A type theory with definitional proof-irrelevance

Gaëtan Gilbert

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THÈSE DE DOCTORAT DE

l'École Nationale Supérieure Mines-Télécom Atlantique
Bretagne Pays de la Loire - IMT Atlantique
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Gaëtan GILBERT

A type theory with definitional proof-irrelevance

Une théorie des types avec insignifiance des preuves définitionnelle

Thèse présentée et soutenue à IMT Atlantique, Nantes, le 20 décembre 2019
Unité de recherche : LS2N (UMR 6004)
Thèse N° : 2019IMTA0169

Rapporteurs avant soutenance :

Andreas Abel	Full Professor, Gothenburg University / Chalmers
Bas Spitters	Assistant Professor, Aarhus University

Composition du Jury :

Président :	Yves Bertot	Directeur de Recherche, Inria Sophia-Antipolis
Examinateurs :	Assia Mahboubi	Chargée de Recherche, Université de Nantes
	Andreas Abel	Full Professor, Gothenburg University / Chalmers
	Bas Spitters	Assistant Professor, Aarhus University
	Gert Smolka	Full Professor, Saarland University
Dir. de thèse :	Nicolas Tabareau	Directeur de Recherche, IMT Atlantique
Co-dir. de thèse :	Matthieu Sozeau	Chargé de Recherche, Université Paris 7

CONTENTS

CHAPTER

1

RÉSUMÉ ÉTENDU

Le texte de cette introduction est en grande partie dérivé de Gilbert et al. [GCST19].

Warning: To English readers:

This chapter is a translation of *Chapter 2* (page 12) from English to French.

1.1 Le Besoin d'insignifiance des preuves

L'insignifiance des preuves, le principe selon lequel deux preuves quelconques d'une même proposition sont égales, est au cœur du mécanisme d'extraction de Coq [Let04]. Dans le calcul des constructions inductives (`CIC`), la théorie sous-jacente de l'assistant de preuve Coq, il existe un type (imprédicatif) Prop utilisé pour caractériser les types pouvant être vus comme des propositions—par opposition aux types dont les habitants ont un sens calculatoire, qui vivent dans l'univers prédicatif Type. Cette information statique est utilisée pour extraire une formalisation écrite en Coq vers un programme purement fonctionnel, en effaçant en toute sécurité toutes les parties impliquant des termes qui habitent des propositions, car un programme ne peut pas utiliser ceux-ci de manière calculatoire.

Afin de garantir concrètement qu'aucun calcul ne peut fuir de preuves de propositions, Coq utilise une restriction de l'élimination dépendante des types inductifs dans Prop dans les prédicats dans Type. Cette restriction, appelée la *condition d'élimination du singleton*, vérifie qu'une proposition inductive peut être éliminée dans un type seulement quand:

- la proposition inductive a au plus un constructeur,

- tous les arguments du constructeur sont eux-mêmes non-calculatoire, c'est-à-dire, sont dans Prop.

Cependant, dans la version actuelle de Coq, l'élimination du singleton permet d'être compatible avec l'insignifiance des preuves, mais n'est pas assez pour l'avoir. Cela signifie que bien que deux preuves de la même proposition ne peuvent pas être distinguées de manière pertinente dans le système, on ne peut pas utiliser le fait qu'elles soient égales dans la logique (sans axiome).

Considérons par exemple un mathématicien ou un informaticien qui définit les entiers bornés de la manière suivante:

```
Definition boundednat (k : nat) : Type := { n : nat & n <= k }.
```

Ici `boundednat k` est la somme dépendante d'un entier `n` avec une preuve (dans Prop) que `n` est plus petit que `k`, en utilisant la définition inductive

```
Inductive le : nat -> nat -> Prop :=
| le0 : forall n, 0 <= n
| leS : forall m n, m <= n -> S m <= S n.
```

Ensuite, notre utilisateur définit une coercition implicite de `boundednat` à `nat` afin qu'il puisse travailler avec des entiers bornés presque comme s'ils étaient des entiers, mis à part des preuves supplémentaires de la borne.

```
Coercion boundednat_to_nat : boundednat >-> nat.
```

Par exemple, il peut définir l'addition d'entiers bornés simplement en en s'appuyant sur l'addition d'entiers, modulo une preuve que le résultat est toujours borné:

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

Malheureusement, quand il s'agit de raisonner sur des entiers bornés, le la situation devient plus difficile. Par exemple, le fait que l'addition de nombres naturels liés soit associative

```
Definition bounded_add_associativity k (n m p: boundednat k)
      e_1 e_2 e'_1 e'_2 :
 add (add n m e_1) p e_2 = add n (add m p e'_1) e'_2.
```

ne découle pas directement de l'associativité de l'addition sur les entiers, car elle nécessite en outre une preuve que `e_2` est égal à `e'_2` (modulo la preuve d'associativité de l'addition de entiers).

Cela devrait être vrai car ce sont deux preuves d'une même proposition, mais ça ne vient pas automatiquement. Au lieu de cela, l'utilisateur doit prouver que la preuve n'est pas pertinente pour `<=` *manuellement*. Cela peut être prouvé (en utilisant le filtrage à la Agda définit par le plugin Equations[1]) par induction sur la preuve de `m <= n`:

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

```
Equations le_hprop {m n}  (e e' : m <= n) : e = e' :=
  le_hprop (le0 _) (le0 _) := eq_refl;
  le_hprop (leS _ _ e) (leS n m e') := ap (leS n m) (le_hprop e e').
```

Notons ici l'utilisation de la fonctorialité de l'égalité `ap: forall f, x = y -> f x = f y` ce qui nécessite un raisonnement explicite sur l'égalité. Même si prouver l'associativité de l'addition était plus compliqué qu'attendu, notre utilisateur est encore assez chanceux de raisonner sur un type inductif qui est en fait une proposition homotopique, au sens de la théorie homotopique des types (HoTT) [UFP13]. En effet, `<=` satisfait la version propositionnelle (par opposition à définitionnelle, qui est vrai par calcul) de l'insignifiance des preuves, comme le montre le lemme `le_hprop`. En revanche, pour un type inductif arbitraire dans Prop, il n'y a aucune raison pour que ce soit une proposition homotopique, et donc l'insignifiance des preuves, même sous sa forme propositionnelle, ne peut être prouvée et doit être déclaré *comme un axiome*.

Dans un contexte où l'insignifiance des preuves pour Prop est native, il devient possible de définir une opération sur les types qui transforme tout type en un type dont le preuve sont insignifiante, et rend donc explicite dans le système le fait qu'une valeur dans `Squash T` ne sera jamais utilisée pour calculer.

```
Definition Squash (T : Type) : Prop := forall P : Prop, (T -> P) -> P.
```

Cette opération vérifie le schéma d'élimination suivant (restreint aux propositions)

```
forall (T : Type) (P : Prop), (T -> P) -> Squash T -> P.
```

ce qui veut dire qu'il correspond à la troncation propositionnelle [UFP13] de HoTT, initialement introduite comme le *bracket type* par Awodey et Bauer [AB04].

L'importance de l'insignifiance (définitionnelle) des preuves pour simplifier le raisonnement a été remarqué depuis longtemps, et divers travaux ont essayé de promouvoir sa mise en œuvre dans des assistants de preuve basés sur la théorie des types [Pfe01][Wer08]. Ceci n'a cependant jamais été réalisé, principalement à cause du malentendu fondamental que l'*élimination des singletons était la bonne contrainte justifiant les propositions qui peuvent être éliminées dans un type.*

En effet, on peut penser à l'élimination des singletons comme une approximation *syntaxique* pour savoir quels types sont naturellement de propositions homotopiques, et donc peuvent être éliminés dans un type arbitraire sans faire fuir des morceaux de calcul ou sans impliquer de nouveaux axiomes. Cependant, l'élimination des singletons ne fonctionne pas pour les types de données indexés, car il s'applique par exemple au type d'égalité de Coq

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

Si l'insignifiance des preuves est valable pour l'égalité, alors chaque égalité a au plus une preuve, un principe qui est connu sous le nom d'unicité des preuves d'identité (UIP). Par conséquent, en supposant l'insignifiance des preuves avec l'élimination des singletons, on obtient un nouvel axiome dans la théorie, qui est par exemple incompatible avec l'axiome d'univalence de HoTT. Cela peut ne pas sembler trop problématique pour certains, mais

une autre conséquence de l'élimination des singletons en présence d'insignifiance définitionnelle des preuves est qu'il brise la décidabilité de la conversion. Par exemple, le prédicat d'accessibilité:

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
   Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

satisfait au critère d'élimination des singletons, mais implémenter l'insignifiance définitionnelle des preuves pour ce prédicat conduit à une conversion indécidable et donc un vérificateur de type indécidable.

Une approche alternative consiste à faire comme dans Lean, où il y a l'insignifiance définitionnelle des preuves avec l'élimination des singletons, mais ils ne mettent en œuvre qu'une version partielle de l'insignifiance des preuves pour les types inductifs récursifs satisfaisant l'élimination des singletons, qui est restreint aux termes fermés.[2]

Mais cette implémentation partielle de l'algorithme de conversion ne satisfait pas la réduction du sujet, ce qui est une propriété plus que souhaitable pour un assistant de preuve.

Enfin, l'élimination des singletons ne parvient pas à capturer les types inductifs avec plusieurs constructeurs tels que `<=` qui sont des propositions homotopiques parfaitement valables et pourraient être éliminés dans les types.

En y regardant de plus près, Coq et son univers imprédicatif Prop ne sont peut-être pas le seul moyen d'implémenter l'insignifiance des preuves dans un assistant de preuve. Agda, qui a seulement une hiérarchie prédicative d'univers et pas de Prop, utilise plutôt une notion d'arguments non pertinents [AS12]. L'idée est de marquer dans le type des fonctions quels arguments peuvent être considérés comme non pertinents. Par exemple, notre utilisateur peut encoder les entiers bornés dans ce cadre, en spécifiant que le deuxième argument de la paire dépendante est sans importance (comme indiqué par `.` dans la définition de `. (n <= k)`):

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

Le fait que l'égalité des entiers naturels sous-jacentes implique l'égalité des entiers naturels bornés est gratuit, grâce à l'insignifiance du deuxième composant:

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

Cependant, dans cette approche, l'insignifiance des preuves n'est pas une propriété du type considéré, mais plutôt une façon d'utiliser l'argument donné une fonction. Nous espérons que SProp donnera plus de flexibilité aux développeurs de preuves formelles.

[2] Une description de ce problème est disponible à https://github.com/leanprover/lean/issues/654

1.2 Leçons de la théorie homotopique des types

Avant de plonger dans la définition précise d'une théorie des types avec insignifiance des preuves, explorons ce qui la rend difficile à introduire tout en maintenant la vérification du typage décidable et en évitant d'induire des axiomes supplémentaires tels que UIP ou l'extensionalité des fonctions.

1.2.1 SProp comme approximation syntaxique des propositions homotopiques

La première leçon de HoTT est que chaque type dans SProp doit être une proposition homotopique, c'est-à-dire avoir le niveau homotopie -1. Formellement, l'univers des propositions homotopiques `hProp` est défini comme

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

et donc il correspond exactement à l'univers des types satisfaisant l'insignifiance des preuves *de manière propositionnelle*. Comme nous l'avons mentionné plus haut, l'opérateur `Squash` qui transforme n'importe quel type en un habitant de SProp, correspond à la troncation propositionnelle.

L'existence de SProp ne devient intéressante que si l'on peut éliminer certaines définitions inductives dans SProp à des types arbitraires. Sinon, SProp constitue une couche logique isolée correspondant à la logique propositionnelle, sans aucune interaction avec le reste de la la théorie des types.

La question est donc:

Quels types inductifs de SProp *peuvent être éliminés sur des types arbitraires?*

Bien entendu, pour préserver la cohérence, il convient de limiter cette option à des types inductifs dont on peut montrer qu'ils sont des propositions homotopiques, mais cela n'est pas suffisant pour préserver la décidabilité de la vérification du typage et l'indépendance enverts UIP.

1.2.2 Flirter avec l'extensionalité

Regardons d'abord une classe apparemment simple de propositions homotopiques, les types contractibles. Ils constituent le niveau le plus bas de la hiérarchie des types, pour lesquels non seulement il y a au plus un habitant à égalité près, mais il y en a exactement un. Bien sûr, le type unité dans SProp qui correspond à la proposition "vraie"

```
Inductive sUnit : SProp := tt : sUnit.
```

est contractible.

Nous dirons que le type d'unité peut être éliminé à n'importe quel type, et donc nous obtenons un type d'unité avec une règle η définitionnelle telle que `u` est convertible à `tt` pour tout `u: sUnit`. Mais en général, il ne faut pas s'attendre à ce que tout type

contractible dans SProp puisse être éliminé à un type arbitraire, comme nous allons le voir ci-dessous.

Types singletons

Un exemple typique d'un type contractible non trivial est le type singleton. Pour tout type `A` et `a: A`, il est défini comme le sous-ensemble de points dans `A` égaux à `a`:

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

Si nous incluons les types singleton dans SProp, et permettons donc son élimination à des types arbitraires, nous sommes conduits à une théorie de type extensionnelle, dans laquelle l'égalité propositionnelle implique l'égalité définitionnelle. Cela revient donc à ajouter UIP et l'indécidabilité de la vérification du typage, à la théorie.[6] En effet, supposons que les types singletons puissent être éliminés de manière arbitraire dans les types, alors il existe une projection `pi1: Sing A a -> A` qui récupère le point du singleton. Mais alors, pour toute preuve d'égalité `e: a = b` entre `a` et `b` dans `A`, en utilisant la congruence de l'égalité définitionnelle, il y a la chaîne suivante d'implications

```
    (a ; refl) ≡ (b ; e) : Sing A a
=>  pi1 (a ; refl) ≡ pi1 (b ; e) : A
=>  a ≡ b : A
```

et par conséquent, `e: a = b` implique `a ≡ b`. À partir de cette analyse, il est clair que SProp ne peut pas inclure tous les types contractibles.

1.2.3 Flirter avec l'indécidabilité

Voyons maintenant d'autres types inductifs qui ne sont que des propositions sans être contractible. Le premier exemple canonique est le type vide

```
Inductive sEmpty : SProp := .
```

qui n'a pas d'habitant, avec un principe d'élimination qui déclare que tout peut être déduit du type vide

```
sEmpty_rect : forall T : Type, sEmpty -> T.
```

Nous verrons que ce type peut être éliminé à n'importe quel type, et c'est réellement le moyen principal de faire communiquer SProp avec Type. En particulier, il permet la construction de programmes par filtrage en utilisant une contraction dans SProp pour les branches absurdes.

L'autre type de définition inductive dans SProp qui peut être éliminée dans n'importe quel type est la somme dépendante d'un type `A` dans SProp et une famille dépendante sur `A` dans SProp, le cas zéro étant le type unité `sUnit`.

Voyons maintenant des définitions inductives plus complexes.

[6] Cette remarque est initialement due à Peter LeFanu Lumsdaine.

Prédicat d'accessibilité

Comme mentionné dans l'introduction, le prédicat d'accessibilité est une proposition homotopique mais elle ne peut être éliminée dans tous les types tout en conservant la décidabilité de la conversion, et donc de la vérification du typage. Intuitivement, c'est parce que le prédicat d'accessibilité permet de définir des points fixes avec une condition de garde *sémantique* (le fait que chaque appel récursif soit sur les termes `y` tels que `R y x`) plutôt qu'une garde *syntaxique* (le fait que chaque appel récursif soit sur un sous-terme syntaxique). Ceci est problématique dans un contexte avec insignifiance des preuves car une fonction qui est définie par induction sur un prédicat d'accessibilité peut être dépliée infiniment souvent. Pour comprendre pourquoi, considérons le lemme d'inversion

```
Definition Acc_inv A R (x : A) (X : Acc x)
  : forall y:A, R y x -> Acc y
  := match X with Acc_intro f => f end.
```

Ce lemme d'inversion est toujours définissable, car même si l'élimination pour `Acc` est restreinte à son univers Prop le type de retour est ce même Prop (par imprédicativité).

Par contre, supposons que l'élimination sur `Acc` ne sois pas restreinte de manière à avoir l'éliminateur général

```
Acc_rect
     : forall (A : Type) (R : A -> A -> Prop)
         (P : A -> Type),
       (forall x : A,
        (forall y : A, R y x -> Acc R y) ->
        (forall y : A, R y x -> P y) -> P x) ->
       forall x : A, Acc R x -> P x
```

Alors, à partir de cette inversion et en utilisant l'insignifiance des preuves, l'égalité définitionnelle suivante peut être déduite pour tout prédicat `P: A -> Type` et fonction `F : forall x, (forall y, r y x -> P y) -> P x` et `X : Acc x`

```
  Acc_rect P F x X
≡ F x (fun y r => Acc_rect P F y (Acc_inv A R x X y r))
```

Dans un contexte ouvert, il est indécidable de savoir combien de fois ce déploiement doit être fait. Même la stratégie selon laquelle il y a au plus un déploiement peut ne pas prendre fin. En effet, supposons que nous nous trouvions dans un contexte où il existe une preuve `R_refl` montrant que `R` est réflexif (ce qui est inconsistent avec la preuve `X`). Ensuite, appliquer le dépliage ci-dessus une fois à `F: = fun x f => f x (R_refl x)` se réduit en

```
Acc_rect P F x X ≡ Acc_rect P F x (Acc_intro x (Acc_inv A R x X))
```

et le dépliage peut recommencer, à jamais.

Comme mentionné ci-dessus, si nous analysons la source de ce dépliage infini, il est dû à l'appel récursif à `Acc` dans l'argument de `Acc_intro` sur une variable arbitraire ‘‘ y‘‘ qui n'est pas *syntaxiquement* plus petite que la variable `x` initiale, mais *sémantiquement*

gardée par la condition `R y x`. Cet exemple montre que l'élimination du singleton n'est pas un critère suffisant pour savoir quand un type inductif dans SProp peut être éliminé dans n'importe quel type, car il faut introduire quelque chose de similaire à la condition de garde syntaxique sur les points fixes.

Voyons maintenant pourquoi ce n'est *pas une condition nécessaire* non plus.

Le bon et le mauvais inférieur ou égal

La définition de inférieur ou égal donnée dans l'introduction ne satisfait pas le critère d'élimination des singletons car elle a deux constructeurs. Cependant, on peut facilement montrer que `m <= n` est une proposition homooptique pour tous les entiers naturels `m` et `n`. C'est donc un bon candidat pour un inductif dans SProp qui soit éliminable dans n'importe quel type. La raison pour laquelle il s'agit d'une proposition homotopique est cependant plus subtile que ce que l'élimination des singletons demande habituellement, car tous les arguments des constructeurs de `le` ne sont pas dans SProp. Pour voir pourquoi c'est une proposition homotopique, il faut distinguer entre arguments de constructeur forcés et non forcés.[7] Un argument forcé est un argument qui peut être déduit des indices du type de retour du constructeur, et qui ne sont pas pertinents sur le plan des calculs. Considérons par exemple le constructeur `leS: forall m n, m le n -> m le S n`, ses deux premiers arguments `m` et `n` peuvent être calculés à partir du type de retour `S m <= S n` et sont donc forcés. En revanche, l'argument de type `m <= n` ne peut pas être déduit à partir du type et doit donc être dans SProp.

Cependant, être une proposition homotopique ne suffit pas, comme nous l'avons vu avec les types singleton et le prédicat d'accessibilité. Ici, la situation est encore plus subtile. Considérons la définition (propositionnellement) équivalente de “ n <= m“ suivante, qui est en fait celle utilisée dans la bibliothèque standard de Coq:

```
Inductive le' : nat -> nat -> SProp :=
  | le'_refl : forall n, n le' n
  | le'S : forall m n,  m le'  n -> m le' S n.
```

On peut aussi montrer que `le' m n` est une proposition homotopique, mais `le` et `le'` ne partagent pas le même principe d'inversion. En effet, dans le contexte (absurde) `e: le' (S n) n`, il y a deux façons de former un terme de type `le' (S n) (S n)`, soit en utilisant le `le'_refl (S n)`, soit en utilisant `le'S (S n) n e`. Cela signifie que permettre à “ le' m n“ d'être éliminé dans n'importe quel type oblige à être capable de décider si le contexte est absurde ou non, ce qui n'est évidemment pas une propriété décidable de la théorie des types. Pour `m <= n`, la situation est différente, car le type de retour des deux constructeurs `le0` et `leS` sont orthogonaux, dans le sens où ils ne peuvent pas être unifiés.

1.2.4 Le filtrage dépendant à la rescousse

Nous proposons un nouveau critère pour savoir quand un type dans SProp peut être éliminé dans un type arbitraire, réparant et généralisant le critère d'élimination des singletons. Ce critère est assez général pour distinguer les définitions de `<= `` et de ``le'`.

[7] Cette terminologie a été introduite par [BMM04].

En général, un type inductif dans SProp peut être éliminé s'il satisfait les trois propriétés suivantes :

1. Chaque argument non forcé doit être dans SProp.
2. Les types de retour des constructeurs doivent être orthogonaux deux à deux.
3. Chaque appel récursif doit satisfaire à une condition de garde syntaxique.

Pour justifier ce critère, nous fournissons une traduction générale de tout type inductif satisfaisant ce critère à un type équivalent défini comme point fixe, en utilisant des idées provenant du filtrage dépendant [CDP14][CD18]. En effet, regarder la définition inductive de la droite (sa conclusion) vers la gauche (ses arguments) nous permet de construire un *arbre des cas possibles* de manière similaire à ce qui est fait dans le filtrage dépendant. Fournir cette traduction signifie également que nous évitons la nécessité d'étendre notre système de base, comme tous les types inductifs peuvent être encodés en utilisant les primitives existantes.

Rejeter les arguments du constructeur qui ne sont pas dans SProp

La première propriété est la plus simple à comprendre: si un constructeur de type inductif peut stocker des informations qui sont pertinente sur le plan du calcul, il ne doit pas pouvoir être dans SProp (ou à minima, il ne doit pas être éliminable à un prédicat dans Type).

Rejeter les définitions non orthogonales

L'idée de la deuxième propriété est que les indices du type de retour de chaque constructeur doit déterminer dans quel constructeur nous sommes, en utilisant des indices disjoints pour les différents constructeurs. C'est une approximation syntaxique du critère d'orthogonalité. C'est la propriété qui ne tient pas pour `le'`.

Rejeter les points de fixation non terminaux

En plus des deux premières propriétés, nous avons également besoin d'une condition de garde syntaxique sur les arguments d'un constructeur récursif. Cette condition de garde impose que la définition de point fixe résultante soit bien fondée. Nous pouvons donc utiliser exactement la même condition syntaxique que celle déjà utilisée pour les points fixes déjà implémentée dans la théorie des types (pas importe laquelle, pour autant qu'elle garantisse la terminaison).

Par exemple, dans le cas de `Acc`, l'arbre des cas possibles induit la définition suivante, qui est automatiquement rejetée par le vérificateur de terminaison en raison de l'appel récursif non gardé `Acc 'y`

```
Fail Equations Acc' (x: A) : SProp :=
  Acc' x := (forall y:A, R y x -> Acc' y).
```

Dérivation automatique des points fixes et des éliminateurs

Si les trois propriétés sont satisfaites, nous pouvons automatiquement déduire un point de fixe dans SProp qui équivaut à la définition inductive. Chaque constructeur correspond à une branche unique d'un cas, et le type de retour dans chaque branche est la somme dépendante des arguments non forcés du constructeur correspondant (le cas zéro étant `sUnit`). Par exemple, pour `<=`, le point fixe est décrit par

```
Equations le_f (n m : nat) : SProp :=
  0 le_f n := sUnit;
  S m le_f S n := m le_f n;
  S _ le_f 0 := sEmpty.
```

Notons que les branches ne correspondant à aucun constructeur reçoivent la valeur `sEmpty`. On peut alors aussi définir des fonctions correspondant aux constructeurs et le principe d'élimination (dans Type) du type inductif de départ.

1.3 Travaux Reliés

L'insignifiance de preuves définitionnelle a été étdudiée par d'autres, bien que son implémentation par des assistants de preuves soit récente:

- En Agda sur le principe des arguments non pertinents, depuis Agda 2.2.8 en 2010, puis étendu pour la version 2.6.0 en 2019 pars un univers de propositions strictes (comme décrit dans cette thèse).
- En Lean (au moins depuis 2015 d'après de Moura et al. [dMKA+15]), en utilisant un univers de propositions strictes qui permet le prédicat d'accessibilité.
- En Coq depuis la version 8.10 en 2019.

Du coté théorique, nous avons par exemple

- Du travail sur les modèles de l'insignifiance de preuves, par example Miquel et Werner [MW03].
- Une preuve de la confluence en théorie des types avec un type d'égalité dans un univers imprédicatif de propositions strictes pars Werner [Wer08], avec une conjecture de normalisation plus tard contredite pars Abel et Coquand [AC19].
- Du travail sur les arguments non pertinents (décrit rapidement plus tôt dans cette introduction), par exemple Abel et Scherer [AS12]. Nous comparons cette approche avec l'univers de propositions strictes dans la section *Comparison With Other Irrelevance Systems* (page 72).
- Du travail par les développeurs de Lean, par exemple la preuve de consistence par Carneiro [Car19].

1.4 Contributions de cette thèse

Dans cette thèse, nous proposons le premier traitement général d'une théorie des types avec un univers des propositions avec des preuves insignifiantes définitionnellement, grâce à des intuitions venant de la notion de niveaux d'homotopie de HoTT et de la troncation propositionnelle. Cette extension de la théorie des types n'ajoute aucun axiome supplémentaire (mis à part l'existence d'un univers de preuves insignifiantes) et est notamment compatible avec l'univalence. Nous prouvons à la fois la cohérence et la décidabilité de la vérification du typage de la théorie des types résultante. Ensuite, nous montrons comment définir un critère presque complet pour détecter quelles définitions inductives dans SProp peuvent être éliminées dans n'importe quel type, en corrigeant et en étendant l'élimination des singletons. Ce critère, inventé en collaboration avec Jesper Cockx, utilise la méthodologie générale de l'unification sensible au calcul et du filtrage dépendant [CDP14][CD18]. Notre présentation n'est pas spécifique à une théorie de type particulière, ce qui est montré par une implémentation à la fois dans Coq, en utilisant un univers imprédicatif des preuves insignifiantes, et dans Agda, en utilisant une hiérarchie de prédicative d'univers de preuves insignifiantes.

CHAPTER

2 EXTENDED OVERVIEW

The text of this introduction is heavily derived from the introduction of Gilbert et al. [GCST19].

2.1 The Need for Proof Irrelevance

Proof-irrelevance, the principle that any two proofs of the same proposition are equal, is at the heart of the Coq extraction mechanism [Let04]. In the Calculus of Inductive Constructions (CIC), the underlying theory of the Coq proof assistant, there is an (impredicative) sort Prop that is used to characterize types that can be seen as propositions—as opposed to types whose inhabitants have a computational meaning, which live in the predicative sort Type. This static piece of information is used to extract a Coq formalization to a purely functional program, erasing safely all the parts involving terms that inhabit propositions, because a program can not use those terms in computationally relevant ways.

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:

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

However, in the current version of Coq, singleton elimination ensures compatibility with proof irrelevance, but does not provide it. 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 (without an axiom).

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

```
Definition boundednat (k : nat) : Type := { n : nat & n <= k }.
```

Here `boundednat 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 le : nat -> nat -> Prop :=
| le0 : forall n, 0 <= n
| leS : forall m n, m <= n -> S m <= S n.
```

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

```
Coercion boundednat_to_nat : boundednat >-> nat.
```

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 : boundednat k) (e : n + m <= k)
  : boundednat 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: boundednat k)
      e_1 e_2 e'_1 e'_2 :
 add (add n m e_1) p e_2 = add n (add m p e'_1) e'_2.
```

does not directly follow from associativity of addition on integers, as it additionally requires a proof that `e_2` equals `e'_2` (modulo the proof of associativity of addition of integers). This should be true because they are two proofs of the same proposition, but it does not follow automatically. Instead, the user has to prove proof-irrelevance for `<=` *manually*. This can be proven (using Agda style pattern matching of the Equations plugin[8] by induction on the proof of `m <= n`:

```
Equations le_hprop {m n}  (e e' : m <= n) : e = e' :=
   le_hprop (le0 _) (le0 _) := eq_refl;
   le_hprop (leS _ _ e) (leS n m e') := ap (leS n m) (le_hprop e e').
```

Note the use of functoriality of equality `ap : forall 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

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

type that is actually a mere proposition, in the sense of Homotopy Type Theory (HoTT) [UFP13]. Indeed, `<=` satisfies the propositional (as opposed to definitional, which holds by computation) version of proof irrelevance, as expressed by the `le_hprop` lemma. For an arbitrary inductive type in Prop, there is no reason anymore why it would be a mere proposition, and thus proof irrelevance, even in its propositional form, can not be proven and must be stated *as an axiom*.

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

```
Definition Squash (T : Type) : Prop := forall P : Prop, (T -> P) -> P.
```

This operator satisfies the following elimination principle (restricted to propositions) given by

```
forall (T : Type) (P : Prop), (T -> P) -> Squash T -> P.
```

which means that it corresponds to the propositional truncation [UFP13] of HoTT, originally introduced as the bracket type by Awodey and Bauer [AB04].

The importance of (definitional) proof irrelevance to simplify reasoning has been noticed for a long time, and various works have tried to promote its implementation in a proof assistant based on type theory [Pfe01][Wer08]. However, this has never been achieved, mainly because of the fundamental misunderstanding that *singleton elimination was the right constraint on which propositions can be eliminated into a type*.

Indeed, one can think of singleton elimination as a *syntactic* approximation of which types are naturally mere propositions, and thus can be eliminated into an arbitrary type without leaking a piece of computation or without implying new axioms. But, singleton elimination does not work for indexed datatypes, as for instance it applies to the equality type of Coq

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

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

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
   Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

satisfies the singleton elimination criterion but implementing definitional proof irrelevance for it leads to an undecidable conversion and thus an undecidable type checker.

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.[9]

But this partial implementation of the conversion algorithm breaks in particular subject reduction, which seems a desirable property for a proof assistant.

Finally, singleton elimination fails to capture inductive types with multiple constructors such as `<=` which are perfectly valid mere propositions and could be eliminated into types.

Looking back, Coq and its impredicative sort Prop may not be the only way to implement proof irrelevance in a proof assistant. Agda, which only has a predicative hierarchy of universes and no Prop, instead uses a notion of irrelevant arguments [AS12]. 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 `.` in the definition of `.(n <= k)`):

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

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

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

However, in this approach proof irrelevance is not a property of the type being considered but rather a way to use the argument of a given function. We hope that SProp will provide more flexibility to formal proof developers.

2.2 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.

2.2.1 SProp as a Syntactical Approximation of Mere Propositions

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

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

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

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.

2.2.2 Flirting with Extensionality

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

```
Inductive sUnit : SProp := tt : sUnit.
```

is contractible.

We will say that the unit type can be eliminated to any type, and thus we get a unit type with a definitional η-law such that `u` is convertible to `tt` for any `u : sUnit`. But in general, it should not be expected that any contractible type in SProp can be eliminated to an arbitrary type, as we can see below.

Singleton types

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

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

If we include singleton types in SProp, and hence permit its elimination over arbitrary types, we are led to an extensional type theory, in which propositional equality implies definitional equality. This would thus add UIP and undecidability of type checking to the theory.[13] Indeed, assume that singleton types can be eliminated to arbitrary types, then there exists a projection `pi1 : Sing A a -> A` which recovers the point from the singleton. But then, for any proof of equality `e: a = b` between `a` and `b` in `A`, using congruence of definitional equality, there is the following chain of implications

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

```
    (a ; refl) ≡ (b ; e) : Sing A a
=>  pi1 (a ; refl) ≡ pi1 (b ; e) : A
=>  a ≡ b : A
```

and hence `e : a = b` implies `a ≡ b`. From this analysis, it is clear that SProp cannot include all contractible types.

2.2.3 Flirting with Undecidability

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

```
Inductive sEmpty : SProp := .
```

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

```
sEmpty_rect : forall T : Type, sEmpty -> T.
```

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

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

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

Accessibility predicate

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

```
Definition Acc_inv A R (x : A) (X : Acc x)
  : forall y:A, R y x -> Acc y
  := match X with Acc_intro f => f end.
```

The inversion lemma is always defined, since even if elimination for `Acc` is restricted to Prop the return type is that same Prop (by impredicativity).

However, suppose elimination on `Acc` is not restricted such that we have the general eliminator

```
Acc_rect
     : forall (A : Type) (R : A -> A -> Prop)
         (P : A -> Type),
       (forall x : A,
        (forall y : A, R y x -> Acc R y) ->
        (forall y : A, R y x -> P y) -> P x) ->
       forall x : A, Acc R x -> P x
```

Then, from the inversion and using definitional proof irrelevance, the following definitional equality is derivable, for any predicate `P : A -> Type` and function `F : forall x, (forall y, r y x -> P y) -> P x` and `X : Acc x`

```
  Acc_rect P F x X
≡ F x (fun y r => Acc_rect P F y (Acc_inv A R x X y r))
```

In an open context, it is undecidable to know how many time this unfolding must be done. Even the strategy that there is at most one unfolding may not terminate. Indeed, suppose that we are in a context where there is a proof `R_refl` showing that `R` is reflexive (which is inconsistent with the proof `X`). Then, applying the unfolding above once to `F := fun x f => f x (R_refl x)` computes to

```
Acc_rect P F x X ≡ Acc_rect P F x (Acc_intro x (Acc_inv A R x X))
```

and the unfolding can start again for ever.

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

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

The Good and the Bad Less Than or Equal

The definition of less than or equal given in introduction does not satisfy the singleton elimination criterion because it has two constructors. However, `m <= n` can easily be shown to be a mere proposition for any natural numbers `m` and `n`. Thus, it is a good candidate for an SProp being eliminable into any type. The reason why it is a mere proposition is however more subtle than what singleton elimination usually requires, as not every argument of the constructors of `le` is in SProp. To see why it is a mere proposition, one needs to distinguish between forced and non-forced constructor arguments.[14] A forced argument is an argument that can be deduced from the indices of the return type of the constructor, and that are not computationally relevant. Consider for instance the

[14] This terminology has been introduced by [BMM04].

constructor `leS : forall m n, m le n -> m le S n`, its two first arguments `m` and `n` can be computed from the return type `S m <= S n` and are thus forced. In contrast, the argument of type `m <= n` cannot be deduced from the type and thus must be in SProp.

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

```
Inductive le' : nat -> nat -> SProp :=
  | le'_refl : forall n, n le' n
  | le'S : forall m n,  m le'  n -> m le' S n.
```

It can also be shown that `le' m n` is a a mere proposition, but `le` and `le'` do not share the same inversion principle. Indeed, in the (absurd) context that `e : le' (S n) n`, there are two ways to form a term of type `le' (S n) (S n)`, either by using `le'_refl (S n)` or by using `le'S (S n) n e`. This means that allowing `le' m n` to be eliminated into any type would require to decide whether the context is absurd or not, which is obviously not a decidable property of type theory. For `m <= n` the situation is different, because the return type of the two constructors `le0` and `leS` are orthogonal, in the sense they cannot be unified.

2.2.4 Dependent Pattern Matching to the Rescue

We propose a new criterion for when a type in SProp can be eliminated to an arbitrary type, fixing and generalizing the singleton elimination criterion. This criterion is general enough to distinguish between the definitions of `<=` and `le'`. In general, an inductive type in SProp may be eliminated if it satisfies three properties:

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

To justify this criterion, we provide a general translation from any inductive type satisfying this criterion to an equivalent type defined as a fixpoint, using ideas coming from dependent pattern matching [CDP14][CD18]. Indeed, looking at the inductive definition from right (its conclusion) to left (its arguments) allows us to construct a *case tree* similarly to what is done with a definition by pattern matching. Providing this translation also means we avoid the need to extend our core language, as all inductive types can be encoded using the existing primitives.

Rejecting constructor arguments not in SProp

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

Rejecting non-orthogonal definitions

The idea of the second property is that the indices of the return type of each constructor should fix in which constructor we are, by using disjoint indices for the different constructors. This is a syntactical approximation of the orthogonality criterion. This is the property that fails to hold for `le'`.

Rejecting non terminating fixpoints

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

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

```
Fail Equations Acc' (x: A) : SProp :=
   Acc' x := (forall y:A, R y x -> Acc' y).
```

Deriving fixpoints and eliminators automatically

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

```
Equations le_f (n m : nat) : SProp :=
  0 le_f n := sUnit;
  S m le_f S n := m le_f n;
  S _ le_f 0 := sEmpty.
```

Note that branches corresponding to no constructor are given the value `sEmpty`. One can then also define functions corresponding to the constructors and the elimination principle (to Type) of the inductive type.

2.3 Related Work

Definitional proof irrelevance has been studied by others, although it is only recently that is has gained implementations in proof assistants:

- In Agda based on modal proof irrelevance, released in Agda 2.2.8 in 2010, then extended in Agda 2.6.0 released in 2019 with a universe of strict propositions (as described in this thesis).

- In Lean (at least from 2015 according to de Moura et al. [dMKA+15]), using a universe of strict propositions allowing the accessibility predicate.
- In Coq since version 8.10 released in 2019.

On the theoretical side, we have for instance

- Work on models of proof irrelevance, for instance Miquel and Werner [MW03].
- A proof of confluence of type theory with equality types in an impredicative universe of strict propositions by Werner [Wer08], together with a conjecture of normalisation later proven false by Abel and Coquand [AC19].
- Work on modal proof irrelevance (i.e. irrelevant arguments, described shortly earlier in this introduction), for instance Abel and Scherer [AS12]. We do a short comparison of this approach with strict propositions in section *Comparison With Other Irrelevance Systems* (page 72).
- Work by the Lean team, such as the proof of consistency by Carneiro [Car19].

2.4 Contributions of this thesis

In this thesis, 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, designed in collaboration with Jesper Cockx, uses the general methodology of proof-relevant unification and dependent pattern matching [CDP14][CD18]. 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.

CHAPTER

3

DEPENDENT TYPE THEORY

3.1 Dependent Type Theory

3.1.1 Terms, Types and Contexts

Type theory is one way to describe the rules of mathematical constructions and proofs in a precise way. In type theory, we manipulate objects called "terms" which each come with a "type": we write $\vdash t : A$ for a term t with type A. Types specify the meaning of the term: it can be a natural number, a function, etc. Types can also correspond to mathematical propositions: we may have types for equality between terms, negation, etc. The type of functions between two propositions can then be interpreted as an implication from the first proposition to the other, and the terms inhabiting that type are the proofs of such implications.

Note that types and terms have the same grammar: types are seen as a special subset of terms.

In order to be able to define functions, we need a way to introduce a fresh term corresponding to the argument. In prose, we may say something like "the average of x and y is $\frac{x+y}{2}$" to define the averaging function.

In type theory we keep track of the introduced terms using a "context": a list of variables each with their type. Then the statements of the type theory (also called "judgments") look like $\Gamma \vdash t : A$ meaning "t has type A under context Γ", with $\Gamma = x_1 : A_1, ..., x_n : A_n$.

In order to make this work, we need 2 new judgments: $\vdash \Gamma$ for "Γ is a well-formed context" and $\Gamma \vdash A$ for "A is a well-formed type under context Γ". Our 3 judgments are mutually defined, for now we have

Ctx-empty: The empty context is well-formed.

$$\frac{}{\vdash}$$

Ctx-extend: Contexts can be extended using a well-formed type.

$$\frac{\Gamma \vdash A \qquad x \text{ free in } \Gamma}{\vdash \Gamma, x : A}$$

Note: In the rest of this thesis we will rename variables implictly to keep the side condition "x free in Γ" true.

Var: variables from a well-formed context get their type from it.

$$\frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

3.1.2 Functions

Now we are ready to describe how functions work. First, the type for functions (also called "product type"):

Pi-type

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash B}{\Gamma \vdash \Pi(x : A), B}$$

$\Pi(x : A), B$ is the type of functions from A to B. Notice that B lives in the context extended by $x : A$: this dependency is why the type theory is called "Dependent type theory". From a logical point of view, Π can be seen as universal quantification $\forall$.

For instance, if we have a type $x = x$ of reflexivity of equality at a variable x, then $\Pi(x : A), x = x$ is the type of reflexivity of equality for type A.

In $\Pi(x : A), B$, type A is the "domain" and B the "codomain" or output type. If x does not appear in B, this non-dependent function type may also be written $A \to B$, with the admissible rule

Arrow-type

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \to B}$$

Then we can create functions using lambda abstractions:

Lambda

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda(x : A), e : \Pi(x : A), B}$$

In the term $\lambda(x : A), e$ and the type $\Pi(x : A), B$, we say that the variable x is "bound" in e and B. Variables which are not bound are called "free". We consider all terms and types up to renaming of bound variables (alpha equivalence).

From the logical point of view, when B does not depend on x: if assuming a proof of A gives us a proof of B, then we have a proof that $A \to B$.

Functions can be used by applying them to an argument. To get the type of such an application, we first need to define substitution: $a[x := b]$ is a (which may be a term or a type) where variable x is substituted by term b. The precise definition of how substitution interacts with binders is subtle and unchanged in this thesis, please refer to your lambda calculus textbook. Informally, it means we replace occurences of x by b in a.

Then we can define application:

App

$$\frac{\Gamma \vdash f : \Pi(x : A), B \qquad \Gamma \vdash v : A}{\Gamma \vdash f \circ v : B[x := v]}$$

In the case where f is a non-dependent function, this becomes

App-Arrow

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash v : A}{\Gamma \vdash f \circ v : B}$$

From the logical point of view: if we have a proof of $A \to B$ and a proof of A, then we have a proof of B.

Usually we write application without the $\circ$ symbol: $f\ v$ instead of $f \circ v$.

We have a dichotomy between the λ terms and the application terms: the first construct values in function types, and the second make use of such values. We call the first kind of term "introduction" terms, and the second "elimination" terms.

3.1.3 Computation and Conversion

Computation and conversion let us identify the application of a lambda term with the body where the argument is substituted in: $(\lambda(x : A), e) \circ v = e[x := v]$. We will extend this as we add more introduction and elimination terms to our theory.

We add 3 new judgments, defined simultaneously with the term, type and context judgments:

- $\Gamma \vdash A \leq B$ "type A is a subtype of B under context Γ".
- $\Gamma \vdash A \equiv B$ "types A and B are convertible under context Γ".
- $\Gamma \vdash t \equiv u : A$ "terms t and u are convertible with type A under context Γ".

and the conversion rule:

Conversion

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \leq B}{\Gamma \vdash t : B}$$

The two conversion judgments are congruences: they are equivalence relations, and if 2 terms or types are identical except for convertible subterms (or subtypes) under Γ extended by whatever variables are bound around the subterms, then they are convertible under Γ. In other words we have rules such as

Lambda-congr

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash e_1 \equiv e_2 : B}{\Gamma \vdash \lambda(x : A), e_1 \equiv \lambda(x : A), e_2 : \Pi(x : A), B}$$

Conversion also contains the beta reduction:

Beta-conv

$$\frac{\Gamma, x : A \vdash e : B \qquad \Gamma \vdash v : A}{\Gamma \vdash (\lambda(x : A), e) \circ v \equiv e[x := v] : B[x := v]}$$

We may also add eta expansion for functions:

Eta-expand

$$\frac{\Gamma \vdash f : \Pi(x : A), B}{\Gamma \vdash f \equiv \lambda(x : A), f \circ x : \Pi(x : A), B}$$

The subtyping judgment is transitive and reflexive, and implied by conversion. It is also compatible with product types in the following way:

Pi-subtype

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash B_1 \leq B_2}{\Gamma \vdash \Pi(x : A), B_1 \leq \Pi(x : A), B_2}$$

Note: We could also have contravariance in the domain:

Pi-subtype-contravariant

$$\frac{\Gamma \vdash A_2 \leq A_1 \qquad \Gamma, x : A_1 \vdash B}{\Gamma \vdash \Pi(x : A_1), B \leq \Pi(x : A_2), B}$$

but this is incompatible with some models like the set model.

Until we have *Universe Cumulativity* (page 34) the separation between subtyping and conversion is not useful, so we may act as though we only have conversion.

Computation (also called reduction) is not a simultaneously defined judgment, rather it is an interesting property. It is generated by the beta reduction (i.e. $(\lambda\ x : A, e) \circ v$ reduces

to $e[x := v]$), transitivity and compatibility with the structure of terms. New type formers also come with additional reductions, for instance inductive types add reductions for the eliminators applied to constructors.

Because reduction is directed, it is possible to ask what a term reduces to. The rules of the type theory should ensure that there is a unique normal form (i.e. a term which does not reduce further), and once the necessary concepts are established we will have results such as "$2 + 2$ reduces to 4". Reduction is included in conversion.

3.1.4 Metatheoretical Properties

The work of this thesis is to extend type theory while maintaining nice properties. To do this, first we need some nice properties to maintain.

Weakening and substitution for judgments

Weakening means if we have a well-formed term or type in some context, we may freely extend the context with variables of well-formed types. In other words, the following rule is admissible:

Wk

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B}{\Gamma, x : B \vdash t : A}$$

(Remember that by convention we have picked an x which does not appear in Γ, and so is not free in t or A.)

We may also substitute a variable by an appropriately typed term:

Subst

$$\frac{\Gamma, x : A, \Delta \vdash t : T \qquad \Gamma \vdash v : A}{\Gamma, \Delta[x := v] \vdash t[x := v] : T[x := v]}$$

Subject Reduction

Reducing a term should not change its type: if $\Gamma \vdash t : A$ and t reduces to u then $\Gamma \vdash u : A$.

Note that when subtyping is not the same as conversion, u may have more types than t.

Strengthening

If x does not appear free in Δ, t and A, and if $\Gamma, x : T, \Delta \vdash t : A$ then $\Gamma, \Delta \vdash t : A$. In other words we can remove unused variables from the context.

Syntactic Validity

Syntactic validity means that the semantic sub-judgments of a valid judgment are also valid. For instance, if $\Gamma \vdash t : A$ then $\vdash \Gamma$ and $\Gamma \vdash A$.

Inversion of Judgments

Inversion of a judgment lets us get judgments about the subterms involved.

- **inversion of context extension:** if $\vdash \Gamma, x : A$ then $\Gamma \vdash A$ (and x is free in Γ).

Note: we can also get $\vdash \Gamma$, but this is redundant with $\Gamma \vdash A$ and syntactic validity so we will omit other such side results for the other inversions.

- **inversion of variables:** if $\Gamma \vdash x : A$ then $x : A \in \Gamma$.
- **inversion of product types:** if $\Gamma \vdash \Pi(x : A), B$ then $\Gamma, x : A \vdash B$.
- **inversion of lambda abstractions:** if $\Gamma \vdash \lambda(x : A), e : T$ then there exists B such that $\Gamma, x : A \vdash e : B$ and $\Gamma \vdash \Pi(x : A), B \leq T$.
- **inversion of application:** if $\Gamma \vdash f \circ v : T$ then there exist A, x and B such that $\Gamma \vdash f : \Pi(x : A), B$ and $\Gamma \vdash v : A$ and $\Gamma \vdash B[x := v] \leq T$.

There are also inversions for the conversion judgments, which look like injectivity rules. For instance:

- **injectivity of product types:** if $\Gamma \vdash \Pi(x : A_1), B_1 \equiv \Pi(x : A_2), B_2$ then $\Gamma \vdash A_1 \equiv A_2$ and $\Gamma, x : A_1 \vdash B_1 \equiv B_2$ (the choice of A_1 to extend Γ is unimportant, by substitution)

Note that we only have injectivity for the introduction forms. With the elimination forms, reduction may get involved. For instance, $\vdash 0 + 0 \equiv 0 * 1$ even though $+$ and $*$ are not convertible, and neither are 0 and 1 (where $0 + 0$ is the infix notation for application of addition to 0 and 0, and so on for $0 * 1$).

We can generalize this issue by introducing the concepts of head normal forms and neutral terms.

- each elimination form has a subterm which is considered the "head". For application $f \circ v$ this is f.
- a term is neutral if it is a variable or if it is an elimination whose head is neutral.
- a term or type is in head normal form if it is neutral or if it is one of the introduction forms (currently Π or λ). If it is a lambda abstraction $\lambda(x : A), e$ the body e must also be head normal (without this condition we would define weak head normal form).

Then each term former is injective for conversion on head normal terms: introduction forms are always injective, elimination forms are injective when the head is neutral, and if 2 variable terms are convertible they are the same variable.

Head normal terms built from different term formers are never convertible, unless one of them is a lambda. In that case we need to eta expand the other: when f and e are in HNF, $\Gamma \vdash f \equiv \lambda(x : A), e : X$ if and only if $\Gamma, x : A \vdash f \circ x \equiv e : B$ (with B gotten from inversion given $\Gamma \vdash \lambda(x : A), e : X$).

Note: Using head normal form instead of weak head normal form means that the above eta expansion only considers normal forms, without having to reduce e after expanding. This is only a convenience for that remark, usually we talk about weak head normal forms and interleave reduction and inversion.

Note: This last "no confusion" property will become weaker as we add types with definitional eta equalities like *Records* (page 45), and of course when we add proof irrelevance.

Uniqueness of Typing

When subtyping coincides with conversion, uniqueness of typing is easily stated:

Uniq-type

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash t : B}{\Gamma \vdash A \equiv B}$$

The situation when we have real subtyping is explained in *Metatheory of Universes* (page 35).

Normalisation, Confluence and Canonicity

All terms reduces to a unique normal form. Equal terms have syntactically equal normal forms, if we include eta expansion in the reduction. Once we introduce proof irrelevance, this becomes syntactic equality up to irrelevant subterms.

Canonicity means that in the empty context normal terms are built from introduction forms (binders such as lambda abstractions will also contain neutral subterms). Combined with inversion lemmas, this implies for instance that every natural number term in the empty context reduces to a finite number of successors applied to zero, i.e. a natural number value.

Note: Normalisation vs Reducibility

Normalisation is not equivalent to having all terms reduce to a weak head normal form. Consider for instance a system with natural numbers and some term `x` which reduces to `S x`. The term `S x` is weak head normal, but `x` is not normal, and has no normal form.

Decidability

The above properties are in support of decidability of the various judgments. Decidability makes it possible to implement proof assistants so that we don't have to check our derivations by hand.

Consistency

If we want to use our terms to represent proofs, we need to ensure that not everything has a proof. As such we soon (in *Notable inductive types* (page 39)) have a type Empty which is not inhabited in the empty context.

3.1.5 Example: functions over natural numbers

We add

- `ℕ` such that $\vdash$ `ℕ`
- `0` such that $\vdash$ `0 : ℕ`
- `S` such that $\Gamma \vdash$ `S : ℕ` $\to$ `ℕ`
- `natrec (x. T) z s n` such that
  - `natrec (x. T) z s 0` reduces to `z`
  - `natrec (x. T) z s (S n)` reduces to `s n (natrec (x. T) z s n)`

  with the typing rule

natrec

$$\frac{\Gamma, x : \mathbb{N} \vdash T \qquad \Gamma \vdash z : T \qquad \Gamma \vdash s : \Pi(a : \mathbb{N}), T[x := a] \to T[x := S\ a] \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \text{natrec}\ (x.T)\ z\ s\ n : T[x := n]}$$

Note: For now we have no way to abstract over types, but if we did we could present `natrec` as something whose type is a Π instead of being forced to treat `(x. T)` specially.

This is enough to define structurally recursive functions on natural numbers. For instance, addition is `add := λ n m. natrec (x. ℕ) n (λ a. S a) m` (with recursion on the second argument).

On the other hand, all our types are formed from `ℕ` and $\to$: we have no way to make a type be about a specific number `n`, and so we have no way to prove properties about the functions we define inside the type theory.

Let's introduce a type for equalities between natural numbers to help:

- `t = u` such that

eq-type

$$\frac{\Gamma \vdash t : \mathbb{N} \qquad \Gamma \vdash u : \mathbb{N}}{\Gamma \vdash t = u}$$

- `refl` such that `⊢ refl : Π (x:ℕ), x = x`

The general form for eliminating equalities is slightly complicated, so we wait until we explain inductive families to present it and for the sake of this example we instead provide injectivity of successor as a primitive:

- `S-inj` such that `⊢ S-inj : Π (x y:ℕ), x = y → S x = S y` and `S-inj a b (refl a)` reduces to `refl (S a)` (NB: when `S-inj a b (refl a)` is well typed `a` and `b` are convertible).

This is enough to show commutativity of `add` internally to our theory. As a first step we show that `add 0 n = n`:

```
⊢ λ (m : ℕ).
  natrec (n. add 0 n = n)
         (refl 0)
         (λ (m' : ℕ) (e : add 0 m' = m'). S-inj (add 0 m') m' e)
         m
  : Π (m : ℕ), add 0 m = m
```

Conversion is involved to make the arguments of `natrec` fit:

- `refl 0 : 0 = 0` needs to have type `add 0 0 = 0`. `add 0 0` reduces to `0` so this subterm has the right type.
- `S-inj (add 0 m') m' e : S (add 0 m') = S m'` needs to have type `add 0 (S m') = S m'`. `add 0 (S m')` reduces to `S (add 0 m')` so this also works.

The remainder of the commutativity proof is left as an exercise to the reader.

With new types and terms, we need to update the definitions of neutral and weak head normal forms:

- `natrec (x. T) z s n` is neutral when `n` is neutral.
- `S-inj a b e` is neutral when `e` is neutral.
- The type `nat` and equality types are weak head normal introduction forms (note that we have no way to define neutral types, and types have no head reductions).
- `0`, the applied successor `S n` and the applied reflexivity proof `refl a` are weak head normal introduction forms.

Then all terms have normal forms, and we have canonicity.

Our other desired properties are also preserved: this is a well-behaved type theory.

The reduction rule of `S-inj` has only been useful for canonicity this far: we have no way to eliminate proofs of equalities, so we don't care about their values.

3.2 Proof assistants, especially Coq

Proof assistants are computer programs which keep track of the details of typing derivations for us, check them to make sure we didn't make a mistake and help us produce them.

There are many proof assistants, not all based on dependent type theory. In this thesis we focus on Coq, and also consider Agda and Lean, all of which use dependent type theory. In the following short presentation of Coq, you should assume that Agda and Lean have similar capabilities.

We don't present the full power of Coq, only enough to understand the code examples we use in the thesis.

A proof assistant carries a global context of checked proof terms bound to names, in other words definitions. Coq has a basic command to add a term as a definition:

```
Definition some_def : nat := 0 + 0.
```

`0 + 0` is syntactic sugar for the application `Nat.add 0 0`, where `Nat.add` is a previous definition (from Coq's standard library).

We can also ask Coq to check a term without making a definition:

```
Check 0 + 1.
    0 + 1
         : nat
```

(`1` is syntactic sugar for `S 0`)

Definitions reduce to their value. We can see what terms reduce to using the following command:

```
Compute 1 + some_def.
    = 1
         : nat
```

Coq has powerful facilities to let us avoid typing parts of terms. Consider the polymorphic identity function

```
Definition id (A:Type) (a:A) : A := a.
```

(the meaning of `Type` is explained in the next section, *Universes* (page 32))

If we apply it as `id A 0`, by inversion we must have `A ≡ nat`. We can indicate to Coq that it should automatically infer a subterm by replacing it with a hole `_`:

```
Check id _ 0.
    id nat 0
         : nat
```

Such holes can be automatically inserted using the "implicit argument" system:

```
Arguments id {_} _.
Check id 0.
    id 0
         : nat
```

Equivalently we could have written

```
Definition id {A} (a:A) := a.
```

Using curly brackets `{}` around an argument means it is implicit. In the above `Definition` we omitted the types for `A` and for the result of `id`: Coq automatically replaced these omissions by holes and solved them.

Coq syntax for the term formers we have already introduced:

- Product types: $\Pi(x : A), B$ is `forall x : A, B`. The type of `x` may be omitted.

  $A \to B$ is `A -> B`.
- Lambda abstractions: $\lambda(x : A), e$ is `fun x : A => e`. The type of `x` may be omitted.
- Applications: $a \circ b$ is `a b`.

We can only ask Coq for judgments about terms, not conversions, except by crafting a term which depends on a conversion. For instance

```
Check (fun (f:A -> nat) (x:B) => f x).
```

succeeds if and only if $B \leq A$.

3.3 Universes

How to formalize a polymorphic concept? Consider the example of lists whose elements all have the same type. We could introduce special type and terms formers `list A` (which is a type when `A` is a type), `nil A : list A`, `cons A : A → list A → list A` and some eliminators. But derived notions about lists also need to depend on the abstract type `A`: shall we continue to introduce a new basic term formers for the `length` function (such that `length A : list A → nat`)?

If only we had a type `Type` to be the type of types. Then we could have `list : Type → Type`, `nil : forall A : Type, list A`, `cons : forall A : Type, A → list A → list A` and whatever eliminator we like as (for now) the primitives providing lists, and `length : forall A : Type, list A → nat` is just some defined notion.

A type of types (also called universe or sort) is itself a type, so it sounds like it should be an element of itself. Sadly having `Type : Type` is inconsistent ([Hur95]) so we must be a bit smarter.

The usual solution, implemented in all of Coq, Agda and Lean is to have a so-called universe hierarchy: we have not just one `Type` for all types, but instead we have some $\texttt{Type}_i$ for varying `i`. We ensure that we never have $\texttt{Type}_i : \texttt{Type}_i$, but we may have $\texttt{Type}_i : \texttt{Type}_j$ when `i` and `j` are different. In Coq $\texttt{Type}_i$ is written `Type@{i}`.

More general systems are possible, but for the purpose of this thesis we will consider that universe indices are natural numbers, or special (and so explained separately). The main point of this thesis is to introduce and examine the behaviour of one such special universe.

Note: In some cases we only have a finite number of universes, which corresponds to only allowing some natural numbers as universe indices. For instance in section *Decidability and Reducibility with Proof Irrelevance* (page 58) we prove decidability of conversion for a type theory simplified to have only $\texttt{Type}_0$ and our special universe, and no $\texttt{Type}_1$. This means not all types belong to some universe.

With `Type : Type` the rules involving universes could be very simple:

univ-is-type

$$\frac{}{\Gamma \vdash \mathsf{Type}}$$

univ-to-type

$$\frac{\Gamma \vdash A : \mathsf{Type}}{\Gamma \vdash A}$$

type-to-univ

$$\frac{\Gamma \vdash A}{\Gamma \vdash A : \mathsf{Type}}$$

With a universe hierarchy, we can keep *univ-is-type* (page 33) and *univ-to-type* (page 33) (annotating `Type` with an arbitrary universe index). However *type-to-univ* (page 33) must change: we can't use an arbitrary level there at risk of going back to `Type : Type`.

We adapt by adding a rule forming an inhabitant of a universe for every rule forming a type. For now we only have 2 type formers: the previously introduced product types, and the newly added universes.

Each universe contains the universes which have smaller indices:

univ-in-univ

$$\frac{i < j}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_j}$$

Note that this means $\texttt{Type}_0$ (called `Set` in Coq) is slightly special: it contains no universe. The types which it does contain are called "small types". For now the only small types are built from assumptions in the context, but for instance once we introduce natural numbers they will be small types.

Product types belong to the maximum of the domain and codomain's universes (when they exist).

Pi-in-univ

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

When we have a universe for every natural number (such as in Coq), every universe has a type (the successor universe), and from this we can show that every type has a universe. We also have all product types: $max(i, j)$ always exists.

3.3.1 Universe Cumulativity

Universe cumulativity means that if `A : Type`$_i$ and `i ≤ j` then we also have `A : Type`$_j$: every universe includes the smaller ones.

Coq has universe cumulativity, but other systems such as Agda and Lean don't. Instead they have explicit lifting: functions `lift`$_{i,j}$ `: Type`$_i \to$ `Type`$_j$ for every `i < j`, such that `lift`$_{i,j}$ `A` is in bijection with `A` (with cumulativity they would be convertible). Usually the lift is provided by tweaking the formation rules of inductive types.

3.3.2 Universe Variables and Universe Polymorphism

Consider the product of type `A × B`. To avoid providing an embedding from larger to smaller universes, we need to have $\times : \mathsf{Type}_i \to \mathsf{Type}_j \to \mathsf{Type}_{max(i,j)}$, but which `i` and `j` are these?

We can avoid picking by using abstract variables for `i` and `j`, keeping a record of the relations between variables (such as `i < j`) to ensure that we can always assign specific numbers to them and get a well-typed judgment.

In fact in Coq, most universes are built from variables and `Set`, with successor and maximum expressions allowed only in the codomain of the type of judgments (such as the type for `×` above).

If we have a type involving universe expressions (also called "algebraic universes") it can be transformed into a type using only variables by generating fresh variables. For instance we can generate a fresh `k` such that `i ≤ k` and `j ≤ k`, then $\times : \mathsf{Type}_i \to \mathsf{Type}_j \to \mathsf{Type}_k$ by cumulativity.

This is still not enough to get the flexibility we want: if we have some type `nat : Type`$_0$, by cumulativity `nat × nat` is well-typed (0 being smaller than every natural, it can also be assumed smaller than every variable) but its type is still $\mathsf{Type}_{max(i,j)}$ even though a pair of natural numbers could be a small type.

To fix this Coq provides universe polymorphism: constants and inductive types may quantify over the universes and universe constraints they use, and later be instantiated by different universes. For instance:

```
Polymorphic Inductive prod@{a b} (A:Type@{a}) (B:Type@{b}) :=
  pair : A -> B -> prod A B.
```

```
About prod.
    prod@{a b} :
    Type@{a} -> Type@{b} -> Type@{max(a,b)}
    (* a b |=  *)

    prod is universe polymorphic
    Arguments prod _%type_scope _%type_scope
    Expands to: Inductive Top.prod

Check prod nat nat.
    prod@{Set Set} nat nat
         : Set
```

Note: The `prod` type found in Coq's standard library does not use universe polymorphism, instead it uses "template polymorphism". It is a similar but more ad-hoc system which does not matter to this thesis.

3.3.3 Universe of Propositions

Coq provides a special universe Prop whose elements are intended to represent propositions, i.e. logical statements whose proof values are of little interest. Prop was introduced in Coq version 4 (1987) to help with extraction: Coq terms can be translated to terms in non-dependent languages such as OCaml while erasing subterms whose types are propositions. As such the rules for defining elements of Prop are focused on ensuring that computation after erasure does not depend on the value of the erased subterms.

Prop is parallel to `Set` in the hierarchy in that `Prop : Type`$_1$.

Prop is impredicative: product types are propositions whenever their codomain is a proposition, regardless of the universe of the domain.

Pi-in-Prop

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash B : \mathsf{Prop}}{\Gamma \vdash \Pi(x : A).B : \mathsf{Prop}}$$

For consistency we must avoid having any universe belong to an impredicative universe, so there is no `Type`$_i$ `: Prop`.

On the other hand nothing stops us from having multiple impredicative universes, and Coq used to have `Set` impredicative (it can still be made impredicative with a command line option).

3.3.4 Metatheory of Universes

Adding a universe hierarchy preserves our expected properties: substitution for judgments, decidability, consistency, etc.

With cumulativity, uniqueness of typing becomes existence of a principal type: if t is typed in Γ then there exists A such that $\Gamma \vdash t : A$ and for all B, if $\Gamma \vdash t : B$ then $\Gamma \vdash A \leq B$. Type_{u+1} (if it exists, for finite hierarchies) is the principal type for Type_u.

Universes add a new introduction form for types.

We have new inversions related to universes:

- If $\Gamma \vdash t \equiv \mathsf{Type}_u$, then t reduces to Type_u (with universe variables, t reduces to Type_v for some v equal to u).

  If $\Gamma \vdash t \leq \mathsf{Type}_u$ (resp. $\Gamma \vdash \mathsf{Type}_u \leq t$) then it reduces to Type_v with $v \leq u$ (resp. with $u \leq v$).

- All types of types are universes: if $\Gamma \vdash A : T$ and $\Gamma \vdash A$ then T is convertible to some universe.

  If we have the full infinite hierarchy then every type has a universe: if $\Gamma \vdash A$ then there exists u such that $\Gamma \vdash A : \mathsf{Type}_u$.

- inversion for Π: if a product type belongs to some universe, then domain and codomain belong to suitable universes.

3.4 Inductive Families

An inductive family *Ind* is given by the following data:

- a context of "parameters" $\Gamma_P = p_1 : P_1, ..., p_m : P_m$.
- a context of "indices" $\Gamma_I = i_1 : I_1, ..., i_n : I_n$, which may depend on the parameters (i.e. $\Gamma_P, i_1 : I_1, ..., i_k : I_k \vdash I_{k+1}$ for $0 \leq k < n$). When there are no indices we may also say that *Ind* is an inductive **type**.
- a universe u.
- a list of "constructors" $\Gamma_C = c_1 : C_1, ..., c_l : C_l$. Each constructor type C_i may not depend on the other constructors, and may depend on the parameters and the inductive: $\Gamma_P, Ind : \Pi\, \Gamma_P\, \Gamma_I, \mathsf{Type}_u \vdash C_i$.

  Additionally, each C_i must be of the form

  $$\Pi\, \Gamma_{X,i}, Ind\, \Gamma_P\, v_{i,1}\, ...\, v_{i,n}$$

  The codomain is the inductive applied to exactly the parameter variables, and then some arbitrary values for the indices.

  Using $\Gamma_{X,i} = x_{i,1} : X_{i,1}, ..., x_{i,a_i} : X_{i,a_i}$, each $X_{i,k}$ must either not refer to *Ind* (in which case it is called a "nonrecursive argument" of c_i), or be of the form

  $$\Pi\, \Gamma_{Y,i,k}, Ind\, \Gamma_P\, w_{i,k,1}\, ...\, w_{i,k,n}$$

  i.e. a product type whose codomain is the inductive at some indices, and such that no variable of $\Gamma_{Y,i,k}$ and no w refers to *Ind*. In this case we say that *Ind* occurs "strictly positively" in $X_{i,k}$ and $x_{i,k}$ is a "recursive argument" of c_i.

Finally there is a condition on the universe: it must be greater or equal than each of the universes of the types of the arguments.

Then $\vdash Ind : \Pi\ \Gamma_P\ \Gamma_I, \mathsf{Type}_u$ and for each i, $\vdash c_i : \Pi\ \Gamma_P, C_i$.

A type theory with inductive families also provides a way to eliminate them, which among other things expresses that all inhabitants are recursively built from the constructors. This can be done in 2 ways:

- Providing primitive elimination operators for each inductive family. The operator $Ind\_elim_{u_T}$ (universe polymorphic over u_T) has type

$$\begin{aligned}
&\Pi\ \Gamma_P\ \left(T : \Pi\ \Gamma_I, Ind\ \Gamma_P\ \Gamma_I \to \mathsf{Type}_{u_T}\right), \\
&\quad \left(\Pi\ \Gamma_{X,1}\ \Gamma_{R,1}, T\ v_{1,1}\ ...\ v_{1,n}\ (c_1\ \Gamma_P\ \Gamma_{X,1})\right) \to \\
&\quad ... \to \\
&\quad \left(\Pi\ \Gamma_{X,l}\ \Gamma_{R,l}, T\ v_{l,1}\ ...\ v_{l,n}\ (c_l\ \Gamma_P\ \Gamma_{X,l})\right) \to \\
&\quad \Pi\ \Gamma_I\ (z : Ind\ \Gamma_P\ \Gamma_I), T\ \Gamma_I\ z
\end{aligned}$$

  where $\Gamma_{R,i}$ contains an "induction hypothesis" variable for each recursive argument $x_{i,k}$ of c_i. The type of the variable is then

$$\Pi\ \Gamma_{Y,i,k}, T\ w_{i,k,1}\ ...\ w_{i,k,n}\ \left(x_{i,k}\ \Gamma_{Y,i,k}\right)$$

  In other words, in order to inhabit a type family T (called the "motive" of the elimination) for all instantiations of the inductive, it suffices to provide functions inhabiting it for each constructor provided it is inhabited for the recursive arguments.

  When applied to a constructor of the inductive type, the elimination operator reduces to the function given for that constructor, applied to the constructor's arguments and to the eliminator applied to the recursive arguments.

  Using $Elim = Ind\_elim_{u_T}\ \Gamma_P\ T\ F_1\ ...\ F_l$:

$$\begin{aligned}
&Elim\ v_{i,1}\ ...\ v_{i,n}\ \ (c_i\ \Gamma_P\ \Gamma_X) \equiv \\
&F_i\ \Gamma_X\ \left(\lambda\ \Gamma_{Y,i,1}, Elim\ w_{i,1,1}\ ...\ w_{i,1,n}\ \ \left(x_{i,1}\ \Gamma_{Y,i,1}\right)\right) \\
&\quad ...\ \left(\lambda\ \Gamma_{Y,i,a_i}, Elim\ w_{i,a_i,1}\ ...\ w_{i,a_i,n}\ \ \left(x_{i,a_i}\ \Gamma_{Y,i,a_i}\right)\right)
\end{aligned}$$

- We can separate case distinction and recursion in 2 related primitive operations.

  First for case distinction we provide a "match" operator:

```
match e as x in Ind _ Γ_I return T with
| c₁ Γₓ,₁ => F₁
...
| cₗ Γₓ,ₗ => Fₗ
end
```

  (because unicode has no capital letter subscripts we use `Γ_I` instead of Γ_I and `Γₓ,ᵢ` instead of $\Gamma_{X,i}$ in code)

`e` is an expression at some instantiation $Ind\ e_{P,1}\ ...\ e_{P,m}\ e_{I,1}\ ...\ e_{I,n}$ of the inductive family. Γ_I and $x : Ind\ e_{P,1}\ ...\ e_{P,m}\ \Gamma_I$ are bound in the type T. $\Gamma_{X,i}$ is bound in each F_i which must have type $T\ v_{i,1}\ ...\ v_{i,n}\ \left(c_i\ e_{P,1}\ ...\ e_{P,m}\ \Gamma_{X,i}\right)$.

The entire `match` then has type $T\ e_{I,1}\ ...\ e_{I,n}\ e$.

The _ in Ind _ stands for the parameters. Since they are invariant for the whole match, we don't bind variables for them in the return type.

We may omit each of the annotations `as x`, `in Ind _ Γ_I` and `return T`, leaving them to be inferred by the proof assistant and the reader.

When `e` is the applied constructor c_i applied to some arguments, the match reduces to $F_{X,i}$ with $\Gamma_{X,i}$ bound to the corresponding arguments.

Then we provide a "fix" operator for recursion:

```
fix F (a₁ : A₁) ... (aᵣ : Aᵣ) {struct aᵢ} : T := e
```

We say that a_i is the "principal argument" or "decreasing argument" of the fixpoint. It must be of some inductive type. $F : \Pi\ (a_1 : A_1)...(a_r : A_r), T$ and each of the a_1 to a_r are bound in `e` whose type must be `T`.

When the fixpoint is applied such that the value for the principal argument is a constructor, it reduces to `e` applied to the same arguments and with the fixpoint substituted for `F`.

There is a side-condition for the fixpoint to be well-formed: appearances of `F` in `e` must be applied such that the value for the principal argument is syntactically smaller than a_i.

"Syntactically smaller" means it comes from a recursive argument of a constructor from a match on a_i.

It is trivial to implement the operation eliminator using match and fixpoint. The other direction is more difficult and depends on the specification of "syntactically smaller". This specification is beyond the scope of this thesis, in which we only need that the translation to elimation operators remains possible.

The proof assistant Lean uses elimination operators, and provides syntactic sugar to provide user syntax matches which are translated to eliminators. Coq uses primitive fixpoints and matches, and automatically provides elimination operators as defined terms.

Agda uses a more complex form of pattern matching than we described. By default it is more powerful than elimination operators (it proves *Uniqueness of Identity Proofs* (page 48)), but with the flag `--without-K` it is equivalent to them.

Note: When the type theory lacks universe cumulativity, we can define lifts using inductive types instead of having a special language construct for lifts:

```
Inductive Lift@{u v|u < v} (A:Type@{u}) : Type@{v} :=
  lift : A -> Lift A.
```

This works so long as the condition on the universe of the inductive uses the cumulative formulation "greater or equal". In other words, we get explicit cumulativity through inductive types.

3.4.1 Notable inductive types

Let us consider some examples (in Coq syntax):

- `Inductive Empty : Prop := .`

  `Empty` is an inductive type with no parameters, no indices and no constructors. Its universe is Prop.

  Because there are no constructors it is impossible to build values of `Empty`, justifying its name. The eliminator expresses that anything may be proved from `Empty`:

  ```
  Empty_rect
       : forall P : Type, Empty -> P
  ```

  Coq generates a non-dependent eliminator for inductive types in Prop, but we could still define the dependent version if necessary.

- `Inductive unit : Set := tt : Unit.`

  `unit` has a unique constructor `tt` which has no arguments. We may omit the type annotation, since there are no output indices to specify:

  ```
  Inductive unit : Set := tt.
  ```

  The eliminator expresses that to prove a property of some element of `unit` it suffices to prove it for `tt`:

  ```
  unit_rect
       : forall P : unit -> Type,
         P tt -> forall u : unit, P u
  ```

- `Inductive bool := true | false.`

  An inductive type with 2 elements. We have omitted the type annotation for the inductive itself this time, as Coq can automatically infer it (its universe is Set).

  The non-dependent eliminator is an `if` combinator:

  ```
  Scheme If := Minimality for bool Sort Type.
  ```

  ```
  If
       : forall P : Type, P -> P -> bool -> P
  ```

Note: The condition is the *last* argument.

```
Definition xor b1 b2 := If _ (If _ false true b2)
                             b2
                             b1.


Compute (xor true true).
    = false
         : bool
```

- `Inductive nat := 0 | S : nat -> nat.`

  The type of Peano natural numbers (i.e. unary numbers). To avoid having to write the output type, we can also specify constructor arguments in the following way:

  ```
  Inductive nat := 0 | S (n : nat).
  ```

  The eliminator is induction over natural numbers:

  ```
  Compute nat_rect.
      = fun (P : nat -> Type) (f : P 0)
              (f0 : forall n : nat, P n -> P (S n)) =>
            fix Ffix (x : nat) : P x :=
              match x as c return (P c) with
              | 0 => f
              | S x0 => f0 x0 (Ffix x0)
              end
           : forall P : nat -> Type,
             P 0 ->
             (forall n : nat, P n -> P (S n)) ->
             forall n : nat, P n
  ```

  Since `S` has a recursive argument, the eliminator uses `fix` in its implementation.

- `Inductive eq {A} a : A -> Prop := eq_refl : eq a a.`

  (`{A}` means `A` is an implicit argument: Coq will automatically infer and insert the correct value by solving typing constraints. For instance `eq 0 0` must have `A ≡ nat`. If we want to provide it explicitly we can use `@eq`: the `@` locally disables insertion of implicit arguments)

  `eq` represents equality between values, as such Coq will print `eq a b` as `a = b`:

  ```
  Check eq 0 1.
      0 = 1
           : Prop
  ```

  The eliminator expresses that `eq` is a Leibnitz equality, i.e. if 2 values are equal any predicate inhabited for one is inhabited for the other:

```
eq_rect
     : forall (A : Type) (x : A) (P : A -> Type),
       P x -> forall y : A, x = y -> P y
```

Equality is especially important to us, since proof irrelevance is about equality. We will inspect the equality inductive with more depth in section *Equality* (page 47).

- `Inductive Sigma A B := sigma : forall a : A, B a -> Sigma A B.`
  `Arguments sigma {_ _} _ _.`

  A type for dependent cartesian products. We can define projections using `match`:

  ```
  Definition pr1 {A B} (x:Sigma A B) : A :=
    match x with sigma a _ => a end.
  Definition pr2 {A B} (x:Sigma A B) : B (pr1 x) :=
    match x with sigma a b => b end.
  ```

  Coq has automatically inferred the appropriate matching predicate to get the second match to typecheck:

  ```
  pr2 =
  fun (A : Type) (B : A -> Type) (x : Sigma A B) =>
  match x as x0 return (B (pr1 x0)) with
  | sigma _ b => b
  end
       : forall (A : Type) (B : A -> Type)
           (x : Sigma A B), B (pr1 x)

  Arguments pr2 {A%type_scope} {B%function_scope}
  ```

- `Inductive Sum A B := inl (a:A) | inr (b:B).`

  A discriminated sum type. It may be interesting to the reader to prove that `Sum A B` is equivalent to `Sigma bool (If Type A B)` (where `If` is partially applied), and that `bool` is equivalent to `Sum unit unit`.

- `Inductive W A (B : A -> Type) :=`
  `w : forall a, (B a -> W A B) -> W A B.`

  `W` may be used as a combinator to express recursive inductive types, instead of providing them as primitives.

  For instance, `nat` is reformulated as `W bool (If Empty unit)` which involves no recursive type other than `W`. We then have `0 = w true (fun e => match e with end)` and `S n = w false (fun _ => n)`.

  Note however that because one of the arguments of `w` is a function, the reformulation of `nat` is not fully equivalent to `nat` without function extensionality. For instance `w true (fun e => 0)` is neither equal to `0` or to a successor.

3.4.2 Inductives and Prop

Because we want to be able to erase propositional values at extraction time, we must put some additional restrictions on inductives declared in Prop.

For instance, if we had `bool : Prop`, then the extraction of `If` would somehow have to work without using the condition argument.

We first restrict inductives in Prop to only those which have at most one constructor, with all arguments themselves in Prop.

Then extraction of a match on a 0-constructor inductive produces an unreachable case (`assert false` in OCaml), and for a match on a 1-constructor inductive we just extract the unique branch.

This condition allows for instance `Empty`, `unit`, and `eq`.

We can then allow all other inductives, but with their eliminations restricted to motives in Prop. For instance:

```
Inductive ex A (B : A -> Prop) : Prop :=
  ex_intro (a : A) (b : B a).
```

(Coq will print `ex A B` as `exists x : A, B x`)

```
ex_ind
     : forall (A : Type) (P : A -> Prop) (P0 : Prop),
       (forall x : A, P x -> P0) ->
       (exists y, P y) -> P0
```

It is impossible to define a corresponding `Exists_rect` with `P : Exists A B -> Type`.

When elimination of a Prop-inductive is restricted to Prop, we say that it is "squashed" to Prop. A squashed inductive is similar to its impredicative encoding, but it has dependent elimination (this is all justified through models of Coq).

For instance the impredicative encoding of `ex` is

```
Definition ex A B : Prop := forall P:Prop, (forall a, B a -> P) -> P.
```

Sometimes we may phrase the restriction on inductives in Prop in the inverse direction: all inductives are allowed in Prop, but only some may be eliminated to non-Prop types. These eliminable inductives are said to have "singleton elimination".

3.4.3 Emulated Inductives

Given the data specifying an inductive (i.e. the parameters, indices, universe, and constructor types), we may consider it emulated by a type family if that family has type $\Pi\, \Gamma_P\, \Gamma_I, \mathsf{Type}_u$, there exist terms which have the types of the constructors, a term which has the type of the eliminator and the expected reduction rules are verified.

Then any judgment using the inductive may be translated to a judgment using the type family instead (if the type theory has primitive fix-and-match, we first need to translate to eliminators).

This may be used to show that certain extensions to the rules of inductive types are conservative.

3.4.4 Extension: non recursively uniform parameters

Consider the following type:

```
Inductive Acc {A} (R : A -> A -> Prop) (x : A) : Prop :=
  Acc_in : (forall y, R y x -> Acc R y) -> Acc R x.
```

We call `Acc` the "accessibility predicate", it is inhabited for `x` when there is no infinite decreasing (according to `R`) chain starting from `x`. It may be used to express strong induction.

`x` is introduced like a parameter, but in the recursive argument of `Acc_in`, `Acc` is applied to a different value `y`. By the rules as we have given them this is not allowed.

We could try interpreting `x` as an index, such that the above code would be syntactic sugar for

```
Inductive Acc {A} (R : A -> A -> Prop) : A -> Prop :=
  Acc_in : forall x, (forall y, R y x -> Acc R y) -> Acc R x.
```

However we now have an argument `x : A : Type` to the constructor, so `Acc` must be squashed.

The type theories implemented in Coq, Adga and Lean allow such non recursively uniform parameters, with the same universe-related rules and match typing rule as ordinary parameters. The difference can be relevant in subtle ways with yet-to-be-introduced nested inductives.

Note: Adga has no universe Prop (it has a universe called Prop but that is actually this thesis's SProp which doesn't permit `Acc`). As such `Acc` must live in some predicative universe, then the argument `y` to the recursive argument of `Acc_in` forces it to live at least in the universe of `A`. As such Agda's use of non recursively uniform parameters has nothing to do with `Acc`.

Note: We could have included this as part of the definition of inductive families, but then the elimination operator needs to vary across some of the parameters so it's easier to present as an extension.

3.4.5 Extension: Mutual and Nested Inductives

Mutual Inductives

We may define multiple inductives in parallel. For instance:

```
Inductive even : nat -> Prop :=
| even_0 : even 0
| even_S : forall n, odd n -> even (S n)

with odd : nat -> Prop :=
| odd_S : forall n, even n -> odd (S n).
```

Where in a single inductive the recursive arguments must produce a value from that inductive, with mutual inductives the recursive arguments of a constructor for one of the inductives being defined may produce values from another inductive in the block. They must share the same parameters.

To be fully equivalent to the fix-and-match presentation (which needs no change beyond recognising which arguments are recursive), elimination operators need to eliminate both inductives at the same time. This mutual elimination is not automatically defined by Coq, but it can be generated on demand:

```
Scheme even_elim := Induction for even Sort Prop
  with odd_elim := Induction for odd Sort Prop.


    even_elim
         : forall (P : forall n : nat, even n -> Prop)
             (Q : forall n : nat, odd n -> Prop),
           P 0 even_0 ->
           (forall (n : nat) (o : odd n),
            Q n o -> P (S n) (even_S n o)) ->
           (forall (n : nat) (e : even n),
            P n e -> Q (S n) (odd_S n e)) ->
           forall (n : nat) (e : even n), P n e
```

A block of mutual inductives may be encoded as a single inductive by modifying the indices. For our example:

```
Inductive even_odd : Sum nat nat -> Prop :=
| even_0 : even_odd (inl 0)
| even_S : forall n, even_odd (inr n) -> even_odd (inl n)
| odd_S : forall n, even_odd (inl n) -> even_odd (inr n).

Definition even n := even_odd (inl n).
Definition odd n := even_odd (inr n).
```

In general, the idea is to first pack up all indices of each inductive into a `Sigma` (for our example both inductives have only 1 index so there is nothing to do), then combine the indices from each inductive using `Sum`.

The combined inductive's universe must be the maximum of the universes of the original inductives. Assuming they are truly mutually dependent, i.e. each has a constructor

using another, the universes must actually all be the same across the block, so this is not a problem.

The above argument does not apply if one of the inductives is squashed. This is more complicated and the reduction from mutual to single inductives is not of particular interest to this thesis so let's move on.

Nested Inductives

We may further generalize to allow recursive arguments returning previously defined inductives instantiated by the one being declared:

```
Inductive list (A:Type) := nil | cons : A -> list A -> list A.
Inductive tree := node : list tree -> tree.
```

We need that the instantiated constructor types of the inner inductive are positive in the outer inductive.

Such nested inductives may be translated to mutual inductives (and from there to single inductives):

```
Inductive tree := node0 : list_tree -> tree
with list_tree :=
| nil_tree
| cons_tree : tree -> list_tree -> list_tree.

Fixpoint relist_tree (l:list tree) : list_tree :=
  match l with
  | nil => nil_tree
  | cons a tl => cons_tree a (relist_tree tl)
  end.

Definition node (l:list tree) : tree := node0 (relist_tree l).
```

(the `Fixpoint` command is a convenient way to make a `Definition` whose body is a `fix`)

Defining the translated eliminator and showing that it has the expected reduction rules is left as an exercise to the reader.

3.4.6 Extension: Records with Eta Expansion

As we have seen with *Sigma types* (page 41), if an inductive has 1 constructor, no indices and is not squashed we can define projections to extract each argument of the constructor.

Such types are called records. They are amenable to an alternative presentation based around "fields" instead of constructors, where each field is an argument of the unique constructor. For instance:

```
Record Sigma A B := sigma { pr1 : A; pr2 : B pr1 }.
```

We can show that every element of such a record is equal to the constructor applied to the projections:

```
Definition Sigma_eta A B (x:Sigma A B) : x = sigma (pr1 x) (pr2 x)
:= match x with sigma a b => eq_refl end.
```

When the record is also *not* recursive, we may turn this equality into a conversion rule. In Coq this must be activated with `Set Primitive Projections.`.

The name of the option stems from the fact that eta conversion uses a primitive representation of projection terms instead of allowing matches on such "primitive records". If we had such matches, we would have

```
  match x with sigma a b => c a b end
= match (sigma (pr1 x) (pr2 x)) with sigma a b => c a b end
= c (pr1 x) (pr2 x)
```

requiring special handling. Since the combination of primitive projections and definitional eta is sufficient to translate matches (to `let a := pr1 x in let b := pr2 x in c a b`) we require that they are translated as syntactic sugar.

3.4.7 Extension: Induction-Induction, Induction-Recursion

Induction-induction is a generalization of mutual induction, where one inductive of the block may use another as an index, and may use the constructors of the other in its constructor types.

Induction-recursion allows to define a fixpoint on an inductive at the same time that we define it, and so allows using that fixpoint in the constructor's type (applying it to the recursive arguments).

These are studied in depth in Forsberg's thesis [For14].

A typical use is to define type theories inside type theory, such that no untyped term may ever appear. For this, we need to simultaneously define well-formed contexts, types in a context, and terms of a certain type in a context:

```
Inductive context :=
| empty : context
| cons : forall Γ, type Γ -> context

with type : context -> Type :=
| Pi : forall Γ (A:type Γ), type (cons Γ A) -> type Γ
| ...

with term : forall Γ, type Γ -> Type :=
| lambda : forall Γ (A:type Γ) (B:type (cons Γ A)),
           term (cons Γ A) B -> term Γ (Pi Γ A B)
| ...
```

The above code sample shows the use of induction-induction. Induction-recursion is also necessary: to define the conversion rule we need to define the conversion judgment.

Conversion, through the beta reduction rule, involves substitution. Substitution is defined recursively on `term`.

Both induction-induction and induction-recursion are stronger than induction alone.

3.4.8 Metatheory of Inductive Families

The usual properties are preserved when adding inductive types.

The inversions for inductive types bring no new concepts. For instance, inductive types and constructors are injective for conversion.

Normalisation for elements of an inductive family is interesting: they are normal when they reduce to neutral terms, or when they reduce to a fully applied constructor whose arguments are normal.

With *natural numbers* (page 40), this means elements of `nat` reduce to a stack of applications of the successor `S`, then either a neutral term or `0`. In the empty context, there are no neutral terms so natural numbers must reduce to $S^n\ 0$ for n a concrete natural number from the metatheory.

For booleans, in the empty context they reduce to either `true` or `false`.

For elements of an identity type $x = y$, in the empty context they reduce to some `eq_refl` (implicitly applied to some type and `z`), then by inversion $x \equiv z \equiv y$.

Partially applied inductives, constructors and fixpoints may be considered as their eta expansion when defining weak head normal forms. In some formulations of type theory instead of having a primitive unapplied `Ind`, the primitive term former is the fully applied inductive (and so on for constructors etc) so the unapplied form never occurs.

Fully applied inductives and constructors are introduction forms. Matches, fully applied fixpoints and fully applied elimination operators are elimination forms with the head being respectively the discriminee for matches, and the principal argument (whose type is the inductive) for fixpoints and elimination operators.

3.5 Equality

With inductive families, we have 3 different concepts of equality between two terms t and u:

- "syntactic" or "structural" equality, when the terms are equal up to equality in the metatheory.
- conversion, also called "definitional" or "extensional" equality, when the terms are indistinguishable inside the type theory.
- "propositional" or "intensional" equality, when there is a term of type $t = u$

Each is included in the next.

The following axioms can make it easier to work with dependent types (including equality itself):

3.5.1 Function Extensionality

Function extensionality means that if $\Pi(x : A), f \circ x = g \circ x$ then $f = g$. Even though this looks closely related to the definitional eta rule, it requires an axiom to prove.

Otherwise, we could have a proof that λ `(x:bool), x = If bool true false`, but by canonicity and inversion in the empty context such a proof would imply that these 2 implementations of identity for booleans are convertible, which they are not.

Function extensionality makes functions less opaque, such that `W bool (If Empty unit)` (from *W types* (page 41)) is fully equivalent to `nat`.

3.5.2 Uniqueness of Identity Proofs

i.e. $\Pi(A : \mathsf{Type})(x\ y : A)(p\ q : x = y), p = q$.

This is equivalent to Streicher's axiom K: $\Pi(A : \mathsf{Type})(x : A)(p : x = x), p =$ `eq_refl`.

Without UIP, some intuitive results such as injectivity of the `sigma` constructor are not available: we only have `sigma a b = sigma a b'` $\rightarrow$ `b = b'` when the type of `a` validates UIP.

Note that all types with decidable equality (such as `bool` and `nat`) validate UIP by Hedberg's lemma [Hed98].

3.5.3 Univalence

Univalence means that equivalent types are equal. Here, equivalence is a refinement of the concept of bijections (and is logically equivalent to bijections, i.e. there is an implication in both directions but composing the implications does not produce the identity function). It is studied in-depth in [UFP13].

Univalence implies function extensionality, and is inconsistent with uniqueness of identity proofs: under univalence, the types of equalities between types are equivalent to the types of equivalences between types. For instance, $bool = bool$ is equivalent to $bool \cong bool$ which has 2 elements (one corresponding to the identity function and one to negation).

This means that when we introduce definitionally proof irrelevant types, if we want to be compatible with univalence we must be careful to avoid allowing proof irrelevant equality types.

3.5.4 Extensional Type Theory

In mainstream mathematical practice, propositional and definitional equality are not distinguished: when two values are equal they may be substituted for one another without

some intervening `match` on the equality. However in type theory we cannot assume or prove definitional equalities, we can only do so with propositional ones.

Extensional type theory (ETT) bridges the gap by adding the "equality reflection" rule:

Equality-Reflection

$$\frac{\Gamma \vdash p : t =_A u}{\Gamma \vdash t \equiv u : A}$$

In other words propositional equality "reflects" the conversion inside the language of the type theory.

The usual type theory without equality reflection is called "intensional" type theory (ITT).

Equality reflection makes type theory undecidable: for instance in order to be able to conclude that variable x is not convertible to variable y, we need to check that context is consistent. Otherwise, all types including $x = y$ would be inhabited.

Equality reflection proves both function extensionality and uniqueness of identity proofs. Function extensionality is trivial by going through eta equality.

We prove UIP by going through axiom K: let $p : a =_A a$. Then

```
match p as px in eq _ _ x return px = eq_refl with
eq_refl => eq_refl
end : p = eq_refl
```

This fails in intensional type theory because $x : A, px : a = x \nvdash px = eq\_refl$: the type of px is not the type of any eq_refl. With equality reflection however since we have the assumption $px : a = x$ everything works out.

ETT is conservative over ITT with UIP and function extensionality axioms by a model argument [Hof97].

ETT can be translated to ITT with UIP, function extensionality and congruence of application for the heterogeneous equality [Our05]. (The last axiom is because in some context we may have $nat \to bool = nat \to nat$, then $(\lambda(x : nat), x) \circ 0 \equiv 0 : bool$. In other words it's an issue with beta reduction and Π-injectivity being unprovable inside the theory.)

With some changes it can be translated to ITT with UIP and function extensionality [WST19]. Specifically, we need to annotate lambda abstractions with their codomain (written $\lambda(x : A).B, e$) and applications with the domain and codomain of the head (written $f \circ_{(x:A).B} v$). Then we block beta reduction unless the indices match, i.e. the *Beta-conv* (page 25) rule becomes

Beta-conv-ett

$$\frac{\Gamma, x : A \vdash e : B \qquad \Gamma \vdash v : A}{\Gamma \vdash (\lambda(x : A).B, e) \circ_{(x:A).B} v \equiv e[x := v] : B[x := v]}$$

3.6 Other Types

There are other types which we may want to add to dependent type theory:

- quotient inductive types (QITs) ([AK16], [ACD+18] for the combination with induction-induction), as the name indicates they formalize the notion of quotienting by some relation.
- higher inductive types (HITs) ([UFP13]), from homotopy type theory. These are incompatible with UIP: it is possible to define a HIT `Circle` with a value `point : Circle` such that `point = point` is equivalent to integers.
- Coinductive types ([Gim95]), which allow to formalize things like infinite streams or non-termination.
- more exotic types which do not look like deformations of the inductive type concept, such as interval spaces from cubical type theory, etc.

We don't study these with any depth, but their addition and the addition of proof irrelevance seem relatively orthogonal, at least for decidability.

CHAPTER

4

DEPENDENT TYPE THEORY WITH DEFINITIONAL PROOF IRRELEVANCE

4.1 Adding Proof Irrelevance

When adding a universe SProp for types with irrelevant proofs, we face a choice: should it be impredicative (like the *Universe of Propositions* (page 35))?

If it is impredicative, we have a unique small universe SProp:

SProp-type

$$\frac{\vdash \Gamma \qquad 0 < u}{\Gamma \vdash \mathsf{SProp} : \mathsf{Type}_u}$$

SProp-impred

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash B : \mathsf{SProp}}{\Gamma \vdash \Pi(x : A), B : \mathsf{SProp}}$$

The formation rules for lambda abstractions and applications are unchanged from *Lambda* (page 23) and *App* (page 24) even when they involve irrelevant types.

Then we add the proof irrelevance rule:

SProp-irrelevant

$$\frac{\Gamma \vdash A : \mathsf{SProp} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t \equiv u : A}$$

Impredicative SProp is used by Coq and by Lean (with Lean calling it Prop).

If we don't want impredicativity (which is how it works in Agda), we cannot have for instance $\vdash \Pi(A : \mathsf{SProp}), A : \mathsf{SProp}$. Instead we have a SProp_i for every universe level i just as we have Type_i. Then we have

SProp-type-pred

$$\frac{\vdash \Gamma \qquad u < v}{\Gamma \vdash \mathsf{SProp}_u : \mathsf{Type}_v}$$

SProp-pred

$$\frac{\Gamma \vdash A : \mathsf{Type}_u \qquad \Gamma, x : A \vdash B : \mathsf{SProp}_v}{\Gamma \vdash \Pi(x : A), B : \mathsf{SProp}_{max(u,v)}}$$

SProp-irrelevance-pred

$$\frac{\Gamma \vdash A : \mathsf{SProp}_u \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a \equiv b : A}$$

Since this thesis focuses on Coq, we focus on the case where SProp is impredicative then present the predicative case as an alternative.

4.1.1 Cumulativity and SProp

In our implementation of Coq, we forbid cumulativity between SProp and the relevant universes: $\not\vdash \mathsf{SProp} \leq \mathsf{Type}_u$.

Cumulativity of SProp is consistent, and it seems decidable: when comparing two terms, we would ask whether the principal type of their principal type reduces to SProp (although we have no full proof that this works).

The reason we forbid it is that it is not efficiently decided: we have to infer types and do potentially costly reductions at every subterm of a conversion problem.

Note: We would also need to do retyping to work with *Records with Eta Expansion* (page 45): in the following context

```
Record Wrap (A:Type) : Type := wrap { unwrap : A }.
Axioms (A:SProp) (x y : Wrap A).
```

the type `Wrap A` is in whatever relevant universe was inferred for `Wrap`, so `x` and `y` are not directly convertible by proof irrelevance. However we have `x ≡ wrap (unwrap x) ≡ wrap (unwrap y) ≡ y` by eta expansion, then proof irrelevance on the `unwrap` subterms. Therefore to decide such a conversion we would need to identify the type as a record, perform the eta expansion and somehow avoid infinitely recursing when `A` is a relevant type (as mentioned it's not certain that this can be made to work).

Once we have forbidden cumulativity, things are much simpler. The essential property is that being a proof irrelevant term is stable by well-typed substitution. The details are further explained in *the Kernel Side section of the implementation chapter* (page 88).

Agda and Lean forbid cumulativity for all universes, so their developers never had to question whether SProp should have it.

4.1.2 SProp related primitives

Empty

The quintessential irrelevant type, we should have $\mathsf{Empty} \to A$ for all A.

The empty type brings power to the theory because its elements can be eliminated to produce a term in any other type, including relevant types. Without this relevant eliminations, we could interpret SProp as a universe containing only the singleton type.

From Empty we can build $\mathsf{Unit} := \mathsf{Empty} \to \mathsf{Empty}$.

Using large elimination, we can build an irrelevant type for any decidable property. Such a property is associated to a boolean b, so the associated type is just

$$\text{if } b \text{ then } \mathsf{Unit} \text{ else } \mathsf{Empty}.$$

See *The Translation* (page 74) for the full generality of what types can be produced from Empty.

Box

The box type, lifting an irrelevant type to the relevant world. It emulates (in the sense of *Emulated Inductives* (page 42)) the following inductive type with unrestricted eliminations:

```
Inductive Box (A:SProp) : Type := box : A -> Box A.
    Box is defined
    Box_rect is defined
    Box_ind is defined
    Box_rec is defined
    Box_sind is defined

Check Box_rect.
    Box_rect
         : forall (A : SProp) (P : Box A -> Type),
           (forall a : A, P (box A a)) ->
           forall b : Box A, P b
```

A primitive Box is equivalent to allowing constructors of inductive types to have irrelevant arguments: we can translate the later to constructors taking boxed arguments.

When we have a single impredicative SProp, Box lives in the lowest relevant universe (in Coq, this means Prop). When we have predicative SProp_i the box of a type lives in the relevant universe at the same predicative level.

Squash

Squashing a relevant type A is to produce an irrelevant approximation Squash A which emulates the following inductive type with elimination restricted to irrelevant predicates:

```
Inductive Squash (A:Type) : SProp := squash : A -> Squash A.


Check Squash_sind.
    Squash_sind
         : forall (A : Type) (P : Squash A -> SProp),
           (forall a : A, P (squash A a)) ->
           forall s : Squash A, P s
```

When SProp is impredicative, `Squash` does not need to be a primitive type:

```
Definition Squash (A:Type) : SProp := forall P : SProp, (A -> P) -> P.
Definition squash A (a:A) : Squash A := fun P f => f a.
Definition Squash_sind A (P : Squash A -> SProp)
  (f : forall a, P (squash A a)) (s:Squash A)
  : P s
  := s (P s) f.
```

(`f` has type `forall a : A, P (squash A a)` which by proof irrelevance is convertible to `A -> P s`)

The restriction on `Squash`'s eliminations is similar to the restriction in *Inductives and Prop* (page 42): we can allow any inductive type (still requiring the positivity condition) in SProp with eliminations restricted to SProp. From this perspective, SProp is similar to Prop but with a stricter condition for singleton elimination.

When SProp is predicative, the squash of a type lives at the same level as the original type, and it eliminates to any level of SProp, including higher ones. These unlimited eliminations require it to be primitive, unlike the impredicative SProp case.

Dependent Pairs

The type of irrelevant dependent pairs

$$A : \mathsf{SProp}, B : A \to \mathsf{SProp} \vdash \Sigma_s(x : A)\ (B\ x) : \mathsf{SProp}$$

may be implemented using the dependent pairs from the relevant world together with Box and Squash as

$$\mathsf{Squash}\ (\Sigma(x : \mathsf{Box}\ A)\ (\mathsf{Box}\ (B\ (\mathsf{unbox}\ x)))).$$

Eliminating the squash to define the projections works since their types are irrelevant. Note that this doesn't use proof irrelevance.

Since we have proof irrelevance, we have definitional η for irrelevant pairs.

Irrelevant pairs may also be defined impredicatively (when the theory allows impredicativity) as

$$\Pi(P : \mathsf{SProp}), (\Pi(x : A), B\ x \to P) \to P.$$

In this formulation proof irrelevance is necessary to get a well-typed second projection.

If for some reason we work in a theory without Box or Squash, dependent pairs in the relevant world are not sufficient to define irrelevant pairs, which must then be provided as a separate primitive.

4.2 Consistency with Proof Irrelevance

Consistency can be obtained as a corollary of the proof in *Decidability and Reducibility with Proof Irrelevance* (page 58), but this is not useful when we want to know which axioms are consistent in a type theory with SProp. For this, we need to come up with some models.

The work in the section has already been published in [GCST19].

4.2.1 Translation to Extensional Type Theory

Following the notion of syntactical translations advocated in [BPT17], we prove consistency of type theory with SProp by a type preserving translation into *Extensional Type Theory* (page 48).

The translation is described in Figure 1.

There must not be cumulativity between SProp and relevant universes: we need to know which Π types are irrelevant. Once we have forbidden cumulativity of SProp, we may introduce a syntactic difference between relevant and irrelevant product types, justified by *Uniqueness of Relevance* (page 66). In the proof of decidability the syntactic mark is about the domain of the products, whereas for the translation it is about the codomain, but the principle is the same.

Carneiro's thesis [Car19] also shows soundness with a pre-processing step (6.1 "Proof splitting") to introduce this syntactic distinction, but then goes directly to a model using ZFC instead of going through ETT.

If we want to handle cumulativity, we need a way to forget the proof of irrelevance when cumulativity is used. This has not been fully investigated.

The idea of the translation is to see an inhabitant of SProp as a pair of type in ETT together with a proof that it has proof irrelevance. Therefore, SProp is translated as the

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

Fig. 1: Syntactical Translation from type theory with SProp to ETT

following dependent sum:

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

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 have proof irrelevance. For instance, the fact that a dependent product whose codomain is irrelevant is itself irrelevant can be proven using functional extensionality in ETT. Identity types are proof irrelevant in ETT so we can also translate identity types in SProp.

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.

If we need to translate without the application congruence axiom we can also translate to ETT with annotated applications.

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

- **Preservation of conversion:**

  For every term t and u of type theory with SProp, if $\Gamma \vdash t \equiv u : A$ then $[\![\Gamma]\!] \vdash [t] \equiv [u]$ in ETT.

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

- **Correctness of the syntactical translation:**

  For every typing derivation of type theory with SProp $\Gamma \vdash M : A$, we have the corresponding derivation $[\![\Gamma]\!] \vdash [M] : [\![A]\!]$ in ETT.

  > By induction on the typing derivation.

Note: If we have impredicative SProp, we use an extensional type theory with impredicative Prop, and use $[\mathsf{SProp}] := \Sigma\, A : \mathsf{Prop}.\ \Pi\, x\ y : A.\ x = y$.

This gives us consistency of type theory with SProp and classical axioms such as function extensionality, UIP, excluded middle or choice.

4.2.2 Univalent Model

We can modify the translation for type theory with SProp and *no UIP* to target the Homotopy Type System (HTS) [Voe13] of Voevodsky.

In HTS, types are divided between pretypes and "fibrant" types. Fibrancy is preserved by the usual constructions (product types, inductives types, etc). However elimination of fibrant inductives is restricted to fibrant types.

There are 2 identity types: the strict equality, a pretype which satisfies the reflection rule of extensional type theory (as such we represent it with $\equiv$), and the fibrant equality $=$, which is the usual inductive.

The separation between strict and fibrant equality allows us to have equality reflection for the strict equality without implying uniqueness of proofs of fibrant equality: the usual proof of UIP from reflection breaks down since we can't eliminate fibrant equalities to produce strict ones.

Fibrant equality satisfies the univalence axiom: 2 fibrant types are fibrantly equal if there is an equivalence between them.

When translating to HTS we have a problem: we want to translate types to fibrant types in order to be able to translate the univalence axiom to its implementation in HTS, but we also want to translate SProp to $\Sigma(A : \mathsf{Prop})\Pi(x\ y : A), x \equiv y$ (where Prop comes from the propositional resizing rule, for impredicativity). By the rules we have given this is not a fibrant type.

For this we rely on work by Thierry Coquand [Coq16] showing that a universe of strict propositions can be built in the cubical model. This then allows us to extend HTS by such a universe which we translate SProp to.

The translation then shows that SProp is compatible with univalence.

4.3 Decidability and Reducibility with Proof Irrelevance

We only prove decidability for a reduced type theory in order to keep the proof at a reasonable size. See *Limitations* (page 70) for more details.

This is formalized in Agda, with source available at https://github.com/SkySkimmer/logrel-tt commit `0cdfbc81e4f3a326e42b92f2c7826ce4953a3e1a`.

4.3.1 Idea of the proof

We base our proof on the proof by Abel et al. [AOV17]. Let us first explain enough of the original proof to understand what we need to change.

The proof applies to a dependent type theory with a universe of small types U and a small type of natural numbers $\mathbb{N} : U$ with large elimination. See *Limitations* (page 70) for discussion of the power of this theory.

Terms and types are in the same syntactic domain. The term formers are

- Product types $\Pi A \rhd B$. Separating the domain and codomain using $\rhd$ instead of a comma distinguishes the De Bruijn index syntax from the named variable syntax.

- Lambda abstractions $\lambda\, t$. The original work by Abel et al. uses lambda abstractions without a type annotation for the domain, losing unique typing. We follow this choice in this thesis.

  Note that adding the type annotations is trivial, touching about 25 lines out of 12000 in the Agda formalization.

- Applications $t \circ u$.

- Variables i_k. We use De Bruijn indices, written as the subscript k, replacing alpha renaming issues with explicit lifting. We may omit the subscript when its specific value is not interesting: for instance we may talk about the newest bound variable i_0, but when we say that a variable is bound to a type in a context we only say $i : A \in \Gamma$.

- The universe of small types U.

- Term formers for natural numbers: the type of natural numbers $\mathbb{N}$, the zero 0 and successor $S\ t$ constructors, and the eliminator $\mathrm{natrec}\, F\ z\ s\ n$ (where a variable of type $\mathbb{N}$ is bound in F).

  Note that the successor and eliminator always appear fully applied. Naturals have large elimination, i.e. F may be a large type such as U.

Neutral terms and weak head normal forms are simple syntactic properties.

Since we have a finite hierarchy of universes, not all types have a type and we need separate judgments for types and terms. As such, the type theory is defined by mutually defined inductive types:

- **Well-formed context:** $\vdash \Gamma$
- **Well-formed type:** $\Gamma \vdash A$
- **Typing:** $\Gamma \vdash t : A$
- **Conversion for types:** $\Gamma \vdash A \equiv B$
- **Conversion for terms:** $\Gamma \vdash t \equiv u : A$

Once we have the typing judgment we can also define well-typed weak head reduction for terms $\Gamma \vdash t \Rightarrow u : A$ and for types $\Gamma \vdash A \Rightarrow B$. It is trivially a subset of conversion. Since we only reduce at the head, it is deterministic and we dodge confluence issues (confluence can be seen semantically as a corollary of some of the lemmas we prove, but is never explicitly stated or proved).

Both conversion judgments are generated by reflexivity, symmetry, transitivity, congruence rules for each term former, eta equality at function types, beta reduction and reduction for natrec. Because of the power of transitivity, we cannot directly prove inversion lemmas (as defined in *Inversion of Judgments* (page 27)) or decidability.

Instead, we do through alternative characterizations of conversion based on reduction:

- **Algorithmic equality** between terms $\Gamma \vdash t \iff u : A$ and between types $\Gamma \vdash A \iff B$. These are mutually defined inductive types which can be seen as the trace

of the conversion decision algorithm: first reduce to weak head normal forms, then apply congruence rules and continue recursively on subterms.

We need to treat neutral terms specially when deciding conversion, so there is also a judgment for algorithmic equality of neutral terms $\Gamma \vdash k \mathrel{\hat{\Longleftrightarrow}} l : A$. There is no type version: large types are never neutral, so equality between neutral terms is enough to account for equality between neutral types.

Normally we compare terms at a unique type, but neutral terms break this invariant: even if we start by comparing two neutral terms at the same type $t \circ u : T$ and $t' \circ u' : T$, the heads must be compared at possibly different types. For instance if f and P are variables such that $f : \Pi(x\ y : \mathbb{N}), P\ (x + y)$ (where $+$ is defined using natrec and therefore computes), the terms $f\ 0\ 1$ and $f\ 1\ 0$ both have type $P\ 1$ even though the heads have different types.

However, the base case for variables does not need type information and if neutral terms are equal then their types must also be equal (even without domain annotations for lambda abstractions neutral terms still have unique typing). Therefore for neutral terms we can decide whether they are convertible at a unique (up to conversion) common type or not convertible at any type.

The original work by Abel et al. uses $\Gamma \vdash t \Longleftrightarrow u : A$ in the paper typesetting and `Γ ⊢ t [conv↑] u : A` in Agda code, we follow this choice.

- **Generic equality** between terms $\Gamma \vdash t \cong u : A$ and between types $\Gamma \vdash A \cong B$. We want to make definitions and proofs about both conversion and algorithmic equality, so we define *generic equalities* which may be instantiated by either.

  Since we want to instantiate them by algorithmic equality generic equalities also have a form for neutral terms $\Gamma \vdash k \sim l : A$. When instantiating generic equality with conversion both forms are instantiated by the conversion judgments.

  Generic equalities must be sound with regards to conversion, satisfy congruence rules (but the congruence rules for the eliminators are restricted to the neutral equality), be symmetric and transitive, be compatible with each other (for instance neutral equality must imply unqualified equality), be compatible with weakening (if two terms or types are equal so is their weakening) and reduction (if two terms or types reduce to equal weak head normal forms then they are equal), and the unqualified equality must satisfy eta equality of functions.

  Note that the congruence rules for term formers without subterms such as 0 amount to reflexivity at that term in well-formed contexts.

- **The logical relation** between terms $\Gamma \Vdash_\ell t \equiv u : A/\mathcal{A}$ and between types $\Gamma \Vdash_\ell A \equiv B/\mathcal{A}$ (which may also be read respectively as "t is reducibly equal to u and A is reducibly equal to B).

  It is defined in successive layers for each type level ℓ (in our case small then large types), with each layer depending on the previous, as part of an inductive-inductive-recursive block. The other components of the block are

  – Reducible types $\Gamma \Vdash_\ell A$ read "A is a reducible type". This is the inductive part of the block, the other components are recursively defined using an argument

$\mathcal{A}$ of this type.

– Reducible terms at a given type $\Gamma \Vdash_\ell t : A/\mathcal{A}$.

Quickly after its definition we prove the *irrelevance lemma*: reducible terms, type equality and term equality at different proofs $\mathcal{A}$ and $\mathcal{A}'$ of $\Gamma \Vdash_\ell A$ are logically equivalent. Additionally we implicitly do existential quantification by the type level.

This justifies omitting the reducible type argument and the type level from our statements: for instance we will talk about reducible equality between terms $\Gamma \Vdash t \equiv u : A$. In the formalization, the existential quantification is explicit and we do not omit the reducible type argument as the proofs are done by induction on it and on the type level.

We have a contructor of reducible type for each type former:

– The universe is a reducible type in well-formed contexts when we are at the level of large types.

– Types which reduce to $\mathbb{N}$ are reducible at any level.

– Types which reduce to any neutral element of the universe which is reflexive for the neutral equality are reducible at any level.

– Small reducible types are also large reducible types.

– Types which reduce to product types are reducible if the domain and codomain are reducible (all at a given level), and some side conditions are verified.

  The side conditions are complex and do not change when adding proof irrelevance so we do not detail them.

  Note that since the domain must be at the same level as the whole type, this proof method is incompatible with impredicativity.

The other components are defined by cases, also based on reduction. For instance:

– A term is reducible at the universe if it is a reducible small type. Because the universe is a reducible large type this is not circular.

– Two terms are reducibly equal at a type which reduces to $\mathbb{N}$ if, inductively, either they reduce to both 0, to successors applied to reducibly equal natural number terms, or to neutral terms equated by the generic equality.

This gives inversion lemmas for reducible equality by definition. For instance injectivity of constructors: from a proof that $\Gamma \Vdash_\ell S\ m \equiv S\ n : \mathbb{N}/[\mathbb{N}]$ for some $[\mathbb{N}] : \Gamma \Vdash_\ell \mathbb{N}$, by the irrelevance lemma we can replace $[\mathbb{N}]$ by the appropriate constructor, then reduce to get a proof of $\Gamma \Vdash_\ell m \equiv n : \mathbb{N}$ (this sketch skips over some proofs of well-formedness of context and the like, which pose no difficulty).

The algorithmic equality is designed to make it easy to show decidability of $\Gamma \vdash t \iff u : A$ given hypotheses $\Gamma \vdash t \iff t : A$ and $\Gamma \vdash u \iff u : A$: the reflexivity hypotheses (which after our discussion of neutral terms you may notice are at the same type) provide structure for induction.

We then need to show soundness and completeness (from completeness and a typing derivation $\Gamma \vdash t : A$ we can get $\Gamma \vdash t \iff t : A$ to use as argument for the decidability proof).

Soundness is relatively trivial (the constructors of algorithmic equality are close to the rules of the type theory combined with reduction), except for some subtlety regarding the treatment of neutral terms which requires inversion lemmas for the typing judgments. Completeness is more difficult.

We use the logical relation to go between typing judgments and algorithmic equality:

- The *escape lemma* shows that reducible types are well-formed types, reducible terms are well-typed, and reducible equality implies the generic equality.
- The *fundamental theorem* shows that the typing judgments imply the corresponding logical relation judgment, for instance conversion implies reducible equality.

  The fundamental theorem is proved using validity judgments $\Gamma \Vdash^{v}_{\ell} t : A/\mathcal{G}/\mathcal{A}$ (and similar for types and both equalities) derived from the logical relation. Essentially the validity judgment is needed for closure by substitution of the logical relation.

Instantiating the generic equality by conversion (which trivially satisfies the expected properties), we get inversion lemmas and reducibility for the typing judgments.

Using these inversion lemmas, we can show soundness and the other generic equality properties for algorithmic equality. Then by the escape lemma and fundamental theorem, algorithmic equality is logically equivalent to conversion.

Together with decidability of algorithmic equality on its reflexive domain, we obtain our desired decidability of conversion.

4.3.2 Adding Proof Irrelevance

We are now ready to add proof irrelevance and show that it does not affect reducibility and decidability of conversion.

This at first glance means adding a universe of small strict propositions Prop and a constructor to the conversion rule:

Prop-Irr

$$\frac{\Gamma \vdash A : \mathsf{Prop} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a \equiv b : A}$$

However we need a predicative system and we only have 1 level of universes, such that $\Pi A : \mathsf{Prop}, A \to A$ is an irrelevant type which itself has no type.

Instead we mark term and type judgments with the relevance of the type: well-formed type $\Gamma \vdash A \frown r$, well-typed term $\Gamma \vdash e : A \frown r$, conversion for types $\Gamma \vdash A \equiv B \frown r$ and for terms $\Gamma \vdash a \equiv b : A \frown r$.

The meta-variable r denotes the relevance, either ! for relevant or $\div$ for irrelevant.

Note: In Agda code we use the slightly different `%` for irrelevance, for better rendering between monospace and math fonts.

We also need to annotate context elements and domains of product types, in order to provide sufficient information to our proofs (such as the irrelevance lemma) before we have shown *Uniqueness of Relevance* (page 66).

For most of the typing rules, adding relevance marks is straightforward. For instance

Pi-Ty

$$\frac{\Gamma \vdash F \frown r_F \qquad \Gamma, F \vdash G \frown r_G}{\Gamma \vdash \Pi F \frown r_F \rhd G \frown r_G}$$

Context-Extend

$$\frac{\Gamma \vdash \qquad \Gamma \vdash A \frown r}{\Gamma, A \frown r \vdash}$$

Var

$$\frac{\Gamma \vdash \qquad i : A \frown r \in \Gamma}{\Gamma \vdash i : A}$$

Of particular interest are the rules for forming a type from a term whose type is a universe:

U-Ty

$$\frac{\Gamma \vdash A : U \frown !}{\Gamma \vdash A \frown !}$$

Prop-Ty

$$\frac{\Gamma \vdash A : \mathsf{Prop} \frown !}{\Gamma \vdash A \frown \div}$$

Note: The relevance is always ! in the premise as types are always relevant terms (i.e. U and Prop are relevant types). A small amount of experimentation indicates that this works better than leaving it free and later proving that is always !, at least when using Agda `--without-K`.

Note: We have 1 universe per relevance: $U = \mathsf{Univ}_!$ and $\mathsf{Prop} = \mathsf{Univ}_\div$.

The irrelevance rule with marks is then

Irrelevance

$$\frac{\Gamma \vdash a : A \frown \div \qquad \Gamma \vdash b : A \frown \div}{\Gamma \vdash a = b : A \frown \div}$$

Well-typed reduction also needs relevance marks, for instance reduction of types is now $\Gamma \vdash A \longrightarrow B \frown r$, such that we can prove it implies $\Gamma \vdash A \equiv B \frown r$ without additional hypotheses.

As a bonus, we may add an empty small irrelevant type $\mathsf{Empty} : \mathsf{Prop}$. Notice that all terms in the empty type must reduce to neutral terms.

4.3.3 Marking the Logical Relation

We still mark everything where a type appears, eg the logical relation is $\Gamma \Vdash_\ell t = u : A \frown r$.

Generic equalities must satisfy an additional property to provide proof irrelevance. Specifically, the equality between neutral terms must relate any two terms within its domain and which have the same irrelevant type.

GenEq-irrelevance

$$\frac{\Gamma \vdash k : A \frown \div \qquad \Gamma \vdash l : A \frown \div \qquad \Gamma \vdash k \sim k : A \frown \div \qquad \Gamma \vdash l \sim l : A \frown \div}{\Gamma \vdash k \sim l : A \frown \div}$$

We do not require an analogous property for the generic equality between arbitrary terms $\_ \vdash \_ \cong \_ : \_ \frown \_$, see *Proof irrelevance of the logical relation* (page 65) for details.

Adapting the definition of the logical relation is mechanical work, let's consider a few cases:

- universe: both universes are relevant large reducible types. The relevance is important for the reducible member of a universe: they must be reducible small types of the correct relevance. In other words the second component of $\Gamma \Vdash_\ell t : Univ_r \frown !/\mathcal{G}$ is $\Gamma \Vdash_{l'} t \frown r$, and analoguously for reducibly equal terms.
- neutrals: the formation rule becomes

  Neutral-Reducible

  $$\frac{\Gamma \vdash A :\longrightarrow^* : N \frown r \qquad \Gamma \vdash N \sim N : Univ_r \frown !}{\Gamma \Vdash_\ell A \frown r}$$

  Once again the relevance and the universe need to be coordinated.

  The sub cases are straightforward to adapt, just add the relevance.

  In the irrelevant case, we still have components $n \sim n$ (resp. $n \sim m$) in the definition of reducible terms (resp. reducibly equal terms). It is the neutral equality which should handle irrelevance.

Most properties (reflexivity, weakening, etc) are trivial to adapt.

Note: We need the relevance marks on domains of product types to get the irrelevance lemma.

When both derivations of $\Gamma \Vdash_\ell A \frown r$ are from the product type case, we know that A reduces to some $\Pi F \frown r_F \rhd G$ from one derivation, and to some $\Pi F' \frown r'_F \rhd G'$ from the

other, along with associated properties such as $\Gamma, F \frown r_F \Vdash_\ell G \frown r$. Since reduction is deterministic we know that the reducts are equal syntactically

$$\Pi F \frown r_F \rhd G = \Pi F' \frown r'_F \rhd G',$$

and so the relevances are also equal.

If we didn't include the mark inside the terms, we would still need to know that some mark exists and is used in the sub-properties, but we would not be able to prove that the one used in $\mathcal{G}$ is the same which is used in $\mathcal{G}'$.

Adapting the rest of the proofs to add relevance marks is essentially a matter of fixing the statements, with no interesting insights. There are then 3 interesting parts: dealing with proof irrelevance in the fundamental theorem (showing that the logical relation is inhabited from a judgment of our type theory), relevance unicity (a new result, analogous to unicity of typing), and dealing with proof irrelevance in the conversion algorithm.

4.3.4 Proof irrelevance of the logical relation

When proving the fundamental theorem, we have a new case where $\Gamma \vdash t \equiv u : A \frown \div$ is given by the proof irrelevance rule. We have sub-derivations $\Gamma \vdash t : A \frown \div$ and $\Gamma \vdash u : A \frown \div$, and by induction

- $\mathcal{G} :: \Vdash^v \Gamma$
- $\mathcal{A} :: \Gamma \Vdash^v_1 A \frown \div / \mathcal{G}$
- $\Gamma \Vdash^v_1 t : A \frown \div / \mathcal{G} / \mathcal{A}$
- $\mathcal{G}' :: \Vdash^v \Gamma$
- $\mathcal{A}' :: \Gamma \Vdash^v_1 A \frown \div / \mathcal{G}'$
- $\Gamma \Vdash^v_1 u : A \frown \div / \mathcal{G}' / \mathcal{A}'$

We seek to establish the following goals:

- $\mathcal{G}'' :: \Vdash^v \Gamma$
- $\mathcal{A}'' :: \Gamma \Vdash^v_1 A \frown \div / \mathcal{G}''$
- $\Gamma \Vdash^v_1 t : A \frown \div / \mathcal{G}'' / \mathcal{A}''$
- $\Gamma \Vdash^v_1 u : A \frown \div / \mathcal{G}'' / \mathcal{A}''$
- $\Gamma \Vdash^v_1 t = u : A \frown \div / \mathcal{G}'' / \mathcal{A}''$

By irrelevance of the validity judgment we have $\Gamma \Vdash^v_1 u : A \frown \div / \mathcal{G} / \mathcal{A}$, so the all but the equality are trivially available.

We therefore seek to establish a lemma of proof irrelevance for the valid equality: if $\Gamma \Vdash^v_\ell t : A \frown \div / \mathcal{G} / \mathcal{A}$ and $\Gamma \Vdash^v_\ell u : A \frown \div / \mathcal{G} / \mathcal{A}$ then $\Gamma \Vdash^v_\ell t = u : A \frown \div / \mathcal{G} / \mathcal{A}$.

This is trivial by definition of validity from proof irrelevance of the logical relation:

Proof-Irrelevance-Logical

$$\frac{\mathcal{A} :: \Gamma \Vdash_\ell A^\frown \div \qquad \Gamma \Vdash_\ell t : A^\frown \div / \mathcal{A} \qquad \Gamma \Vdash_\ell u : A^\frown \div / \mathcal{A}}{\Gamma \Vdash_\ell t \equiv u : A^\frown \div / \mathcal{A}}$$

This final lemma is proven by induction on $\mathcal{A}$

- A is neither reducibly a universe, or reducibly $\mathbb{N}$ as these are at the wrong relevance.
- if A is reducibly the empty type or a neutral type, we just need to use proof irrelevance for the generic equality on neutral terms.
- A is reducible as a type from a lower level than ℓ we recurse onwards.

This leaves reducible product types as the interesting case. We have $A \Rightarrow^* \Pi F^\frown r_F \rhd G$, reductions $t \Rightarrow f$ and $u \Rightarrow g$, and knowing that f and g under any well-typed weakening are reducible when applied to a reducible argument.

We need to establish that f and g are reducibly equal under any weakening when applied to a reducible argument. This is done by proof irrelevance, recursing on the proof that G is reducible.

Then we prove that f and g are related by the generic equality, using η of the generic equality, $f\ i_0$ and $g\ i_0$ being reducibly equal as we just showed.

4.3.5 Uniqueness of Relevance

We don't have *Uniqueness of Typing* (page 28) due to our un-annotated lambdas. For instance the identity term $\lambda\ i_0$ has type both $\mathbb{N} \to \mathbb{N}$ and $Univ_! \to Univ_!$.

We do have uniqueness of relevance:

Uniq-Relevance

$$\frac{\Gamma \vdash A^\frown r \qquad \Gamma \vdash A^\frown r'}{r = r'}$$

Note: Unicity of relevance is not used in the proof of decidability of conversion.

The proof is not especially interesting. One factor for this is that, due to the lack of large universes, being convertible to a universe is the same as being syntactically equal to it.

4.3.6 Algorithmic Equality and Proof Irrelevance

Definition of algorithmic equality

We need to modify the algorithmic equality on neutrals to have proof irrelevance, starting from

```
-- Neutral equality.
data _⊢_~_↑_ (Γ : Con Term) : (k l A : Term) → Set where
  var-refl    : ∀ {x y A}
              → Γ ⊢ var x ∷ A
              → x PE.≡ y
              → Γ ⊢ var x ~ var y ↑ A
  app-cong    : ∀ {k l t v F G}
              → Γ ⊢ k ~ l ↓ Π F ▹ G
              → Γ ⊢ t [conv↑] v ∷ F
              → Γ ⊢ k ∘ t ~ l ∘ v ↑ G [ t ]
  natrec-cong : ∀ {k l h g a₀ b₀ F G}
              → Γ • ℕ ⊢ F [conv↑] G
              → Γ ⊢ a₀ [conv↑] b₀ ∷ F [ zero ]
              → Γ ⊢ h [conv↑] g
                   ∷ Π ℕ ▹ (F ▹▹ F [ suc (var 0) ]↑)
              → Γ ⊢ k ~ l ↓ ℕ
              → Γ ⊢ natrec F a₀ h k ~ natrec G b₀ g l ↑ F [ k ]
  Emptyrec-cong : ∀ {k l F G}
                → Γ ⊢ F [conv↑] G
                → Γ ⊢ k ~ l ↓ Empty
                → Γ ⊢ Emptyrec F k ~ Emptyrec G l ↑ F
```

(where `Γ ⊢ t [conv↑] v ∷ F` is the non-neutral algorithmic equality, and `PE.≡` is syntactic equality)

We could just add relevance marks and a rule

```
  irrelevance : ∀ {k l A}
              → Neutral k
              → Neutral l
              → Γ ⊢ k ∷ A ^ %
              → Γ ⊢ l ∷ A ^ %
              → Γ ⊢ k ~ l ↑ A ^ %
```

However there are then 2 ways to prove equalities for some terms: by congruence and by irrelevance. This redundancy adds more cases to our proofs, especially when they involve 2 neutral equality premises (like transitivity or decidability). Instead, we separate neutral equality into its relevant part (inhabited by congruence as previously) and its irrelevant part (inhabited only by irrelevance). As such we get the following definition (where `↓%` is `↓` specialised to irrelevant terms):

```
-- Neutral equality.
data _⊢_~_↑!_ (Γ : Con Term) : (k l A : Term) → Set where
  var-refl : ∀ {x y A}
           → Γ ⊢ var x ∷ A ^ !
           → x PE.≡ y
           → Γ ⊢ var x ~ var y ↑! A
  app-cong : ∀ {k l t v F rF G}
           → Γ ⊢ k ~ l ↓! Π F ^ rF ▹ G
           → Γ ⊢ t [conv↑] v ∷ F ^ rF
           → Γ ⊢ k ∘ t ~ l ∘ v ↑! G [ t ]
  natrec-cong : ∀ {k l h g a₀ b₀ F G}
```

(continues on next page)

(continued from previous page)

```
              → Γ • ℕ ^ ! ⊢ F [conv↑] G ^ !
              → Γ ⊢ a₀ [conv↑] b₀ :: F [ zero ] ^ !
              → Γ ⊢ h [conv↑] g
                   :: Π ℕ ^ ! ▹ (F ^ ! ▹▹ F [ suc (var 0) ]↑) ^ !
              → Γ ⊢ k ~ l ↓! ℕ
              → Γ ⊢ natrec F a₀ h k ~ natrec G b₀ g l ↑! F [ k ]
  Emptyrec-cong : ∀ {k l F G}
                → Γ ⊢ F [conv↑] G ^ !
                → Γ ⊢ k ~ l ↓% Empty
                → Γ ⊢ Emptyrec F k ~ Emptyrec G l ↑! F

record _⊢_~_↑%_ (Γ : Con Term) (k l A : Term) : Set where
  inductive
  constructor %~↑
  field
    neK : Neutral k
    neL : Neutral l
    ⊢k : Γ ⊢ k :: A ^ %
    ⊢l : Γ ⊢ l :: A ^ %

data _⊢_~_↑_^_ (Γ : Con Term)
  : (k l A : Term) → Relevance → Set where
  ~↑! : ∀ {k l A} → Γ ⊢ k ~ l ↑! A → Γ ⊢ k ~ l ↑ A ^ !
  ~↑% : ∀ {k l A} → Γ ⊢ k ~ l ↑% A → Γ ⊢ k ~ l ↑ A ^ %
```

Note: It may work just as well to have a single type for both relevant and irrelevant neutral equality, separating the relevances only by restricting the congruence and variable constructors to relevant terms. However I didn't experiment with this alternative.

Note: Since we don't add an irrelevance rule for the non-neutral generic equality, the decision algorithm invoked on irrelevant terms inhabits the conversion by reducing the terms and η expanding functions before invoking irrelevance. If we care about not doing extra work (for instance if we want to extract the decision algorithm) we would need to add such a shortcut rule.

Most properties are trivially preserved. For instance soundness of neutral equality `soundness~↑ : ∀ {k l A rA Γ} → Γ ⊢ k ~ l ↑ A ^ rA → Γ ⊢ k ≡ l :: A ^ rA` has just 1 new case to consider where `Γ ⊢ k ~ l ↑ A ^ rA` is between irrelevant terms, so inhabited by `~↑%` applied to some proofs that `k` and `l` irrelevantly inhabit `A`, so we can just apply the proof irrelevance rule of the typing judgments.

Congruence of Algorithmic Equality

We need to show that congruence is admissible for equality of neutrals at any relevance (in order to show that it's part of a generic equality, considering it isn't trivial by constructor

after the relevance separation).

This needs a small amount of reasoning about substitutions. Consider the case of congruence for applications:

```
app-cong′ : ∀ {Γ k l t v F rF G rG}
          → Γ ⊢ k ~ l ↓ Π F ^ rF ▹ G ^ rG
          → Γ ⊢ t [conv↑] v :: F ^ rF
          → Γ ⊢ k ∘ t ~ l ∘ v ↑ G [ t ] ^ rG
```

The relevant case `rG = !` is trivial by constructor `app-cong`. In the irrelevant case, we need to show that `k ∘ t` and `l ∘ v` both inhabit `G [ t ]`. We can show this in 2 ways:

- by soundness, $\Gamma \vdash$ `k ≡ l :: Π F ^ rF ▹ G ^ %` and $\Gamma \vdash$ `t ≡ v :: F ^ rF`. By syntactic validity, $\Gamma \vdash$ `k :: Π F ^ rF ▹ G ^ %` (and so on for the other terms), so $\Gamma \vdash$ `k ∘ t :: G [ t ] ^ %` and $\Gamma \vdash$ `l ∘ v :: G [ v ] ^ %`. Then we need a lemma showing that substituting convertible terms `t` and `v` in `G` produces convertible results.
- alternatively, we use the congruence rule of the object type theory on the conversion proofs gotten by soundness, proving $\Gamma \vdash$ `k ∘ t ≡ l ∘ v :: G [ t ] ^ %`. Then by syntactic validity both sides have type `G [ t ]`: the syntactic validity lemma invokes the substitution lemma for us.

The second method (invoking soundness, congruence in the object theory and finally syntactic validity) requires less thought to extend to new neutral term formers such as `natrec` compared to building a ad-hoc reasoning for each.

Decidability of Algorithmic Equality

Our main statement is

```
decConv↑ : ∀ {A B r Γ Δ}
         → ⊢ Γ ≡ Δ
         → Γ ⊢ A [conv↑] A ^ r → Δ ⊢ B [conv↑] B ^ r
         → Dec (Γ ⊢ A [conv↑] B ^ r)
decConv↑Term : ∀ {t u A r Γ Δ}
             → ⊢ Γ ≡ Δ
             → Γ ⊢ t [conv↑] t :: A ^ r
             → Δ ⊢ u [conv↑] u :: A ^ r
             → Dec (Γ ⊢ t [conv↑] u :: A ^ r)
```

We decide conversion between types at the same relevance, and between terms at the same type and relevance. We use the reflexivity hypotheses to structure our recursion.

Note: About the context equality hypothesis:

The 2 sides do not live in the same context, instead we require their contexts to be have pointwise convertible terms:

```
-- Equality of contexts.
data ⊢_≡_ : (Γ Δ : Con Term) → Set where
```

(continues on next page)

(continued from previous page)

```
ϵ : ⊢ ϵ ≡ ϵ
_•_ : ∀ {Γ Δ A B r} → ⊢ Γ ≡ Δ
    → Γ ⊢ A ≡ B ^ r
    → ⊢ Γ • A ^ r ≡ Δ • B ^ r
```

Using contexts equal up to conversion makes the comparison of product types easier: the reflexivity hypotheses for the codomains are in contexts extended by different domains, but once we have decided conversion for the domains we know that the contexts are equal.

When comparing neutrals terms, even when the terms have a common type the subterms may have different types. For instance we may be comparing applications of a variable of type $\mathsf{Empty} \to \mathbb{N}$ and one of type $\mathbb{N} \to \mathbb{N}$: both applications have type $\mathbb{N}$. This is troublesome when comparing irrelevant terms, as they are equal exactly when their types are equal so we would need to complicate the recursion to go back to comparing types.

Thankfully our separation of neutral equality between the relevant and the irrelevant worlds ensures that we only need to compare irrelevant terms at the same type. For instance, `Emptyrec-cong` (in the *definition of neutral equality* (page 67)) involves irrelevant terms which are always at type Empty, and since they are irrelevant there will be no further recursion.

4.3.7 Necessary Conditions: Why Booleans are not Irrelevant

We may wonder where exactly the proof would break down if we tried to change the theory in an inconsistent or undecidable way, for instance adding an irrelevant type of booleans.

The critical part is the fundamental lemma:

- In the case for congruence of the eliminator for booleans, we need that applying the eliminator to reducibly equal booleans produces reducibly equal results. To do this, we define reducible equality on booleans such that two terms are equal when they both reduce to the same constructor or to neutral terms.
- With the above definition for reducible equality, we do not have proof irrelevance for the logical relation.

4.3.8 Limitations

The object type theory is significantly weaker than the theories implemented in Coq and Agda, so we should consider the potential impact of these differences.

- Impredicativity of the irrelevant universe: the above proof method is unsuited to dealing with impredicativity.
- Universes: we only have 1 level of universes, which means being convertible to a universe is the same as being syntactically equal to it. Adding more universes would require fixing the proofs where this is used, especially in the definition of a

type being reducibly equal to a universe. The levels used for the logical relation would also change (there is one for each universe, and a top level). Universes and proof relevance are related so this is something to be aware of (most importantly, since we can't deal with impredicativity we would need an irrelevant universe as well as a relevant one at each level), but generalizing the proofs seems quite feasible.

- Inductive familes and sigma types with definitional eta extensionality: this seems orthogonal to the issue of proof relevance and as such should not present any particular issues. Induction-recursion is a crucial feature of the meta-theory used in the proof, and as such would probably run into Gödel incompleteness if we tried to add it to the object theory. It is unclear whether induction-induction could be made to work.
- Type annotations on lambda abstractions: we can prove decidability of conversion, but not decidability of typechecking as the lack of type annotations on lambda abstractions can be used to encode undecidable unification problems. However a type theory with annotated lambdas may be translated to the un-annotated type theory while preserving head reductions, so this is not a problem.
- Relevance marks: the type theory implemented in Coq has relevances marks in contexts and for product type domains as in the object theory, and also a few more (notably the domain of lambda abstractions and for case eliminations). However they are not visible in the user syntax: by relevance unicity they are inferrable data. The translation between type theories with and without relevance marks could be more carefully described, but I don't have the time.
- Definitional uniqueness of identity proofs: having an irrelevant identity type changes the reduction significantly, such that it is difficult or maybe impossible to adapt this proof for it. Notice for instance that `eq_rect x P v y e` is in weak head normal form if and only if `x` is *not* convertible to `y`, and as such weak head normal form is not a simple syntactic predicate.
- Box and squash: neither require anything fundamentally new from the logical relation. Squash is interesting in that it adds irrelevant terms which do not reduce to lambdas or neutral terms (such as `squash 0`), so we would need generic equalities to have proof irrelevance even on non-neutrals. Extending the proof to allow squash may also increase the understanding of how exactly adding irrelevant booleans would break it. However I don't have the time.

4.4 Canonicity and irrelevant axioms

Note: The following only holds when we do *not* have definitional UIP.

First we prove a lemma:

Inconsistency from neutral terms: if there is a neutral term of relevant type in a context Γ containing only irrelevant variables, then that context is inconsistent.

Proof:

By induction on the term:

- It cannot be a variable or an elimination of `Squash` since it is a relevant term.
- If it is an elimination from an inductive type (which are all relevant), Box or an application, then the head is a smaller relevant neutral term also in Γ, so we recurse on it.
- If it is an elimination from the empty type then the head is a term in the empty type in Γ.

This gives us **Canonicity in irrelevant contexts**: given a term t with a relevant type in a consistent context Γ which contains only irrelevant variables, t reduces to an introduction form. For instance if t is in the type natural numbers, then t reduces to some $S^n 0$.

Proof:

By normalisation, either t reduces to an introduction form or to a neutral term. In the later case, Γ must be inconsistent.

For instance this means the axiom of proof irrelevant double negation elimination $\forall P : \mathsf{SProp}, ((P \to Empty) \to Empty) \to P$ preserves canonicity.

4.5 Comparison With Other Irrelevance Systems

Abel and Scherer's Irrelevant Intensional Type Theory [AS12] directs definitional irrelevance around applications: applications can be relevant $t \circ u$ or irrelevant $t \div u$, irrelevant applications are equal regardless of the arguments values. To form an irrelevant application, the head must be an irrelevant function of type $\Pi(x \div A), B$, and such functions may be formed by irrelevant lambda abstractions $\lambda(x \div A), t$. Such irrelevant binders come with restrictions: the variable is introduced irrelevantly to the context producing $\Gamma, x \div A$. Irrelevant variables may not be directly used, but when typing the argument of an irrelevant application all variables of the context are made relevant, and therefore accessible.

Squash types may also be expressed in terms of irrelevant arguments by allowing them in the type of constructors of inductive types: then the squash has a unique constructor of type $\Pi(x \div A), \mathsf{Squash}\ A$. Note that such a type has only propositional proof irrelevance (behaving more like Box (Squash A) according to our definitions).

It seems possible to translate from a theory with irrelevant arguments to one with SProp, by translating bindings $x \div A$ to bindings $x : \mathsf{Squash}\ A$. Then the translation of irrelevant application must unsquash all irrelevant variables from the context before squashing the translated argument. For instance if the context has one irrelevant variable y:

$$[t \div u] = [t] \circ (\mathsf{match}\ y\ \mathsf{with}\ \mathtt{squash}\ y \Rightarrow \mathtt{squash}\ [u]\ \mathsf{end})$$

In this example y is shadowed inside the match.

A full investigation of such a translation has not yet been done.

In the other direction, we need to deal with quantification over SProp somehow while preserving conversions. An interesting exercise is translating the statement of proof irrelevance using Leibnitz equality:

$$\Pi(P : \mathsf{SProp})(Q : P \to \mathsf{Type})(x\ y : P), Q\ x \to Q\ y$$

Suppose we translate SProp to a relevant universe, and keep irrelevant bindings and applications, then we get

$$\Pi(P : \mathsf{Type})(Q : \Pi(\_ \div P), \mathsf{Type})(x\ y \div P), Q \div x \to Q \div y$$

A function producing a value in an irrelevant type would be translated to a regular (relevant) function, but all uses would be translated to irrelevant applications. This seems like a sensible output but I have not checked that it works in general.

In this sketch of a translation, we rely on the lack of cumulativity from irrelevance to relevance in order to decide how to translate bindings and applications. What does this mean for translations of cumulative theories?

We may also compare with Miquel's "Implicit Calculus of Constructions" [Miq01]: in this system, irrelevant function types may use the introduced variable relevantly in the codomain, such that irrelevant bindings are only introduced by lambda abstractions. This additional power indicates that it is stronger than simply having a universe of strict propositions.

CHAPTER

5

INTERPRETING INDUCTIVE FAMILIES AS FIXPOINTS

We've defined a theory with a small number of primitive SProp-related types. To make use of it, we need to be able to express more complicated types. Thankfully, SProp is a universe and as such we can construct its inhabitants by large eliminations from the usual inductive types such as $\mathbb{N}$. In this chapter we detail a general way of defining types by large eliminations which behave like certain inductive types. This allows us to encode such inductive types without needing them as a primitive in the theory, and so giving us inductive types in SProp.

Most of this material has already been published as Gilbert et al. [GCST19].

Note that there are other ways to define datatypes with a dependent elimination principle, as for instance by using an impredicative encoding in a type theory enriched with dependent intersections types as done in Cedille [Stu18].

5.1 The Translation

We define elimination by case trees according to [CDP14].

5.1.1 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.

For instance, the following inductive and fixpoint are equivalent:

```
Inductive le : nat -> nat -> Prop :=
| le_0 : forall n, le 0 n
| le_S : forall m n, le m n -> le (S m) (S n).

Fixpoint le (m n : nat) : Prop := match m, n with
| 0, _ => True
| S m', S n' => le m' n'
| S _, 0 => False
end.
```

One challenge in this translation is to determine which constructor arguments should appear on the right-hand side: for the constructor `le_S`, the argument of type `le m 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 `le`:

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

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

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

where:

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

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

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

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

- **Done**: If all constraints are solved, elaboration returns a leaf node containing the disjoint sum of the remaining constructors:

$$\Gamma \vdash \{\mathtt{c}_1\ \Delta_1\ []; \ldots; \mathtt{c}_k\ \Delta_k\ []\} \rightsquigarrow \mathtt{c}_1\ \Delta_1 \uplus \ldots \uplus \mathtt{c}_k\ \Delta_k$$

  The empty disjoint sum being the empty case tree: $\Gamma \vdash \{\} \rightsquigarrow \bot$.

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

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

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

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

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

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

- **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:

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

- **Split**: Finally, if $(x : \mathtt{D}\ \bar{w}) \in \Gamma$ and each constraint set Φ_i contains a constraint of the form $x \stackrel{?}{/} \mathtt{d_j}\ \bar{u}_j : \mathtt{D}\ \bar{w}$ where $\mathtt{d_j}$ is a constructor of the datatype $\mathtt{D}$, elaboration continues by performing a case split on x. For each constructor $\mathtt{d_j} : \Delta'_j \to \mathtt{D}\ \bar{w}'_j$, we use proof-relevant unification [CD18] to determine whether this constructor can be used at indices $\bar{w}$. For each of the constructors for which unification succeeds positively, elaboration continues to construct a subtree for that constructor.

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

- **Implicit Identity**: If there are unsolved constraints but it is not possible to split on any variable, we may turn each remaining constraint $v \stackrel{?}{/} u : A$ into a new constructor argument of type $u =_A v$. In other words, the constraint implicitly introduces an identity-typed argument.

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

The elaboration algorithm repeats the steps above whenever they are applicable until it produces a complete case tree. Thanks to the *Implicit Identity* (page 76) rule it cannot get stuck.

5.1.2 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 $\mathtt{lz} : (n : \mathbb{N}) \to 0 \leq n$ and $\mathtt{ls} : (m\ n : \mathbb{N}) \to m \leq n \to \mathtt{S}\ m \leq \mathtt{S}\ n$ by $\mathtt{lz} = \lambda n.\ \mathtt{tt}$ and $\mathtt{ls} = \lambda m\ n\ p.\ p$ respectively. Note that these terms are type-correct since $0 \leq n = \mathsf{sUnit}$ and $\mathtt{S}\ m \leq \mathtt{S}\ n = m \leq n$. We can also define the eliminator $\leq\_\mathtt{rect}$ by performing the same case splits as in the translation of the type $\leq$:

$$\begin{aligned}
&\leq_\mathtt{rect} : (P : (m\ n : \mathbb{N}) \to m \leq n \to \mathsf{Type})(m_0 : (n : \mathbb{N}) \to P\ 0\ n\ (\mathtt{lz}\ n)) \\
&\qquad (m_S : (m\ n : \mathbb{N})(H : m \leq n) \to P\ m\ n\ x \to P\ (\mathtt{S}\ m)\ (\mathtt{S}\ n)\ (\mathtt{ls}\ m\ n\ x)) \\
&\qquad (m\ n : \mathbb{N})(x : m \leq n) \to P\ m\ n\ x \\
&\leq_\mathtt{rect}\ P\ m_0\ m_S\ m\ n\ x \\
&\quad = \mathtt{case}_m \left\{ \begin{array}{lcl} 0 & \mapsto & m_0\ n \\ \mathtt{S}\ m' & \mapsto & \mathtt{case}_n \left\{ \begin{array}{lcl} 0 & \mapsto & \mathtt{sEmpty_rect}\ x \\ \mathtt{S}\ n' & \mapsto & m_S\ m'\ n'\ x\ (\leq_\mathtt{rect}\ P\ m_0\ m_S\ m'\ n'\ x) \end{array} \right\} \end{array} \right\}
\end{aligned}$$

Note that the eliminator applied to the constructors computes as expected for the inductive type.

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

Constructing the constructors

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

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

Constructing the eliminator

The eliminator `D_rect` for the elaborated datatype $\mathtt{D} : \Gamma \to \mathsf{Type}$ with constructors $\mathtt{c_i} : \Delta_i \to \mathtt{D}\ \bar{v}_i$ for $i = 1 \dots k$ takes the following arguments:

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

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

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

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

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

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

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

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

evaluates to

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

Note: In Coq the elimination principles are derived from fixpoint and match primitives. Matches are trivial to translate once we know how to translate elimination principles since they're less general. Fixpoints can also be translated but the justification is harder as it involves the complex structural recursion rules.

5.1.3 Non translatable types

The resulting case tree is not always well-founded. For instance, from the inductive

```
Inductive Loop := loop : Loop -> Loop.
```

we get

$$\vdash \{\texttt{loop Loop []}\} \rightsquigarrow \texttt{loop Loop}.$$

This would make us define the translation as

```
Fixpoint Loop := Loop.
```

which of course is rejected. Our algorithm leaves deciding termination of the produced case tree to the type-checking of the host type theory. Note that if the translated inductive terminates, then so does the translated eliminator as it recurses on the same arguments.

Termination checking can be complex, for instance Coq recognises that the applied fixpoint `n - m` (subtraction for natural numbers) is (non strictly) structurally smaller than `n`. Conversely, there may be inductives whose translation is terminating but for which termination checking is not smart enough to finish the translation automatically. In such cases we may be able to manually adapt the translation's result, based on our argument for why the translation is terminating (e.g. by using the accessibility predicate and well-founded recursion). Note that such an adaption may destroy the definitional computation rules of the eliminator, although they remain provable.

Other inductives such as `Loop`, Peano natural numbers or the accessibility predicate provide expressive power through their inductive structure which is not available from the primitive data types used by the translation (disjoint sum, sigma types, empty type, unit type and identity type). As such it is expected that they cannot be translated by our algorithm.

5.2 Application to Strict Propositions

A case tree represents a type in SProp when it is is either a leaf node of the form $\mathsf{c}\ \Delta$ where c is a constructor name and Δ is a telescope of types in SProp, an empty node $\bot$, or an internal node of the form

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

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

The translation fails to produce a type which can live in SProp when either:

- one of the constructors has a non-SProp argument. Allowing the inductive in SProp would mean either inconsistency (if the telescope of arguments is not a mere proposition) or risk undecidability such as with *Singleton types* (page 16).
- there are multiple constructors remaining when we produce a leaf node in the *Done* (page 75) rule. From this we can get instantiations of different constructors at the

same type (possibly in an inconsistent context). Then either the theory is inconsistent or conversion is undecidable (we need to verify consistency of the context to ensure it has not collapsed).

- the translation produces a non-terminating case tree (or at least one that we cannot prove terminating). Then the conversion is undecidable as we explained in *Flirting with Undecidability: the Accessibility Predicate* (page 17).
- the translation used the *Implicit Identity* (page 76) rule and the host theory does not have definitional UIP.

As such, when we have UIP the translation together with our primitive SProp types are complete: all inductive types which could be accepted as SProp inductives without breaking decidability or consistency can be translated. (There may still be irrelevant types which would preserve decidability and consistency, but they cannot be expressed as inductive types.)

When we do not wish to have UIP, the impact of allowing specific types involving implicit identities (for instance having an identity type for booleans in SProp) is less clear, see chapter *Reduction Behaviour of Irrelevant Identities* (page 81).

CHAPTER

6

DEFINITIONAL UNIQUENESS OF IDENTITY PROOFS

6.1 Reduction Behaviour of Irrelevant Identities

Extending a type theory with SProp to have definitional uniqueness of identity proofs at first glance seems to be just about allowing the inductive family representing identity in SProp:

```
Inductive eq {A} (a:A) : A -> SProp := eq_refl : eq a a.
```

However unlike the primitives we have seen previously (`Empty`, `Squash`, etc) it also requires special reduction behaviour to preserve good properties of reduction.

Usually, when we match some discriminee which does not reduce to a constructor, the match's reduction is also blocked:

```
Eval lazy in fun x : nat => match x with 0 => true | S _ => false end.
    = fun x : nat =>
          match x with
          | 0 => true
          | S _ => false
          end
        : nat -> bool
```

Now suppose we match on an hypothesis of type `eq 0 0`:

```
Definition m1 := fun x : eq 0 0 =>
  match x with eq_refl _ => nat -> nat end.
```

By proof irrelevance and since they have the same type, the discriminee `x : eq 0 0` may be replaced by `eq_refl 0`, so `m1` should be convertible to

```
Definition m2 := fun _ : eq 0 0 =>
  match eq_refl 0 with eq_refl _ => nat -> nat end.
```

then by reduction to

```
Eval lazy in m2.
    = fun _ : eq 0 0 => nat -> nat
        : eq 0 0 -> Set
```

Because we check conversion by reducing both sides then inspecting their structure, and as the match in `m1` does not have the same structure as `nat -> nat`, we have to make sure that it does reduce to `nat -> nat`. Additionally, suppose we wished to apply some `f : m1 x` to `0` (in a context where `x : eq 0 0`), to check this we need to find a way to convert `m1 x` to a product type: that way is reduction.

Of course, we can't reduce every match on an `eq` to its single branch: that would be ill-typed in general and so would break subject reduction. We need to reduce

```
match e : eq a b as x in (eq _ y) return (P y x) with
  | eq_refl _ => v : P a (eq_refl a)
end : P b e
```

to `v` if and only if `a` and `b` are convertible:

- if they are convertible `e` is convertible to `eq_refl a`.
- if they are not convertible, `e` and `eq_refl a` do not have convertible types so there is no reduction to do.

This special reduction behaviour was already noted in [Wer08].

Note: Perhaps we could also reduce when `P a (eq_refl a)` and `P b e` are convertible (for instance when `P` doesn't use its arguments), but we don't need to in order to keep the type theory tractable.

Since this makes reduction depend on conversion, there are severe consequences on concepts used in metatheoretical proofs. For instance the property of being in weak head normal form now involves the negation of conversion. Since the proof in section *Decidability and Reducibility with Proof Irrelevance* (page 58) deeply uses weak head normal forms, it seems quite difficult and perhaps impossible to add `eq` to it.

Another lost property: applying a substitution from variables to neutral terms may unlock reductions, for instance the substitution $y \mapsto x$ applied to a match on `eq x y`. This may have an impact when attempting to adapt the reduction machine used in Coq [Cre91] as it uses neutrality to control the sharing optimisations.

6.2 Termination of the Special Reduction

Normalization of a theory with definitional UIP breaks down in the presence of another impredicative universe: using

```
Axiom all_eq : forall P Q : Prop, eq P Q.

Definition K (A B : Prop) (x:A): B := match all_eq A B with eq_refl => x end.

Definition bot : Prop := forall P : Prop, P.

Definition c : bot := fun X : Prop => K (bot -> bot) X (fun z : bot => z (bot -> bot) z).
```

we would have `c (bot -> bot) c` infinitely reduce to itself.

Using SProp instead of Prop poses the same issues.

This issue was identified by Abel and Coquand [AC19].

There is no such known issue in predicative theories.

Alternatively, perhaps restricting irrelevant equalities to small types (such that `all_eq` cannot be stated any more, since Prop is a large type) may be sufficient to recover termination. This may be related to the speculation in the following section, where we focus on compatibility with univalence.

6.3 Hierarchy of Strictly Truncated Universes

Allowing specific types with implicit identities may not imply definitional UIP for all types. For instance, we could have definitional UIP for natural numbers

```
Inductive nateq (n:nat) : nat -> SProp := nateq_refl : nateq n n.
```

without having UIP on arbitrary types.

More generally, we could imitate the homotopy levels defined in HoTT to have a concept of "strictly truncated universe". Where in HoTT we define

```
Record Contr (A:Type) := { center : A; contr : forall y, center = y }.

Inductive hlevel := -2 | +1 : hlevel -> hlevel.

Fixpoint IsTrunc n A := match n with
  | -2 => Contr A
  | +1 k => forall x y : A, IsTrunc k (x = y)
  end.

Record hType n := { htype : Type; htrunc : IsTrunc n htype }.
```

We could have universe Type_i^h with a predicative level i and a strict homotopy truncation level h. The strict homotopy level ranges from -1 to ∞ (there is no use for a type of

contractible types so we start at -1, and ∞ is for types which are not truncated at any level). The rules are

hUniv-in-hUniv

$$\frac{i < j \qquad h < k \vee k = \infty}{\Gamma \vdash \mathsf{Type}_i^h : \mathsf{Type}_j^k}$$

Pi-in-hUniv

$$\frac{\Gamma \vdash A : \mathsf{Type}_i^k \qquad \Gamma, x : A \vdash B : \mathsf{Type}_j^h}{\Gamma \vdash \Pi(x : A).B : \mathsf{Type}_{max(i,j)}^h}$$

In other words the homotopy level is impredicative.

Then SProp_i is just Type_i^{-1}, i.e.

hUniv-irrelelance

$$\frac{\Gamma \vdash A : \mathsf{Type}_i^{-1} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t \equiv u : A}$$

(if we want impredicativity of SProp we can add that all SProp_i are convertible to each other or some such collapsing rule)

Finally we use the strict homotopy level when defining equality types: if $A : \mathsf{Type}_i^h$ and $x, y : A$ then $x =_A y : \mathsf{Type}_i^{h-1}$ (at the edges $-1 - 1 = -1$ and $\infty - 1 = \infty$).

This is difficult to implement: when using universe variables with a graph of constraints, $h - 1$ is not cleanly encoded as new graph constraints. We could try to generalize the graph system, but any such generalization would risk performance. Universes already have a significant performance cost with little visible benefit (mathematicians rarely need to think about universes when working) so type theory with strict homotopy levels may be better suited to experimental proof assistants than to heavily used Coq. The practical use of a hierarchy of strictly truncated types is also unclear.

CHAPTER

7

IMPLEMENTING DEFINITIONAL PROOF IRRELEVANCE IN COQ

7.1 Implemented Theory

We add a new universe of strict propositions SProp. It is impredicative, like the pre-existing universe of propositions Prop. Unlike Prop it is isolated in the cumulativity relation: $\mathsf{SProp} \leq u$ if and only if $u = \mathsf{SProp}$.

7.1.1 Irrelevant Inductive Types

Like with Prop, any inductive type with well-typed and strictly positive constructors may be declared in SProp, but elimination may be restricted to motives in SProp. The condition for unrestricted elimination is much stronger than for Prop: only inductives with no constructors – in other words, empty types – avoid being squashed.

Note: When an inductive type is squashed in SProp we must prevent defining proof relevant fixpoints (and not just matches) whose principal argument is that inductive. Otherwise it becomes possible to define fixpoints over the squashed accessibility predicate:

```
Inductive Acc (x:A) : SProp :=
  Acc_in : (forall y, R y x -> Acc y) -> Acc x.

Definition Acc_inv {x} (a : Acc x) : forall y, R y x -> Acc y
  := match a with Acc_in _ f => f end.
```

(continues on next page)

(continued from previous page)

```
Fixpoint Fix {P:A -> Type}
  (f : forall x, (forall y, R y x -> P y) -> P x)
  x (a:Acc x) {struct a} : P x
  := f x (fun y Rxy => Fix f y (Acc_inv a y Rxy)).
```

This is because Coq is smart enough to know that `Acc_inv a y Rxy` is structurally smaller than `a`.

This implicit squashing lets us define the `Squash` type as an inductive itself.

About the elimination restriction

The *translation to fixpoints* (page 74) gives a criterion allowing unrestricted eliminations for inductives like

```
Inductive Unit : SProp := .

Inductive NonZero : nat -> SProp := S_NonZero : forall n, NonZero (S n).
```

or even

```
Inductive le : nat -> nat -> SProp :=
| le0 : forall n, le 0 n
| leS : forall n m, le n m -> le (S n) (S m).
```

without affecting the power of the theory. The reduction rules for the eliminators can be derived from the translation: for instance `Unit_rect` reduces unconditionally. Once the constructors take arguments it becomes more complicated:

```
match p : NonZero (S n) with S_NonZero k => k end
```

must reduce to `n` even when `p` is blocked (analogously to the *Reduction Behaviour of Irrelevant Identities* (page 81)), such that we need to extract `n` from the index of the type of the discriminee.

The reduction for matches on `p : le (S n) (S m)` is even more complex: it needs to produce a proof of `le n m`. Such a proof is itself a match, but infinite recursion may be avoided by restricting the special reduction to relevant matches, leaving irrelevant matches such as the proof of `le n m` to reduce only when `p` reduces to a constructor (which by typing must be `leS`).

This restriction works because thanks to proof irrelevance we only need to reduce irrelevant terms for canonicity, in which case we are in an empty context and the special reduction is not needed.

A second difficulty comes from the need to detect which types are allowed by the translation. A fully general implementation would make the kernel depend on dependent unification, introducing significant complexity.

I originally implemented a version of Coq with SProp with an approximation of the criterion based on "forced arguments" (as explained by Cockx [CDP14]). This approximations accepted `Unit` and `NonZero`, but rejects types needing projections to reduce (such as for the recursive argument of `leS`), types with multiple constructors (such as `le`) and a variety of types involving unification problems beyond the capability of forced arguments.

In the end I decided to drop this system, judging that the implementation complexity outweighed its benefits considering it neither improved the power of the theory nor prevented the needed for encodings on even simple examples.

Identity Types

An implementation allowing non squashed identity types in SProp (with the special reduction behaviour) was also made, but the combination with impredicativity turned out to be undecidable. Perhaps it will be reused if we can build confidence that restricting irrelevant identities to be on small types is unproblematic (i.e. `eq : forall A : Set, A -> A -> SProp` with `Set` a universe which is not the type of any other universe may have unrestricted eliminations), or if we implement a fully predicative mode for Coq.

The implementation is also incomplete: the VM and native conversion reduction systems do not take the special reduction into account, causing incompleteness of the respective conversion checkers. It is unclear how difficult changing this will be.

Until then, we can locally ignore universe constraints to get irrelevant inductive types with unrestricted eliminations but without the special reduction:

```
Unset Universe Checking.
Inductive eq {A} (a:A) : A -> SProp := eq_refl : eq a a.
Set Universe Checking.
```

This does not threaten consistency but breaks transitivity of the implemented conversion and subject reduction. Canonicity may also be preserved but I don't have a full proof.

Note: The `Universe Checking` flag requires Coq version 8.11 or above, expected release early January 2020.

7.1.2 Relevant Inductive Types

Proof relevant inductive type may have their constructors take proof irrelevant arguments: they are implictly boxed. This lets us declare Box as a normal inductive.

7.1.3 Primitive Records

Primitive records may be declared in SProp if all their fields are irrelevant. This includes the type of irrelevant pairs:

```
Set Primitive Projections.
Record sSigma (A:SProp) (B:A -> SProp) : SProp
  := ssigma { spr1 : A; spr2 : B spr1 }.
```

There is no squashing for primitive records, so records in SProp with a relevant field are rejected.

This is the preferred way to declare irrelevant pairs, as the inductive version is squashed, making it harder to work with (although proof irrelevance and definability of its projections mean it is no less powerful than the record version).

There is also a restriction for relevant records to avoid definitional proof irrelevance escaping from SProp through the η expansion, for instance with the following record:

```
Fail Record Box (A:SProp) : Set := box { unbox : A }.
```

Relevant records are allowed to have irrelevant fields so long as they also have at least one relevant field. For instance

```
Record sigS (A:Type) (B:A -> SProp) : Type := existS { pr1S : A; pr2S : B pr2S }.
```

7.2 Kernel Side

We mark elements of the environment with their relevance: variables, constants, inductive types (with their relevance as types, which is the relevance as terms of the constructors) and record projections.

There are also marks inside terms for every binder (product types, lambda abstractions, and (co)fixpoints. Note that the binders involved in case eliminations such as pattern variables are internally represented as lambdas, at least for now). Case eliminations are also marked with their relevance (i.e. the relevance of their return type): this could often be recovered from the branches but eliminations on the empty type have no branches.

With all these marks, determining the relevance of a term is a syntactic recursion along the head of applications down to either something marked with its own relevance or something whose relevance can be looked up in the environment. Critically for performance, it involved no substitution or reduction.

Coq's kernel has 3 implementations of reduction: a walk over the abstract syntax implementing the lazy Krivine abstract machine [Cre91][Bar99], a bytecode compiler/interpreter [GL02] and a compiler to OCaml which then calls the OCaml compiler and dynamically links the result [BDG11]. Each is associated with a decision procedure for conversion trusted by the kernel.

For each of these conversions, the basic idea is to weak head normalize the arguments, compare the shape of the heads then recurse on the "stack" of applications, case eliminations, etc around around the heads. To implement proof irrelevance, we simply check whether the terms to convert are marked irrelevant around each such recursive step. In simplified code (from `kernel/reduction.ml`, function and exception names unchanged):

```
let rec ccnv env x y =
  try eqappr env x y
  with NotConvertible
    when is_irrelevant env x && is_irrelevant env y -> ()

and eqappr env x y =
  let hx, sx = whd_stack env x in
  let hy, sy = whd_stack env y in
  match hx, hy with
  | Var vx, Var vy ->
    if vx <> vy then raise NotConvertible;
    convert_stacks env sx sy
    (* calls ccnv on each argument for application,
       on branches for matches, etc *)
  | ...
```

Note: As a performance optimisation, we do not unfold constants in the `whd_stack` calls. Then we converting two applications of the same constant, we can try converting their arguments and unfold only if that fails. When doing that unfolding, relevance has not changed so we directly recurse with `eqappr` instead of `ccnv`.

Note: Assuming the starting conversion problem is between terms sharing a common type, when we check for irrelevance it is always for terms which have a common type (just as it works in *Algorithmic Equality and Proof Irrelevance* (page 66)), so we do not need to check both terms. However the untrusted layers of Coq (for instance the tactic `set (x:=term)`, see past issue[15]) use the kernel conversion between terms which do not share a type. Checking relevance for both sides of the conversion problem then reduces SProp-related problems.

A similar addition needs to be made to the VM and native compute based conversions.

7.3 Issues with non-cumulativity

During normal term elaboration (transforming user syntax to a term of type theory), we don't always know that a type is a strict proposition early enough. For instance:

```
Definition constant_0 : ?[T] -> nat := fun _ : sUnit => 0.
```

While checking the type of the constant, we only know that `?[T]` must inhabit some sort. Putting it in some floating universe `u` would disallow instantiating it by `sUnit : SProp`.

In order to make the system usable without having to annotate every instance of SProp, we consider SProp to be a subtype of every universe during elaboration (i.e. outside the kernel). Then once we have a fully elaborated term it is sent to the kernel which will

[15] https://github.com/SkySkimmer/coq/issues/13

check that we didn't actually need cumulativity of SProp (in the example above, u doesn't appear in the final term).

This means that some errors will be delayed until Qed:

```
Lemma foo : Prop.
Proof.
pose (fun A : SProp => A : Type); exact True.

Fail Qed.
    The command has indeed failed with message:
    In environment
    A : SProp
    The term "A" has type "SProp"
    while it is expected to have type "Type".

Abort.
```

By using `Unset Elaboration StrictProp Cumulativity.` Coq keeps cumulativity disabled for SProp even during elaboration. This ensures that conversion is complete even outside the kernel, at the cost of surprising errors.

7.3.1 Incorrect Marks

If cumulativity is used during elaboration, there is a possibility that the conversion checker will see incorrect marks:

```
Fail Definition late_mark :=
  fun (A:SProp) (P:A -> Prop) x y (v:P x) => v : P y.
    The command has indeed failed with message:
    In environment
    A : SProp
    P : A -> Prop
    x : A
    y : A
    v : P x
    The term "v" has type "P x"
    while it is expected to have type "P y".
```

The binders for x and y are created before their type is known to be A, so they're not marked irrelevant. This can be avoided with sufficient annotation of binders (see irrelevance at the beginning of this chapter) or by bypassing the conversion check in tactics.

```
Definition late_mark :=
  fun (A:SProp) (P:A -> Prop) x y (v:P x) =>
  ltac:(exact_no_check v) : P y.
```

The kernel will re-infer the marks on the fully elaborated term, and so correctly converts x and y.

This system is still more limited than we like, for future work we will attempt to add a

concept of elaboration-time universe variables to Coq (analogous to existential variables when compared with regular variables). Such variables would be allowed to unify with SProp and help us keep track of such unifications to update marks. As a bonus, we hope to solve the following long-standing error:

```
Fail Check forall A B, A -> B -> exists x : A, B.
    The command has indeed failed with message:
    In environment
    A : Type
    B : Type
    x : A
    The term "B" has type "Type"
    while it is expected to have type "Prop"
    (universe inconsistency).
```

CHAPTER

8

OTHER CONTRIBUTIONS

At the beginning of my thesis I published [Gil17] following up on a preceding internship with Bas Spitters.

During my thesis I also participated in Coq development, becoming a core developer with particular expertise in the implementation of the universe system and the Continuous Integration system.

8.1 Inconsistencies

I worked on identifying and removing inconsistencies:

- I found and fixed an incorrect optimisation when checking module subtyping with inductive types living in algebraic universes (bug report #7695[16], fix #7798[17]).
- I found and fixed a missing freshness check in universe polymorphism (fix #8341[18], no separate bug report).

  The idea of the bug expressed with lambda calculus instead of universe polymorphism is that in `let x := 0 in λ x, t`, while type checking `t` the variable `x` would be considered convertible to `0` according to the `let`-binding, but after type checking reducing an application would allow to substitute it by anything else.
- I found a bug about the freshness of universes in "template universe polymorphism" (#9294[19], fixed by Matthieu Sozeau in #9918[20]).

[16] https://github.com/coq/coq/issues/7695
[17] https://github.com/coq/coq/pull/7798
[18] https://github.com/coq/coq/pull/8341
[19] https://github.com/coq/coq/issues/9294
[20] https://github.com/coq/coq/pull/9918

- I found a further bug in template universe polymorphism due to not taking into account implicit universe constraints caused by using a universe non-linearly (#11039[21], not yet fixed).

8.2 Universe System Improvements

- Preserving user-given universe names (as opposed to the automatic `Filename.123` names) across module and file boundaries (#1033[22], this comes with a number of internal API cleanups)
- Generating automatic names for universes belonging to specific constants (which allows to identify their origin and meaning more easily) (#8760[23])
- Removing "phantom" universe quantification of universe polymorphic objects ("phantom" universes are not used by the object, typically they are generated by elaboration or tactics and do not appear in the final result. Their presence can cause significant performance issues and generally makes universe polymorphic definitions more noisy.) (#7495[24], with some preliminary work in the aforementioned #1033)
- Private universes for opaque universe polymorphic definitions (#8850[25]). When an opaque (never-unfoldable) proof uses a universe which doesn't appear in its statement (for instance to define an auxiliary type by large case elimination), such universes do not need to be given as argument: a valid such universe can always be assumed to exist if the transitive constraints between the "public" universes hold, and since the constant cannot be unfolded we never need a concrete value for the private universe.

  As a contrived example, consider

```
Polymorphic Lemma foo : Type.
Proof. exact Type. Qed.
```

  If we made the constant transparent, we would get `foo@{u v | Set <= v < u} : Type@{u} = Type@{v}` (polymorphic universes are always above `Set`). However with an opaque `foo` we never care about the value of `v`, so it becomes `foo@{u | Set < u} : Type@{u}` with the inhabitance certificate `{v | Set <= v < u} |= Type@{v} : Type@{u}`.

8.3 Cumulativity of Inductive Types Improvements

Improvements for cumulativity of universe polymorphic inductive types: the original feature, by Mattieu Sozeau and Amin Timany (#613[26]), is typically useful for "container"

[21] https://github.com/coq/coq/issues/11039
[22] https://github.com/coq/coq/pull/1033
[23] https://github.com/coq/coq/pull/8760
[24] https://github.com/coq/coq/pull/7495
[25] https://github.com/coq/coq/pull/8850
[26] https://github.com/coq/coq/pull/613

inductives types. Consider

```
Polymorphic Inductive Prod@{i} (A B : Type@{i})
  := pair : A -> B -> Prod A B.
```

Without the feature, two different universe instantiations of `Prod` (with the same term arguments), such as `Prod@{u} nat bool` and `Prod@{v} nat bool`, are not convertible. However we can trivially build an equivalence between them, and model arguments and presentations of CIC with anonymous inductive types justify making them convertible. Enabling such conversions in Coq is the goal of #613.

For `Prod`, the universe instantiation does not participate in conversion. Other types can introduce inequality constraints, for instance

```
Polymorphic Inductive Univ@{i}
  := univ : Type@{i} -> Univ.
```

We have `Univ@{u} <= Univ@{v}` if an only if `u <= v`.

As first implemented, the kernel would correctly accept the new conversions, but the elaboration engine on `fun x : Prod@{u} nat bool => x : Prod@{v} nat bool` would generate a constraint `u = v` unless that constraint was prevented by the ambiant universe constraints. In other words, the constrained transport

```
Polymorphic Definition Transport@{u v|u < v}
  : Prod@{u} nat bool -> Prod@{v} nat bool
  := fun x => x.
```

could be defined, as well as the opposite direction (with `v < u`), but

```
Polymorphic Definition Transport@{u v}
  : Prod@{u} nat bool -> Prod@{v} nat bool
  := fun x => x.
```

would infer `u = v`, and requiring the absence of constraints by using `Definition Transport@{u v|}` would be rejected.

I fixed this incompleteness in the elaboration #6775[27].

Originally, the cumulativity information for an inductive type would be stored as a pair of universe instantiations of the inductive together with the constraints required for a subtyping to hold (i.e. in the way I explained it for `Univ`). I simplified (#6128[28]) this based on the observation that these constraints always boil down to variance information for each universe argument (so for instance we never have `Ind@{u v} <= Ind@{u' v'}` from constraints relating different arguments like `u <= v'`, they must all be constraints on `u` and `u'` or on `v` and `v'`).

Thus for instance Coq will say

[27] https://github.com/coq/coq/pull/6775
[28] https://github.com/coq/coq/pull/6128

```
Prod@{i} :
Type@{i} -> Type@{i} -> Type@{i}
(* *i |=  *)

Prod is universe polymorphic
Arguments Prod _%type_scope _%type_scope
Expands to: Inductive Top.Prod

Univ@{i} : Type@{i+1}
(* +i |=  *)

Univ is universe polymorphic
Expands to: Inductive Top.Univ
```

`*u` means that universe `u` is irrelevant in conversion problems, `+u` means it is covariant. We can also have `=u` for invariant universes. Contravariant universes are not possible, because conversion is not contravariant in the domain of product types: `forall x : A, B <= forall x : A', B'` requires `A = A'` and `A' <= A` is not sufficient (to be compatible with the classical set model, justifying classical axioms).

8.4 Miscellaneous Coq Work

- work on cleaning up the primitive projection "compatibility layer", eg #7908[29]
- internal API work (attributes #8515[30], proof state handling #10215[31], etc for instance #9027[32] is a nice cleanup)
- Continuous Integration: for instance moving from Travis to Gitlab to take advantage of the artifact system #687[33], experimenting with Circle CI #6400[34], using Azure Pipelines for non-linux free testing #9215[35].

  At the time of this writing, the original Travis CI system and the Circle CI system were dropped (respectively due to being underpowered compared to Gitlab, and due to opacity of the free tier offering). We use principally Gitlab CI, with custom (Inria-owned) workers for Windows testing, and Azure Pipelines for free Windows and OSX testing (unlike the Gitlab workers, Azure gives 3rd party developers a way to run CI for Windows. It also requires less maintenance so we are considering fully switching Windows testing to Azure.)

[29] https://github.com/coq/coq/pull/7908
[30] https://github.com/coq/coq/pull/8515
[31] https://github.com/coq/coq/pull/10215
[32] https://github.com/coq/coq/pull/9027
[33] https://github.com/coq/coq/pull/687
[34] https://github.com/coq/coq/pull/6400
[35] https://github.com/coq/coq/pull/9215

CHAPTER

9

FUTURE PERSPECTIVES

9.1 Practical Concerns

The Coq implementation has some issues:

- The VM and "native" conversion decision procedures in the kernel ignore proof irrelevance, making them incomplete (but still sound). This needs some design work to tag function domains and/or free variables with their relevance. For instance, in "native" mode functions are represented by OCaml functions, so extracting information without applying them is difficult.
- Most of the Coq system still assumes cumulativity, leading to various issues described in *Issues with non-cumulativity* (page 89). We have some ideas for how to progress (by introducing a separation between universe variables and meta-variables) but it's unclear how difficult this will be. Hopefully this will allow us to improve usability in other ways, for instance allowing to declare Box as an implicit coercion from SProp to other types.
- Tactic support will also need improvement, for instance setoid rewriting along relations in SProp is currently impossible since all the lemmas are stated with relations in Prop.
- Bugs are certainly lurking in the implementation, an overlook in De Bruijn index lifting has already been found leading to a proof of `False` (now fixed) (issue #10904[36]).

  In the future perhaps the MetaCoq [SAB+19] project will help prevent such issues?

[36] https://github.com/coq/coq/issues/10904

9.2 Usage of SProp

So far SProp has been used in small scale experiments with type theory in type theory, for instance an (unpublished) implementation of the setoid model by Simon Boulier.

In general we can expect 2 classes of uses for SProp:

- Small improvements from being able to replace axioms like UIP by proofs, or better computation in the presence of axioms (for instance function extensionality breaks canonicity even with definitional UIP, but definitional UIP means that if a use of function extensionality reduces to an identity it will still compute). For instance [SdM16] (congruence closure with dependent functions in Lean) make use of UIP, but axiomatic UIP would also work.
- Developments which deeply use proof irrelevance, which means they also deeply use dependent types. Use of UIP is likely to feature heavily. For instance it should be possible to formalize Two Level Type Theory [ACK17] using equality in SProp for strict equality.

In general we probably have much to learn from the Lean world, which has experience with definitional proof irrelevance. It will be interesting to see the impact of the different choices we have made regarding decidability of conversion with the irrelevant universe.

BIBLIOGRAPHY

[AC19] Andreas Abel and Thierry Coquand. Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality. 2019. arXiv:1911.08174v1[37].

[AS12] Andreas Abel and Gabriel Scherer. On Irrelevance and Algorithmic Equality in Predicative Type Theory. *Logical Methods in Computer Science*, 03 2012. URL: http://dx.doi.org/10.2168/LMCS-8(1:29)2012, doi:10.2168/lmcs-8(1:29)2012[38].

[AOV17] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of Conversion for Type Theory in Type Theory. *Proc. ACM Program. Lang.*, 2(POPL):23:1–23:29, December 2017. URL: http://doi.acm.org/10.1145/3158111, doi:10.1145/3158111[39].

[ACD+18] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient Inductive-Inductive Types. *Foundations of Software Science and Computation Structures*, pages 293–310, 2018. URL: http://dx.doi.org/10.1007/978-3-319-89366-2_16, doi:10.1007/978-3-319-89366-2_16[40].

[AK16] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. *ACM SIGPLAN Notices*, 51(1):18–29, Jan 2016. URL: http://dx.doi.org/10.1145/2914770.2837638, doi:10.1145/2914770.2837638[41].

[37] https://arxiv.org/abs/1911.08174v1
[38] https://doi.org/10.2168/lmcs-8(1:29)2012
[39] https://doi.org/10.1145/3158111
[40] https://doi.org/10.1007/978-3-319-89366-2_16
[41] https://doi.org/10.1145/2914770.2837638

[ACK17] Danil Annenkov, Paolo Capriotti, and Nicolai Kraus. Two-Level Type Theory and Applications. *CoRR*, 2017. URL: http://arxiv.org/abs/1705.03307, arXiv:1705.03307[42].

[AB04] Steven Awodey and Andrej Bauer. Propositions As [Types]. *J. Log. and Comput.*, 14(4):447–471, August 2004. URL: https://doi.org/10.1093/logcom/14.4.447, doi:10.1093/logcom/14.4.447[43].

[Bar99] Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7, 1999.

[BDG11] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full Reduction at Full Throttle. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of Lecture Notes in Computer Science, 362–377. Springer, 2011. URL: http://dx.doi.org/10.1007/978-3-642-25379-9_26, doi:10.1007/978-3-642-25379-9_26[44].

[BPT17] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Certified Programs and Proofs (CPP 2017)*, 182 – 194. Paris, France, January 2017. URL: https://hal.inria.fr/hal-01445835, doi:10.1145/3018610.3018620[45].

[BMM04] Edwin Brady, Conor McBride, and James McKinna. Inductive Families Need Not Store Their Indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, 115–129. Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Car19] Mario Carneiro. *The Type Theory of Lean*. PhD thesis, Carnegie Mellon University, 2019. URL: https://github.com/digama0/lean-type-theory/releases/download/v1.0/main.pdf.

[CD18] Jesper Cockx and Dominique Devriese. Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. *Journal of Functional Programming*, 28:e12, 2018. doi:10.1017/S095679681800014X[46].

[CDP14] Jesper Cockx, Dominique Devriese, and Frank Piessens. Pattern matching without K. In *ACM SIGPLAN Notices*, volume 49, 257–268. ACM, 2014.

[Coq16] Thierry Coquand. Universe of bishop sets. www.cse.chalmers.se/ coquand/bishop.pdf, 2016.

[Cre91] Pierre Crégut. *Machines à environnement pour la réduction symbolique et l'évaluation partielle*. PhD thesis, Université Paris 7, 1991.

[dMKA+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). *Lecture*

[42] https://arxiv.org/abs/1705.03307
[43] https://doi.org/10.1093/logcom/14.4.447
[44] https://doi.org/10.1007/978-3-642-25379-9_26
[45] https://doi.org/10.1145/3018610.3018620
[46] https://doi.org/10.1017/S095679681800014X

Notes in Computer Science, pages 378–388, 2015. URL: http://dx.doi.org/10.1007/978-3-319-21401-6_26, doi:10.1007/978-3-319-21401-6_26[47].

[For14] Fredrik Nordvall Forsberg. *Inductive-Inductive Definitions*. PhD thesis, Swansea University, 2014. URL: https://personal.cis.strath.ac.uk/fredrik.nordvall-forsberg/thesis/thesis.pdf.

[Gil17] Gaëtan Gilbert. Formalising Real Numbers in Homotopy Type Theory. In *6th ACM SIGPLAN Conference on Certified Programs and Proofs* , 112–124. Paris, France, 2017. URL: https://hal.inria.fr/hal-01449326, doi:10.1145/3018610.3018614[48].

[GCST19] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional Proof Irrelevance Without K. *Proc. ACM Program. Lang.*, 3(POPL):3:1–3:28, 2019. URL: http://doi.acm.org/10.1145/3290316.

[Gim95] Eduardo Giménez. Codifying guarded definitions with recursive schemes. *Types for Proofs and Programs*, pages 39–59, 1995. URL: http://dx.doi.org/10.1007/3-540-60579-7_3, doi:10.1007/3-540-60579-7_3[49].

[GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In Mitchell Wand and Simon L. Peyton Jones, editors, *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002.*, 235–246. ACM, 2002. URL: http://doi.acm.org/10.1145/581478.581501, doi:10.1145/581478.581501[50].

[Hed98] Michael Hedberg. A Coherence Theorem for Martin-Löf's Type Theory. *J. Funct. Program.*, 8(4):413–436, July 1998. URL: http://dx.doi.org/10.1017/S0956796898003153, doi:10.1017/S0956796898003153[51].

[Hof97] Martin Hofmann. *Extensional Constructs in Intensional Type Theory*. Springer London, 1997. ISBN 9781447109631. URL: http://dx.doi.org/10.1007/978-1-4471-0963-1, doi:10.1007/978-1-4471-0963-1[52].

[Hur95] Antonius J. C. Hurkens. A simplification of Girard's paradox. *Typed Lambda Calculi and Applications*, pages 266–278, 1995. URL: http://dx.doi.org/10.1007/bfb0014058, doi:10.1007/bfb0014058[53].

[Let04] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004.

[Miq01] Alexandre Miquel. The Implicit Calculus of Constructions Extending Pure Type Systems with an Intersection Type Binder and Subtyping. *Typed*

[47] https://doi.org/10.1007/978-3-319-21401-6_26
[48] https://doi.org/10.1145/3018610.3018614
[49] https://doi.org/10.1007/3-540-60579-7_3
[50] https://doi.org/10.1145/581478.581501
[51] https://doi.org/10.1017/S0956796898003153
[52] https://doi.org/10.1007/978-1-4471-0963-1
[53] https://doi.org/10.1007/bfb0014058

Lambda Calculi and Applications, pages 344–359, 2001. URL: http://dx.doi.org/10.1007/3-540-45413-6_27, doi:10.1007/3-540-45413-6_27[54].

[MW03] Alexandre Miquel and Benjamin Werner. The Not So Simple Proof-Irrelevant Model of CC. *Types for Proofs and Programs*, pages 240–258, 2003. URL: http://dx.doi.org/10.1007/3-540-39185-1_14, doi:10.1007/3-540-39185-1_14[55].

[Our05] Nicolas Oury. Extensionality in the Calculus of Constructions. *Theorem Proving in Higher Order Logics*, pages 278–293, 2005. URL: http://dx.doi.org/10.1007/11541868_18, doi:10.1007/11541868_18[56].

[Pfe01] Frank Pfenning. Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, LICS '01, 221–. Washington, DC, USA, 2001. IEEE Computer Society. URL: http://dl.acm.org/citation.cfm?id=871816.871845.

[SdM16] Daniel Selsam and Leonardo de Moura. Congruence Closure in Intensional Type Theory. *Lecture Notes in Computer Science*, pages 99–115, 2016. URL: https://arxiv.org/abs/1701.04391, doi:10.1007/978-3-319-40229-1_8[57].

[SAB+19] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. working paper or preprint, June 2019. URL: https://hal.inria.fr/hal-02167423.

[Stu18] Aaron Stump. From realizability to induction via dependent intersection. *Ann. Pure Appl. Logic*, 169(7):637–655, 2018. URL: https://doi.org/10.1016/j.apal.2018.03.002, doi:10.1016/j.apal.2018.03.002[58].

[UFP13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013.

[Voe13] Vladimir Voevodsky. A simple type system with two identity types. Unpublished note, 2013.

[Wer08] Benjamin Werner. On the Strength of Proof-Irrelevant Type Theories. *Logical Methods in Computer Science*, 4:1–20, 09 2008.

[WST19] Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. Eliminating Reflection from Type Theory. In *CPP 2019 - 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*, 91–103. Lisbonne, Portugal, January 2019. ACM. URL: https://hal.archives-ouvertes.fr/hal-01849166, doi:10.1145/3293880.3294095[59].

[54] https://doi.org/10.1007/3-540-45413-6_27
[55] https://doi.org/10.1007/3-540-39185-1_14
[56] https://doi.org/10.1007/11541868_18
[57] https://doi.org/10.1007/978-3-319-40229-1_8
[58] https://doi.org/10.1016/j.apal.2018.03.002
[59] https://doi.org/10.1145/3293880.3294095

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Une théorie des types avec insignifiance des preuves définitionnelle

Mot clés : Assistant de preuve, Coq, insignifiance des preuves, univers, UIP

Résumé : L'égalité définitionnelle, aussi appelée conversion, pour une théorie des types avec une vérification de type décidable est l'outil le plus simple pour prouver que deux objets sont les mêmes, laissant le système décider en utilisant uniquement le calcul. Par conséquent, plus il y a de choses égales par conversion, plus il est simple d'utiliser un langage basé sur la théorie des types. L'insignifiance des preuves, indiquant que deux preuves quelconques de la même proposition sont égales, est une manière possible d'étendre la conversion afin de rendre une théorie des types plus puissante. Cependant, ce nouveau pouvoir a un prix si nous l'intégrons naïvement, soit en rendant la vérification de type indécidable, soit en réalisant de nouveaux axiomes—tels que l'unicité des preuves d'identité (UIP)—qui sont incompatibles avec d'autres extensions, comme l'univalence. Dans cette thèse, nous proposons un moyen général d'étendre une théorie des types avec l'insignifiance des preuves, de manière à ce que la vérification du type soit décidable et est compatible avec l'univalence. Nous fournissons un nouveau critère pour décider si une proposition peut être éliminée sur un type (en corrigeant et en améliorant la règle d'élimination des singletons de Coq) en utilisant des techniques provenant de développements récents du filtrage dépendant sans UIP. Nous fournissons aussi une preuve de la décidabilité du typage à base de relations logiques. Cette extension de la théorie des types a été implementée dans les assistants de preuve Coq et Agda.

Title: A type theory with definitional proof-irrelevance

Keywords: Proof assistant, Coq, proof-irrelevance, universes, UIP

Abstract: Definitional equality, a.k.a 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 thesis, 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 decidability of type checking using logical relations. This extension of type theory has been implemented both in Coq and Agda.