



**HAL**  
open science

## Definitional Proof-Irrelevance without $K$

Gaëtan Gilbert, Jesper Cockx, Sozeau Matthieu, Nicolas Tabareau

► **To cite this version:**

Gaëtan Gilbert, Jesper Cockx, Sozeau Matthieu, Nicolas Tabareau. Definitional Proof-Irrelevance without  $K$ . 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2019, Jan 2019, Lisbon, Portugal). hal-01859964v1

**HAL Id: hal-01859964**

**<https://inria.hal.science/hal-01859964v1>**

Submitted on 22 Aug 2018 (v1), last revised 16 Oct 2018 (v2)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Definitional Proof-Irrelevance without K

GAËTAN GILBERT, Gallinette Project-Team, Inria, France

JESPER COCKX, Chalmers / Gothenburg University, Sweden

MATTHIEU SOZEAU, Pi.R2 Project-Team, Inria and IRIF, France

NICOLAS TABAREAU, Gallinette Project-Team, 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.

## 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 Construction (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.

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.

---

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

---

2019. 2475-1421/2019/1-ART1 \$15.00

<https://doi.org/>

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 `e₂` 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<sup>1</sup>) 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 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 build 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.

<sup>1</sup><http://mattam82.github.io/Coq-Equations/>

**Definition** Squash  $(T : \text{Type}) : \text{Prop} := \forall P : \text{Prop}, (T \rightarrow P) \rightarrow P$ .

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

$$\forall (T : \text{Type}) (P : \text{Prop}), (T \rightarrow P) \rightarrow \text{Squash } T \rightarrow P.$$

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 since a long time, and various work 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  $(A : \text{Type}) (x : A) : A \rightarrow \text{Prop} := \text{eq\_refl} : \text{eq } A \ x \ x$

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  $(A : \text{Type}) (R : A \rightarrow A \rightarrow \text{Prop}) (x : A) : \text{Prop} :=$   
 Acc\_intro :  $(\forall y : A, R \ y \ x \rightarrow \text{Acc } R \ y) \rightarrow \text{Acc } R \ x$

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.<sup>2</sup> 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  $\leq$  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 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

<sup>2</sup>See a description of this issue in <https://github.com/leanprover/lean/issues/654>.

. in the definition of  $\cdot.(n \leq k)$ :

```
data boundedNat (k : ℕ) : Set where
  pair : (n : ℕ) →  $\cdot.(n \leq k)$  → 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}(nm : 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.<sup>3</sup> 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<sup>4</sup>, which, as observed recently by the Agda community<sup>5</sup>, 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 ( Bool → Squash Bool )
          (λi → (Squash Bool → Bool) (λu → (a : Bool) → u (i a) ≡ a)))
bizarre = sq (sq, unsq, (λa → 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 `sProp` 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. This criterion uses the general methodology

<sup>3</sup>Agda 2.2.8, see <https://github.com/agda/agda/blob/master/CHANGELOG.md#release-notes-for-agda-2-version-228>.

<sup>4</sup>Added in Agda 2.2.10.

<sup>5</sup>Issue #2170 at <https://github.com/agda/agda/issues/2170>.

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.

The Coq and Agda formalizations are available as a virtual machine ([https://drive.google.com/open?id=16z7nhdWjgWAfGMJhqia9a404FyLsi\\_Q2](https://drive.google.com/open?id=16z7nhdWjgWAfGMJhqia9a404FyLsi_Q2)) or as non-anonymous source code.

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**  $\text{hProp} := \{ A : \text{Type} \ \& \ \forall x \ y : A, \ x = y \}$ .

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**  $\text{sUnit} : \text{sProp} := \text{tt} : \text{sUnit}$ .

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  $\eta$ -law such that  $u \equiv \text{tt}$  for any  $u : \text{sUnit}$ . But in general,

246 it should not be expected that any contractible type in `sProp` can be eliminated to an  
 247 arbitrary type, as we can see below.

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

250 **Definition** `Sing (A : Type) (a : A) := { b : A & a = b }.`  
 251

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

259  $(a ; \text{refl}) \equiv (b ; e) : \text{Sing } A \ a \Rightarrow \pi_1 (a ; \text{refl}) \equiv \pi_1 (b ; e) : A \Rightarrow a \equiv b : A$   
 260

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

### 263 2.3 Flirting with Undecidability

264 Let us now look at other inductive types that are mere proposition without being con-  
 265 tractible. The first canonical example is the empty type

266 **Inductive** `sEmpty : sProp := .`  
 267

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

270 `sEmpty_rect : ∀ T : Type, sEmpty → T.`  
 271

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

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

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

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

288 **Definition** `Acc_inv A R (x : A) (X : Acc x) : ∀ y:A, R y x → Acc y :=`  
 289

290 `Acc_rect (fun x => ∀ y, R y x → Acc y) (fun x X _ => X) x X.`  
 291

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

293 <sup>6</sup>This remark is originally due to Peter LeFanu Lumsdaine.  
 294

295  $\text{Acc\_rect} : \forall P : A \rightarrow \text{Type}, (\forall x : A, (\forall y : A, R\ y\ x \rightarrow \text{Acc}\ y) \rightarrow (\forall y : A, R\ y\ x \rightarrow$   
 296  $P\ y) \rightarrow P\ x)$   
 297  $\rightarrow \forall x : A, \text{Acc}\ x \rightarrow P\ x$

298 But then, from this inversion and using definitional proof irrelevance, the following defini-  
 299 tional equality is derivable, for any predicate  $P : A \rightarrow \text{Type}$  and function  $F : \forall x, (\forall y, R$   
 300  $y\ x \rightarrow P\ y) \rightarrow P\ x$  and  $X : \text{Acc}\ x$

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

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

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

310 and the unfolding can start again for ever.

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

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

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

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

335 **Inductive**  $\leq_{bad} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{sProp} :=$   
 336  $\quad | \leq_{bad\_refl} : \forall n, n \leq_{bad}\ n$   
 337  $\quad | \leq_{bad} S : \forall m\ n, m \leq_{bad}\ n \rightarrow m \leq_{bad}\ S\ n.$

338 It can also be shown that  $m \leq_{bad}\ n$  is a mere proposition, but  $\leq$  and  $\leq_{bad}$  do not share  
 339 the same inversion principle. Indeed, in the (absurd) context that  $e : S\ n \leq_{bad}\ n$ , there are  
 340 two ways to form a term of type  $S\ n \leq_{bad}\ S\ n$ , either by using  $\leq_{bad\_refl}\ (S\ n)$  or by using

342 <sup>7</sup>This terminology has been introduced by [Brady et al. 2004].

344  $\leq_{bad} S (S\ n)\ n\ e$ . This means that allowing  $m \leq_{bad} n$  to be eliminated into any type would  
 345 require to decide whether the context is absurd or not, which is obviously not a decidable  
 346 property of type theory. For  $m \leq n$  the situation is different, because the return type of the  
 347 two constructors  $\leq 0$  and  $\leq S$  are orthogonal, in the sense they cannot be unified.

#### 348 2.4 Dependent Pattern Matching to the Rescue

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

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

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

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

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

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

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

380 Fail **Equations**  $Acc' (x: A) : sProp :=$   
 381  $Acc' x := (\forall y:A, R\ y\ x \rightarrow Acc' y).$

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

387 **Equations**  $\leq_{fix} (n\ m : \mathbb{N}) : sProp :=$

388



### 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  $\Gamma$  is well formed, while  $\Gamma \vdash M : A$  means that the term  $M$  has type  $A$  in environment  $\Gamma$  and the statement  $\Gamma \vdash A \equiv B$  means  $A$  is convertible to  $B$  in context  $\Gamma$ .

MLTT features dependent products, (a negative presentation of) dependent sums and a predicative hierarchy of universes  $\text{Type}_i$ . The conversion judgment features  $\beta$ -reduction, traditional rules of computation for dependent sums (surjective pairing and projections), together with congruence rules for every constructor. As usual, we note  $A \rightarrow B$  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  $\text{sProp}_i$  by adding the rule

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

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

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

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:

$$\frac{\Gamma \vdash A : \text{sProp}_i \quad \Gamma \vdash x : A \quad \Gamma \vdash y : A}{\Gamma \vdash x \equiv y : A}$$

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):

$$\frac{}{\Gamma \vdash \text{sEmpty} : \text{sProp}_i} \quad \frac{\Gamma \vdash A : \text{Univ}_i \quad \Gamma \vdash e : \text{sEmpty}}{\Gamma \vdash \text{sEmpty\_rect } A \ e : A}$$

491 This is enough to define many other types in sProp. For instance sUnit is just sEmpty  $\rightarrow$   
 492 sEmpty (whose inhabitant  $\lambda x : \text{sEmpty}. x$  is unique up to conversion due to proof irrele-  
 493 vance).

### 494 3.4 Squash

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

$$\begin{array}{c}
 \frac{\Gamma \vdash A : \text{Type}_i}{\Gamma \vdash \|A\| : \text{sProp}} \qquad \frac{\Gamma \vdash x : A}{\Gamma \vdash \text{sq } x : \|A\|} \\
 \\
 \frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma \vdash P : \text{sProp} \quad \Gamma \vdash f : A \rightarrow P \quad \Gamma \vdash x : \|A\|}{\Gamma \vdash \text{unsq } P f x : P}
 \end{array}$$

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

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

$$\begin{array}{lcl}
 508 & \|A\| & := \Pi(P : \text{sProp}), (A \rightarrow P) \rightarrow P \\
 509 & \text{sq } x & := \lambda P : \text{sProp}. \lambda f : A \rightarrow P. f x \\
 510 & \text{unsq } P f x & := x P f
 \end{array}$$

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

$$\begin{array}{c}
 \frac{\Gamma \vdash A : \text{Type}_i}{\Gamma \vdash \|A\|_{i,j} : \text{Prop}_{\max(i,j+1)}} \\
 \\
 \frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma \vdash P : \text{sProp}_j \quad \Gamma \vdash f : A \rightarrow P \quad \Gamma \vdash x : \|A\|_{i,j} \quad j < i}{\Gamma \vdash \text{unsq } P f x : P}
 \end{array}$$

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

### 522 3.5 Box

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

$$\begin{array}{c}
 \frac{\Gamma \vdash A : \text{Prop}_i}{\Gamma \vdash \Box A : \text{Type}_i} \qquad \frac{\Gamma \vdash x : A}{\Gamma \vdash \text{box } x : \Box A} \\
 \\
 \frac{\Gamma \vdash A : \text{sProp}_i \quad \Gamma \vdash P : \Box A \rightarrow \text{Type}_j \text{ or } \Gamma \vdash P : \Box A \rightarrow \text{sProp}_j \quad \Gamma, x : A \vdash f : P (\text{box } x) \quad \Gamma \vdash x : \Box A}{\Gamma \vdash \text{unbox } P f x : P x}
 \end{array}$$

534 together with the conversion rule

$$535 \qquad \Gamma \vdash \text{unbox } P f (\text{box } a) \equiv f a$$

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

539

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  $\Sigma(x : A).\text{Unit}$  or  $\Sigma(x : \text{Unit}).A$ , and instead of  $\Sigma(x : A).B$  with  $A : \text{sProp}$  we could use  $\Sigma(y : \Box A).B[x := \text{unbox } A (\lambda z : A. z) y]$ .

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

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

$A \rightarrow A \rightarrow \text{sProp}$  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:

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

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:

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

Note that  $\pi_{s2}$  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:

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

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

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

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

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

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

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  $\eta$ -law for functions, one can derive functional extensionality, that is given two functions  $f, g : \Pi x : A. B$  there is a term

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

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.

638	$[\text{Type}_i]$	$:= \Sigma A : \text{Type}_i. \text{Unit}$
639	$[\text{sProp}_i]$	$:= \Sigma A : \text{Type}_i. \Pi x y : A. x = y$
640	$[x]$	$:= x$
641	$[\lambda x : A. B : \text{Type}]$	$:= (\Pi x : \llbracket A \rrbracket. \llbracket B \rrbracket; \text{tt})$
642	$[\Pi x : A. B : \text{sProp}]$	$:= (\Pi x : \llbracket A \rrbracket. \llbracket B \rrbracket;$
643		$\lambda f g : \Pi x : \llbracket A \rrbracket. \llbracket B \rrbracket. \pi_{\Pi} \llbracket A \rrbracket \llbracket B \rrbracket_{\text{pf}} f g)$
644	$[\lambda x : A. M]$	$:= \lambda x : \llbracket A \rrbracket. \llbracket M \rrbracket$
645	$[MN]$	$:= \llbracket M \rrbracket \llbracket N \rrbracket$
646	$[\text{sEmpty}]$	$:= (\text{Empty}; \lambda x y : \text{Empty}. \pi_{\text{Empty}} x y)$
647	$[\text{sEmpty\_rect } P t]$	$:= \text{Empty\_rect } [P] [t]$
648	$[\Box A]$	$:= (\pi_1[A], \text{tt})$
649	$[\text{box } x]$	$:= [x]$
650	$[\text{unbox } P f a]$	$:= [f] [a]$
651	$[\Sigma x : A. B : \text{Type}]$	$:= (\Sigma x : \llbracket A \rrbracket. \llbracket B \rrbracket; \text{tt})$
652	$[\Sigma x : A. B : \text{sProp}]$	$:= (\Sigma x : \llbracket A \rrbracket. \llbracket B \rrbracket;$
653		$\lambda X Y : \Sigma x : \llbracket A \rrbracket. \llbracket B \rrbracket. \pi_{\Sigma} \llbracket A \rrbracket_{\text{pf}} \llbracket B \rrbracket_{\text{pf}} X Y)$
654	$[\lambda x : A. M]$	$:= \lambda x : \llbracket A \rrbracket. \llbracket M \rrbracket$
655	$[MN]$	$:= \llbracket M \rrbracket \llbracket N \rrbracket$
656	$\llbracket A \rrbracket$	$:= \pi_1[A]$
657	$\llbracket A \rrbracket_{\text{pf}}$	$:= \pi_2[A]$
658		
659	$\pi_{\text{Empty}} x y$	$:= \text{Empty\_rect } (x = y) x$
660	$\pi_{\Pi} A \pi_B f g$	$:= \text{funext } (\lambda x : A. \pi_B(f x)(g x))$
661	$\pi_{\Sigma} \pi_A \pi_B X Y$	$:= \text{eq}_{\Sigma} (\pi_A(\pi_1 X)(\pi_1 Y)) (\pi_B(\pi_2 X)(\pi_2 Y))$
662		
663		
664		
665		
666		
667		
668		
669		
670		
671		
672		
673		

Fig. 3. Syntactical Translation from sMLTT to ETT

## 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:

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

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.

687 The translation is described in Figure 3, where  $\text{eq}_\Sigma$  is defined as the witness that equal-  
 688 ity between elements of an dependent sum is given by the equality of the corresponding  
 689 projections (which is provable using  $J$ )

$$690 \quad \text{eq}_\Sigma : \Pi x y : \Sigma a : A. B. \pi_1 x =_A \pi_1 y \rightarrow \pi_2 x =_{B\{a := \pi_1 x\}} \pi_2 y \rightarrow x =_{\Sigma a:A. B} y.$$

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

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

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

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

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

712 Proof. By induction on the typing derivation.  $\square$

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

$$720 \quad [\Gamma] \vdash [\Pi x : A. B] : \Sigma A : \text{Type}_i. \Pi x y : A. x = y$$

722 To remedy this situation, we can make use of the propositional resizing rule introduced by  
 723 Vladimir Voevodsky [Voevodsky 2011], which says that every mere proposition can be put  
 724 in the universe  $\text{Type}_0$ .

$$725 \quad \frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma \vdash e : \Pi x y : A. x =_A y}{\Gamma \vdash \text{rr}(A, e) : \text{Type}_0}$$

728 The propositional resizing is justified by any classical model of ETT, that is a model which  
 729 satisfies the law of excluded middle, because in such a model, every mere proposition is  
 730 either sEmpty or Unit and thus lives in the lowest universe.<sup>9</sup> Using propositional resizing,  
 731

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

733 <sup>9</sup>Vladimir Voevodsky has also introduced other resizing rules involving type equivalences which are a bit  
 734 more controversial as their justifications have not been completely written up.

$$\begin{array}{ll}
\text{736} & [\text{Type}_i]_{HTS} & := \Sigma A : \text{FType}_i. \text{Unit} \\
\text{737} & & \\
\text{738} & [\text{sProp}_i]_{HTS} & := \Sigma A : \text{FType}_i. \Pi x y : A. x = y \\
\text{739} & & 
\end{array}$$

Fig. 4. Syntactical Translation from sMLTT to HTS

it is thus possible to define the coercion<sup>10</sup>

$$\begin{array}{l}
\text{744} \quad \text{rr}_{\text{sProp}} : (\Sigma A : \text{Type}_i. \Pi x y : A. x = y) \rightarrow \Sigma A : \text{Type}_0. \Pi x y : A. x = y \\
\text{745} \quad \quad \quad := \lambda A : \_ . (\text{rr}(\pi_1 A, \pi_2 A)), \lambda x y : \text{rr}(\pi_1 A, \pi_2 A). \pi_2 A x y \\
\text{746}
\end{array}$$

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 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  $\text{Type}_i$  for which equality is strict, and fibrant types which leave in  $\text{FType}_i$  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:

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

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  $[\text{sProp}_i]_{HTS}$  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

$$[\text{sProp}_i]_{HTS} : \text{FType}_i.$$

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).

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

$$\begin{array}{c}
785 \quad \Gamma \vdash A : \text{Type}_i \quad \Gamma \vdash x : A \quad \Gamma \vdash y : A \\
786 \quad \hline \\
787 \quad \Gamma \vdash x =_A^s y : \text{sProp}_i \\
788 \\
789 \quad \Gamma \vdash e : M =_A^s N \quad \Gamma, y : A, q : M =_A^s y \vdash P : \text{Type}_i \quad \Gamma \vdash u : P\{y := M\}\{q := \text{refl}_M^s\} \\
790 \quad \hline \\
791 \quad \Gamma \vdash J_s(y.q.P, e, u) : P\{y := N\}\{q := e\} \\
792 \\
793 \quad \Gamma \vdash J_s(y.q.P, \text{refl}_M, u) \equiv u \\
794 \\
795 \\
796 \\
797 \\
798 \\
799 \\
800 \\
801 \\
802 \\
803 \\
804 \\
805 \\
806 \\
807 \\
808 \\
809 \\
810 \\
811 \\
812 \\
813 \\
814 \\
815 \\
816 \\
817 \\
818 \\
819 \\
820 \\
821 \\
822 \\
823 \\
824 \\
825 \\
826 \\
827 \\
828 \\
829 \\
830 \\
831 \\
832 \\
833
\end{array}$$

Fig. 5. Definition of a strict equality type.

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

#### 4.4 Strict Identity and UIP

We now consider an optional extension, adding propositional equality in  $\text{sProp}$ . 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.

In this context, the computation rule for equality in  $\text{sProp}$  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

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

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

#### 4.5 Relevance Marks and Decidability of Type Checking

A naïve implementation of type theory with  $\text{sProp}$  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

834 syntax with sProp and the inference rules for adding a variable to a context are described  
 835 in Figure 6. Adapting the other rules and adding extensions is straightforward and left as  
 836 an exercise to the reader.

$$\begin{array}{l}
 837 \\
 838 \quad * ::= \text{Relevant} \mid \text{Irrelevant} \\
 839 \\
 840 \quad A, B, M, N ::= \text{sProp}_i \mid \text{Type}_i \mid x \mid M N \mid \lambda x^* : A. M \\
 841 \quad \quad \quad \mid \Pi x^* : A. B \mid \Sigma x : A. B \mid \pi_1 M \mid \pi_2 M \mid (M, N) \\
 842 \\
 843 \quad \Gamma, \Delta ::= . \mid \Gamma, x^* : A \\
 844 \\
 845 \quad \frac{\Gamma \vdash A : \text{Type}_i}{\vdash \Gamma, x^{\text{Relevant}} : A} \qquad \frac{\Gamma \vdash A : \text{sProp}_i}{\vdash \Gamma, x^{\text{Irrelevant}} : A}
 \end{array}$$

848 Fig. 6. sMLTT with relevance marks

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

- 853 •  $\text{relevance}_\Gamma(x) := *$  when  $x^* \in \Gamma$
- 854 •  $\text{relevance}_\Gamma(M N) := \text{relevance}_\Gamma(M)$
- 855 •  $\text{relevance}_\Gamma(\lambda x^* : A. M) := \text{relevance}_{\Gamma, x^* : A}(M)$
- 856 • Everything else (sorts, product types, relevant sigma types, relevant pairs and relevant  
 857 projections) is relevant.

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

862 Proposition 4.4. For  $M$  well typed in  $\Gamma$ ,  $\text{relevance}_\Gamma(M) = \text{Irrelevant}$  if and only if there  
 863 exists  $A$  such that  $\Gamma \vdash M : A$  and  $\Gamma \vdash A : \text{sProp}$ .

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

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

874 Using relevance marks, we can replay the proof of decidability of type checking of Andreas  
 875 Abel and Gabriel Scherer using algorithmic equality and logical relations [Abel and Scherer  
 876 2012]—for sMLTT without a strict equality, as the work of Abel et al. does not deal with  
 877 identity types.

878 Their setting was designed to handle irrelevant arguments, where irrelevance was marked  
 879 on the typing annotation  $\Gamma \vdash t \div A$  which says that  $t$  is used irrelevantly. Here we replace  
 880 this irrelevance annotation by  $\Gamma \vdash t : A \wedge \text{relevance}_\Gamma(t) = \text{Irrelevant}$ . That is, the annotation  
 881 is not on the typing judgment but directly on the term. Concretely, we replace the rule for  
 882

dealing with irrelevance in the definition of structural equality on neutral terms by

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

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. In this section, we give a 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.

### 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:

**Equations**  $\leq_{\text{fix}} (n \ m : \mathbb{N}) : \text{sProp} :=$   
 $0 \leq_{\text{fix}} n := \text{sUnit};$   
 $\text{S } m \leq_{\text{fix}} \text{S } n := m \leq_{\text{fix}} n;$   
 $\text{S } \_ \leq_{\text{fix}} 0 := \text{sEmpty}.$

One challenge in this translation is to determine which constructor arguments should appear on the right-hand side: for the constructor  $\leq\text{S}$  (in the second equation), the argument of type  $m \leq n$  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  $\leq$ :

$$m \leq n := \text{case}_m \left\{ \begin{array}{l} 0 \quad \mapsto \leq 0 \\ \text{S } m' \quad \mapsto \text{case}_n \left\{ \begin{array}{l} 0 \quad \mapsto \perp \\ \text{S } n' \quad \mapsto \leq\text{S } (p : m' \leq n') \end{array} \right\} \end{array} \right\}$$

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

node  $\perp$ , or an internal node of the form

$$\text{case}_x \{c_1 \hat{\Delta}_1 \mapsto^{\tau_1} Q_1; \dots; c_n \hat{\Delta}_n \mapsto^{\tau_n} Q_n\}$$

where  $x$  is a variable,  $c_i$  are constructors with fresh variables  $\hat{\Delta}_i$  for arguments,  $\tau_i$  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

$$\Gamma \vdash \{c_1 \Delta_1 [\Phi_1]; \dots; c_k \Delta_k [\Phi_k]\}$$

where:

- $\Gamma$  is the ‘outer’ telescope of datatype indices,
- $c_1, \dots, c_k$  are the names of the constructors,
- $\Delta_i$  is the ‘inner’ telescope of arguments of  $c_i$ , and
- $\Phi_i$  is a set of constraints  $\{w_{ij} / v_{ij} : A_{ij} \mid j = 1 \dots l\}$ .

To transform the definition of an inductive datatype  $D$  to a case tree, the initial elaboration problem has for  $\Gamma$  the index telescope of  $D$ ,  $c_1, \dots, c_k$  all constructors of  $D$ ,  $\Delta_i$  the argument telescope of  $c_i$ , and  $\Phi_i = \{x_j / v_{ij} : A_j \mid j = 1 \dots l\}$  where  $\Gamma = (x_1 : A_1) \dots (x_l : A_l)$  and  $v_{i1}, \dots, v_{il}$  are the indices targeted by  $c_i$ , i.e.  $c_i : \Delta_i \rightarrow D \ v_{i1} \dots v_{il}$ . For example, for  $\leq$  we start with the following elaboration problem:

$$(m \ n : \mathbb{N}) \vdash \left\{ \begin{array}{l} \leq 0 \ (n' : \mathbb{N}) \quad [m / 0 : \mathbb{N}, n / n' : \mathbb{N}] \\ \leq S \ (m' \ n' : \mathbb{N})(p : m' \leq n') \quad [m / S \ m' : \mathbb{N}, n / S \ n' : \mathbb{N}] \end{array} \right\}$$

From this initial problem, the elaboration proceeds by successive case splitting on variables in  $\Gamma$  and simplification of constraints in  $\Phi_i$  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  $\perp$ :

$$\Gamma \vdash \{\} \rightsquigarrow \perp$$

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

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

Solve Constraint If there is a constraint of the form  $y / x$  where  $x$  is bound in  $\Delta$  and  $y$  bound in  $\Gamma$ , we can instantiate the variable  $x$  to  $y$ , removing it from  $\Delta$  in the process:

$$\frac{\Gamma \vdash \{c \ \Delta_1(\Delta_2[y/x]) \ [\Phi]\} \rightsquigarrow Q \quad (x : A) \in \Gamma}{\Gamma \vdash \{c \ \Delta_1(x : A)\Delta_2[y / x : A, \Phi]\} \rightsquigarrow Q}$$

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

$$\frac{d : \Delta' \rightarrow D \ \bar{w}' \quad \Gamma \vdash \{c \ \Delta \ [\bar{v} / \bar{u} : \Delta', \Phi]; P\} \rightsquigarrow Q}{\Gamma \vdash \{c \ \Delta \ [d \ \bar{v} / \bar{u} : D \ \bar{w}, \Phi]; P\} \rightsquigarrow Q}$$

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

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

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:

$$\frac{\Gamma \vdash \{P\} \rightsquigarrow Q}{\Gamma \vdash \{c \Delta [d_2 \bar{v} / ? d_1 \bar{u} : D \bar{w}, \Phi]; P\} \rightsquigarrow Q}$$

Split Finally, if  $(x : D \bar{w}) \in \Gamma$  and each constraint set  $\Phi_i$  contains a constraint of the form  $x / ? d_j \bar{u}_j : D \bar{w}$  where  $d_j$  is a constructor of the datatype  $D$ , elaboration continues by performing a case split on  $x$ . For each constructor  $d_j : \Delta'_j \rightarrow D \bar{w}'_j$ , 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.

$$\frac{\begin{array}{l} (x : D \bar{w}) \in \Gamma \quad D : \Psi \rightarrow \text{Type} \\ d_j : \Delta'_j \rightarrow D \bar{w}'_j \quad \text{for } j = 1 \dots l \\ \text{unify}(\Gamma \Delta'_j \vdash \bar{w} = \bar{w}'_j : \Psi) \Rightarrow \text{yes}(\Gamma_j, \sigma_j, \tau_j) \quad \text{for } j = 1 \dots k \quad (k \leq l) \\ \text{unify}(\Gamma \Delta'_j \vdash \bar{w} = \bar{w}'_j : \Psi) \Rightarrow \text{no} \quad \text{for } j = k + 1 \dots l \\ \Gamma_j \vdash \{c_i \Delta_i \sigma_j [\Phi_i \sigma_j] \mid i = 1 \dots m\} \rightsquigarrow Q_j \quad \text{for } j = 1 \dots k \end{array}}{\Gamma \vdash \{c_i \Delta_i [\Phi_i] \mid i = 1 \dots m\} \rightsquigarrow \text{case}_x \{d_j \hat{\Delta}'_j \mapsto^{\tau_j} Q_j \mid j = 1 \dots k\}}$$

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  $\text{sProp}$ , 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  $w / ? v : A$  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  $m \leq n$  we can define terms  $\text{lz} : (n : \mathbb{N}) \rightarrow 0 \leq n$  and  $\text{ls} : (m n : \mathbb{N}) \rightarrow m \leq n \rightarrow S m \leq S n$  by  $\text{lz} = \lambda n. \text{tt}$  and  $\text{ls} = \lambda m n p. p$  respectively. Note that these terms are type-correct since  $0 \leq n = \text{sUnit}$  and  $S m \leq S n = m \leq n$ . We can also define the eliminator  $\leq\_rect$  by performing the same case splits as in the translation of the type

1030  $\leq$ :

$$\begin{aligned}
1031 & \leq\_rect : (P : (m\ n : \mathbb{N}) \rightarrow m \leq n \rightarrow \text{Type}) (m_{\leq 0} : (n : \mathbb{N}) \rightarrow P\ 0\ n\ (\text{lz}\ n)) \\
1032 & \quad (m_{\leq S} : (m\ n : \mathbb{N}) (H : m \leq n) \rightarrow P\ m\ n\ x \rightarrow P\ (S\ m)\ (S\ n)\ (\text{ls}\ m\ n\ x)) \\
1033 & \quad (m\ n : \mathbb{N}) (x : m \leq n) \rightarrow P\ m\ n\ x \\
1034 & \leq\_rect\ P\ m_{\leq 0}\ m_{\leq S}\ m\ n\ x \\
1035 & = \text{case}_m \left\{ \begin{array}{l} 0 \quad \mapsto \quad m_{\leq 0}\ n \\ S\ m' \quad \mapsto \quad \text{case}_n \left\{ \begin{array}{l} 0 \quad \mapsto \quad \text{sEmpty\_rect}\ x \\ S\ n' \quad \mapsto \quad m_{\leq S}\ m'\ n'\ x\ (\leq\_rect\ P\ m_{\leq 0}\ m_{\leq S}\ m'\ n'\ x) \end{array} \right\} \end{array} \right\}
\end{aligned}$$

1039 Since the eliminator is the defining property of a datatype, being able to construct the  
1040 eliminator shows the correctness of the translation. In particular, the eliminator can be used  
1041 to show that the generated type is equivalent to its inductive version.  
1042

1043 Constructing the constructors. Let  $c : \Delta \rightarrow D\ \bar{w}$  be one of the constructors of  $D$ . By  
1044 construction, the case tree of  $D$  will have a leaf of the form  $c\ \Delta'$  where the variables bound  
1045 by  $\Delta'$  form a subset of those bound in  $\Delta$ . We thus define the term  $c$  as  $\lambda x_1 \dots x_n. (x_{i_1}, \dots, x_{i_m})$   
1046 where  $\Delta = (x_1 : A_1) \dots (x_n : A_n)$  and  $\Delta' = (x_{i_1} : A'_{i_1}) \dots (x_{i_m} : A'_{i_m})$ .

1047 Beware, as it is not immediately obvious that this term is type-correct:  $A'_i$  is not necessarily  
1048 equal to  $A_i$ , since the variables in  $\Delta \setminus \Delta'$  have been substituted in the process. However, since  
1049 the Solve Constraint step only applies when both sides of the constraint are variables, this  
1050 substitution is just a renaming. We can apply the same renaming to  $\Delta'$ , thus ensuring that  
1051 the term  $c$  is indeed of type  $\Delta \rightarrow D\ \bar{w}$ .  
1052

1053 Constructing the eliminator. The eliminator  $D\_rect$  for the elaborated datatype  $D : \Gamma \rightarrow$   
1054  $\text{Type}$  with constructors  $c_i : \Delta_i \rightarrow D\ \bar{v}_i$  for  $i = 1 \dots k$  takes the following arguments:

- 1055 • The motive  $P : \Gamma \rightarrow D\ \hat{\Gamma} \rightarrow \text{Type}$
- 1056 • The methods

$$\begin{aligned}
1057 & \\
1058 & m_i \quad : \quad \Delta_i \rightarrow (H_1 : \Psi_{i1} \rightarrow P\ \bar{w}_{i1}\ (x_{ij_1}\ \hat{\Psi}_{i1})) \rightarrow \dots \rightarrow \\
1059 & \quad (H_{q_i} : \Psi_{iq_i} \rightarrow P\ \bar{w}_{iq_i}\ (x_{ij_{q_i}}\ \hat{\Psi}_{iq_i})) \rightarrow P\ \bar{v}_i\ (c_i\ \hat{\Delta}_i)
\end{aligned}$$

1060 where  $(x_{ij_p} : \Psi_{ip} \rightarrow D\ \bar{w}_{ip}) \in \Delta_i$  for  $p = 1 \dots q_i$  are the recursive arguments of the  
1061 constructor  $c_i$ .  
1062

1063 and produces a function of type  $\Gamma \rightarrow (x : D\ \hat{\Gamma}) \rightarrow P\ \hat{\Gamma}\ x$ .

1064 To construct this eliminator, we proceed by induction on the case tree defining  $D$ : for  
1065 each case split in the elaboration of  $D$ , we perform the same case split on the corresponding  
1066 index in  $\Gamma$ . At each leaf, we are either in an empty node or a constructor node.

- 1067 • In an empty node, we have  $x : \text{sEmpty}$ , hence we can conclude by  $\text{sEmpty\_rect}$ .
- 1068 • In a constructor node for constructor  $c_i$ ,  $x$  is a nested tuple consisting of the non-forced  
1069 arguments of  $c_i$ . Moreover, the remaining telescope of indices  $\Gamma'$  contains the forced  
1070 arguments of  $c_i$ . We thus apply the motive  $m_i$  to arguments for  $\Delta_i$  taken from  $x$  and  
1071  $\Gamma'$ . For the induction hypotheses  $H_p$  we call the eliminator recursively with arguments  
1072  $\bar{w}_{ip}\ (x_{ij_p}\ \hat{\Psi}_{ip})$  where  $x_{ij_p}$  is again taken from either  $x$  or  $\Gamma'$ . Note that this recursion is  
1073 well-founded if and only if the recursive definition of the datatype itself is so.  
1074

1075 This finishes the construction of the eliminator. It can easily be checked that the eliminator  
1076 we just constructed also has the appropriate computational behaviour:  $D\_rect\ P\ m_1 \dots m_k\ \bar{v}_i\ (c_i\ \hat{\Delta}_i)$   
1077 evaluates to  $m_i\ \hat{\Delta}_i\ (\lambda \hat{\Psi}_{i1}. D\_rect\ \bar{w}_{i1}\ (x_{ij_1}\ \hat{\Psi}_{i1})) \dots (\lambda \hat{\Psi}_{iq_i}. D\_rect\ \bar{w}_{iq_i}\ (x_{ij_{q_i}}\ \hat{\Psi}_{iq_i}))$ .  
1078

## 6 IMPLEMENTATION IN Coq AND Agda

### 6.1 Implementation in Coq

The universe hierarchy. Coq comes with an infinite hierarchy of predicative universes  $\text{Type}_i$  and an impredicative universe  $\text{Prop} : \text{Type}_1$ . Cumulativity makes each universe a subtype of the next, with  $\text{Prop}$  a subtype of  $\text{Type}_0$ . We add to this  $\text{sProp} : \text{Type}_1$  but do not include  $\text{sProp}$  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  $\text{sEmpty}$ :

**Definition**  $\text{sEmpty\_rect\_type} := \forall (P : \text{sEmpty} \rightarrow \text{Type}) x, P x$ .

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

The system works around this issue by allowing cumulativity of  $\text{sProp}$  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  $\text{sMLTT}$  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  $\text{sProp}$ , we reduce terms to weak head normal form then if they have the same shape we recurse for each pair of subterms. With  $\text{sProp}$ , 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  $\text{sEmpty}$ , squash, box and irrelevant pairs are subsumed by restrictions on the inductive types which may be defined.

In the system without  $\text{sProp}$ , 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  $\text{Prop}$  but its elimination is restricted to  $\text{Prop}$  unless it satisfies singleton elimination: 0 or 1 constructor with all arguments in  $\text{Prop}$ . Inductive types in  $\text{Prop}$  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  $\text{sProp}$  by considering  $\text{sProp}$  as smaller than other universes when checking the level of an inductive type. This allows defining box types:

**Inductive**  $\text{Box} (A : \text{sProp}) : \text{Prop} := \text{box} : A \rightarrow \text{Box } A$ .

**Definition**  $\text{unbox } A (x : \text{Box } A) : A := \text{match } x \text{ with } \text{box } y \Rightarrow y \text{ end}$ .

The level  $\text{sProp}$  of the argument  $A$  is smaller than the level  $\text{Prop}$  of the inductive.

Like with  $\text{Prop}$ , any inductive type may be defined in  $\text{sProp}$  but its elimination is restricted to  $\text{sProp}$  when it is not the empty type. This amounts to implicitly using the squash (which

1128 can be already defined using the impredicative encoding). For primitive records in sProp we  
 1129 allow fields in sProp. This provides a generalization to the unit type and dependent sums.  
 1130 To get other inductive types in sProp that can be eliminated into any type, the user has to  
 1131 use the automatic encoding described in Section 5, implemented on top of the Equations  
 1132 plugin (see Section 6.4).

1133

## 1134 6.2 Example 1: Prime Numbers

1135 To illustrate a simple use of sProp in Coq, together with the automatic translation of inductive  
 1136 types into fixpoints described in Section 5, let us consider the definition of primality.

1137 First, we can define the  $n \mid m$  predicate, which states that the natural number  $n$  is a  
 1138 divisor of  $m$ .

1139

1140 **Inductive** Divide :  $\mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{SProp} :=$

1141 | divide<sub>0</sub> :  $\forall n, \text{Divide } n \ 0$

1142 | divides<sub>S</sub> :  $\forall n \ m \ (e: \text{S } n \leq \text{S } m), \text{Divide } (\text{S } n) \ (m - n) \rightarrow \text{Divide } (\text{S } n) \ (\text{S } m).$

1143 **Infix** "|" := Divide.

1144

1145 This definition satisfies the conditions described in Section 5, so the corresponding fixpoint  
 1146 definition can be automatically generated, together with its eliminator into Type.

1147 Note that we have definitional proof irrelevance for  $n \mid m$ , but we can still extract the  
 1148 natural number that witnesses the fact that  $n$  is a divisor of  $m$  out of the proof that  $n \mid m$ .

1149

1150 **Definition** divide\_to\_ℕ :  $\forall n \ m, n \mid m \rightarrow \mathbb{N}.$

1151

1152 together with the correctness of this definition.

1153

1154 **Lemma** divide\_to\_ℕ\_correct n m (e:n | m): divide\_to\_ℕ n m e \* n = m.

1155

1156 Then, it is easy to define primality in sProp in a way that satisfies the conditions described  
 1157 in Section 5.

1158

1159 **Inductive** is\_gcd (a b g:ℕ) : SProp :=

1160 is\_gcd\_intro :  $g \mid a \rightarrow g \mid b \rightarrow (\forall x, x \mid a \rightarrow x \mid b \rightarrow x \mid g) \rightarrow \text{is\_gcd } a \ b \ g.$

1161

1162 **Inductive** prime (p:ℕ) : SProp :=

1163 prime\_intro :  $1 < p \rightarrow (\forall n, 1 \leq n \rightarrow n < p \rightarrow \text{is\_gcd } n \ p \ 1) \rightarrow \text{prime } p.$

1164

1165 This definition gives us definitional proof irrelevance for prime, without paying the price  
 1166 of the definition of a decision procedure into booleans (for instance using the sieve of Er-  
 1167 atosthenes) and a proof that it corresponds to primality. Here, we have a direct definition  
 1168 instead, and the decision procedure is only a useful addition to prove primality of a partic-  
 1169 ular natural number and may be implemented as a tactic.

1170

## 1171 6.3 Example 2: The Setoid Model

1172 The previous example show the use of sProp to define definitionally proof irrelevant pred-  
 1173 icates while still being able to extract relevant values out of it. We now turn to a more  
 1174 critical use of sProp to avoid higher coherence issues in syntactical models of type theory.

1175 As an example, using sProp, we can formally define in Coq the setoid translation presented  
 1176 on paper twenty years ago by Thorsten Altenkirch [Altenkirch 1999]. We define a setoid as  
 1177 a type carrier together with a notion of equality which is reflexive, symmetric, transitive,  
 1178 and definitionally proof irrelevant.

1179

```

1177 Record Setoid :=
1178   { carrier : Type;
1179     eq : carrier → carrier → SProp;
1180     refl : ∀ x, eq x x;
1181     sym : ∀ {x y}, eq x y → eq y x;
1182     trans : ∀ {x y z}, eq x y → eq y z → eq x z
1183   }.

```

1185 This way, we can define a category with families (CwF), as introduced by Peter Dybjer [Dy-  
1186 bjer 1996], where contexts are `Setoid`, types are setoid families (over a context) and terms  
1187 are sections of setoid families. This CwF features dependent products  $\Pi$ , a universe, and  
1188 identity types `Eq`. We can then prove that this CwF satisfies functional extensionality, thus  
1189 providing a formal proof that functional extensionality is admissible in sMLTT.

#### 1190 6.4 Extension of the Equations Plugin

1191 The Equations plugin of Coq [Mangin and Sozeau 2018] provides an implementation of  
1192 dependent-pattern matching compilation. We reuse the tools of Equations to automatically  
1193 derive the translation of inductive definitions that fit in `sProp` to fixpoint definitions. Con-  
1194 cretely, we have extended Equations with a new `Derive Inversion` command that tries to  
1195 produce a case tree from an inductive definition and succeeds if and only if the resulting  
1196 recursive definition can be typed in `sProp`.

1197 Using this tool, it is possible to automatically derive the definition of the recursive variants  
1198 of  $\leq$  and `Divide`, along with their constructors. Coming back to the example of  $\leq_{\text{fix}}$ , our  
1199 framework provides automatically the two definitions corresponding to the two constructor  
1200 of  $\leq$ :

1201 **Definition**  $\leq_{\text{fix}0} : \forall n, 0 \leq_{\text{fix}} n := \text{fun } n \Rightarrow \text{I}$ .

1202 **Definition**  $\leq_{\text{fix}S} : \forall m n, m \leq_{\text{fix}} n \rightarrow \text{S } m \leq_{\text{fix}} \text{S } n := \text{fun } n m e \Rightarrow e$ .

1203 We are currently also working on implementing the automatic generation of the eliminator  
1204 as described in Section 5.2.

#### 1207 6.5 Implementation in Agda

1208 Aside from adding `sProp` to Coq, we also implemented a variant of the same concept for  
1209 Agda. Since previously there was no `Prop` universe, the `sProp` universe is simply called `Prop`  
1210 in Agda. This new universe has been integrated in the current development version of Agda  
1211 on <https://github.com/agda/agda> and is planned to be released as part of Agda 2.6.0 in  
1212 November 2018.

1213 User perspective. On the syntactic level, the main change to Agda is the addition of the  
1214 new sort `Prop` next to the old sort `Set`. Unlike `sProp` in Coq, `Prop` in Agda is predicative.  
1215 For example,  $(A : \text{Prop}) \rightarrow A$  is not itself an element of `Prop`. Instead, there is a hierarchy  
1216 of universes `Prop0` ( $= \text{Prop}$ ), `Prop1`, `Prop2`, ...analogous to the hierarchy of `Seti`. Like `Set`,  
1217 `Prop` also supports universe polymorphism, so for each  $\ell : \text{Level}$  we have the sort `Prop  $\ell$` .  
1218 As an example, we can define the universe polymorphic squash type and its eliminator:

```

1219 data Squash {ℓ}(A : Set ℓ) : Prop ℓ where
1220   sq : A → Squash A
1221 unsquash : ∀{ℓ ℓ'}{A : Set ℓ}(P : Prop ℓ')(A → P) → Squash A → P
1222 unsquash P f (sq x) = f x

```

1225

1226 Note that this is more powerful than the predicative encoding of the squash type described  
 1227 in Section 3.4 since we can eliminate it into any `Propi`.

1228 Like in the Coq implementation, there is no implicit conversion from `Prop` to `Set`. Instead,  
 1229 `Prop` can be embedded through the definition of a record type `Box (A : Prop) : Set` with  
 1230 one field `unbox : A`. More generally, when defining a record type in `Seti`, the fields can be  
 1231 in both `Propj` or `Setj` for any  $j \leq i$ . On the other hand, for record types in `Prop` all fields  
 1232 must be in `Prop` themselves.

1233  
 1234 Implementation details. Since Agda uses a type-directed conversion check internally and  
 1235 types are annotated with their sorts, adding the rule that any two elements of a type in  
 1236 `Prop` are equal was straightforward. In particular, the relevance marks used in the Coq  
 1237 implementation are not required here. In contrast, making `Prop` impredicative is currently  
 1238 problematic since Agda's termination checker assumes predicativity. This was our main  
 1239 reason to implement the predicative hierarchy `Propi` instead.

1240  
 1241 Interaction with irrelevant arguments. As mentioned in the introduction, Agda has an-  
 1242 other another facility for definitional proof irrelevance in the form of irrelevant function  
 1243 types [Abel and Scherer 2012] and irrelevant fields. `Prop` can be used in situations where  
 1244 Agda's pre-existing irrelevance cannot, for example:

- 1245 • We can use `Prop` to give a function an irrelevant return type.
- 1246 • We can define new datatypes and record types in `Prop`, such as `Squash`.
- 1247 • We can construct new types in `Prop` by pattern matching, such as  $\leq$ .
- 1248 • We can quantify over all types in `Prop`.

1249 In our implementation, it is allowed to use irrelevant arguments when constructing an  
 1250 element of a type in `Prop`. This allows us to convert irrelevant arguments into elements of  
 1251 the squash type:  
 1252

```
1253     irrToSquash : {A : Set} → .A → Squash A
1254     irrToSquash x = sq x
```

1256 In fact, irrelevant functions and irrelevant fields can be encoded in terms of `Prop` by using  
 1257 the squash type: each irrelevant argument or field of type `.A` is turned into a squashed  
 1258 argument `Squash A`. This encoding uses the fact that `Squash` is an applicative functor with  
 1259 identity `sq : A → Squash A` and sequential application

```
1260     _<*>_ : {A B : Set} → Squash (A → B) → Squash A → Squash B
1261     sq f <*> sq x = sq (f x)
```

1264 For example, if  $g : .B \rightarrow C$  then the expression  $\lambda f x. g (f x)$  of type  $(A \rightarrow B) \rightarrow A \rightarrow C$  is  
 1265 encoded by  $\lambda f x. g (f \text{<*> } sq x)$  of type `Squash (A → B) → A → C`.

1266 In addition to irrelevant functions and fields, Agda also provides experimental support  
 1267 for irrelevant definitions and irrelevant projections, which were already mentioned in the  
 1268 introduction. In contrast to irrelevant functions, not everything that's possible with these  
 1269 features can be encoded in terms of `Squash`, even after fixing the bug mentioned in the  
 1270 introduction. In particular, they can be used to construct a function `.choice : .A → A` for  
 1271 any type `A`, but the corresponding statement `Squash (Squash A → A)` is not provable. If  
 1272 this were postulated as an axiom, we conjecture that these experimental ways of using  
 1273 irrelevance could be encoded in terms of `Squash` as well.

1274

## 1275 7 CONCLUSION AND FUTURE WORK

1276 We have given a general way to extend a type theory with a predicative hierarchy of uni-  
1277 verses (or an impredicative universe) satisfying definitional proof irrelevance, while keeping  
1278 decidability of type checking. We have shown various metatheoretical properties of our ex-  
1279 tension using syntactic translations into extensional type theories ETT and HTS. We have  
1280 then described a new way to decide whether a proposition can be eliminated over a type  
1281 using techniques coming from recent development on dependent pattern matching without  
1282 UIP. We have implemented our approach both in Coq and in Agda, and illustrated its useful-  
1283 ness on several examples, in particular to formalize the setoid model of type theory, which  
1284 can not be done without definitional proof irrelevance.

1285 Regarding future work, the main extension of our framework that we would like to address  
1286 in a near future is the definition of a strict equality that remains compatible with univalence.  
1287 The idea is to detect syntactically which types are mere sets—using again ideas coming from  
1288 dependent pattern matching—in order to allow elimination of strict equality on those types.  
1289 This should require the addition of a hierarchy of universes of mere (strict) sets, and maybe  
1290 other hierarchies for higher homotopy levels as well, but it is not clear how to do it in a  
1291 good way from the moment.

1292

1293

1294

1295

1296

1297

1298

1299

1300

1301

1302

1303

1304

1305

1306

1307

1308

1309

1310

1311

1312

1313

1314

1315

1316

1317

1318

1319

1320

1321

1322

1323

## 1324 REFERENCES

- 1325 Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of Conversion for Type Theory  
 1326 in Type Theory. *Proc. ACM Program. Lang.* 2, POPL, Article 23 (Jan. 2018), 29 pages. DOI:<http://dx.doi.org/10.1145/3158111>
- 1327 Andreas Abel and Gabriel Scherer. 2012. On Irrelevance and Algorithmic Equality in Predicative Type  
 1328 Theory. *Logical Methods in Computer Science* 8, 1 (03 2012). DOI:[http://dx.doi.org/10.2168/lmcs-8\(1:](http://dx.doi.org/10.2168/lmcs-8(1:29)2012)  
 1329 [29\)2012](http://dx.doi.org/10.2168/lmcs-8(1:29)2012)
- 1330 T. Altenkirch. 1999. Extensional equality in intensional type theory. In *Proceedings. 14th Symposium on*  
 1331 *Logic in Computer Science (Cat. No. PR00158)*. 412–420. DOI:[http://dx.doi.org/10.1109/LICS.1999.](http://dx.doi.org/10.1109/LICS.1999.782636)  
 1332 [782636](http://dx.doi.org/10.1109/LICS.1999.782636)
- 1333 Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. 2016. Extending Homotopy Type Theory with  
 1334 Strict Equality. In *CSL*.
- 1335 Steven Awodey and Andrej Bauer. 2004. Propositions As [Types]. *J. Log. and Comput.* 14, 4 (Aug. 2004),  
 1336 447–471. DOI:<http://dx.doi.org/10.1093/logcom/14.4.447>
- 1337 Simon Boulrier, Pierre-Marie Pédrot, and Nicolas Tabareau. 2017. The Next 700 Syntactical Models of Type  
 1338 Theory. In *Proceedings of Certified Programs and Proofs*. ACM, 182–194.
- 1339 Edwin Brady, Conor McBride, and James McKinna. 2004. Inductive Families Need Not Store Their Indices.  
 1340 In *Types for Proofs and Programs*, Stefano Berardi, Mario Coppo, and Ferruccio Damiani (Eds.). Springer  
 1341 Berlin Heidelberg, Berlin, Heidelberg, 115–129.
- 1342 Paolo Capriotti. 2017. Models of type theory with strict equality. Ph.D. Dissertation. University of Not-  
 1343 tingham.
- 1344 Jesper Cockx and Andreas Abel. 2018. Elaborating Dependent (Co)pattern matching. In *Proceedings of*  
 1345 *the 23th ACM SIGPLAN Conference on Functional Programming (ICFP 2018)*. ACM Press, St. Louis,  
 1346 Missouri, United States.
- 1347 Jesper Cockx and Dominique Devriese. 2018. Proof-relevant unification: Dependent pattern matching with  
 1348 only the axioms of your type theory. *Journal of Functional Programming* 28 (2018), e12. DOI:<http://dx.doi.org/10.1017/S095679681800014X>
- 1349 Jesper Cockx, Dominique Devriese, and Frank Piessens. 2014. Pattern matching without K. In *ACM*  
 1350 *SIGPLAN Notices*, Vol. 49. ACM, 257–268.
- 1351 Thierry Coquand. 2016. Universe of Bishop sets. (2016). [www.cse.chalmers.se/~coquand/bishop.pdf](http://www.cse.chalmers.se/~coquand/bishop.pdf).
- 1352 Peter Dybjer. 1996. Internal type theory. In *Types for Proofs and Programs*, Stefano Berardi and Mario  
 1353 Coppo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 120–134.
- 1354 Martin Hofmann. 1995. Conservativity of equality reflection over intensional type theory. In *International*  
 1355 *Workshop on Types for Proofs and Programs*. Springer, 153–164.
- 1356 P. Letouzey. 2004. Programmation fonctionnelle certifiée – L’extraction de programmes dans l’assistant  
 1357 Coq. Ph.D. Dissertation. Université Paris-Sud.
- 1358 Cyprien Mangin and Matthieu Sozeau. 2018. Equations Reloaded. (2018). [http://mattam82.github.io/](http://mattam82.github.io/Coq-Equations/)  
 1359 [Coq-Equations/](http://mattam82.github.io/Coq-Equations/)
- 1360 Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium ’73*,  
 1361 H.E. Rose and J.C. Shepherdson (Eds.). *Studies in Logic and the Foundations of Mathematics*, Vol. 80.  
 1362 Elsevier, 73 – 118. DOI:[http://dx.doi.org/https://doi.org/10.1016/S0049-237X\(08\)71945-1](http://dx.doi.org/https://doi.org/10.1016/S0049-237X(08)71945-1)
- 1363 Frank Pfenning. 2001. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. In  
 1364 *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS ’01)*. IEEE  
 1365 Computer Society, Washington, DC, USA, 221–. <http://dl.acm.org/citation.cfm?id=871816.871845>
- 1366 The Univalent Foundations Program. 2013. Homotopy Type Theory: Univalent Foundations of Mathematics.  
 1367 Institute for Advanced Study.
- 1368 Vladimir Voevodsky. 2011. Resising Rules - their use and semantic justification. [www.math.ias.edu/](http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/2011_Bergen.pdf)  
 1369 [~vladimir/Site3/Univalent\\_Foundations\\_files/2011\\_Bergen.pdf](http://www.math.ias.edu/~vladimir/Site3/Univalent_Foundations_files/2011_Bergen.pdf). (2011).
- 1370 Vladimir Voevodsky. 2013. A simple type system with two identity types. (2013). [https://ncatlab.org/](https://ncatlab.org/homotopytypetheory/files/HTS.pdf)  
 1371 [homotopytypetheory/files/HTS.pdf](https://ncatlab.org/homotopytypetheory/files/HTS.pdf)
- 1372 Benjamin Werner. 2008. On the Strength of Proof-Irrelevant Type Theories. 4 (09 2008), 1–20.

## 1368 A LEAN SUBJECT REDUCTION FAILURE

1369 axiom A : Type  
 1370 axiom r : A → A → Prop

1372

```

1373
1374 axiom C : A → Type
1375 axiom F : Pi x, (Pi y, r y x → C y) → C x
1376
1377
1378 lemma fix_F_eq1 (x : A) (acx : acc r x) :
1379   well_founded.fix_F F x acx =
1380   well_founded.fix_F F x (acc.intro x ( y, acc.inv acx)) :=
1381   eq.refl _
1382
1383 lemma fix_F_eq2 (x : A) (acx : acc r x) :
1384   well_founded.fix_F F x (acc.intro x ( y, acc.inv acx)) =
1385   F x ( ( y : A) (p : r y x), well_founded.fix_F F y (acc.inv acx p))
1386   := eq.refl _
1387
1388 lemma fix_F_eq3 (x:A) (acx : acc r x) :
1389   well_founded.fix_F F x acx =
1390   F x ( ( y : A) (p : r y x), well_founded.fix_F F y (acc.inv acx p))
1391   := eq.trans (fix_F_eq1 x acx) (fix_F_eq2 x acx)
1392
1393 lemma fix_F_eq4 (x:A) (acx : acc r x) :
1394   well_founded.fix_F F x acx =
1395   F x ( ( y : A) (p : r y x), well_founded.fix_F F y (acc.inv acx p))
1396   := eq.refl _ -- fails
1397
1398 B INCONSISTENCY OF IRRELEVANT FIELDS
1399
1400 data  : Set where ¬
1401
1402 _ : Set → Set ¬
1403   A = (A → )
1404 truefalse : ¬ (true false)
1405 truefalse ()
1406
1407 infixr 30 ____
1408 ____ : {A : Set} {x y z : A} → x y → y z → x z
1409 refl q = q
1410
1411 infix 42 !_
1412 !_ : {A : Set} {x y : A} → x y → y x
1413 !_ refl = refl
1414
1415
1416
1417
1418 record Squash (A : Set) : Set where
1419   constructor sq
1420   field .unsq : A
1421

```

```

1422
1423 open Squash
1424
1425 squash_pi : {A : Set} → (x y : Squash A) → x = y
1426 squash_pi x y = refl
1427
1428 squash_invol : {B : Set} → (Squash (Squash B)) → Squash B
1429 squash_invol (sq (sq x)) = sq x
1430
1431 squash_elim : {A B : Set} → (A → Squash B) → Squash A → Squash B
1432 squash_elim f (sq x) = squash_invol (sq (f x))
1433
1434 map-sq : {A B : Set} → (A → B) → Squash A → Squash B
1435 map-sq f (sq x) = sq (f x)
1436
1437 factivity : Squash →
1438 factivity (sq ())
1439
1440
1441
1442 bizarre : Squash Σ( (ℕ → Squash ℕ) ( i → Σ (Squash ℕ → ℕ) ( u → (a : ℕ) → u (i a)
1443   a)))
1444 bizarre = sq (sq , unsq , ( a → refl {x = a}))
1445
1446 bad : Squash (true = false)
1447 bad = map-sq ( { (i , u , s) → !(s = true) = s = false }) bizarre
1448
1449 boom :
1450 boom = factivity (map-sq truefalse bad)
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470

```