]\!]_{\mathsf{pf}} & := \pi_2 [A]
\end{array}
$$

$$
\begin{array}{ll}
\pi_{\mathsf{Empty}}\ x\ y & := \mathsf{Empty\_rect}\ (x = y)\ x \\
\pi_\Pi\ A\ \pi_B\ f\ g & := \mathsf{funext}\ (\lambda\, x : A.\ \pi_B (f\ x)(g\ x)) \\
\pi_\Sigma\ \pi_A\ \pi_B\ X\ Y & := \mathsf{eq}_\Sigma\ (\pi_A(\pi_1 X)(\pi_1 Y))\ (\pi_B(\pi_2 X)(\pi_2 Y))
\end{array}
$$

Fig. 3. Syntactical Translation from sMLTT to ETT

The rest of the translation is rather straightforward, but for the fact that we need to provide an additional proof that type constructors which end in sProp build mere propositions. For instance, the fact that a dependent product whose codomain is a mere proposition is itself a mere proposition can be proven using functional extensionality in ETT. Definitional proof irrelevance is then modeled by the fact that every inhabitant of sProp is a mere proposition together with the reflection rule of ETT.

The translation is described in Figure 3, where eq_Σ is defined as the witness that equality between elements of an dependent sum is given by the equality of the corresponding projections (which is provable using J)

$$\mathsf{eq}_\Sigma\ :\ \Pi\, x\ y : \Sigma\, a : A.\ B.\ \pi_1 x =_A \pi_1 y \to \pi_2 x =_{B\{a:=\pi_1 x\}} \pi_2 y \to x =_{\Sigma\, a:A.\ B} y.$$

Note that this statement is slightly more involved in intentional type theory[8] as the second equality does not type-check in MLTT, because $\pi_2 y$ does not have type $B\{a := \pi_1 x\}$. Here, the situation is simpler as we can use the reflection rule to convert $B\{a := \pi_1 y\}$ into $B\{a := \pi_1 x\}$ using the first equality.

The following two properties are proved by mutual induction (although we state them separately for readability).

Lemma 4.1 (preservation of conversion). *For every term t and u of* sMLTT, *if* $\Gamma \vdash t \equiv u : A$ *then* $[\![\Gamma]\!] \vdash [t] \equiv [u]$ *in* ETT.

Proof. By induction on the proof of congruence. The structure of the term is preserved by the translation, so β-reduction and congruence rules are automatically preserved. The only interesting case is the rule for definitional proof-irrelevance which says that when $\Gamma \vdash A : \mathrm{sProp}$, then t and u are convertible. But by correctness of the translation (Theorem 4.2), we know that $[\![\Gamma]\!] \vdash [A] : \Sigma\, A : \mathrm{Type}_i.\ \Pi\, x\ y : A.\ x = y$ and $[t]$, $[u]$ have type $[\![A]\!]$. Thus we can form of proof of $[t] = [u]$ by $[\![A]\!]_{\mathrm{pf}}\ [t]\ [u]$. From which we deduce $[t] \equiv [u]$ by the reflection rule. □

Theorem 4.2 (correctness of the syntactical translation). *For every typing derivation of* sMLTT $\Gamma \vdash M : A$, *we have the corresponding derivation* $[\![\Gamma]\!] \vdash [M] : [\![A]\!]$ *in* ETT.

Proof. By induction on the typing derivation. □

About impredicativity and resizing rules. ETT is not sufficient to interpret impredicativity of sProp in sMLTT, because if we state $[\mathrm{sProp}] := \Sigma\, A : \mathrm{Type}_0.\ \Pi\, x\ y : A.\ x = y$, where Type_0 is some fixed universe (let say the first one) in the hierarchy, then the typing rule for impredicativity of dependent product for sProp can not be interpreted by the translation because for $\Gamma \vdash A : \mathrm{Type}_i$ and $\Gamma,\ x : A \vdash B : \mathrm{sProp}$, one has that

$$[\![\Gamma]\!] \vdash [\Pi\, x : A.\ B] : \Sigma\, A : \mathrm{Type}_i.\ \Pi\, x\ y : A.\ x = y$$

To remedy this situation, we can make use of the propositional resizing rule introduced by Vladimir Voevodsky [Voevodsky 2011], which says that every mere proposition can be put in the universe Type_0.

$$\frac{\Gamma \vdash A : \mathrm{Type}_i \qquad \Gamma \vdash e : \Pi\, x\ y : A.\ x =_A y}{\Gamma \vdash \mathsf{rr}(A, e) : \mathrm{Type}_0}$$

The propositional resizing is justified by any classical model of ETT, that is a model which satisfies the law of excluded middle, because in such a model, every mere proposition is either sEmpty or Unit and thus lives in the lowest universe.[9] Using propositional resizing, it is thus possible to define the coercion[10]

$$\begin{array}{lcl} \mathsf{rr}_{\mathrm{sProp}} & : & (\Sigma\, A : \mathrm{Type}_i.\ \Pi\, x\ y : A.\ x = y) \to \Sigma\, A : \mathrm{Type}_0.\ \Pi\, x\ y : A.\ x = y \\ & := & \lambda\, A : \_.\ (\mathsf{rr}(\pi_1 A, \pi_2 A)), \lambda\, x\ y : \mathsf{rr}(\pi_1 A, \pi_2 A).\ \pi_2 A\ x\ y) \end{array}$$

which is enough to interpret impredicativity of sProp.

4.3 sMLTT is Compatible with Univalence

We now show that sMLTT is compatible with univalence, and thus in particular, definitional proof irrelevance does not necessary imply UIP. To that end, we consider an extension of ETT with a

[8] The precise statement and a proof of it can be found for instance in [Univalent Foundations Program 2013].

[9] Vladimir Voevodsky has also introduced other resizing rules involving type equivalences which are a bit more controversial as their justifications have not been completely written up.

[10] For simplicity, we assume that inhabitants of A and $\mathsf{rr}A$ are the same, but to keep type-checking decidable, we should use explicit wrapper from one to the other.

$$
\begin{aligned}
[\textsc{Type}_i] &:= \Sigma\, A : \textsc{FType}_i.\ \mathsf{Unit} \\
[\textsc{sProp}_i] &:= \Sigma\, A : \textsc{FType}_i.\ \Pi\, x\ y : A.\ x = y
\end{aligned}
$$

Fig. 4. Syntactical Translation from sMLTT to HTS

second notion of equality, but restricted to *fibrant* types. This extension has first been proposed by Vladimir Voevodsky under the name Homotopy Type System (HTS) [Voevodsky 2013] and has later been reworked by Paolo Capriotti *et. al.* under the name two-level type theory [Altenkirch et al. 2016; Capriotti 2017] (which may or may not be extensional for the strict equality). Those two systems differ slightly. In particular, in HTS, there is only one type of integers which is fibrant, whereas two level type theory distinguishes between fibrant and non-fibrant types. However, those differences are not important for our purpose, so even though we refer to HTS, we could also have used an extensional two-level type theory.

The main idea of HTS is to distinguish between types which live in $\textsc{Type}_i$ for which equality is strict, and fibrant types which live in $\textsc{FType}_i$ for which there are two notions of equality: strict equality $a =_A b$ and homotopical equality $a \sim_A b$. The rules for $a \sim_A b$ are the same as for propositional equality, except that the formation of $a \sim_A b$ is only possible when A is fibrant, and the eliminator $J_\sim$ is restricted to fibrant predicates:

$$
\frac{\Gamma \vdash e : M \sim_A N \qquad \Gamma, y : A, q : M \sim_A y \vdash P : \textsc{FType}_i \qquad \Gamma \vdash u : P\{y := M\}\{q := \mathrm{refl}^{\sim}_M\}}{\Gamma \vdash J_\sim(y.q.P, e, u) : P\{y := N\}\{q := e\}}
$$

Then, the type of propositional equality is not fibrant, so it is not possible to derive propositional equalities from homotopical ones.

We can modify the translation of Figure 3 to target the *fibrant fragment* of HTS. In this case, types and propositions are interpreted as fibrant types, as shown in Figure 4. Note that $[\textsc{sProp}_i]$ is not by definition fibrant in HTS, so before stating the correctness of this modified translation, we need to introduce a new rule in HTS in order to have

$$[\textsc{sProp}_i] : \textsc{FType}_i.$$

This rule is admissible as has been proven by Thierry Coquand in his note on the universes of Bishop sets and strict propositions [Coquand 2016]. Indeed, using the cubical model of type theory (which is a model of HTS), Thierry Coquand shows in particular that the universe of strict propositions is fibrant (Theorem 6.2).

It is now easy to replay the proof of Theorem 4.2 to show the correctness of the syntactical translation in HTS and conclude the following corollary.

Corollary 4.3. sMLTT *is compatible with univalence.*

Proof. As the translation of $\textsc{Type}_i$ is isomorphic to $\textsc{FType}_i$ and the translation of a dependent product in $\textsc{Type}_i$ is directly translated as a dependent product, we can add a univalent fibrant equality in $\textsc{Type}_i$ in sMLTT which is simply translated as the univalent equality in $\textsc{FType}_i$. □

4.4 Strict Identity and UIP

We now consider an optional extension, adding propositional equality in sProp. Since this implies uniqueness of identity proofs, this extension is not always desired, in particular if we want to stay compatible with univalence, but we mention it because it provides a very simple and modular way to add UIP in the system.

$$\frac{\Gamma \vdash A : \textsc{Type}_i \qquad \Gamma \vdash x : A \qquad \Gamma \vdash y : A}{\Gamma \vdash x \ =^s_A \ y : \textsc{sProp}_i} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{refl}^s_M \ : M \ =^s_A \ M}$$

$$\frac{\Gamma \vdash e : M \ =^s_A \ N \qquad \Gamma, y : A, q : M \ =^s_A \ y \vdash P : \textsc{Type}_i \qquad \Gamma \vdash u : P\{y := M\}\{q := \text{refl}^s_M\ \}}{\Gamma \vdash J_s(y.q.P, e, u) : P\{y := N\}\{q := e\}}$$

$$\Gamma \vdash J_s(y.q.P, \text{refl}_M\ , u) \equiv u$$

Fig. 5. Definition of a strict equality type.

In this context, the computation rule for equality in sProp can equivalently be stated as $\Gamma \vdash J(x.q.P, e, u) \equiv u$ when $e : x \ =^s \ x$, as already noted by Benjamin Werner in [Werner 2008]. The consistency of this extension is direct as it is justified by the translation presented in Figure 3 by simply adding

$$[x \ =^s_A \ y] := ([x] =_{[\![A]\!]} [y], \text{uip}\ [\![A]\!]\ [x]\ [y])$$

where $\text{uip}\ A\ x\ y : \Pi\, e\ e' : x =_A y.\ e =_{x=y} e'$ is a proof of UIP in ETT. Then refl^s and J_s are directly translated using refl and J.

4.5 Relevance Marks and Decidability of Type Checking

A naïve implementation of type theory with sProp requires to check during conversion if the terms we are comparing are relevant or not. To compute this information requires 2 rounds of typing: first to get its type, then to get the sort of the type, but this is not very efficient. There is another issue, this time on the theoretical side: conversion can not be defined independently from typing, and the standard technique to prove decidability of type checking developed by Andreas Abel and others [Abel et al. 2018; Abel and Scherer 2012] based on algorithmic equality and logical relations does not apply. We now introduce a notion of term annotation that allows to solve both issues at the same time, thus making it possible to decide irrelevance syntactically, without relying on type checking.

Specifically, we annotate lambdas and products according to the relevance of the bound type. The same is done for each variable in a context. When adding a variable to a context and when typing a function or product type we need to check the mark on the variable. The syntax with sProp and the inference rules for adding a variable to a context are described in Figure 6. Adapting the other rules and adding extensions is straightforward and left as an exercise to the reader.

$$\begin{array}{rcl} * & ::= & \textsf{Relevant} \mid \textsf{Irrelevant} \\ A, B, M, N & ::= & \textsc{sProp}_i \mid \textsc{Type}_i \mid x \mid M\ N \mid \lambda\, x^* : A.\ M \\ & & \mid \Pi\, x^* : A.\ B \mid \Sigma x : A.\ B \mid \pi_1\ M \mid \pi_2\ M \mid (M, N) \\ \Gamma, \Delta & ::= & \ . \mid \Gamma, x^* : A \end{array}$$

$$\frac{\Gamma \vdash A : \textsc{Type}_i}{\vdash \Gamma, x^{\textsf{Relevant}} : A} \qquad \frac{\Gamma \vdash A : \textsc{sProp}_i}{\vdash \Gamma, x^{\textsf{Irrelevant}} : A}$$

Fig. 6. sMLTT with relevance marks

The procedure to decide the relevance of a term in a context is defined by induction on the syntax:

- $\mathsf{relevance}_\Gamma(x) \coloneqq *$ when $x^* \in \Gamma$
- $\mathsf{relevance}_\Gamma(M\ N) \coloneqq \mathsf{relevance}_\Gamma(M)$
- $\mathsf{relevance}_\Gamma(\lambda\, x^* : A.\ M) \coloneqq \mathsf{relevance}_{\Gamma, x^*:A}(M)$
- Everything else (sorts, product types, relevant sigma types, relevant pairs and relevant projections) is relevant.

As binders for product types are annotated, we can pass this annotation to the context when we go under the binder (for instance in a conversion procedure). Binders in Σ types need no annotation since they are always relevant.

Proposition 4.4. *For M well typed in Γ,* $\mathsf{relevance}_\Gamma(M) = \mathsf{Irrelevant}$ *if and only if there exists A such that $\Gamma \vdash M : A$ and $\Gamma \vdash A : s\mathrm{Prop}$.*

Proof. By induction on the typing derivation of M. Only the case for application is of interest. The crucial property is that relevance is stable by well-typed substitution: if $\Gamma, x^* : A \vdash T : s$, $\Gamma \vdash e : A$ and $\Gamma \vdash T[x \coloneqq e] : s'$ then s is sProp if and only if s' is sProp. This is given by uniqueness of typing and the fact that substitution is type preserving. □

In a setting with cumulativity, for instance in Coq, uniqueness of typing expresses that two types of the same term have a common upper bound, so for correctness of marks, implicit cumulativity from sProp to Type must be forbidden, instead relying on explicit cumulativity through $\square$.

Using relevance marks, we can replay the proof of decidability of type checking of Andreas Abel and Gabriel Scherer using algorithmic equality and logical relations [Abel and Scherer 2012]—for sMLTT without a strict equality, as the work of Abel *et al.* does not deal with identity types.

Their setting was designed to handle irrelevant arguments, where irrelevance was marked on the typing annotation $\Gamma \vdash t \div A$ which says that t is used irrelevantly. Here we replace this irrelevance annotation by $\Gamma \vdash t : A \wedge \mathsf{relevance}_\Gamma(t) = \mathsf{Irrelevant}$. That is, the annotation is not on the typing judgment but directly on the term. Concretely, we replace the rule for dealing with irrelevance in the definition of structural equality on neutral terms by

$$\frac{\Gamma \vdash n \leftrightarrow n : A \qquad \mathsf{relevance}_\Gamma(n) = \mathsf{Irrelevant} \qquad \Gamma \vdash n' \leftrightarrow n' : A}{\Gamma \vdash n \leftrightarrow n' : A}$$

Once this change has been done, we can replay their proof directly to get:

Theorem 4.5 (From Theorem 6.7, [Abel and Scherer 2012]). *$\Gamma \vdash t \equiv t' : T$ is decidable.*

5 FROM INDUCTIVE TYPE TO A FIXPOINT

So far, we have described the type theory sMLTT with a universe sProp consisting of function types, the empty type, the unit type, dependent sum types, and a squash type. But in general we also want to define arbitrary inductive types in sProp. On one hand, we can always define an inductive type in Type and then apply the propositional squash to bring it into sProp, but this restricts its elimination principle to target types in sProp. On the other hand, we mentioned before that certain inductive types can be translated to fixpoints using just the empty type, unit type, and dependent sum as the basic building blocks in sProp. This translation has already been intuitively sketched in Section 2.4. In this section, we give a formal criterion for when this translation is possible and show how it can be automated.

The algorithm described here is inspired by the elaboration of definitions by dependent pattern matching to a case tree described by Cockx and Abel [2018], and makes heavy use of much of the same machinery for case analysis and proof-relevant unification.

Note that there are other ways to define datatypes with a dependent elimination principle, as for instance by using an impredicative encoding in a type theory enriched with dependent intersections types as done in Cedille [Stump 2018].

5.1 Constructing the Case Tree

From a high-level view, the idea of the translation is to view the definition of the inductive type as a *function* that does some case analysis on the indices and returns the argument types of the appropriate constructor in each case. The types for which this translation succeeds are exactly the types in sProp that can be eliminated into arbitrary types. For example, recall from Section 2.4 that we can view the definition of $\leq$ as the following definition:

$$
\begin{array}{l}
\text{Equations } \leq_{\text{fix}} (\mathrm{n}\ \mathrm{m} : \mathbb{N}) : \text{sProp} := \\
\quad 0 \leq_{\text{fix}} \mathrm{n} := \text{sUnit}; \\
\quad \mathrm{S}\ \mathrm{m} \leq_{\text{fix}} \mathrm{S}\ \mathrm{n} := \mathrm{m} \leq_{\text{fix}} \mathrm{n}; \\
\quad \mathrm{S}\ \_ \leq_{\text{fix}} 0 := \text{sEmpty}.
\end{array}
$$

One challenge in this translation is to determine which constructor arguments should appear on the right-hand side: for the constructor $\leq\mathrm{S}$ (in the second equation), the argument of type $\mathrm{m} \leq \mathrm{n}$ makes an appearance but the first two arguments m and n do not. These disappearing arguments correspond exactly to the *forced arguments* of the constructor: their values can be uniquely determined from the type.

To tackle this problem in general, we choose not to translate the inductive definition to a list of clauses, but directly to a case tree. For example, we construct the following case tree for $\leq$:

$$
m \leq n := \text{case}_m \left\{ \begin{array}{lcl} 0 & \mapsto & \leq 0 \\ \mathrm{S}\ m' & \mapsto & \text{case}_n \left\{ \begin{array}{lcl} 0 & \mapsto & \bot \\ \mathrm{S}\ n' & \mapsto & \leq \mathrm{S}\ (p : m' \leq n') \end{array} \right\} \end{array} \right\}
$$

In general, a case tree representing an inductive datatype in sProp is either a leaf node of the form $\mathrm{c}\ \Delta$ where c is a constructor name and Δ is a telescope of types in sProp, an empty node $\bot$, or an internal node of the form

$$
\text{case}_x \left\{ \mathrm{c}_1\ \hat{\Delta}_1 \mapsto^{\tau_1} Q_1; \ldots; \mathrm{c}_\mathrm{n}\ \hat{\Delta}_n \mapsto^{\tau_n} Q_n \right\}
$$

where x is a variable, c_i are constructors with fresh variables $\hat{\Delta}_i$ for arguments, τ_i are substitutions (these will be explained later), and Q_i are again case trees.

For constructing a case tree from the declaration of the inductive type, we work on an elaboration problem P of the form

$$
\Gamma \vdash \{\mathrm{c}_1\ \Delta_1\ [\Phi_1]; \ldots; \mathrm{c}_\mathrm{k}\ \Delta_k\ [\Phi_k]\}
$$

where:

- Γ is the 'outer' telescope of datatype indices,
- $\mathrm{c}_1, \ldots, \mathrm{c}_\mathrm{k}$ are the names of the constructors,
- Δ_i is the 'inner' telescope of arguments of c_i, and
- Φ_i is a set of constraints $\{w_{ij} \stackrel{?}{/} v_{ij} : A_{ij} \mid j = 1 \ldots l\}$.

To transform the definition of an inductive datatype D to a case tree, the initial elaboration problem has for Γ the index telescope of D, $\mathrm{c}_1, \ldots, \mathrm{c}_\mathrm{k}$ all constructors of D, Δ_i the argument telescope of c_i, and $\Phi_i = \{x_j \stackrel{?}{/} v_{ij} : A_j \mid j = 1 \ldots l\}$ where $\Gamma = (x_1 : A_1) \ldots (x_l : A_l)$ and $v_{i1}, \ldots, v_{il}$ are the indices targeted by c_i, i.e. $\mathrm{c}_\mathrm{i} : \Delta_i \to \mathrm{D}\ v_{i1}\ \ldots\ v_{il}$. For example, for $\leq$ we start

with the following elaboration problem:

$$(m\ n : \mathbb{N}) \vdash \left\{ \begin{array}{ll} \leq 0\ (n' : \mathbb{N}) & [m \stackrel{?}{/} 0 : \mathbb{N}, n \stackrel{?}{/} n' : \mathbb{N}] \\ \leq \mathrm{S}\ (m'\ n' : \mathbb{N})(p : m' \leq n') & [m \stackrel{?}{/} \mathrm{S}\ m' : \mathbb{N}, n \stackrel{?}{/} \mathrm{S}\ n' : \mathbb{N}] \end{array} \right\}$$

From this initial problem, the elaboration proceeds by successive case splitting on variables in Γ and simplification of constraints in Φ_i until there are only zero or one constructors left and all constraints have been solved. More specifically, the elaboration algorithm may perform the following steps:

Empty If there are no constructors left, elaboration is done and returns the case tree $\bot$:

$$\Gamma \vdash \{\} \rightsquigarrow \bot$$

Done If there is a single constructor left and all constraints are solved, elaboration checks if all remaining arguments of the constructor are in sProp. If this is the case, it returns a leaf node containing this constructor:

$$\frac{\forall (x : A) \in \Delta. \quad A : \mathrm{sProp}}{\Gamma \vdash \{\mathrm{c}\ \Delta\ []\} \rightsquigarrow \mathrm{c}\ \Delta}$$

Solve Constraint If there is a constraint of the form $y \stackrel{?}{/} x$ where x is bound in Δ and y bound in Γ, we can instantiate the variable x to y, removing it from Δ in the process:

$$\frac{\Gamma \vdash \{\mathrm{c}\ \Delta_1 (\Delta_2[y/x])\ [\Phi]\} \rightsquigarrow Q \qquad (x : A) \in \Gamma}{\Gamma \vdash \left\{\mathrm{c}\ \Delta_1 (x : A) \Delta_2\ [y \stackrel{?}{/} x : A, \Phi]\right\} \rightsquigarrow Q}$$

Simplify constraint If the left- and right-hand side of a constraint are applications of the same constructor, we can simplify the constraint:

$$\frac{\mathrm{d} : \Delta' \to \mathrm{D}\ \bar{w}' \qquad \Gamma \vdash \left\{\mathrm{c}\ \Delta\ [\bar{v} \stackrel{?}{/} \bar{u} : \Delta', \Phi]; P\right\} \rightsquigarrow Q}{\Gamma \vdash \left\{\mathrm{c}\ \Delta\ [\mathrm{d}\ \bar{v} \stackrel{?}{/} \mathrm{d}\ \bar{u} : \mathrm{D}\ \bar{w}, \Phi]; P\right\} \rightsquigarrow Q}$$

Remove constraint If a constraint is trivially satisfied, it can be removed:

$$\frac{\Gamma \vdash u = v : A \qquad \Gamma \vdash \{\mathrm{c}\ \Delta\ [\Phi]; P\} \rightsquigarrow Q}{\Gamma \vdash \left\{\mathrm{c}\ \Delta\ [v \stackrel{?}{/} u : A, \Phi]; P\right\} \rightsquigarrow Q}$$

Remove constructor If a constraint is unsolvable because the left- and right-hand side are applications of distinct constructors, the constructor does not contribute to this branch of the case tree and can be safely removed:

$$\frac{\Gamma \vdash \{P\} \rightsquigarrow Q}{\Gamma \vdash \left\{\mathrm{c}\ \Delta\ [\mathrm{d}_2\ \bar{v} \stackrel{?}{/} \mathrm{d}_1\ \bar{u} : \mathrm{D}\ \bar{w}, \Phi]; P\right\} \rightsquigarrow Q}$$

Split Finally, if $(x : \mathrm{D}\ \bar{w}) \in \Gamma$ and each constraint set Φ_i contains a constraint of the form $x \stackrel{?}{/} \mathrm{d_j}\ \bar{u}_j : \mathrm{D}\ \bar{w}$ where $\mathrm{d_j}$ is a constructor of the datatype D, elaboration continues by performing a case split on x. For each constructor $\mathrm{d_j} : \Delta'_j \to \mathrm{D}\ \bar{w}'_j$, we use proof-relevant unification [Cockx and Devriese 2018] to determine whether this constructor can be used at indices $\bar{w}$. For each of the constructors for which unification succeeds positively, elaboration

continues to construct a subtree for that constructor.

$$\frac{\begin{array}{ll}(x : \mathrm{D}\ \bar{w}) \in \Gamma \qquad \mathrm{D} : \Psi \to \textsc{Type} & \\ \mathrm{d_j} : \Delta'_j \to \mathrm{D}\ \bar{w}'_j & \text{for } j = 1 \ldots l \\ \textsc{unify}(\Gamma\Delta'_j \vdash \bar{w} = \bar{w}'_j : \Psi) \Rightarrow \textsc{yes}(\Gamma_j, \sigma_j, \tau_j) & \text{for } j = 1 \ldots k \quad (k \le l) \\ \textsc{unify}(\Gamma\Delta'_j \vdash \bar{w} = \bar{w}'_j : \Psi) \Rightarrow \textsc{no} & \text{for } j = k+1 \ldots l \\ \Gamma_j \vdash \left\{\mathrm{c_i}\ \Delta_i\sigma_j\ [\Phi_i\sigma_j] | i = 1 \ldots m\right\} \rightsquigarrow Q_j & \text{for } j = 1 \ldots k\end{array}}{\Gamma \vdash \{\mathrm{c_i}\ \Delta_i\ [\Phi_i] | i = 1 \ldots m\} \rightsquigarrow \mathrm{case}_x\left\{\mathrm{d_j}\ \hat{\Delta}'_i \mapsto^{\tau_j} Q_j | j = 1 \ldots k\right\}}$$

The elaboration algorithm repeats the steps above whenever they are applicable until it either produces a complete case tree, or gets stuck because no rules apply.

We can make the above elaboration algorithm more powerful in various places by introducing additional squash operators:

- If not all constructor arguments are in sProp, we can squash the types of those that are not.
- If there are multiple constructors left but no unsolved constraints, we may take the disjoint sum of the constructor telescopes and squash the resulting type.
- If there are unsolved constraints but it is not possible to split on any variable, we may turn each remaining constraint $w \stackrel{?}{/} v : A$ into a new constructor argument of type $\|v =_A w\|$.

However, each of these options may reduce the ways in which we can eliminate the resulting type, so they may not always be desirable.

5.2 Generating the Constructors and the Eliminator

To make practical use of the type constructed by the elaboration algorithm from the previous section, we also need terms representing the constructors and the eliminator for the translated type. For example, for $m \le n$ we can define terms $\mathrm{lz} : (n : \mathbb{N}) \to 0 \le n$ and $\mathrm{ls} : (m\ n : \mathbb{N}) \to m \le n \to \mathrm{S}\ m \le \mathrm{S}\ n$ by $\mathrm{lz} = \lambda n.\ \mathrm{tt}$ and $\mathrm{ls} = \lambda m\ n\ p.\ p$ respectively. Note that these terms are type-correct since $0 \le n = \mathrm{sUnit}$ and $\mathrm{S}\ m \le \mathrm{S}\ n = m \le n$. We can also define the eliminator $\le\_\mathrm{rect}$ by performing the same case splits as in the translation of the type $\le$:

$$\begin{array}{l}\le\_\mathrm{rect} : (P : (m\ n : \mathbb{N}) \to m \le n \to \textsc{Type})(m_{\le 0} : (n : \mathbb{N}) \to P\ 0\ n\ (\mathrm{lz}\ n)) \\ \qquad\qquad (m_{\le \mathrm{S}} : (m\ n : \mathbb{N})(H : m \le n) \to P\ m\ n\ x \to P\ (\mathrm{S}\ m)\ (\mathrm{S}\ n)\ (\mathrm{ls}\ m\ n\ x)) \\ \qquad\qquad (m\ n : \mathbb{N})(x : m \le n) \to P\ m\ n\ x \\ \le\_\mathrm{rect}\ P\ m_{\le 0}\ m_{\le \mathrm{S}}\ m\ n\ x \\ \quad = \mathrm{case}_m\left\{\begin{array}{lcl} 0 & \mapsto & m_{\le 0}\ n \\ \mathrm{S}\ m' & \mapsto & \mathrm{case}_n\left\{\begin{array}{lcl} 0 & \mapsto & \mathrm{sEmpty\_rect}\ x \\ \mathrm{S}\ n' & \mapsto & m_{\le \mathrm{S}}\ m'\ n'\ x\ (\le\_\mathrm{rect}\ P\ m_{\le 0}\ m_{\le \mathrm{S}}\ m'\ n'\ x)\end{array}\right\}\end{array}\right\}\end{array}$$

Since the eliminator is the defining property of a datatype, being able to construct the eliminator shows the correctness of the translation. In particular, the eliminator can be used to show that the generated type is equivalent to its inductive version.

Constructing the constructors. Let $\mathrm{c} : \Delta \to \mathrm{D}\ \bar{w}$ be one of the constructors of D. By construction, the case tree of D will have a leaf of the form $\mathrm{c}\ \Delta'$ where the variables bound by Δ' form a subset of those bound in Δ. We thus define the term c as $\lambda x_1\ \ldots\ x_n.\ (x_{i_1}, \ldots, x_{i_m})$ where $\Delta = (x_1 : A_1) \ldots (x_n : A_n)$ and $\Delta' = (x_{i_1} : A'_{i_1}) \ldots (x_{i_m} : A'_{i_m})$.

Beware, as it is not immediately obvious that this term is type-correct: A'_i is not necessarily equal to A_i, since the variables in $\Delta \backslash \Delta'$ have been substituted in the process. However, since the Solve Constraint step only applies when both sides of the constraint are variables, this substitution is just a renaming. We can apply the same renaming to Δ', thus ensuring that the term c is indeed of type $\Delta \to \mathrm{D}\ \bar{w}$.

Constructing the eliminator. The eliminator D_rect for the elaborated datatype $\mathrm{D} : \Gamma \to \textsc{Type}$ with constructors $\mathrm{c_i} : \Delta_i \to \mathrm{D}\ \bar{v}_i$ for $i = 1 \ldots k$ takes the following arguments:

- The *motive* $P : \Gamma \to \mathrm{D}\ \hat{\Gamma} \to \textsc{Type}$
- The *methods*

$$\begin{aligned} m_i \quad : \quad & \Delta_i \to (H_1 : \Psi_{i1} \to P\ \bar{w}_{i1}\ (x_{ij_1}\ \hat{\Psi}_{i1})) \to \ldots \to \\ & (H_{q_i} : \Psi_{iq_i} \to P\ \bar{w}_{iq_i}\ (x_{ij_{q_i}}\ \hat{\Psi}_{iq_i})) \to P\ \bar{v}_i\ (\mathrm{c_i}\ \hat{\Delta}_i) \end{aligned}$$

  where $(x_{ij_p} : \Psi_{ip} \to \mathrm{D}\ \bar{w}_{ip}) \in \Delta_i$ for $p = 1 \ldots q_i$ are the recursive arguments of the constructor $\mathrm{c_i}$.

and produces a function of type $\Gamma \to (x : \mathrm{D}\ \hat{\Gamma}) \to P\ \hat{\Gamma}\ x$.

To construct this eliminator, we proceed by induction on the case tree defining D: for each case split in the elaboration of D, we perform the same case split on the corresponding index in Γ. At each leaf, we are either in an empty node or a constructor node.

- In an empty node, we have x : sEmpty, hence we can conclude by sEmpty_rect.
- In a constructor node for constructor $\mathrm{c_i}$, x is a nested tuple consisting of the non-forced arguments of $\mathrm{c_i}$. Moreover, the remaining telescope of indices Γ' contains the forced arguments of $\mathrm{c_i}$. We thus apply the motive m_i to arguments for Δ_i taken from x and Γ'. For the induction hypotheses H_p we call the eliminator recursively with arguments $\bar{w}_{ip}\ (x_{ij_p}\ \hat{\Psi}_{ip})$ where x_{ij_p} is again taken from either x or Γ'. Note that this recursion is well-founded if and only if the recursive definition of the datatype itself is so.

This finishes the construction of the eliminator. It can easily be checked that the eliminator we just constructed also has the appropriate computational behaviour:

$$\mathrm{D\_rect}\ P\ m_1\ \ldots\ m_k\ \bar{v}_i\ (\mathrm{c_i}\ \hat{\Delta}_i)$$

evaluates to

$$m_i\ \hat{\Delta}_i\ (\lambda \hat{\Psi}_{i1}.\ \mathrm{D\_rect}\ \bar{w}_{i1}\ (x_{ij_1}\ \hat{\Psi}_{i1}))\ \ldots\ (\lambda \hat{\Psi}_{iq_i}.\ \mathrm{D\_rect}\ \bar{w}_{iq_i}\ (x_{ij_{q_i}}\ \hat{\Psi}_{iq_i})).$$

6 IMPLEMENTATION IN Coq AND Agda

6.1 Implementation in Coq

The universe hierarchy. Coq comes with an infinite hierarchy of predicative universes $\textsc{Type}_i$ and an impredicative universe $\textsc{Prop} : \textsc{Type}_1$. Cumulativity makes each universe a subtype of the next, with Prop a subtype of $\textsc{Type}_0$. We add to this $\textsc{sProp} : \textsc{Type}_1$ but do not include sProp in the cumulativity relation. This lack of cumulativity is necessary to use relevance marks in the conversion (as already mentioned in Section 4.5).

The lack of cumulativity can cause issues during the elaboration phase, as the system assumes cumulativity throughout. For instance, consider the type of the dependent eliminator for sEmpty:

```
Definition sEmpty_rect_type := ∀ (P:sEmpty → Type) x, P x.
```

During elaboration an existential variable $?T : \textsc{Type}$ is generated for the type of x, then it must be unified with sEmpty : sProp. For this to be well-typed we must have cumulativity.

The system works around this issue by allowing cumulativity of sProp during elaboration but not when checking definitions. In the above case the use of cumulativity disappears when the existential variable is instantiated so there is no error. When cumulativity is truly necessary the error is delayed until the kernel check.

6.1.1 Conversion. Conversion in Coq is untyped so we use relevance marks as described in Section 4.5 to implement definitional proof irrelevance. Since Coq is a richer system than sMLTT there are a few additional cases. For instance, constants are annotated with their relevance as part of the context, and fixpoints are annotated as binders since they bind themselves in their bodies.

The change to the conversion algorithm is simple: without sProp, we reduce terms to weak head normal form then if they have the same shape we recurse for each pair of subterms. With sProp, we just insert a check for irrelevance before the reduction step.

Because we allow cumulativity during elaboration the generated relevance marks can be incorrect, leading to incompleteness of conversion during conversion. However the marks are fixed by the kernel.

6.1.2 Inductive types. The extensions with sEmpty, squash, box and irrelevant pairs are subsumed by restrictions on the inductive types which may be defined.

In the system without sProp, an inductive type may be defined at any universe level greater than the universes of the arguments of its constructors. As a special case, any inductive type may be defined in Prop but its elimination is restricted to Prop unless it satisfies singleton elimination: 0 or 1 constructor with all arguments in Prop. Inductive types in Prop with restricted elimination are called "squashed".

Coq also provides "primitive records" with surjective pairing. They are inductive types with exactly one constructor which has at least one argument (such arguments are called fields of the record), and must not be squashed. We extend this to handle sProp by considering sProp as smaller than other universes when checking the level of an inductive type. This allows defining box types:

```
Inductive Box (A:SProp) : Prop := box : A → Box A.
Definition unbox A (x:Box A) : A := match x with box y ⇒ y end.
```

The level sProp of the argument A is smaller than the level Prop of the inductive.

Like with Prop, any inductive type may be defined in sProp but its elimination is restricted to sProp when it is not the empty type. This amounts to implicitly using the squash (which can be already defined using the impredicative encoding). For primitive records in sProp we allow fields in sProp. This provides a generalization to the unit type and dependent sums. To get other inductive types in sProp that can be eliminated into any type, the user has to use the automatic encoding described in Section 5, implemented on top of the Equations plugin (see Section 6.4).

6.2 Example 1: Prime Numbers

To illustrate a simple use of sProp in Coq, together with the automatic translation of inductive types into fixpoints described in Section 5, let us consider the definition of primality.

First, we can define the $n \mid m$ predicate, which states that the natural number n is a divisor of m.

```
Inductive Divide : ℕ → ℕ → SProp:=
| divide₀ : ∀ n, Divide n 0
| divideS : ∀ n m (e: S n ≤ S m), Divide (S n) (m − n) → Divide (S n) (S m).
Infix "|" := Divide.
```

This definition satisfies the conditions described in Section 5, so the corresponding fixpoint definition can be automatically generated, together with its eliminator into Type.

Although we have definitional proof irrelevance for $n \mid m$, we can still extract the natural number that witnesses the fact that n is a divisor of m out of the proof that $n \mid m$, which is precisely the goal of the criterion developed in Section 5.

```
Definition divide_to_ℕ : ∀ n m, n | m → ℕ.
```

together with the correctness of this definition.

```
Lemma divide_to_ℕ_correct n m (e:n | m): divide_to_ℕ n m e * n = m.
```

Note however that being an element of sProp implies both proof irrelevance *and* computational irrelevance, so an element of m | n is completely erased during compilation. It does not store the quotient, but the (mere) property that the euclidean division of n by m has no remainder. Thus, divide_to_ℕ computes the quotient from m and n. Essentially finding the quotient by deconstructing the proof of n | m is no easier than by deconstructing n and m. It can be seen as a side condition to guarantee we have no missing case.

Then, it is easy to define primality in sProp in a way that satisfies the conditions described in Section 5.

```
Inductive is_gcd (a b g:ℕ) : SProp :=
 is_gcd_intro : g | a → g | b → (∀ x, x | a → x | b → x | g) → is_gcd a b g.

Inductive prime (p:ℕ) : SProp :=
  prime_intro : 1 < p → (∀ n, 1 ≤ n → n < p → is_gcd n p 1) → prime p.
```

This definition gives us definitional proof irrelevance for prime, without paying the price of the definition of a decision procedure into booleans (for instance using the sieve of Eratosthenes) and a proof that it corresponds to primality. Here, we have a direct definition instead, and the decision procedure is only a useful addition to prove primality of a particular natural number and may be implemented as a tactic.

6.3 Example 2: The Setoid Model

The previous example shows the use of sProp to define definitionally proof irrelevant predicates while still being able to extract relevant values out of them. We now turn to a more critical use of sProp to avoid higher coherence issues in syntactical models of type theory. As an example, using sProp, we can formally define in Coq the setoid translation presented on paper twenty years ago by Thorsten Altenkirch [Altenkirch 1999]. We define a setoid as a type carrier together with a notion of equality which is reflexive, symmetric, transitive, *and* definitionally proof irrelevant.

```
Record Setoid :=
  { carrier : Type;
    eq : carrier → carrier → SProp;
    refl : ∀ x, eq x x;
    sym : ∀ {x y}, eq x y → eq y x;
    trans : ∀ {x y z}, eq x y → eq y z → eq x z
  }.
```

This way, we can define a category with families (CwF), as introduced by Peter Dybjer [Dybjer 1996], where contexts are Setoid, types are setoid families (over a context) and terms are sections of setoid families. This CwF features dependent products Π, a universe, and identity types Eq. We can then prove that this CwF satisfies functional extensionality, thus providing a formal proof that functional extensionality is admissible in sMLTT.

6.4 Extension of the Equations Plugin

The Equations plugin of Coq [Mangin and Sozeau 2018] provides an implementation of dependent-pattern matching compilation. We reuse the tools of Equations to automatically derive the translation of inductive definitions that fit in sProp to fixpoint definitions. Concretely, we have extended Equations with a new Derive Inversion command that tries to produce a case tree from an inductive definition and succeeds if and only if the resulting recursive definition can be typed in sProp.

Using this tool, it is possible to automatically derive the definition of the recursive variants of $\leq$ and Divide, along with their constructors. Coming back to the example of $\leq_{\text{fix}}$, our framework provides automatically the two definitions corresponding to the two constructor of $\leq$:

```
Definition ≤fix0 : ∀ n, 0 ≤fix n := fun n ⇒ I.
Definition ≤fixS : ∀ m n, m ≤fix n → S m ≤fix S n := fun n m e ⇒ e.
```

We are currently also working on implementing the automatic generation of the eliminator as described in Section 5.2.

6.5 Implementation in Agda

Aside from adding sProp to Coq, we also implemented a variant of the same concept for Agda. Since previously there was no Prop universe, the sProp universe is simply called Prop in Agda. This new universe has been integrated in the current development version of Agda on https://github.com/agda/agda and is planned to be released as part of Agda 2.6.0 in November 2018.

User perspective. On the syntactic level, the main change to Agda is the addition of the new sort Prop next to the old sort Set. Unlike sProp in Coq, Prop in Agda is *predicative*. For example, $(A : \text{Prop}) \to A$ is not itself an element of Prop. Instead, there is a hierarchy of universes Prop_0 $(= \text{Prop})$, Prop_1, Prop_2, ...analogous to the hierarchy of Set_i. Like Set, Prop also supports universe polymorphism, so for each $\ell : \text{Level}$ we have the sort $\text{Prop}\ \ell$. As an example, we can define the universe polymorphic squash type and its eliminator:

```
data Squash {ℓ}(A : Set ℓ) : Prop ℓ where
  sq : A → Squash A

unsquash : ∀{ℓ ℓ'}{A : Set ℓ}(P : Prop ℓ')(A → P) → Squash A → P
unsquash P f (sq x) = f x
```

Note that this is more powerful than the predicative encoding of the squash type described in Section 3.4 since we can eliminate it into any Prop_i.

Like in the Coq implementation, there is no implicit conversion from Prop to Set. Instead, Prop can be embedded through the definition of a record type $\text{Box}\ (A : \text{Prop}) : \text{Set}$ with one field $\text{unbox} : A$. More generally, when defining a record type in Set_i, the fields can be in both Prop_j or Set_j for any $j \leq i$. On the other hand, for record types in Prop all fields must be in Prop themselves.

Implementation details. Since Agda uses a type-directed conversion check internally and types are annotated with their sorts, adding the rule that any two elements of a type in Prop are equal was straightforward. In particular, the relevance marks used in the Coq implementation are not required here. In contrast, making Prop impredicative is currently problematic since Agda's termination checker assumes predicativity. This was our main reason to implement the predicative hierarchy Prop_i instead.

Interaction with irrelevant arguments. As mentioned in the introduction, Agda has another another facility for definitional proof irrelevance in the form of irrelevant function types [Abel and

Scherer 2012] and irrelevant fields. Prop can be used in situations where Agda's pre-existing irrelevance cannot, for example:

- We can use Prop to give a function an irrelevant return type.
- We can define new datatypes and record types in Prop, such as Squash.
- We can construct new types in Prop by pattern matching, such as $\leq$.
- We can quantify over all types in Prop.

In our implementation, it is allowed to use irrelevant arguments when constructing an element of a type in Prop. This allows us to convert irrelevant arguments into elements of the squash type:

$$\begin{aligned}&\mathsf{irrToSquash} : \{A : \mathsf{Set}\} \to .A \to \mathsf{Squash}\ A\\&\mathsf{irrToSquash}\ x\ =\ \mathsf{sq}\ x\end{aligned}$$

In fact, irrelevant functions and irrelevant fields can be encoded in terms of Prop by using the squash type: each irrelevant argument or field of type $.A$ is turned into a squashed argument Squash A. This encoding uses the fact that Squash is an applicative functor with identity $\mathsf{sq} : A \to \mathsf{Squash}\ A$ and sequential application

$$\begin{aligned}&\_<^*>\_ : \{A\ B : \mathsf{Set}\} \to \mathsf{Squash}\ (A \to B) \to \mathsf{Squash}\ A \to \mathsf{Squash}\ B\\&\mathsf{sq}\ f\ <^*>\ \mathsf{sq}\ x\ =\ \mathsf{sq}\ (f\ x)\end{aligned}$$

For example, if $g : .B \to C$ then the expression $\lambda f\ x.\ g\ (f\ x)$ of type $.(A \to B) \to A \to C$ is encoded by $\lambda f\ x.\ g\ (f\ <^*>\ \mathsf{sq}\ x)$ of type $\mathsf{Squash}\ (A \to B) \to A \to C$.

In addition to irrelevant functions and fields, Agda also provides experimental support for irrelevant definitions and irrelevant projections, which were already mentioned in the introduction. In contrast to irrelevant functions, not everything that's possible with these features can be encoded in terms of Squash, even after fixing the bug mentioned in the introduction. In particular, they can be used to construct a function $.\mathsf{choice} : .A \to A$ for any type A, but the corresponding statement $\mathsf{Squash}\ (\mathsf{Squash}\ A \to A)$ is not provable. If this were postulated as an axiom, we conjecture that these experimental ways of using irrelevance could be encoded in terms of Squash as well.

7 CONCLUSION AND FUTURE WORK

We have given a general way to extend a type theory with a predicative hierarchy of universes (or an impredicative universe) satisfying definitional proof irrelevance, while keeping decidability of type checking. We have shown various metatheoretical properties of our extension using syntactic translations into extensional type theories ETT and HTS. We have then described a new way to decide whether a proposition can be eliminated over a type using techniques coming from recent development on dependent pattern matching without UIP. We have implemented our approach both in Coq and in Agda, and illustrated its usefulness on several examples, in particular to formalize the setoid model of type theory, which can not be done without definitional proof irrelevance.

Regarding future work, the main extension of our framework that we would like to address in a near future is the definition of a strict equality that remains compatible with univalence. The idea is to detect syntactically which types are mere sets—using again ideas coming from dependent pattern matching—in order to allow elimination of strict equality on those types. This should require the addition of a hierarchy of universes of mere (strict) sets, and maybe other hierarchies for higher homotopy levels as well, but it is not clear how to do it in a good way at the moment.

A LEAN SUBJECT REDUCTION FAILURE

```
axiom A : Type
```

```
axiom r : A → A → Prop

axiom C : A → Type
axiom F : forall x, (forall y, r y x → C y) → C x

lemma fix_F_eq1 (x : A) (acx : acc r x) :
  well_founded.fix_F F x acx =
  well_founded.fix_F F x (acc.intro x (λ y, acc.inv acx)) := eq.refl _

lemma fix_F_eq2 (x : A) (acx : acc r x) :
  well_founded.fix_F F x (acc.intro x (λ y, acc.inv acx)) =
  F x (λ (y : A) (p : r y x), well_founded.fix_F F y (acc.inv acx p))
   := eq.refl _

lemma fix_F_eq3 (x:A) (acx : acc r x) :
  well_founded.fix_F F x acx =
  F x (λ (y : A) (p : r y x), well_founded.fix_F F y (acc.inv acx p))
  := eq.trans (fix_F_eq1 x acx) (fix_F_eq2 x acx)

lemma fix_F_eq4 (x:A) (acx : acc r x) :
  well_founded.fix_F F x acx =
  F x (λ (y : A) (p : r y x), well_founded.fix_F F y (acc.inv acx p))
  := eq.refl _ -- fails
```

ACKNOWLEDGMENTS

The authors want to thank Simon Boulier for his implementation of the Setoid model.

REFERENCES

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of Conversion for Type Theory in Type Theory. *Proc. ACM Program. Lang.* 2, POPL, Article 23 (Jan. 2018), 29 pages. DOI:http://dx.doi.org/10.1145/3158111

Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science* 8, 1 (03 2012). DOI:http://dx.doi.org/10.2168/lmcs-8(1:29)2012

T. Altenkirch. 1999. Extensional equality in intensional type theory. In *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*. 412–420. DOI:http://dx.doi.org/10.1109/LICS.1999.782636

Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. 2016. Extending Homotopy Type Theory with Strict Equality. In *CSL*.

Steven Awodey and Andrej Bauer. 2004. Propositions As [Types]. *J. Log. and Comput.* 14, 4 (Aug. 2004), 447–471. DOI: http://dx.doi.org/10.1093/logcom/14.4.447

Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The Next 700 Syntactical Models of Type Theory. In *Proceedings of Certified Programs and Proofs*. ACM, 182–194.

Edwin Brady, Conor McBride, and James McKinna. 2004. Inductive Families Need Not Store Their Indices. In *Types for Proofs and Programs*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 115–129.

Paolo Capriotti. 2017. *Models of type theory with strict equality.* Ph.D. Dissertation. University of Nottingham.

Jesper Cockx and Andreas Abel. 2018. Elaborating Dependent (Co)pattern matching. In *Proceedings of the 23th ACM SIGPLAN Conference on Functional Programming (ICFP 2018)*. ACM Press, St. Louis, Missouri, United States.

Jesper Cockx and Dominique Devriese. 2018. Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. *Journal of Functional Programming* 28 (2018), e12. DOI:http://dx.doi.org/10.1017/S095679681800014X

Jesper Cockx, Dominique Devriese, and Frank Piessens. 2014. Pattern matching without K. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 257–268.

Thierry Coquand. 2016. Universe of Bishop sets. (2016). www.cse.chalmers.se/~coquand/bishop.pdf.

Peter Dybjer. 1996. Internal type theory. In *Types for Proofs and Programs*, Stefano Berardi and Mario Coppo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 120–134.

Martin Hofmann. 1995. Conservativity of equality reflection over intensional type theory. In *International Workshop on Types for Proofs and Programs*. Springer, 153–164.

P. Letouzey. 2004. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. Ph.D. Dissertation. Université Paris-Sud.

Cyprien Mangin and Matthieu Sozeau. 2018. Equations Reloaded. (2018). http://mattam82.github.io/Coq-Equations/

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:http://dx.doi.org/https://doi.org/10.1016/S0049-237X(08)71945-1

Frank Pfenning. 2001. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*. IEEE Computer Society, Washington, DC, USA, 221–. http://dl.acm.org/citation.cfm?id=871816.871845

Aaron Stump. 2018. From realizability to induction via dependent intersection. *Ann. Pure Appl. Logic* 169, 7 (2018), 637–655. DOI:http://dx.doi.org/10.1016/j.apal.2018.03.002

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study.

Vladimir Voevodsky. 2011. Resising Rules - their use and semantic justification. www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/2011_Bergen.pdf. (2011).

Vladimir Voevodsky. 2013. A simple type system with two identity types. (2013). https://ncatlab.org/homotopytypetheory/files/HTS.pdf

Benjamin Werner. 2008. On the Strength of Proof-Irrelevant Type Theories. 4 (09 2008), 1–20.