université PARIS-SACLAY

Translating Proofs from Lean to Dedukti

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n° 580, Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat: Informatique
Unité de recherche: Deducteam

Thèse présentée et soutenue à ENS Paris-Saclay,
le 10 Mars 2026, par

Rishikesh VAISHNAV

Composition du jury

Herman GEUVERS Professor, Radboud University Nijmegen	Rapporteur
Nicolas TABAREAU Directeur de recherche, Inria	Rapporteur
Christine PAULIN Professeure, Université Paris-Saclay	Examinatrice
Jesper COCKX Maître de conférences, Delft University of Technology	Examinateur
Sebastian ULLRICH Docteur, Lean FRO	Examinateur
Mario CARNEIRO Docteur, Chalmers University of Technology	Examinateur

Direction de la thèse

Frédéric BLANQUI	Directeur

Thèse de doctorat

NNT: 2020UPASA000

Contents

Chapter 1

Introduction

This thesis concerns the topic of proof translation between proof assistants, specifically the task of translating proofs expressed in the proof assistant Lean [27] to an intermediate logical framework known as Dedukti [10]. In this first chapter, we will start with a broad introduction to formal methods and our translation task, followed by a description of the theories that we aim to translate between and a high-level overview of our translation plan.

1.1 Formal Methods and Proof Translation

The area of computer science known as "formal methods" broadly concerns the specification, development, and verification of computer software/hardware, cyber-physical systems, and formal mathematical theories. Formal methods often involve the use of programs and/or well-established procedures to verify the correctness of some structured representation of a particular system (be it a program, hardware description, formal mathematical proof, etc.) according to a user-provided specification. This verification can be done through automated means, user-directed proof steps, or a combination of the two. Formal methods have applications particularly in the verification of correctness properties of safety-critical software and hardware systems. More recently, formal methods have also seen renewed interest for use in guiding and verifying machine learning systems through the rigorous symbolic interpretation and formal verification of the input-output representations used by artificial intelligence agents.

Proof Assistants

A particular application of formal methods is in the use of software tools to both verify and guide the construction of formal proofs of mathematical theorems. Such tools are known as "proof assistants" or "interactive theorem provers", with most being based on the use of "kernel" programs that verify that "proof terms" expressed in a particular syntax correctly prove mathematical theorem statements. Proof assistant kernel implementations can be based on a variety of theoretical foundations, with many of them using certain extensions of type theory that exploit what is known as the Curry-Howard correspondence [24] between programming language types and theorem statements. Starting from a base kernel implementation, many proof assistants feature a stack of tooling to ease the process of formal mathematical proof, including implicit arguments, interactive proof state inspection, library search tools, proof

automation tactics and meta-programming facilities, as well as more advanced forms of automation and proof search algorithms.

Proof assistants provide particular value to mathematicians in their ability to keep explicit track of proof obligations and the set of current assumptions/derived properties in an evolving "proof state" during the course of proving a mathematical theorem. This allows the user to essentially "offload" the responsibility keeping track of all of these details onto the proof assistant, greatly reducing the chance of errors and providing rigorous correctness requirements and guarantees that contribute highly to the trust that can be placed in the truth of the formally proven mathematical theorem. Proof assistants also present some novel opportunities in the area of automated reasoning and mathematical discovery, as the feedback that they provide can be used to guide the automated search for provable theorems deriving from foundational axioms and mathematical objects.

Proof Translation: Motivations and Challenges

With the wide variety of existing proof assistants and independent formalization efforts being made in different systems, interoperability becomes a major concern. It would be highly desirable to avoid duplicating work between libraries that have been formalized in different systems – ideally, as soon as a mathematical theory has been formalized in one proof assistant, it should be possible to immediately export (i.e., translate) it to other systems through fully automated means, in a way that not only preserves the computational content of mathematical definitions and proofs, but also produces a translated output that can be independently verified by the target system. Additionally, having reliable translation tools between different systems would grant greater freedom of choice to users in terms of which proof assistant they can use for their particular formalization task, providing them with the confidence that their formal results will remain relevant beyond the scope of their chosen system.

Beyond just these practical benefits, however, having a complete translation could also be interesting from a proof-theoretical perspective. Firstly, translating a mathematical library to another system, with the translation being successfully typechecked by that system's independently implemented kernel, would afford more trust in the library's formal results, in addition to improving our confidence in the respective kernel implementations. On the other hand, if a proof is deemed correct in the source system but has a translation that is considered incorrect by the target system, this could highlight the existence of bugs in the implementation of the kernels of either system (assuming the correctness of the translation itself). Having a translation between different proof assistants that is additionally *formally verified* would also open some interesting possibilities around formal meta-theoretical analyses of the respective systems, as it may enable certain meta-theoretical results (such as consistency) to be transferred between them.

The actual design and implementation of such translations between proof assistants is often a highly involved and exacting task. While proof assistants often share in many aspects of their theoretical foundations, they often also exhibit significant differences that unfortunately make it difficult to share formal results between them. These differences may exist on the syntactic level – that is, differences in the possible ways in which proof terms and mathematical statements and definitions may be formed, according to the low-level program syntax used by the proof assistant – and also at the level of the particular theories underlying kernel implementations. These theories may be categorically distinct – for instance, proof

assistants may be based on dependent type theory (for instance, the Rocq [31] proof assistant), set theory (for instance, Mizar [42]), higher-order logic (for instance, Isabelle/HOL [28]), and a number of other possible theoretical foundations (and combinations thereof). Such fundamental differences obviously greatly hinder interoperability between these systems. However, even amongst theories with closely aligned foundations, differences may exist that make the task of exporting proofs between systems highly non-trival.

In order to implement a correct translation between systems, we need to consider all of the differences between them and design a translation that "reconciles" them in some way. In many cases, there are similar features between systems that can be used to emulate one another to some extent. In other cases, the differences between the systems are so great that a complete translation is not possible (or, only possible with the addition of certain axioms or additional kernel features in the target system). Such a case can arise in particular when systems fundamentally differ in their expressive power, with one system able to prove properties that the other cannot (and vice versa). As such, translation should only be attempted between "compatible" systems, with a theoretical analysis of translation feasibility (or, one that identifies a subset of feasibly translatable proofs from the source system) preceding the actual design and implementation of a translation.

Our Task: Translating from Lean to Dedukti

This thesis describes the translation from the proof assistant Lean to the Dedukti logical framework, with the eventual goal of enabling export from Dedukti of these translated proofs to several different proof assistants. Lean [27] is a proof assistant developed by the Lean FRO and designed for formalizing general mathematics, with a large and quickly growing library of formal mathematics known as Mathlib [38]. Lean features a rich standard library, along with extensive tooling for user interactivity, proof automation, and metaprogramming. Lean has become especially popular amongst professional mathematicians, with several advanced/fundamental theoretical results in mathematics having been formalized in Lean in recent years.

Dedukti [10] is a logical framework developed specifically for proof system interoperability with a minimal type theory that is based on the $\lambda\Pi$ calculus with rewrite rules. Dedukti allows for the definition of various rewrite system encodings within it corresponding to various different type theories, with translation from one proof assistant to another via Dedukti generally taking the following three steps: first, proofs are translated to Dedukti via a translation from the source system to Dedukti, assuming some encoding of the source type theory within Dedukti. Then, translation is performed from the source theory encoding to the target theory encoding within Dedukti. Finally, the translated proofs in the target theory's encoding are exported from Dedukti to the target system[1].

This centralized approach has benefits over directly translating between the different systems as it greatly cuts down on the number of translations that need to be implemented, as we can focus solely on translating each system to and from Dedukti. This is preferable to the approach of implementing a translation between every possible pair of systems, which would involve a quadratic number of translations. It is also better than the approach of implementing a linear chain of translations, which would accumulate large overheads as we translate across multiple systems. While our approach still requires us to consider the trans-

[1]Some existing export tools have been implemented from Dedukti to PVS [36], Matita [3], Rocq [31], OpenTheory [25, 40], and Agda [11, 16]

lation between different theory encodings within Dedukti itself, this is attenuated somewhat (relative to the direct translation approach) thanks to the fact that we are working with a single framework and a unified syntax. Additionally, general tools for translating between different Dedukti theories (following, for instance, recent work by Traversié [41]) can greatly reduce this burden wherever they are applicable.

Both Lean and Dedukti enable the typechecking of proofs through the propositions-as-types principle, and they share some type-theoretic foundations, both being based on extensions of Church's simply typed λ-calculus with dependent function types. However, their theories differ in a number of ways that we will have to account for when defining our translation. Below, we will provide a mostly complete presentation of both Lean and Dedukti's type theories that will serve as a useful reference to us as we go about defining our translation from Lean to Dedukti in the later chapters.

1.2 Lean's Type Theory

Let's start with a description of Lean, the source theory of our translation. Lean's type theory takes closely after that of the Rocq proof assistant [31], being based based on the Calculus of Inductive Constructions with an infinite universe hierarchy and an impredicative universe of propositional types. However, it has a number of unique features of its own, most of which serve as user conveniences that enable it to be practical for large-scale formalizations. Let's lay out a full formal presentation of Lean's type theory below. Our presentation below will be quite similar to that of Carneiro [13], who provided the first formal description of Lean's type theory w.r.t. the implementation of the Lean 3 kernel[2].

The Calculus of Constructions with Inductive Types

Lean's type theory is based on the Calculus of Inductive Constructions (CIC) [30], which is a theory that allows for general dependent types and the declaration of expressive inductive types representing mathematical objects, along with their associated elimination principles. CIC is based on the Calculus of Constructions, which enables the representation of higher-order intuitionistic logic via the "propositions-as-types" principle, as was first described by Coquand and Huet [14].

Core to CIC is the use of dependent types, which allows function types to have codomains with dependencies on previously bound domain variables. For instance, if we have a type `Vec : Type → Nat → Type` corresponding to the type of a list with a specified number of elements of a specified type, we could describe the following function type:

```
def duplicate : {T : Type} → (n : Nat) → T → Vec T n := ...
```

whose output type `Vec T n` depends on previously bound domain variables `T` and `n`. CIC is made usable as a system for formal proof through the "propositions-as-types" principle (also known as the Curry-Howard correspondence), in which types belonging to a designated propositional type universe (referred to in Lean as `Prop` or `Sort 0`) can be interpreted as logical propositions, with terms inhabiting these types being interpretable as proofs. For instance, we can prove that any proposition implies itself as follows:

[2]The most recent version of Lean, Lean 4, has a kernel with a few additional features which we will have to account for in our translation – namely, those of struct-η, struct-like reduction, and unit-η.

```
theorem tauto (P : Prop) : P → P := fun p => p
```

We provide the proof as a λ-function which can be thought of as a `Prop`-specific identity function, taking in a proof of `P` and returning that same proof. The function type `P → P`, which lives in `Prop`, can be corresponded to an implication, with a proof of this implication corresponding to a λ-abstraction. This interpretation of types as propositions and terms as proofs is central to the use of CIC as a system of formal proof.

Inductive Types

Lean's rich expressivity can largely be attributed to its support for the declaration of user-defined "inductive types", which enable the definition of mathematical objects as specific Lean types. These inductive types are assigned a set of "constructors" that can be used to construct instances of the inductive type. For instance, the natural numbers in Lean are defined as follows:

```
inductive Nat where
  | zero : Nat
  | succ (n : Nat) : Nat
```

Here, the constructor `Nat.zero` represents the natural number zero, and the recursive constructor `Nat.succ` represents the successor of another natural number (with the number one being represented as `Nat.succ Nat.zero`). These inductive types generate associated elimination symbols, referred to in Lean as "recursors", that enable the definition of functions that operate on the contents of inductive type constructions. For instance, Lean generates the following recursor for `Nat`:

```
-- Nat.rec.{u} {motive : Nat → Sort u}
--   (zero : motive Nat.zero) (succ : (n : Nat) → motive n → motive n.succ)
--   (t : Nat) : motive t
#check Nat.rec
```

This recursor can be used to define, for instance, an addition function:

```
def Nat.add (n m : Nat) : Nat :=
  Nat.rec n (fun _ prec => Nat.succ prec) m
```

If we instantiate the recursor's `motive` argument with some predicate `P : Nat → Prop`, this recursor corresponds to the inductive principle on natural numbers:

```
axiom P : Nat → Prop
-- Nat.rec : P Nat.zero → ((n : Nat) → (ih : P n) → P n.succ) → (t : Nat) → P t
#check Nat.rec (motive := P)
```

The type above indicates that to prove that `P n` holds for all `Nat` instances `n`, we must show that it holds in the `Nat.zero` case and the `Nat.succ` case, where the successor case is provided with the inductive hypothesis `ih`.

Inductive types can also have parameters and indices, which are used to define parameterized "families" of inductive types. Consider, for instance, the following inductive type declaration corresponding to the type `Vec` shown above:

```
inductive Vec (T : Type) : Nat → Type where
| nil  : Vec T Nat.zero
| cons : (n : Nat) → Vec T n → T → Vec T (Nat.succ n)
```

Here, the inductive type constant `Vec` itself (a.k.a., the type constructor of the inductive type) takes one parameter for the type of the elements of the vector, and one index for the number of elements in the vector. The difference between parameters and indices is that parameters are specified as the initial arguments to the constructors, whereas indices are fully determined by the constructor itself.

`Vec` has two constructors, `Vec.nil` for declaring an empty vector and `Vec.cons` for appending an element to the end of a vector. `Vec.cons` takes another instance of `Vec` as one of its arguments, making `Vec` what we call a "recursive" inductive type. Note that the output type of a constructor's function type must be the inductive type we are currently declaring. There are also some additional restrictions placed on the form of constructor function types (e.g. the positivity requirement), though we do not cover them here[3].

Lean also allows for the definition of more complex inductive types, such as mutual inductive types, which is a generalization of recursive inductive types where one inductive type is defined in terms of a second inductive type, which is in turn defined in terms of the first one. For instance, the following mutual inductive types encode the property of a natural number being even or odd:

```
mutual
  inductive Even : Nat → Prop where
  | zero : Even Nat.zero
  | succ : {n : Nat} → Odd n → Even (Nat.succ n)

  inductive Odd : Nat → Prop where
  | succ : {n : Nat} → Even n → Odd (Nat.succ n)
end
```

Term and Type Context Syntax

Lean terms are taken from the following grammar:

$$
\begin{aligned}
&S_{\mathrm{L}} = t \text{ where} \\
&\quad t ::= x \mid c.\{\ell_1, \ldots, \ell_n\} \mid \texttt{Sort}\ \ell \mid t_1\ t_2 \mid \texttt{fun}\ (x : t_1)\ \texttt{=>}\ t_2 \mid (x : t_1) \to t_2 \mid \\
&\qquad \texttt{let}\ (x : A)\ :=\ v\ \texttt{in}\ b \mid t.i \\
&\quad \ell ::= u \mid \texttt{z} \mid \texttt{s}\ \ell \mid \texttt{max}\ \ell_1\ \ell_2 \mid \texttt{imax}\ \ell_1\ \ell_2
\end{aligned}
$$

where $i \in \mathbb{N}$, $x \in \mathcal{X}$, $u \in \mathcal{U}$, and $c \in \mathcal{C}$, with $\mathcal{X}$ denoting a set of bound variable names, $\mathcal{U}$ being a set of universe level parameter symbols, and $\mathcal{C}$ denoting a set of constant names (with $\mathcal{X}$, $\mathcal{U}$, and $\mathcal{C}$ disjoint). A Lean term can either be a bound variable reference x, a level-instantiated constant reference $c.\{\ell_1, \ldots, \ell_n\}$, a type universe, a.k.a. "sort" `Sort` ℓ, an application $t_1\ t_2$, a λ-function `fun` $(x : t_1)$ `=>` t_2, a dependent function type $(x : t_1) \to t_2$, a let binding, a.k.a. "local definition" `let` $(x : A)\ :=\ v$ `in` b, or a structure projection expression $t.i$. Universe level expressions are taken from the grammar ℓ, and can be a universe level

[3] See here for the exact specifications on the declaration of inductive types in Lean:
`https://lean-lang.org/doc/reference/latest/The-Type-System/Inductive-Types/`

parameter reference u, the zero universe level `z`, the successor of some other universe level `s` ℓ, the maximum of two universe levels `max` ℓ_1 ℓ_2, or the impredicative maximum of two universe levels `imax` ℓ_1 ℓ_2. A formal interpretation of universe level constructions is provided in Section 1.2.2.

In our presentation of Lean's type theory, we will also make use of a grammar for typing contexts, denoted by the symbol Δ, defined as follows:

$$
\begin{aligned}
C_{\mathrm{L}} &= \Delta \text{ where} \\
\Delta &::= (\Sigma; \Gamma) \\
\Gamma &::= (\Gamma_U; \Gamma_B) \\
\Sigma &::= () \mid (\Sigma, \sigma_C) \\
\Gamma_U &::= () \mid (\Gamma_U, u) \\
\Gamma_B &::= () \mid (\Gamma_B, x : t) \mid (\Sigma_R, x : t_1 := t_2)
\end{aligned}
$$

A Lean typing context consists of two components: a global typing context Σ, and a local typing context Γ. The global context is a list of constant declarations, with the symbol σ_C representing the syntax category of constant declarations. A Lean constant may be declared an inductive type, definition, theorem, axiom, or quotient type. For simplicity's sake, we do not provide a fully formal description of the constant declaration grammar here. The local context Γ consists of a universe level context Γ_U, together with a bound variable context Γ_B. The universe level context is simply a list of named universe level parameters that can be introduced by constant declarations, while the bound variable context is a list of variables along with their associated types (in the case of a bound variable being introduced by a λ-function or dependent function type), and possibly a value as well (in the case of a bound variable being introduced by a `let`-binding). For some typing context $\Delta = (\Sigma; (\Gamma_U; \Gamma_B))$, we will use the shorthand $\Delta, x : T$ to mean $(\Sigma; (\Gamma_U; (\Gamma_B, x : T)))$, as well as the shorthand $\Delta, x : T := v$ to mean $(\Sigma; (\Gamma_U; (\Gamma_B, x : T := v)))$, to append binders to the local context. We will also use the shorthand notation c, Δ to mean $((c :: \Sigma); \Gamma)$ (where $c :: \Sigma$ indicates *prepending* c to the list of constant declarations Σ).

1.2.1 Type Inference Rules

To describe Lean's type theory, we will use the syntax $\Delta \vdash t : T$ as our type inference judgment, read as "term t has type T in typing context Δ". The notation $t[a/x]$ denotes the substitution of the bound variable x in t by the value a. The full set of type inference rules in Lean are as follows:

$$\frac{\Delta \vdash f : (x : A) \to B \quad \Delta \vdash e : A}{\Delta \vdash f\ e : B[e/x]}\ [\text{APP}] \qquad \frac{\Delta \vdash A, B : \texttt{Sort}\ \ell \quad \Delta \vdash A \equiv B \quad \Delta \vdash t : A}{\Delta \vdash t : B}\ [\text{CONV}]$$

$$\frac{\Delta \vdash A : \texttt{Sort}\ \ell \quad \Delta, x : A \vdash e : B}{\Delta, x : A \vdash \texttt{fun}\ (x : A)\ \texttt{=>}\ e : (x : A) \to B}\ [\text{LAM}] \qquad \frac{\Delta \vdash A : \texttt{Sort}\ \ell \quad \Delta, x : A \vdash B : \texttt{Sort}\ \ell'}{\Delta \vdash (x : A) \to B : \texttt{Sort}\ (\texttt{imax}\ \ell\ \ell')}\ [\text{ALL}]$$

$$\frac{\Delta \vdash A : \texttt{Sort}\ \ell \quad \Delta \vdash e : A \quad \Delta, x : A := e \vdash b : B}{\Delta \vdash \texttt{let}\ (x : A)\ :=\ e\ \texttt{in}\ b : B\ [x/e]}\ [\text{LET}]$$

$$\frac{}{\Delta \vdash \texttt{Sort}\ \ell : \texttt{Sort}\ (\texttt{s}\ \ell)}\ [\text{SORT}] \qquad \frac{\Delta \vdash A : \texttt{Sort}\ \ell}{\Delta, x : A \vdash x : A}\ [\text{VAR}]$$

$$\frac{S \text{ structure-like} \quad \Delta \vdash t : S\ p'_1\ \ldots\ p'_n \quad \Delta \vdash S.\texttt{mk} : (p_1 : P_1) \to \ldots \to (p_n : P_n) \to (a_1 : A_1) \to \ldots \to (a_m : A_m)}{\Delta \vdash t.i : A_i\ [p_1/p'_1,\ \ldots,\ p_n/p'_n,\ a_1/t.1,\ \ldots,\ a_{i-1}/t.(i-1)]}\ [\text{PROJ}]$$

The rule [APP] describes the typing of applications, whose type is determined as an instantiation of the possibly dependent codomain type of the application head's function type with the application argument. The rule [LAM] determines the type of a λ-function, whose type is a function type whose domain is the annotated binder type of the λ-function, and whose codomain is the inferred type of the λ-function body. The rule [ALL] states that a function type inhabits a type universe whose index is the `imax` of the domain and codomain universe indices (the semantics of `imax` are described below). The rule [LET] concerns the typing of let-bindings, which are typed as the inferred type of the body of the let-binding, with the let-bound variable instantiated with the bound value. The rule [PROJ] defines the typing of projections, applied to terms of structure-like types (defined below). The type of a projection is the type of the field of the unique structure constructor at the specified projection index, with the parameters and previous field values instantiated within it according to the inferred type of the structure term and projections of the previous fields. The rule [SORT] expresses Lean's infinite universe hierarchy, in which each sort is typed within its successor sort. The bottommost sort, `Sort 0`, abbreviated `Prop`, represents propositional types ("propositions" for short), with higher-level sorts being used for more "standard" types representing mathematical objects, as well as for higher-order function types. The need for such an infinite universe hierarchy can be attributed to Russell's paradox [32], a result from set theory which demonstrates how inconsistencies can arise from circular set membership rules[4].

Type Conversion and Definitional Equality

Lastly, Lean's [CONV] rule expands the class of types that a term could feasibly have by allowing a term to type as any type that can be identified with its normally inferred type[5] according to Lean's "definitional equality" judgment. We use the notation $\Delta \vdash t \equiv s$ to denote this judgment, stating that the terms t and s are considered definitionally equal in context Δ.

The rule [CONV] is quite powerful, as it enables a derivation to effectively "swap out" the normally inferred type of a term with another one, as long as it can determine that the new type is equal to the old one according to some externally defined (and possibly extensible) judgment allowing for identities that extend beyond just syntactic equivalence. This is particularly relevant for rules where certain typings are enforced, allowing them to be used more broadly, thus greatly improving the expressive capability of the type system and allowing more terms to be typeable.

For instance, recall the rule [APP]:

$$\frac{\Delta \vdash f : (x : A) \to B \quad \Delta \vdash e : A}{\Delta \vdash f\ e : B[e/x]}\ [\text{APP}]$$

This rule requires the function domain type to match the type of the argument, which is enforced in the rule by using the same symbol A for both types. Making use of [CONV],

[4] More directly applicable to Lean, there is a version of Russell's paradox adapted for type theory that is known as Girard's Paradox [20], a Lean formalization of which can be found here:
`https://leanprover-community.github.io/mathlib4_docs/Counterexamples/Girard.html`

[5] That is, the typing that can be derived for the term without any top-level uses of the rule [CONV].

however, we can relax this requirement, deriving the following equivalent rule:

$$\frac{\Delta \vdash f : T \quad \Delta \vdash T \equiv (x : A) \to B \quad \Delta \vdash e : U \quad \Delta \vdash U \equiv A}{\Delta \vdash f\ e : B[e/x]}\ \text{[APP']}$$

Here, we introduce a new type symbol T that we assert to be definitionally equal to $(x : A) \to B$, as well as U, that we assert to be definitionally equal to A. We can see that this new set of premises allows us to derive the original ones via [CONV], which then allows us to prove the same typing via [APP]:

$$\frac{\dfrac{\Delta \vdash f : T \quad \Delta \vdash T \equiv (x : A) \to B}{\Delta \vdash f : (x : A) \to B} \quad \dfrac{\Delta \vdash e : U \quad \Delta \vdash U \equiv A}{\Delta \vdash e : A}}{\Delta \vdash f\ e : B[e/x]}$$

In general, thanks to [CONV] we can always replace a premise of the form $\Delta \vdash t : T$ with the premises $\Delta \vdash t : T'$, $\Delta \vdash T \equiv T'$. For instance, the following rule is equivalent to [ALL]:

$$\frac{\Delta \vdash A : T \quad \Delta \vdash T \equiv \texttt{Sort}\ \ell \quad \Delta, x : A \vdash B : U \quad \Delta, x : A \vdash U \equiv \texttt{Sort}\ \ell'}{\Delta \vdash (x : A) \to B : \texttt{Sort}\ (\texttt{imax}\quad \ell\ \ell')}\ \text{[ALL']}$$

[CONV] is especially useful when it comes to user-provided type annotations, e.g. those appearing in the rule [LET], which we can equivalently state as:

$$\frac{\Delta \vdash A : T \quad \Delta \vdash T \equiv \texttt{Sort}\ \ell \quad \Delta \vdash e : U \quad \Delta \vdash U \equiv A \quad \Delta, x : A \coloneqq e \vdash b : B}{\Delta \vdash \texttt{let}\ (x : A)\ :=\ e\ \texttt{in}\ b : B\ [x/e]}\ \text{[LET']}$$

Here, we have relaxed the premise typing requirement on the value expression e to allow for any type definitionally equal to the annotated type. This affords the user of the proof assistant much more freedom in how they posit the type of the let value, as it is sometimes the case that types that are automatically inferred by type inference routines have large and complex forms, while being definitionally equivalent to more natural and intuitive representations that would be expressed by a user.

So, as we can see from the [CONV]-derived rules above, the presence of [CONV] in our type theory doesn't just allow for alternate typings of fixed terms – it also greatly expands the class of terms which are typable in our theory to begin with, allowing for a more expressive type theory that makes for more convenient mathematical formalization. Of course, the extent of this additional expressivity is highly dependent on the specifics of the definitional equality judgment, which we describe below.

1.2.2 Definitional Equality Rules

The particular rules constituting Lean's definitional equality judgment are as follows:

$$\frac{}{\Delta \vdash t \equiv t}\ \text{[RFL]} \qquad \frac{\Delta \vdash t \equiv s}{\Delta \vdash s \equiv t}\ \text{[SYMM]}$$

$$\frac{\Delta \vdash f \equiv f' \quad \Delta \vdash a \equiv a'}{\Delta \vdash f\ a \equiv f'\ a'}\ \text{[CGR-APP]} \qquad \frac{\Delta \vdash A \equiv A' \quad \Delta, x : A \vdash e \equiv e'}{\Delta \vdash \texttt{fun}\ (x : A)\ \texttt{=>}\ e \equiv \texttt{fun}\ (x : A')\ \texttt{=>}\ e'}\ \text{[CGR-LAM]}$$

$$\frac{\Delta \vdash A \equiv A' \quad \Delta, x : A \vdash B \equiv B'}{\Delta \vdash (x : A) \to B \equiv (x : A') \to B'}\ \text{[CGR-ALL]} \qquad \frac{\Delta \vdash t \equiv s}{\Delta \vdash t.i \equiv s.i}\ \text{[CGR-PROJ]}$$

$$\frac{\ell_1 \approx \ell'_1 \quad \cdots \quad \ell_n \approx \ell'_n}{\Delta \vdash C.\{\ell_1, \ldots, \ell_n\} \equiv C.\{\ell'_1, \ldots, \ell'_n\}}\ [\text{CGR-CONST}] \qquad \frac{\ell \approx \ell'}{\Delta \vdash \texttt{Sort}\ \ell \equiv \texttt{Sort}\ \ell'}\ [\text{CGR-SORT}]$$

$$\frac{\Delta \vdash P : \texttt{Prop} \quad \Delta \vdash p, q : P}{\Delta \vdash p \equiv q}\ [\text{PI}] \qquad \frac{}{\Delta \vdash (\texttt{fun}\ (x : A)\ \texttt{=>}\ e\ x) \equiv e}\ [\text{FUN-ETA}]$$

$$\frac{S \text{ struct-like} \quad \Delta \vdash t : S\ p_1\ \ldots\ p_n \quad \Delta \vdash t.1 \equiv a_1 \quad \ldots \quad \Delta \vdash t.m \equiv a_m}{\Delta \vdash t \equiv S.\texttt{mk}\ p_1\ \ldots\ p_n\ a_1\ \ldots\ a_m}\ [\text{ETA-S}]$$

$$\frac{U \text{ unit-like} \quad \Delta \vdash t, s : U}{\Delta \vdash t \equiv s}\ [\text{UNIT}] \qquad \frac{\Delta \vdash t \rightsquigarrow^* t' \quad \Delta \vdash t' \equiv s}{\Delta \vdash t \equiv s}\ [\text{RED}]$$

Each rule also implicitly carries an additional premise requiring the well-typedness of the terms in the left- and right-hand sides of the conclusion judgment; for the sake of brevity, we omit these premises in our presentation of the rules above. The rule [RFL] ensures that the definitional equality relation is reflexive, while [SYMM] ensures that it is symmetric (enabling the asymmetric definitional equality rules to apply in both directions). The rules [CGR-APP], [CGR-LAM], [CGR-ALL], and [CGR-PROJ] are "congruence identities" that allow two terms with the same root syntax to be identified on basis of the definitional equality of their corresponding subterms.

The Universe Level Equality Judgment

The congruence identities [CGR-CONST] and [CGR-SORT] are defined using a judgment for the equivalence of universe level terms, which we denote using the syntax $\ell \approx \ell'$:

$$\frac{\ell_1 \approx \ell'_1 \quad \cdots \quad \ell_n \approx \ell'_n}{\Delta \vdash C.\{\ell_1, \ldots, \ell_n\} \equiv C.\{\ell'_1, \ldots, \ell'_n\}}\ [\text{CGR-CONST}] \qquad \frac{\ell \approx \ell'}{\Delta \vdash \texttt{Sort}\ \ell \equiv \texttt{Sort}\ \ell'}\ [\text{CGR-SORT}]$$

This universe level judgment is defined in terms of an interpretation function $\texttt{eval}_\sigma(\ell)$, where $\sigma : \mathcal{U} \to \mathbb{N}$ is a "universe level parameter instantiation" function that assigns each universe level parameter to a natural number. The interpretation function is defined as follows:

$$\begin{aligned}
\texttt{eval}_\sigma(u) &:= \sigma(u) \\
\texttt{eval}_\sigma(\texttt{z}) &:= 0 \\
\texttt{eval}_\sigma(\texttt{s}\ \ell) &:= \texttt{eval}_\sigma(\ell) + 1 \\
\texttt{eval}_\sigma(\texttt{max}\ \ell\ \ell') &:= \max(\texttt{eval}_\sigma(\ell), \texttt{eval}_\sigma(\ell')) \\
\texttt{eval}_\sigma(\texttt{imax}\ \ell\ \ell') &:= \begin{cases} 0 & \texttt{eval}_\sigma(\ell') = 0 \\ \max(\texttt{eval}_\sigma(\ell), \texttt{eval}_\sigma(\ell')) & \text{otherwise} \end{cases}
\end{aligned}$$

The universe level equivalence relation asserts that two universe level expression evaluate to the same number under all possible universe level parameter instantiations σ, making them "semantically equivalent":

$$\ell \approx \ell' \iff \forall \sigma, \texttt{eval}_\sigma(\ell) = \texttt{eval}_\sigma(\ell')$$

As one would expect, the rules [CGR-CONST] and [CGR-SORT] establish definitional equality between level-instantiated constant references and sorts based on the semantic equivalence of their corresponding pairs of universe level expressions. The [CGR-SORT] rule in particular, in combination with the rules [ALL] and [CONV], is responsible for the impredicativity of Lean's `Prop` universe. We can see this in the typing $\Delta \vdash (P : \texttt{Prop}) \to P \to P : \texttt{Prop}$,

as this function type is able to type within `Prop` despite having quantified over `Prop`. Directly applying [ALL], we obtain $\Delta \vdash (P : \texttt{Prop}) \to P \to P : \texttt{Sort imax (s z) (imax z z)}$, and we can also show that $\texttt{imax (s z) (imax z z)} \approx \texttt{z}$, from which it follows by [CGR-SORT] that $\Delta \vdash \texttt{Sort imax (s z) (imax z z)} \equiv \texttt{Prop}$, and lastly from [CONV] that $\Delta \vdash (P : \texttt{Prop}) \to P \to P : \texttt{Prop}$.

Proof Irrelevance

The rule [PI], referred to as "proof irrelevance", enables the kernel to ignore the computational content of proofs when comparing terms, only concerning itself with the equality of their propositional types:

$$\frac{\Delta \vdash P : \texttt{Prop} \quad \Delta \vdash p, q : P}{\Delta \vdash p \equiv q} \text{ [PI]}$$

[PI] can be justified using Lean's propositional extensionality axiom, which states that any two propositions that are logically equivalent are also propositionally equal:

```
structure Iff (a b : Prop) : Prop where
  intro ::
  mp : a → b
  mpr : b → a
axiom propext {P Q : Prop} : Iff P Q → P = Q
```

We can use `propext` to prove that any provable proposition is propositionally equal to the proposition `True`:

```
inductive True : Prop where
| intro : True
theorem propEqTrue (P : Prop) (p : P) : P = True := by
  apply propext
  apply Iff.intro <;> intro
  trivial
  exact p
```

This lemma can then be used to prove a propositional version of [PI], by first replacing the propositional type of the proof terms with `True`, and then performing elimination on the proof terms to convert them both to the unique constructor `True.intro`:

```
theorem prfIrrel (P : Prop) (p q : P) : p = q := by
  have this := propEqTrue P p -- this : P = True
  subst this
  -- elimnates `p : True` and `q : True` to `True.intro`
  cases p
  cases q
  rfl
```

Proof irrelevance is useful, for example, in establishing the definitional equality of predicate subtype instances with equal values, but differing membership proofs. Subtypes in Lean can be defined as a parametric inductive type as follows:

```
inductive Subtype {A : Type} (p : A → Prop) where
| mk : (val : A) → (prf : p val) : Subtype p
```

Suppose that we use this to define a subtype for natural numbers less than five:

```
def NatLT5 : Type := Subtype (fun n => n < 5)
def NatLT5.mk (n : Nat) (p : n < 5) : NatLT5 :=
  @Subtype.mk Nat (fun n => n < 5) n p
```

Now, suppose we have two syntactically distinct proofs `p1 p2 : 3 < 5`. Proof irrelevance gives us a definitional equality between `NatLT5.mk 3 p1` and `NatLT5.mk 3 p2`, as one would expect, since when we consider the equality of these subtype constructions, all that we care about is the equality of their underlying values.

Forms of proof irrelevance are supported in a number of other proof assistants. Until recently, the use of proof irrelevance in Rocq had to be made explicit with an axiom[6], with, optional support for definitional proof irrelevance having recently been added with the `SProp` type[7]. Agda supports user-annotated irrelevant function arguments and struct fields[8], in addition to a proof-irrelevant type universe `Prop` (analogous to Rocq's `SProp`). F* erases the details of SMT solver-generated equality proofs [37]. The PVS proof assistant [36] features a special case of proof irrelevance in identifying predicate subtype constructions.

Function η

Lean's definitional equality judgment features the rule [FUN-ETA] rule for "function η", which equates a term of a function type with a λ-function whose body is the term applied to the abstracted variable:

$$\frac{}{\Delta \vdash (\texttt{fun}\ (x : A)\ \texttt{=>}\ e\ x) \equiv e}\ \text{[FUN-ETA]}$$

We refer to `fun` $(x : A)$ `=>` $f\ x$ as the "η-expansion" of f.

This equality is justified through the "function extensionality" principle, which states that two functions are equal if they are equal under every possible application. In Lean, function extensionality is provided by the theorem `funext`, which is proven through the use of quotient types:

```
theorem funext {A : Sort u} {B : A → Sort v} {f g : (x : A) → B x}
    (h : (x : Nat) → f x = g x) : f = g := ...
```

From this axiom, it is clear that the definitional equality encoded by [FUN-ETA] holds propositionally, since if we instantiate g with `fun` $(x : A)$ `=>` $f\ x$, the premise `h` is satisfied definitionally via β-reduction (described below).

Struct-η

The rule [ETA-S] takes into account a characterization of certain "structure-like" inductive types in Lean, abbreviated "structs". In other proof assistants, e.g. Rocq, these are referred

[6] See `https://rocq-prover.org/doc/V9.0.0/stdlib/Stdlib.Logic.ProofIrrelevance.html`.
[7] See `https://rocq-prover.org/doc/V9.0.0/refman/addendum/sprop.html`.
[8] See `https://agda.readthedocs.io/en/v2.5.4/language/irrelevance.html`.

to as "record types"[9]. Unlike Rocq, however, Lean does not distinguish between struct-like inductive types and normal inductive types at the syntactic level. Rather, it is the responsibility of the typechecker to determine whether a struct-specific rule is applicable by investigating the form of the inductive type in question.

For an inductive type in Lean to be considered struct-like, it must have a single constructor and no inductive type indices. For instance, consider the following structure type declaration:

```
structure Point where (x : Nat) (y : Nat)
```

Note that the `structure` syntax is simply syntactic sugar that Lean's elaborator expands to the following, including definitions corresponding to each of the projections:

```
inductive Point where
| mk : Nat → Nat → Point
def Point.x (p : Point) : Nat := p.1
def Point.y (p : Point) : Nat := p.2
```

Essentially, structs can be thought of as simple collections of objects of certain (possibly dependent) types. The fact that they lack indices means that their types cannot depend on the particular values provided for their fields, which allows for a powerful elimination principle based upon the property that any term of a struct-like type can be shown to be equivalent to a constructor application, as demonstrated in the following proof:

```
theorem structEtaPoint (p : Point) : Point.mk p.x p.y = p :=
  Point.rec
    (fun x y =>
      rfl
    -- ^ proof of `Point.mk (Point.mk x y).x (Point.mk x y).y = Point.mk x y`
    ) p
```

Because this is quite a useful principle to work with when using struct-like types in Lean, Lean's kernel promotes this propositional equality to a definitional one. Given a struct-like inductive type S with n parameters m fields, a constructor `mk` and a term $\Delta \vdash t : S\ p_1\ \ldots\ p_n$, we have the following definitional equality:

$$\frac{S \text{ struct-like} \quad \Delta \vdash t : S\ p_1\ \ldots\ p_n}{\Delta \vdash t \equiv S.\texttt{mk}\ p_1\ \ldots\ p_n\ t.1\ \ldots\ t.m} \text{ [ETA-S']}$$

We refer to the expression $S.\texttt{mk}\ p_1\ \ldots\ p_n\ t.1\ \ldots\ t.m$ as the "struct-η-expansion" of the term t. In words, this equality says that the kernel can identify a term of struct type with a particular explicit construction using the unique struct constructor, with the type parameters provided by the inferred struct type and the fields values extracted from the term via their corresponding projections. As we have demonstrated by example in `structEtaPoint` above, these must always be equal for any explicit construction taking the place of t, as in this case the projections on the RHS reduce, making the LHS and RHS syntactically equal.

Note that the rule [ETA-S'] is actually a bit less general than the check that is actually implemented by the Lean kernel, corresponding to the rule [ETA-S]:

$$\frac{S \text{ struct-like} \quad \Delta \vdash t : S\ p_1\ \ldots\ p_n \quad \Delta \vdash t.1 \equiv a_1 \quad \ldots \quad \Delta \vdash t.m \equiv a_m}{\Delta \vdash t \equiv S.\texttt{mk}\ p_1\ \ldots\ p_n\ a_1\ \ldots\ a_m} \text{ [ETA-S]}$$

[9] See `https://rocq-prover.org/doc/master/refman/language/core/records.html`.

This rule is more general than [ETA-S'], requiring that each of the constructor field arguments can be identified with a corresponding projection of the structure term, and it can also be shown to hold propositionally for any struct-like type.

Unit η

The rule [UNIT] takes into account a special distinction that Lean makes for so-called "unit-like" inductive types, which are inductive types with a single constructor without any fields. For instance, the Lean standard library provides the following universe-polymorphic unit type:

```
inductive PUnit : Sort u where
  | unit : PUnit
```

Such types are useful especially with type-parameterized inductive types, where they can be used as "trivial" instantiations of these type parameters[10]. For such unit-like types, Lean enables a special definitional equality rule that allows any two instances of the same unit type to be identified, giving us the rule [UNIT]:

$$\frac{U \text{ unit-like} \quad \Delta \vdash t, s : U}{\Delta \vdash t \equiv s} \text{ [UNIT]}$$

Similarly to struct-η, this rule can be justified as it corresponds to a propositional equality that can be shown by elimination on unit-like types.

1.2.3 The Reduction Relation

Lastly, we have the rule [RED], which allows a term t to be identified with a term s that is definitionally equal to a reduced form of t:

$$\frac{\Delta \vdash t \rightsquigarrow^* t' \quad \Delta \vdash t' \equiv s}{\Delta \vdash t \equiv s} \text{ [RED]}$$

This rule introduces a new piece of notation $\Delta \vdash t \rightsquigarrow^* s$, denoting the reflexive, transitive closure of a single-step reduction relation $\Delta \vdash t \rightsquigarrow s$. This relation is used to decide equality between Lean terms through the (partial) computation of normal forms, which greatly expands the set of identifiable terms as Lean lacks a rule for general transitivity of definitional equality[11].

The reduction relation is defined by a set of "reduction rules", a subset of which are shown below:

$$\frac{}{\Delta \vdash (\texttt{fun } (x : A) \texttt{ => } e)\ a \rightsquigarrow e[x/a]} \text{ [BETA]} \qquad \frac{\Delta \vdash t \rightsquigarrow s}{\Delta \vdash C[t] \rightsquigarrow C[s]} \text{ [CTX]}$$

$$\frac{\Delta = (\Sigma; \Gamma) \quad \Sigma \text{ contains } C.\{u_1, \ldots, u_n\}, \text{ defined with value } v}{\Delta \vdash C.\{l_1, \ldots, l_n\} \rightsquigarrow v[[u_1/l_1, \ldots, u_n/l_n]]} \text{ [DELTA]}$$

$$\frac{S \text{ structure-like}}{\Delta \vdash (S.\texttt{mk}\ p_1\ \ldots\ p_n\ a_1\ \ldots\ a_i\ \ldots\ a_m).i \rightsquigarrow a_i} \text{ [RPROJ]}$$

[10] For instance, they can be used with monadic function type signatures to signify that the function lacks a return value.

[11] In fact, we cannot have a theory for Lean exhibiting general transitivity, as it would render the theory undecidable, as was shown by Carneiro [13].

$$\frac{K \text{ K-like} \quad \Delta \vdash K.\texttt{mk}\ p_1\ \ldots\ p_n : \texttt{K}\ p_1\ \ldots\ p_n\ i_1\ \ldots\ i_m \quad \Delta \vdash t : K.\texttt{mk}\ p_1\ \ldots\ p_n\ i_1\ \ldots\ i_m}{\Delta \vdash t \rightsquigarrow K.\texttt{mk}\ p_1\ \ldots\ p_n} \text{ [KLR]}$$

$$\frac{S \text{ struct-like} \quad \Delta \vdash t : S\ p_1\ \ldots\ p_n}{\Delta \vdash t \rightsquigarrow S.\texttt{mk}\ p_1\ \ldots\ p_n\ t.1\ \ldots\ t.m} \text{ [R-ETA-S]}$$

$$\frac{\Delta = (\Sigma; \Gamma) \quad \Sigma \text{ quotient-declared}}{\Delta \vdash \texttt{@Quot.ind}\ A\ r\ B\ p\ (\texttt{@Quot.mk}\ A\ r\ a) \rightsquigarrow p\ a} \text{ [QIND]}$$

$$\frac{\Delta = (\Sigma; \Gamma) \quad \Sigma \text{ quotient-declared}}{\Delta \vdash \texttt{@Quot.lift}\ A\ r\ B\ f\ h\ (\texttt{@Quot.mk}\ A\ r\ a) \rightsquigarrow f\ a} \text{ [QLIFT]}$$

$$\ldots$$

As in the definitional equality judgment, these reduction rules implicitly take as premises the well-typedness of the LHS and RHS of the conclusion judgment.

The rule [BETA] corresponds to "β-reduction", which is quite standard to many type-theory based proof assistants. This rule allows an application of a λ-function to a term (known as a "β-redex") to be reduced by substituting the application for the body of the λ-function and replacing the bound variable with the argument. The rule [CTX] enables the reduction relation to apply between any two terms which contain subterms that directly[12] reduce to one another, where we use the notation $C[t]$ to indicate the substitution of a singular subterm hole in C with the term t.

δ-Reduction

The rule [DELTA] expresses "δ-reduction", otherwise known as "δ-expansion", which involves expanding a previously defined constant to its body in the process of reduction:

$$\frac{\Delta = (\Sigma; \Gamma) \quad \Sigma \text{ contains } C.\{u_1, \ldots, u_n\}, \text{ defined with value } v}{\Delta \vdash C.\{l_1, \ldots, l_n\} \rightsquigarrow v[[u_1/l_1, \ldots, u_n/l_n]]} \text{ [DELTA]}$$

While [DELTA] does not strictly expand the class of provable propositions, the use of defined constant names allows terms to be reused in several different places without duplication. Strictly speaking, [DELTA] itself is only really useful for constants whose type is not in `Prop`, as definitions corresponding to proofs (i.e., theorems) do not ever need to be expanded, thanks to the rule [PI], which makes the actual content of proof terms irrelevant for typechecking.

Projection Reduction

The rule [RPROJ] describes how structure type projections reduce whenever they are applied to an explicitly constructed structure type instance, directly extracting the field from the construction corresponding to the projection index:

$$\frac{S \text{ structure-like}}{\Delta \vdash (S.\texttt{mk}\ p_1\ \ldots\ p_n\ a_1\ \ldots\ a_i\ \ldots\ a_m).i \rightsquigarrow a_i} \text{ [RPROJ]}$$

For instance, recall our example `Point` struct from earlier. The term `(Point.mk 1 2).1` is able to reduce immediately to 1 via [RPROJ], as the projection directly extracts the first field from the `Point.mk` constructor application, without performing standard reduction via

[12] By "directly", we mean to say that that some rule other than [CTX] is applied to the subterm (or, equivalently, that reduction changes the function head of an n-ary application).

recursors (described below). Projection reduction has been implemented in the Lean kernel as an important optimization, as structure types are very frequently used in larger-scale formalizations.

K-Like Reduction

Lean also features a special reduction rule known as "K-like reduction" ([KLR] above), which is based on the characterization of so-called "K-like" inductive types in Lean, which are defined as inductive types that that live in `Prop` and have a single constructor without any (non-parametric) arguments. Lean's equality inductive type, recalled below, is an example of such a K-like inductive type:

```
inductive Eq {A : Sort u} (a : A) : A → Prop where
| refl : Eq a a
```

This inductive type has two parameters, found to the left of the colon in the inductive type signature: the polymorphic type `A` and the LHS element `a : A`. It also has an index, which is the RHS element of the equality. In the particular case of K-like inductive types, where constructors have no arguments, indices are effectively functions of the parameters.

Many proof assistants feature support for what is known as "Axiom K", which states that any proof of equality between identical terms[13] is propositionally equal to the unique reflexive constructor for equality[14]:

```
theorem axiomK (p : t = t) : p = Eq.refl t := ...
```

It is equivalent to the notion of uniqueness of identity proofs (UIP), which allows for equating any two proofs of the same equality type:

```
theorem UIP (p q : t = s) : p = q := ...
```

This is a useful property, since it is often the case that there is more than one way to show a given equality, while the equality type only has a single constructor, implying that the proofs should (in principle) be the same. Both properties are trivially provable in Lean, as they are a special case of proof irrelevance (as we can see in the use of the `rfl` proofs above).

Lean generalizes the axiom K *as a reduction rule* for any K-like inductive type. In general, suppose we have a K-like type `K` with n parameters, m indices and the unique constructor `mk`. We can express K-like reduction as the rule:

$$\frac{K \text{ K-like} \quad \Delta \vdash K.\texttt{mk}\ p_1\ \ldots\ p_n : \texttt{K}\ p_1\ \ldots\ p_n\ i_1\ \ldots\ i_m \quad \Delta \vdash t : K.\texttt{mk}\ p_1\ \ldots\ p_n\ i_1\ \ldots\ i_m}{\Delta \vdash t \rightsquigarrow K.\texttt{mk}\ p_1\ \ldots\ p_n} \text{ [KLR]}$$

Note the requirement that the type of t corresponds to the type of a constructor application of the K-like inductive type. This is important for the subject reduction property, because if this is not the case, then the reduction would not respect preservation of typing, with the constructor application on the RHS having a different type than the LHS.

For example, the above rule applies to the equality type with $n = 2$, $m = 1$, $\texttt{K} = \texttt{Eq}$, $\texttt{mk} = \texttt{Eq.refl}$, $p_1 = \texttt{Nat}$, $p_2 = 0$, and $i_1 = 0$, allowing any term `t : 0 = 0` to be reduced[15]

[13] Due to [CONV], it effectively applies up to definitional equality of the LHS and RHS.

[14] Despite being called an axiom, this property is provable using the eliminator for equality.

[15] With respect to a practical typechecker implementation, during reduction we must ensure that `t` is not already an application of `Eq.refl` before applying [KLR] in order to avoid non-termination.

to `Eq.refl Nat 0`. Note that K-like reduction is related to proof irrelevance, since t and `mk` $p_1 \ \ldots \ p_n$ are already definitionally equal in Lean by [PI] (as K-like inductive types must live in `Prop`). However, as a reduction rule, it enables a more powerful elimination principle on K-like inductive types. For instance, in the case of `Eq`, it enables the recursor to reduce on *any* well-typed major premise argument, without needing an explicit construction:

```
theorem KLR (h : t = t) : Eq.rec refl h = refl := rfl
```

Here, the major premise is simply the variable `h`. This is a practically useful reduction because, for instance, it allows us to eliminate redundant casts around terms, since applications of the explicit type conversion operator `cast` reduce to applications of `Eq.rec`:

```
def cast {A B : Sort u} (h : A = B) (a : A) : B := ...
theorem elimCast (T : Type) (t : T) (h : T = T) :
   t = cast h t :=
  -- `t` and `cast h t` are defeq thanks to [KLR]
  rfl
```

In reality, Lean's kernel does not implement such a reduction rule in general when computing the WHNF of terms of K-like types; instead, it only uses it on major premise arguments of recursors in the course of reducing recursor applications, equivalent to the following reduction rule:

$$\frac{K \text{ K-like} \quad \Delta \vdash K.\texttt{mk}\ p_1 \ \ldots \ p_n : K\ p_1 \ \ldots \ p_n\ i_1 \ \ldots \ i_m \quad \Delta \vdash t : \texttt{K}\ p_1 \ \ldots \ p_n\ i_1 \ \ldots \ i_m}{\Delta \vdash K\ \texttt{rec}\ p_1 \ \ldots \ p_n\ \_\ f\ i_1 \ \ldots \ i_m\ t \rightsquigarrow f} \text{ [KLR-REC]}$$

This rules specifies that we are able to apply recursor reduction on a $K.\texttt{mk}$ application given any well-typed major premise t. To apply it, the kernel has to verify the second premise of the rule, which it does by inferring the type of t, using the parameter arguments to construct a K `rec` application and checking that the index arguments match up with those of the inferred type of this construction[16]. While seemingly more limited in scope, this alternate reduction rule is entirely sufficient, as it is only in the case of recursor reduction that an application of [KLR] would make any difference in a judgment of definitional equality.

K-like reduction, in combination with Lean's impredicative `Prop` universe, results in non-termination of reduction, as shown by Abel and Coquand [1]. While its use in Lean has proven to be quite successful, such a theoretical lack of strong normalization may be part of the reason why very few other proof assistants support it. It does, however, exist to a limited extent in the Rocq proof assistant, where it can be enabled with the "Definitional UIP" flag[17] (this is not enabled by default to preserve certain theoretical properties, e.g. normalization).

Struct-Like Reduction

Structure types in Lean impose a special reduction rule that is similar to the one for K-like inductive types: in the same way that we can reduce a term of a well-indexed K-like inductive

[16] Note that for K-like inductive types, the fact that the unique constructor takes no non-parametric arguments means that the index values are uniquely determined as a function of the parameters.

[17] `https://rocq-prover.org/doc/V8.18.0/refman/addendum/sprop.html#definitional-uip`.

type to a constructor application for the purpose of recursor reduction, we can apply struct-η-expansion to identify a term of structure type with the unique structure constructor applied to the corresponding projections of the term, giving us the rule [R-ETA-S]:

$$\frac{S \text{ struct-like} \quad \Delta \vdash t : S\ p_1\ \ldots\ p_n}{\Delta \vdash t \rightsquigarrow S.\texttt{mk}\ p_1\ \ldots\ p_n\ t.1\ \ldots\ t.m} \text{ [R-ETA-S]}$$

As with [KLR], this rule is only really relevant for the reduction of recursor applications, with the Lean kernel only actually ever applying this reduction to the major premise argument of structure recursor applications. Therefore, [R-ETA-S] can be equivalently stated as follows:

$$\frac{S \text{ struct-like} \quad \Delta \vdash t : S\ p_1\ \ldots\ p_n}{\Delta \vdash S.\texttt{rec}\ p_1\ \ldots\ p_n\ \_\ f\ t \rightsquigarrow f\ t.1\ \ldots\ t.m} \text{ [R-ETA-S-REC]}$$

Quotient Reduction

Lean features a special built-in polymorphic type, referred to as a "quotient", that enables the construction of terms that represent equivalence classes of a type under a specified equivalence relation. Quotients types in Lean are constructed with the symbol `Quot`, with quotient instances being constructed using the symbol `Quot.mk`. These symbols have the following types:

```
#check Quot
-- Quot.{u} {A : Sort u} (r : A → A → Prop) : Sort u
#check Quot.mk
-- Quot.mk.{u} {A : Sort u} (r : A → A → Prop) (a : A) : Quot r
```

The quotient type `@Quot A r` corresponds to a type of equivalence classes of `A` under the relation `r`, where two constructions `Quot.mk r a` and `Quot.mk r a'` are considered equivalent if and only if `a` and `a'` are related by `r`. This is expressed as a propositional equivalence by the axiom `Quot.sound`:

```
#check Quot.sound
-- Quot.sound.{u} {A : Sort u} {r : A → A → Prop} {a b : A} :
--   r a b → Quot.mk r a = Quot.mk r b
```

Lean includes two special elimination symbols for quotient types. The first is `Quot.ind`, which encodes the induction principle on quotient types:

```
#check Quot.ind
-- Quot.ind.{u} {A : Sort u} {r : A → A → Prop} {B : Quot r → Prop},
--   ((a : A) → B (Quot.mk r a)) : (q : Quot r) → B q
```

To allow for the definition of data-computing functions on quotient types, a function lifting operator `Quot.lift` is also provided:

```
#check Quot.lift
-- Quot.lift.{u, v} : {A : Sort u} → {r : A → A → Prop} → {B : Sort v} →
--   (f : A → B) → ((a b : A) → r a b → f a = f b) → Quot r → B
```

Both of these functions have special reduction rules associated with them, that apply when an explicit quotient construction is provided as the final argument. These can be expressed as follows:

$$\frac{\Delta = (\Sigma; \Gamma) \quad \Sigma \text{ quotient-declared}}{\Delta \vdash \texttt{@Quot.ind}\ A\ r\ B\ p\ (\texttt{@Quot.mk}\ A\ r\ a) \rightsquigarrow p\ a} \text{ [QIND]}$$

$$\frac{\Delta = (\Sigma; \Gamma) \quad \Sigma \text{ quotient-declared}}{\Delta \vdash \texttt{@Quot.lift}\ A\ r\ B\ f\ h\ (\texttt{@Quot.mk}\ A\ r\ a) \rightsquigarrow f\ a} \text{ [QLIFT]}$$

Recursor Reduction

For every inductive type that is defined, Lean generates "recursors" (a.k.a. eliminators) that allow for defining functions/performing inductive proofs on terms of that inductive type. For instance, the type `Vec` declared above has generates the following recursor:

```
#check Vec.rec
recursor Vec.rec.{u} : {T : Type} → {motive : (a : Nat) → Vec T a → Sort u} →
  (nil :  motive Nat.zero Vec.nil) → (cons : (n : Nat) → (v : Vec T n) → (t : T) →
  motive n v → motive n.succ (Vec.cons n v t)) →
  {n : Nat} → (v : Vec T n) → motive n v
```

The `motive` argument describes the output type of the recursor application, which may depend on the index and particular `Vec` instance that the recursor is applied to. The next two arguments, `nil` and `cons`, are the "minor premises" of the recursor, which describe how elimination is to be performed in the case of a `Vec` being constructed as `Vec.nil` or `Vec.cons`. These are followed by an index argument `n` and a "major premise" `t`, which is the `Vec` instance that is actually eliminated upon.

For every inductive type recursor, Lean generates a set of associated reduction rules, one for each of the inductive type constructors. When a major premise is provided whose weak-head normal form is a constructor application, the recursor is able to reduce, according to the reduction rule that was defined for that constructor – this is referred to as recursor reduction, a.k.a. ι-reduction. For example, with our `Vec` inductive type, the recursor reduction rule for `Vec.nil` simply reduces to the minor premise argument that corresponds to the `Vec.nil` case:

```
example : @Vec.rec T motive nil cons Vec.nil = nil := rfl
```

The recursor rule for `Vec.cons` is more complex:

```
example : @Vec.rec T motive nil cons (Vec.cons n v t)
          = cons v t (@Vec.rec motive nil cons v) := rfl
```

This recursor rule applies the minor premise argument corresponding to the `Vec.cons` case to the data arguments of the `Vec.cons` application, with the inductive hypothesis argument being provided by a recursive call of the same recursor application, swapping in the recursive argument for the major premise. In general, inductive hypothesis arguments are generated for each recursive instance in the function type of the constructor.

This reduction is implemented in the kernel by applying a special function corresponding to the reduction rule to the motive types, minor premises, and extracted constructor arguments. The body of this function applies the minor premise corresponding to the major

premise constructor application to the extracted data arguments and recursive calls for each of the inductive hypothesis arguments.

Recursor rules are also generated for the other categories of inductive types, like mutual inductive types. For instance, the `Even`/`Odd` mutual inductive types shown earlier generate the following recursors, where each recursor must be provided with elimination rules for *both* mutually defined types:

```
-- Even.rec :
--   {mtv_e : (a : Nat) → Even a → Sort u} → {mtv_o : (a : Nat) → Odd a → Sort u} →
--   mtv_e Nat.zero Even.zero →
--   ({n : Nat} → (a : Odd n) → mtv_o n a → mtv_e (Nat.succ n) (Even.succ n)) →
--   ({n : Nat} → (a : Even n) → mtv_e n a → mtv_o (Nat.succ n) (Odd.succ n)) →
--   {a : Nat} → (t : Even a) → mtv_e a t
#print Even.rec
-- Odd.rec :
--   {mtv_e : (a : Nat) → Even a → Sort u} → {mtv_o : (a : Nat) → Odd a → Sort u} →
--   mtv_e Nat.zero Even.zero →
--   ((n : Nat) → (a : Odd n) → mtv_o n a → mtv_e (Nat.succ n) (Even.succ n)) →
--   ((n : Nat) → (a : Even n) → mtv_e n a → mtv_o (Nat.succ n) (Odd.succ n)) →
--   {a : Nat} → (t : Odd a) → mtv_o a t
#print Odd.rec
```

These recursors generate the following mutually referencing rules:

```
example : Even.rec z succ_e succ_o Even.zero = z := rfl
example : Even.rec z succ_e succ_o (Even.succ n)
           = succ_e n (Odd.rec z succ_e succ_o n) := rfl
example : Odd.rec z succ_e succ_o (Odd.succ n)
           = succ_o n (Even.rec z succ_e succ_o n) := rfl
```

A formal rule characterizing the general form of recursor reduction in Lean was first described by Carneiro [13]. However, the presentation is rather complex and requires the use of new notation to formally describe inductive type/constructor declarations. As we will see later in Section 4.2, it turns out that we do not need to account for all of the specifics of the theoretical structure of recursor reduction in Lean in defining our translation, as Lean generates recursor rules that we can directly translate into rewrite rules. As such, we omit from this thesis a full formal presentation recursor reduction in Lean.

1.3 Dedukti's Type Theory

Dedukti's type theory is much more minimal than Lean's, being based on the $\lambda\Pi$-calculus modulo rewriting ($\lambda\Pi/R$). Its type theory is based on the $\lambda\Pi$ calculus, which is a variant of the simply typed λ-calculus in which both types and terms can depend on terms, enabling an encoding of first-order logic. The addition of rewrite rules to this theory enables a representation of higher-order logic through the use of specialized symbols and associated rewrite rules, for instance via a pure type system encoding (see Section 2.2).

Term and Type Context Syntax

Dedukti terms are taken from the following grammar:

$$
\begin{aligned}
&S_{\text{DK}} = t \text{ where} \\
&\quad t ::= x \mid c \mid \texttt{Type} \mid \texttt{Kind} \mid t_1\ t_2 \mid (x : t_1) \texttt{ => } t_2 \mid (x : t_1)\ \rightarrow\ t_2
\end{aligned}
$$

where $x \in \mathcal{X}$ and $c \in \mathcal{C}$, with $\mathcal{X}$ denoting a set of bound variable names and $\mathcal{C}$ denoting a set of declared symbol names. A Dedukti term can either be a variable or declared symbol reference, a special type universe symbol `Type` or `Kind`, a λ-function, or a dependent function type expression. In presenting Dedukti's type theory, we will also make use of a grammar for typing contexts, denoted by the symbol Δ:

$$
\begin{aligned}
C_{\text{DK}} &= \Delta \text{ where} \\
\Delta &::= (\Sigma; \Gamma) \\
\Sigma &::= (\Sigma_D; \Sigma_R) \\
\Sigma_D &::= () \mid (\Sigma_D, \sigma_D) \\
\Sigma_R &::= () \mid (\Sigma_R, \sigma_R) \\
\Gamma &::= () \mid (\Sigma_R, x : t)
\end{aligned}
$$

Similarly to Lean, a Dedukti typing context consists of two main components: a global typing context Σ, and a local typing context Γ. The global context contains a list of symbol declarations Σ_D and a set of rewrite rules Σ_R that together constitute a "rewrite system" that will factor into Dedukti's type conversion rule. The local context consists of a list of named variable typings introduced in the course of constructing typing derivations for binder terms. As we did for Lean, we will use the shorthand $\Delta, x : T$ to mean $(\Sigma; (\Gamma, x : T))$.

Symbol declarations and rewrite rules are taken from the following syntax:

$$
\begin{aligned}
\sigma_D &::= c\ \texttt{:}\ t. \mid \texttt{def}\ c\ \texttt{:}\ t. \\
\sigma_R &::= [x_1, \ldots, x_n]\ t_1\ \hookrightarrow\ t_2.
\end{aligned}
$$

A symbol declaration of the form c `:` $t.$ declares c as a "static symbol" of type t and a declaration of the form `def` c `:` $t.$ declares c as a "defined symbol" of type t. Rewrite rules in Dedukti consist of a set of "pattern variables", a left-hand side (LHS) and a right-hand side (RHS), where the LHS must be an application headed by a defined symbol, and any of the pattern variables appearing in the RHS must also appear in the LHS, with the RHS having the same type as the LHS[18].

For instance, we might declare the type of natural numbers in Dedukti as follows:

```
Nat : Type.
zero : Nat.
succ : Nat → Nat.
```

This corresponds to a standard Peano-style definition, where a natural number can either be constructed as zero using the constructor `zero`, or as the successor of some other natural

[18]There are additional restrictions placed on the statement of rewrite rules in Dedukti, though these are a bit more complex to state and not presented here.

number n using the constructor application `succ` n, Now, we can use rewrite rules to declare a predecessor operator as follows:

$$\begin{aligned}&\texttt{def pred : Nat} \to \texttt{Nat}.\\ &[]\ \texttt{pred zero} \hookrightarrow \texttt{zero}.\\ &[n]\ \texttt{pred (succ}\ n) \hookrightarrow n. \quad (r_1)\end{aligned}$$

Note that `pred` is declared as a defined symbol because we later define rewrite rules on it.

1.3.1 Type Inference Rules

The basic typing rules of $\lambda\Pi/R$ are as follows[19]:

$$\frac{\Delta \vdash^{\hookrightarrow} A, B : \texttt{Type} \quad \Delta \vdash^{\hookrightarrow} A \equiv B \quad \Delta \vdash^{\hookrightarrow} t : A}{\Delta \vdash^{\hookrightarrow} t : B}\ [\text{CONV}] \qquad \frac{\Delta \vdash^{\hookrightarrow} f : (x : A) \to B \quad \Delta \vdash^{\hookrightarrow} e : A}{\Delta \vdash^{\hookrightarrow} f\ e : B[e/x]}\ [\text{APP}]$$

$$\frac{}{\Delta \vdash^{\hookrightarrow} \texttt{Type} : \texttt{Kind}}\ [\text{TYPE}] \qquad \frac{\Delta \vdash^{\hookrightarrow} A : \texttt{Type}}{\Delta, x : A \vdash^{\hookrightarrow} x : A}\ [\text{VAR}] \qquad \frac{c \text{ declared in } \Delta \text{ with type } T}{\Delta \vdash^{\hookrightarrow} c : T}\ [\text{CONST}]$$

$$\frac{\Delta \vdash^{\hookrightarrow} A : \texttt{Type} \quad \Delta, x : A \vdash^{\hookrightarrow} e : B}{\Delta, x : A \vdash^{\hookrightarrow} (x : A) \texttt{ => } e : (x : A) \to B}\ [\text{LAM}] \qquad \frac{\Delta \vdash^{\hookrightarrow} A : \texttt{Type} \quad \Delta, x : A \vdash^{\hookrightarrow} B : \texttt{Type}}{\Delta \vdash^{\hookrightarrow} (x : A) \to B : \texttt{Type}}\ [\text{ALL}]$$

These rules closely resemble the corresponding rules from Lean's theory, with a few key differences no note. Firstly, the rule for typing universe sorts has been replaced by the rule [TYPE] which simply types the symbol `Type` as `Kind`, with `Kind` itself being left untyped. Secondly, the rules [LAM] and [ALL] are not able to quantify over types, as domain types must themselves inhabit `Type`. Lastly, while the type conversion rule [CONV] is identical in form to the type conversion rule in Lean, and serves the same purpose of expanding the set of typeable terms, it uses a different definitional equality judgment, which we define below.

1.3.2 Definitional Equality Rules

Dedukti's definitional equality judgment is defined more or less solely in terms of reduction to syntactically equivalent forms. Let's formally present this judgment, starting from a base notion of reduction in Dedukti.

For some $\Delta = ((\Sigma_D; \Sigma_R); \Gamma)$, we use the notation $\Delta \vdash^{\hookrightarrow} t \hookrightarrow^{\mathrm{H}}_{\beta} s$ to say that t "head-reduces" to s in a single step through the direct application of β-reduction or some rewrite rule in Σ_R. Precisely, it is characterized by the following two rules:

$$\frac{\Delta \vdash^{\hookrightarrow} ((x : A) \texttt{ => } e)\ a : T}{\Delta \vdash^{\hookrightarrow} ((x : A) \texttt{ => } e)\ a \hookrightarrow^{\mathrm{H}}_{\beta} e[x/a]}\ [\text{BETA}] \qquad \frac{[x_1, \ldots, x_n]\ l \hookrightarrow r. \text{ in } \Sigma_R \quad t = l[a_1/x_1, \ \ldots, \ a_n/x_n]}{\Delta \vdash^{\hookrightarrow} t \hookrightarrow^{\mathrm{H}}_{\beta} r[a_1/x_1, \ \ldots, \ a_n/x_n]}\ [\text{RW}]$$

The rule [BETA] corresponds to standard β-reduction, while the rule [RW] enables terms to reduce based on syntactic matching with the LHS of rewrite rules, where pattern variables may be replaced by arbitrary subterms, rewriting to the RHS with these same pattern variables replaced with the same subterms. For instance, by the rewrite rule r_1 defined above, we have for all terms n that `pred` (`succ` n) reduces to n.

[19] For simplicity's sake, we use the same rule labels for similar rules in different theories; it should always be clear from the context which rule from which theory is being referred to.

Extending this relation with with context closure, we obtain our single-step reduction relation $\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta s$, stating that some subterm of t reduces to s in a single step, through β-reducing or rewriting of one of its subterms. Formally, we can specify this relation with the following rule, which is analogous to the rule [CTX] from Lean's type theory:

$$\frac{\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^{\mathrm{H}} s}{\Delta \vdash^{\hookrightarrow} C[t] \hookrightarrow_\beta C[s]} \text{ [CTX]}$$

Let's also introduce the relation $\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^* s$ to indicate that t reduces to s after some number of steps (zero or more). Formally, we have the following three rules:

$$\frac{}{\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^* t} \text{ [RED\_REFL]} \qquad \frac{\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta s}{\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^* s} \text{ [RED\_UNIT]}$$

$$\frac{\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^* s \quad \Delta \vdash^{\hookrightarrow} s \hookrightarrow_\beta u}{\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^* u} \text{ [RED\_TRANS]}$$

Verbally, we simply say that "t reduces to s" in context Δ if $\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^* s$.

We can now define Dedukti's equality judgment, which is based on the syntactic equivalence of reduced forms:

$$\frac{\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^* u_1 \quad \Delta \vdash^{\hookrightarrow} s \hookrightarrow_\beta^* u_2 \quad \Delta \vdash^{\hookrightarrow} u_1 =_{\mathrm{S}} u_2}{\Delta \vdash^{\hookrightarrow} t \equiv s} \text{ [DEQ]}$$

The judgment $\Delta \vdash^{\hookrightarrow} t =_{\mathrm{S}} s$ simply checks for the syntactic equality of t and s[20]:

$$\frac{}{\Delta \vdash^{\hookrightarrow} t =_{\mathrm{S}} t} \text{ [SYN]}$$

Dedukti has also been extended for support for function-η, which can be enabled with a special command-line flag, amounting to the following rule:

$$\frac{\Delta \vdash^{\hookrightarrow} f : (x : A) \ \to \ B}{\Delta \vdash^{\hookrightarrow} ((x : A) \texttt{=>} \ f \ x) =_{\mathrm{S}} f} \text{ [FUN-ETA]}$$

Lastly, we add an important condition to typing requiring that our typing context Δ is confluent. To state this, we use the judgment $\texttt{conf}(\Delta)$, defined by the following rule:

$$\frac{\forall t_1, t_2, (\Delta \vdash^{\hookrightarrow} s \hookrightarrow_\beta^* t_1 \wedge \Delta \vdash^{\hookrightarrow} s \hookrightarrow_\beta^* t_2) \implies (\exists u, \Delta \vdash^{\hookrightarrow} t_1 \hookrightarrow_\beta^* u \wedge \Delta \vdash^{\hookrightarrow} t_2 \hookrightarrow_\beta^* u)}{\texttt{conf}(\Delta)} \text{ [CONF]}$$

In words, the confluence property ensures that any pair of terms that are arrived at by applying rewrite rules/β-reduction to an initial term in different orders/locations are eventually able to be "joined" by continuing to apply reduction to both terms. In general, determining the confluence of a given rewrite system is an undecidable problem, however it is a critical property to ensure that the rewrite system defined by our global typing context is "well-behaved," enabling important theoretical properties to hold. In particular, it gives us the following uniqueness of normal forms property:

[20]Note that this rule implicitly accounts for syntactic equivalence up to α-equivalence (that is, renaming of variables bound by λ-functions or dependent function type expressions).

Lemma 1.3.1 (Uniqueness of Normal Forms)**.** If $\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^{*\mathrm{N}} u_1$ and $\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^{*\mathrm{N}} u_2$, then $\Delta \vdash^{\hookrightarrow} u_1 =_{\mathrm{S}} u_2$.

Here, we use the notation $\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^{*\mathrm{N}} s$ to indicate that t reduces to s after some number of steps, with s being in normal form – that is, t "normalizes to" s. Formally, $\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^{*\mathrm{N}} s$ means that $\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^{*} s$ with $\Delta \vdash^{\hookrightarrow} \mathrm{NF}(s)$, where we define the notation $\Delta \vdash^{\hookrightarrow} \mathrm{NF}(t)$ to indicate that a term is in normal form, meaning that neither β-reduction nor any rewrite rules can apply to it. [21]. From the uniqueness of normal forms property, we can also derive the following, which will be useful to us in showing the correctness of our translation:

Lemma 1.3.2. If $\Delta \vdash^{\hookrightarrow} t \equiv s$, with $\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^{*\mathrm{N}} t'$ and $\Delta \vdash^{\hookrightarrow} s \hookrightarrow_\beta^{*\mathrm{N}} s'$, then $\Delta \vdash^{\hookrightarrow} t' =_{\mathrm{S}} s'$.

That is, any two definitionally equal terms have syntactically identical normal forms.

Because confluence is such an important property, we implicitly add the premise $\texttt{conf}(\Delta)$ to all of our typing rules defined earlier, making our typing judgment as a whole require a confluent typing context, which will ease later analyses. Note, however, that this condition makes our Dedukti theory much more restrictive than the theory that is actually decided by Dedukti's kernel, as it implies that *nothing* is typeable in the case of a non-confluent typing context. Dedukti's typechecker, on the other hand, will still allow for typing in a non-confluent context, as it does not perform any confluence check itself during typechecking.

The fact that Dedukti does not verify confluence can sometimes result in some unexpected behavior. Namely, in the event of non-confluence, where a term can reduce to distinct normal forms, then depending on the arbitrary order in which the kernel chooses to apply specific rewrite rules/β-reduction on certain subterms, it will identify the term with only *one* of them, the one it happened to reduce it to.

For instance, if we have a non-confluent rewrite system and the kernel is performing a conversion check in which it attempts to identify some term t with some normal form term u_1 such that t reduces to u_1, but in doing so applies reduction in such a way that t is reduced to some syntactically distinct normal form term u_2 instead, it would not make an attempt to "backtrack" its reduction steps and perform some kind "search" over all possible reductions to try to instead arrive at u_1. Instead, it would simply report that t and u_1 are not definitionally equal, resulting in a type mismatch error.

On the other hand, if we *do* have confluence, then we can be sure that the above situation can never arise, causing our more restrictive theory to exactly coincide with the typing judgment implemented by the Dedukti kernel. In this case, it suffices to apply rules in any arbitrary order (with backtracking search being unnecessary), and we can be sure that the equational theory defined by the reflexive, transitive, symmetric closure of our set of rewrite rules is effectively decided by the Dedukti kernel. So, because all of our reasoning about translation correctness will be done w.r.t. this theory that requires confluence as a typing premise, we will want to to be sure that the rewrite system established by our translation respects the confluence property.

A necessary property for confluence is "local confluence", expressed as follows:

$$\forall t_1, t_2, (\Delta \vdash^{\hookrightarrow} s \hookrightarrow_\beta t_1 \wedge \Delta \vdash^{\hookrightarrow} s \hookrightarrow_\beta t_2) \implies (\exists u, \Delta \vdash^{\hookrightarrow} t_1 \hookrightarrow_\beta^{*} u \wedge \Delta \vdash^{\hookrightarrow} t_2 \hookrightarrow_\beta^{*} u)$$

This is identical to the confluence condition, but using a single-step reduction from s to derive the divergent terms t_1 and t_2. A well-known result states that confluence follows

[21] Formally, we would say that there does not exist any s such that $\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta s$

from the properties of local confluence and termination. While local confluence itself is not sufficient for confluence, it is nevertheless useful to ensure that local confluence holds, as an important "sanity check" as we go about defining our encoding. We will do this by analyzing the presence and joinability of "critical pairs" upon adding new rewrite rules to our encoding.

Critical pairs arise when two rewrite rules overlap on the same term. For instance, consider the following rewrite rule:

$$[n]\ \texttt{pred}\ (\texttt{pred}\ (\texttt{succ}\ (\texttt{succ}\ n))) \hookrightarrow n. \quad (r_2)$$

Now, if Dedukti is faced with the term `pred (pred (succ (succ zero)))`, we can either directly apply this rule, or we can can instead apply the earlier defined rule r_1 on `pred`, resulting in two different terms, giving us following critical pair:

$$\texttt{pred}\ (\texttt{pred}\ (\texttt{succ}\ (\texttt{succ}\ \texttt{zero}))) \overset{r_1}{\hookrightarrow} \texttt{zero}.$$
$$\texttt{pred}\ (\texttt{pred}\ (\texttt{succ}\ (\texttt{succ}\ \texttt{zero}))) \overset{r_2}{\hookrightarrow} \texttt{pred}\ (\texttt{succ}\ \texttt{zero}).$$

Critical pairs are not problematic *per se* – in this instance, the additional rule r_2, while redundant, does not present any issues for our ability to decide equality between `Nat`-typed terms because the above critical pair is *joinable*. Namely, we can apply r_1 again to the second term to reduce it to the first term.

In general, however, the presence of *unjoinable* critical pairs points to ambiguity in our rewrite system w.r.t. the way that it arrives at a normal form, which may lead to terms not being identified that *should be* according to the equational theory that we are trying to encode. That is to say, it is not sufficient for us to only have that the rewrite rules that we define are correct according to the intended equational theory – they must also be defined in such a way that they can *effectively decide* this theory through the computation of normal form terms. In some cases (as we will see in Chapter 3), it is even necessary to define *new* symbols, beyond those originally present in the grammar of the encoded equational theory, in order to represent these normal form terms.

Translation Overview

Our translation from Lean to Dedukti will involve the encoding of certain aspects of Lean's type theory within a Dedukti rewrite system, along with the implementation of a translation from Lean terms to Dedukti syntax. We will assume well-typed input Lean terms, which will then be translated to Dedukti and verified with the Dedukti typechecker `dkcheck` [33]. From the translated Dedukti proofs, we can then translate them to Dedukti encodings of other systems and finally export these to the target systems. As we will see, however, things don't play out quite as neatly as this, as there are some aspects of Lean's type theory that do not seem to be directly encodable within Dedukti. Instead, we will have to use a strategy that involves a preliminary translation to a subset of Lean's theory, which we call "Lean$^-$". At each step along the way, we verify our output with a corresponding kernel. Starting from and input environment that is well-typed according the Lean kernel, we will translate it to an Lean$^-$ environment that is checked against a Lean$^-$ kernel. Next, we will translate this Lean$^-$ environment into a Dedukti library using our Lean$^-$ encoding, which we verify with the Dedukti kernel `dkcheck`. Finally, we translate this Dedukti library to the corresponding encodings of target systems (also verifying these libraries against `dkcheck`), and use previously implemented tools for exporting these libraries to be checked by the

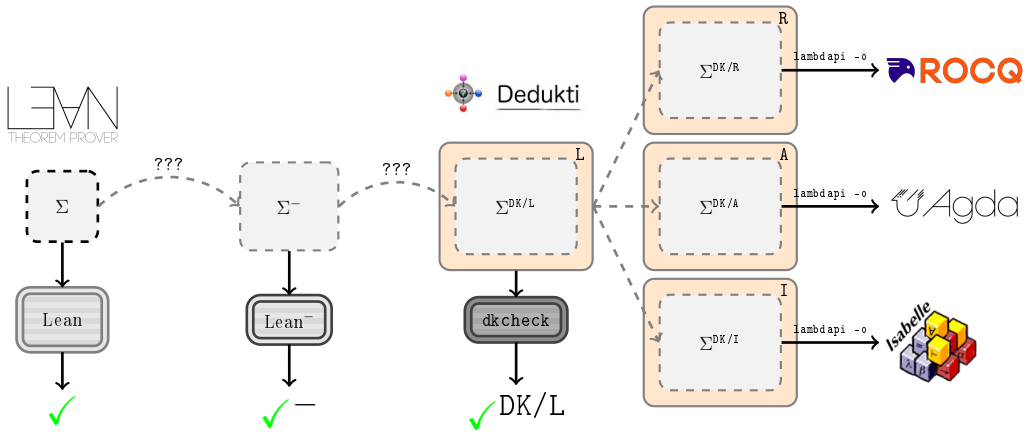

Figure 1.1: Exporting Lean to other systems via Dedukti.

target systems themselves. This overall process is visualized in Figure 1.1. More details on the Lean$^-$ theory and the details of the pre-translation strategy will be provided in later chapters.

Chapter 2

Translation Framework

In this chapter, we will introduce the basic framework of our translation from Lean to Dedukti. This translation will have to translate Lean terms to Dedukti terms in such a way that certain important translation correctness properties are satisfied, accounting for the differences between Lean and Dedukti's type theories. Let's start with a description of these desired properties, which will motivate the overall translation strategy that we subsequently describe.

2.1 Theoretical Motivations

In order for the output of our translation to be useful (i.e. amenable to the eventual export of Dedukti proofs to other systems), it must satisfy certain properties that capture some formal notion of translation "correctness". To introduce these properties, let's first establish our translation notation. Supposing t is some well-typed Lean term[1], let us use the notation $|t|^{\hookrightarrow}$ to represent the translation of t from Lean to Dedukti. Defined as such, $|\cdot|^{\hookrightarrow}$ is only a partial function, and we do not consider the translation of ill-typed terms. We use the same function applied to a Lean typing context, that is, $|\Delta|^{\hookrightarrow}$, to signify the translation of Δ to a Dedukti typing context.

We will also use the notation $\|t\|^{\hookrightarrow}$ to indicate a separate "as-type" translation of a Lean term t to a Dedukti term. This translation will be used in places where expressions of type `Type` are expected – specifically, in λ-function and function type binder domain types, as well as in symbol type declarations, and in the translation of Lean types that appear as types in Dedukti typing judgments[2]. We cannot use our ordinary "object-level" translation $|\cdot|^{\hookrightarrow}$ in these as we expect this translation always to produce terms typed as some encoding of a type in Dedukti, never `Type` itself. We will see that this as-type translation is in fact closely related to the translation $|\cdot|^{\hookrightarrow}$.

Note that both translations implicitly take typing contexts as arguments – this is necessary because, as we will see, part of our translation will depend on information that is not purely syntactic, requiring in particular the use of type inference subroutines to extract certain additional typing information that is needed by our translation. In general, inferring types requires complete knowledge of the current typing context, which is why our translation must also accept as input a representation of the Lean typing context that is applicable

[1] For convenience, from here on we will also take "Lean term" to always mean "well-typed Lean term".

[2] This can be seen, for instance, in our use of the as-type translation in presenting the translation soundness property below.

to the term that is currently being translated.

2.1.1 Completeness

One important property that we would like to ensure of our translation is that it produces Dedukti terms that have the same "meaning", in some sense, as the original Lean terms they were translated from. While it is important for our translation to output terms that are well-typed in our target system, this is certainly not the only criteria, as we would also like to ensure that the semantic interpretation of a term is preserved following its translation to Dedukti.

One way to characterize the "meaning" of propositions is in terms of provability. A property that we should certainly satisfy is that a proposition is provable in Lean if and only if its as-type translation to Dedukti is provable. Stated more generally in terms of types, we would say that a Lean type is inhabited if and only if its as-type translation to Dedukti is inhabited:

Property 2.1.1 (Correctness of the Encoding)**.** For all terms T such that $\Delta \vdash T : \texttt{Sort}\ \ell$ we have:

$$(\exists t, \Delta \vdash t : T) \iff (\exists t, |\Delta|^{\hookrightarrow} \vdash^{\hookrightarrow} t : \|T\|^{\hookrightarrow})$$

The forward direction is easy to show when we are able to show correctness of the translation, as characterized by the soundness property described below. So, let's think about the reverse direction, which is known the "completeness" property (also referred to in the literature as "conservativity"):

Property 2.1.2 (Translation Completeness)**.** For all terms T such that $\Delta \vdash T : \texttt{Sort}\ \ell$ we have:

$$(\exists t, |\Delta|^{\hookrightarrow} \vdash^{\hookrightarrow} t : \|T\|^{\hookrightarrow}) \implies (\exists t, \Delta \vdash t : T)$$

That is, the inhabitation of the translation of a type in the Dedukti encoding should imply the inhabitation of the type in the source theory.

Ensuring this property eliminates a certain class of undesirable translations, e.g. a translation that simply translates every proposition to an encoding of the inductive type `True` within Dedukti, allowing us to trivially satisfy the soundness property. Under such a translation, we would translate `False` to our encoding of `True`, which would be inhabited in Dedukti, while `False` is not inhabited in Lean, thus violating completeness. At a high level, the completeness property helps ensure that our translation is not "cheating" by translating Lean types to types with different semantics in the encoding.

Additionally, satisfying the completeness ensures that we have an encoding of Lean in Dedukti that is at least as sound as the source encoding. Supposing that we do translate types faithfully, a lack of completeness would point to a problem in our base encoding of Lean's type theory within Dedukti. Assuming that Lean is consistent, and our translation steps accurately preserve the semantics of terms, the ability to prove `False` within Dedukti would necessarily arise from an inconsistent encoding. As such, it is also important to ensure that we respect completeness as we proceed with the definition of our Dedukti encoding.

2.1.2 Soundness

While the completeness property is crucial to having a correct translation, it quite far from being sufficient on its own. Its usefulness is severely limited by its antecedent, which supposes the existence of a proof of a translated proposition. However, what if our as-type translation simply outputs `False` for every input Lean type? Then, completeness is trivially satisfied (assuming that `False` is not provable in our encoding), as the antecedent is always false for any term T.

In order to protect against this possible scenario, we need to verify another important translation property, known as "soundness". Soundness ensures that any inhabited Lean type has an as-type translation that is also inhabited in Dedukti:

Property 2.1.3 (Soundness)**.** For all terms T such that $\Delta \vdash T : \texttt{Sort}\ \ell$, we have:

$$(\exists t, \Delta \vdash t : T) \implies (\exists t, |\Delta|^{\hookrightarrow} \vdash^{\hookrightarrow} t : \|T\|^{\hookrightarrow})$$

This theorem can be formulated in terms of the translation by requiring that the translated inhabitant serves as the inhabitant in our target theory:

Property 2.1.4 (Translation Soundness)**.** For all terms t, T such that $\Delta \vdash T : \texttt{Sort}\ \ell$, we have:

$$\Delta \vdash t : T \implies |\Delta|^{\hookrightarrow} \vdash^{\hookrightarrow} |t|^{\hookrightarrow} : \|T\|^{\hookrightarrow}$$

With respect to our base Lean encoding, while completeness helps us to ensure that our encoding is not *excessively* permissive (so as to allow for a proof of `False`), soundness is a complementary property which ensures that our encoding is *sufficiently* permissive, in the sense that we are able to prove any property that we were originally able to prove.

2.1.3 Encoding Properties

To show the desired translation properties of soundness and completeness, we would like to verify certain properties of our encoding of Lean within Dedukti. Namely, we would like to show that our rewrite system satisfies the properties of termination and confluence, described below. Verifying these properties also give us greater confidence that the rewrite system is "bug-free" and correctly implemented w.r.t. its intended behavior, and incrementally analyzing whether they hold as we proceed in deriving our encoding will help us ensure that our translation as a whole has the desired correctness properties.

Termination

An important property to ensure about our Dedukti translation is that it produces a terminating rewrite system in combination with our base Lean encoding. Termination is required to ensure that a translated term can be practically typechecked. While a term may *theoretically* be well-typed in the $\lambda\Pi/R$ theory according to some typing derivation and constituent rewrite sequence, the actual typechecking computation performed by Dedukti may not follow these exact steps, which may be problematic if there are infinite rewrite sequences.

With respect to our translation, the termination property can be formally stated as follows:

Property 2.1.5 (Termination)**.** For all valid Lean typing contexts Δ, there exists no infinite sequence of terms $t_1, t_2, \ldots$ such that, for all $i \in \mathbb{N}$:

$$|\Delta|^{\hookrightarrow} \vdash^{\hookrightarrow} t_i \hookrightarrow_\beta t_{i+1}$$

While termination guarantees that we can always arrive at an irreducible form, it does not say anything about how "useful" this irreducible form is. In particular, we do not know whether this irreducible form is a "canonical" representative of the set of all terms that are equivalent to the original term in some sense. For that, we need an additional property to hold, known as confluence.

Confluence

For the existence of an irreducible form to be of practical use to us, it should help us decide some notion of equality between terms. The specific notion we care about here is equivalence modulo β-reduction/rewriting via the syntactic comparison of reduced forms which characterizes the definitional equality judgment used in Dedukti's conversion rule, recalled below:

$$\frac{\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta^* u_1 \quad \Delta \vdash^{\hookrightarrow} s \hookrightarrow_\beta^* u_2 \quad \Delta \vdash^{\hookrightarrow} u_1 =_{\mathrm{S}} u_2}{\Delta \vdash^{\hookrightarrow} t \equiv s} \text{ [DEQ]}$$

As rewrite rules encode judgmental equivalences between terms, the relation $\Delta \vdash^{\hookrightarrow} t \equiv s$ should ideally be an equivalence relation – that is, it should be reflexive, transitive and symmetric. As this relation is based on the syntactic comparison of reduced forms, for this to be the case, we must satisfy the property that any two terms related by the equivalence closure of $\Delta \vdash^{\hookrightarrow} t \hookrightarrow_\beta s$ eventually rewrite/β-reduce to unique and identical irreducible forms. That is, any irreducible form should correspond exactly to a "normal form", which is a unique representative of the equivalence class of terms defined by our rewrite rules and β-reduction.

Whether this property actually holds, however, depends on particular properties of our rewrite system. Specifically, it must be the case that our rewrite system satisfies the "confluence" property, which states that if a term s can be rewritten to two separate forms t_1 and t_2, then t_1 and t_2 can always be "unified" by rewriting to some term u. In terms of our translation, we can formalize this property as:

Property 2.1.6 (Confluence)**.** For all valid Lean typing contexts Δ and terms s, t_1, t_2,

$$|\Delta|^{\hookrightarrow} \vdash^{\hookrightarrow} s \hookrightarrow_\beta^* t_1 \wedge |\Delta|^{\hookrightarrow} \vdash^{\hookrightarrow} s \hookrightarrow_\beta^* t_2 \implies \exists u, |\Delta|^{\hookrightarrow} \vdash^{\hookrightarrow} t_1 \hookrightarrow_\beta^* u \wedge |\Delta|^{\hookrightarrow} \vdash^{\hookrightarrow} t_2 \hookrightarrow_\beta^* u$$

If we have confluence, we can be assured that any two terms that are related by the equivalence closure of β-reduction/rewriting will reduce to the same normal form. Satisfying this confluence property will help ensure soundness of our translation, as we defined our typing judgment $\Delta \vdash^{\hookrightarrow} t : T$ to require confluence of the $\lambda\Pi/R$ context Δ via the judgment $\texttt{conf}(\Delta)$– w.r.t. the translation soundness property, this requires confluence of the translated Lean context $|\Delta|^{\hookrightarrow}$.

Confluence is also an important "sanity check" to ensure that a rewrite system has been correctly implemented. It essentially encodes the notion that the particular order in which the rewrite rules of the rewrite system are applied has no effect[3] on the normal form that

[3] An exception to this is that Dedukti also allows for sequential rewrite rules, in which the order in which rewrite rules are applied can be specified by the user, resolving any ambiguity that may otherwise result in non-confluence. We do use sequential rewrite rules in some parts of our encoding, see Section 3.3.3.

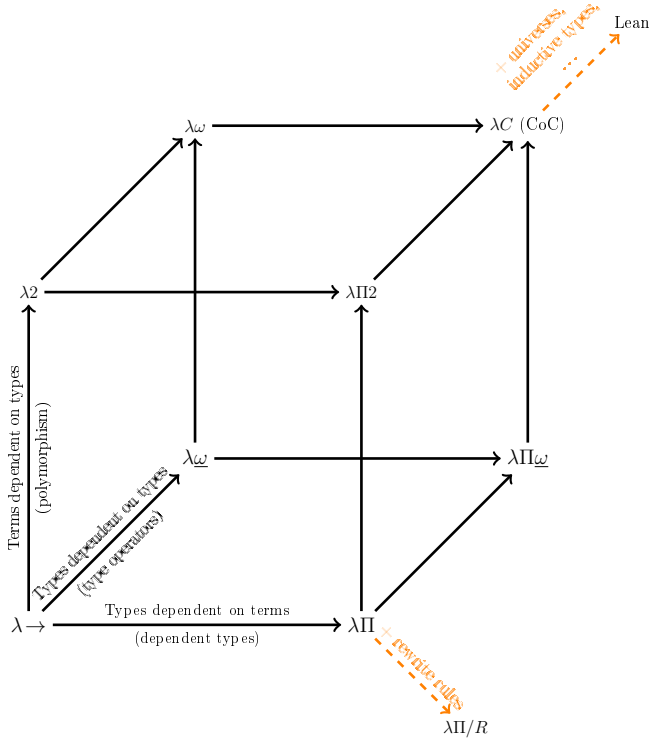

Figure 2.1: Barendregt's λ-cube

is ultimately arrived at. This is a good sign that the rewrite system has an unambiguous interpretation, and is therefore more likely to correctly implement the intended equivalence relation.

2.2 A Pure Type System Encoding

Let's start with the task of defining our base encoding, consisting of the initial set of type definitions and rewrite rules that will form the basis of our representation of Lean in Dedukti. To get started, it will be useful to first frame our translation task as a special instance of a translation from a more general theory that is well-studied, and for which a known translation already exists. For this, we look to a particular generalization of dependent type theories referred as "pure type systems".

2.2.1 Pure Type Systems

The type theories used by proof assistants often fall into a few generalized categories that can be seen as varieties or extensions of the eight points of Barendregt's λ-cube [6], shown in Figure 2.1.

The systems represented in the λ-cube use two type universes, "`Kind`" and "`Type`", with user-declared types living in `Type` and with `Type` having type `Kind` (with `Kind` itself being untypable). Each of the axes of the cube characterize the types of dependencies which are allowed, that is, the permissible forms of quantification/abstraction. When discussing the

theories of the lambda cube, we say that "types" are expressions that live in `Type`, while "terms" are expressions that live in some type.

Dedukti's type theory, the $\lambda\Pi$-calculus modulo rewrite rules, is an extension of the front-bottom-right corner of the cube, the $\lambda\Pi$ calculus. The $\lambda\Pi$ calculus allows for dependent types, where types can depend on terms. This is characterized by Dedukti typing rule for function types, recalled below:

$$\frac{\Delta \vdash^{\hookrightarrow} A : \texttt{Type} \quad \Delta, x : A \vdash^{\hookrightarrow} B : \texttt{Type}}{\Delta \vdash^{\hookrightarrow} (x : A) \ \to \ B : \texttt{Type}} \ [\text{ALL}]$$

In this rule, the typing of the codomain can utilize a variable that is bound to the domain type.

Notably, however, Dedukti does *not* allow for types to depend on types, meaning that types cannot be quantified over[4]. For instance, the following type declaration that attempts to declare a type-polymorphic identity function is not valid in Dedukti:

$$\texttt{id} \ : \ (t : \texttt{Type}) \to t \to t.$$

Lean, on the other hand, can be seen as an extension of the back-top-right corner of the lambda cube, known as the Calculus of Constructions (CoC). In CoC, types and terms can depend on bound variables representing types or terms, which can be seen as the most general form of dependent types. In particular, within CoC, it is possible to quantify over types, corresponding to the rule for function types in Lean:

$$\frac{\Delta \vdash A : \texttt{Sort} \ \ell \quad \Delta, x : A \vdash B : \texttt{Sort} \ \ell'}{\Delta \vdash (x : A) \ \texttt{→} \ B : \texttt{Sort} \ (\texttt{imax} \ \ell \ \ell')} \ [\text{ALL}]$$

Here, there is no restriction on the universe that the domain type inhabits, allowing in particular for the definition of type-polymorphic functions. This powerful capability makes CoC a common type-theoretic basis for several practical proof assistants.

Many of these systems can be interpreted as what are known as "pure type systems" (PTSs). A PTS is a generalized extension of CoC with a (possibly infinite) set of type universes (not necessarily just `Type` and `Kind`), along with special universe relations describing the typing rules relating to these universes. Precisely, a PTS is described by a set of universes S and a set of axioms $A \subseteq S \times S$ such that, for all $(s_1, s_2) \in A$:

$$\frac{}{\Delta \vdash^{\text{PTS}} s_1 : s_2}$$

And, for all $(s_1, s_2, s_3) \in R$:

$$\frac{\Delta \vdash^{\text{PTS}} A : s_1 \quad \Delta, x : A \vdash^{\text{PTS}} B : s_2}{\Delta \vdash^{\text{PTS}} (x : A) \ \to \ B : s_3}$$

where we use the notation $\Delta \vdash^{\text{PTS}} t : T$ to indicate the typing judgment of a generic PTS.

In particular, CoC can be considered a specific instance of a PTS defined by the following assignments:

$S := \{\texttt{Type}, \texttt{Kind}\}$

$A := \{(\texttt{Type}, \texttt{Kind})\}$

$R := \{(\texttt{Type}, \texttt{Type}, \texttt{Type}), (\texttt{Type}, \texttt{Kind}, \texttt{Kind}), (\texttt{Kind}, \texttt{Type}, \texttt{Type}), (\texttt{Kind}, \texttt{Kind}, \texttt{Kind})\}$.

[4]However, it is possible to achieve the equivalent of type quantification through the use of rewrite rules, see Section 2.2.3.

2.2.2 Lean as a Pure Type System

Lean's type theory can be shown to generally fit the mold of a PTS. Lean's type universes come in the form of an infinite hierarchy of "sorts" indexed by "universe levels". Sorts in Lean derive from the $\texttt{Sort}\ \ell$ expression variant:

$$e := \ldots \mid \texttt{Sort}\ \ell \mid \ldots$$

Recall that level expressions ℓ are taken from the following grammar:

$$\ell := u \mid \texttt{s}\ \ell \mid \texttt{max}\ \ell\ \ell \mid \texttt{imax}\ \ell\ \ell \quad u \in \mathcal{U}$$

with a particular interpretation under a universe level parameter instantiation (as was described in Section 1.2.2).

In terms of a PTS, Lean's sorts correspond to a universe set S such that

$$S := \{\texttt{Sort}\ \ell\},$$

that is, the set of all sorts indexed by arbitrary universe level expressions. Each sort is typable as the sort with the successive universe level, as expressed by the sort typing rule:

$$\frac{}{\Delta \vdash \texttt{Sort}\ \ell : \texttt{Sort}\ (\texttt{s}\ \ell)}\ [\text{SORT}]$$

This corresponds to the axiom set:

$$A := \{(\texttt{Sort}\ \ell, \texttt{Sort}\ (\texttt{s}\ \ell))\}$$

Lean allows for arbitrary quantification over types at every universe level, as expressed in the rule typing function types:

$$\frac{\Delta \vdash A : \texttt{Sort}\ \ell \quad \Delta, x : A \vdash B : \texttt{Sort}\ \ell'}{\Delta \vdash (x : A) \to B : \texttt{Sort}\ (\texttt{imax}\ \ell\ \ell')}\ [\text{ALL}]$$

This is consistent with Lean's type theory being interpreted as an extension of CoC. [ALL] corresponds to the following PTS rule set:

$$R := \{(\texttt{Sort}\ \ell, \texttt{Sort}\ \ell', \texttt{Sort}\ (\texttt{imax}\ \ell\ \ell'))\}.$$

2.2.3 Encoding Pure Type Systems in $\lambda\Pi/R$

Rather than extending the $\lambda\Pi$ calculus with additional quantifications to obtain CoC (which then generalizes to theories resembling that of Lean), an alternative way to extend it to improve its representational power is to add rewrite rules; in this way, we obtain the $\lambda\Pi$ calculus modulo rewriting that Dedukti is based on. As mentioned earlier, $\lambda\Pi$ does not allow for quantifying over types, and thus neither does $\lambda\Pi/R$. If our goal is to translate from PTS-like type theories such as Lean, where such quantifications *are* possible, how, then, are we to represent these quantifications within $\lambda\Pi/R$? Can we use the newly introduced rewrite rules to enable us to simulate this kind of quantification?

Meta-Types

The key observation made by Cousineau and Dowek is that while we cannot quantify over types in $\lambda\Pi/R$, we *can* quantify over a "meta-type" whose elements themselves represent types, making use of rewrite rules to effectively encode higher-order function types. In their PTS encoding, they create a type U_s for every universe in S:

$$\forall s \in S, \quad U_s \ : \ \texttt{Type}.$$

These are the meta-types that we will be able to quantify over in $\lambda\Pi/R$. To recover actual types from the elements of each meta-type, we also declare a "decoding function" symbol ϵ_s for each universe as follows:

$$\forall s \in S, \quad \epsilon_s \ : \ U_s \to \texttt{Type}.$$

Other than the U_s, we will never define a new type directly in `Type`: they will always be declared at the object-level as a member of some U_s, and their constructors will use ϵ_s to produce the output type. For instance, a type representing the natural numbers might be translated as:

$$\begin{aligned}
&\texttt{Nat} \ : \ U_s. \\
&\texttt{Nat.zero} \ : \ \epsilon_s \ \texttt{Nat}. \\
&\texttt{Nat.succ} \ : \ \epsilon_s \ \texttt{Nat} \to \epsilon_s \ \texttt{Nat}.
\end{aligned}$$

Axiom Encoding

We also need "type variable" symbols for each of the universes, which are typed as the universe they are meant to be typed as according to the axioms:

$$\forall (s_1, s_2) \in A, \quad \dot{s_1} \ : \ U_{s_2}.$$

These are accompanied by rewrite rules on the decoding function $\epsilon_s \ t$ hat allow them to be rewritten to the type universes they represent:

$$\forall (s_1, s_2) \in A, \quad [\dots] \ \epsilon_{s_2} \ \dot{s_1} \ \hookrightarrow \ U_{s_2}.$$

These symbols are useful, for instance, in defining certain type-polymorphic functions, which may themselves be applied to universes. For instance, suppose we add to CoC an additional `Prop` universe that lives inside of `Type`, giving rise to the axiom (`Prop`, `Type`) and the type variable:

$$\dot{\texttt{Prop}} \ : \ U_{\texttt{Type}}.$$

which produces the accompanying decoding function rewrite rule:

$$[] \ \epsilon_{\texttt{Type}} \ \dot{\texttt{Prop}} \ \hookrightarrow \ U_{\texttt{Prop}}.$$

Suppose that `Prop` is inhabited by the propositions `True` and `False`. This would be encoded as:

$$\begin{aligned}
\texttt{True} \ &: \ U_{\texttt{Prop}}. \\
\texttt{False} \ &: \ U_{\texttt{Prop}}.
\end{aligned}$$

Now, suppose we want to define a type-polymorphic identity function, encoded as:

$$\texttt{id} \ : \ (T : U_{\texttt{Type}}) \to \epsilon_{\texttt{Type}} \ T \to \epsilon_{\texttt{Type}} \ T.$$
$$[] \ \texttt{id} \ \hookrightarrow \ \lambda \ (T : U_{\texttt{Type}}) \ (t : \epsilon_{\texttt{Type}} \ T). \ t.$$

If we wish to instantiate this as the identity function for `Prop`, we need to use the $\dot{\texttt{Prop}}$ symbol, passed as the first argument to `id`. Thanks to the rewrite rule, this application types as:

$$\Delta \vdash^{\hookrightarrow} \texttt{id} \ \dot{\texttt{Prop}} : U_{\texttt{Prop}} \to U_{\texttt{Prop}},$$

enabling the application $\texttt{id} \ \dot{\texttt{Prop}} \ \texttt{True}$ to be well-typed and reduce to `True`.

Rule Encoding

To enable the representation of function type dependencies beyond those available to $\lambda\Pi/R$, we need to use a deep encoding of PTS function types, via the definition of the following "Pi variable" symbols:

$$\forall (s_1, s_2, s_3) \in R, \dot{\Pi}_{s_1,s_2} \ : \ (T : U_{s_1}) \to (\epsilon_{s_1} \ T \to U_{s_2}) \to U_{s_3}.$$

Note that the second argument of the Pi variable is a function that allows for the encoding of dependent types. These Pi variables come with the following set of rewrite rules:

$$\forall (s_1, s_2, s_3) \in R, [T, f] \ \epsilon_{s_3} \ (\dot{\Pi}_{s_1,s_2} \ T \ f) \ \hookrightarrow \ (t : \epsilon_{s_1} \ T) \to \epsilon_{s_2} \ (f \ t).$$

which enable them to rewrite to Dedukti's native dependent function types when applied to our

We can see the utility of Pi variables by again returning to our identity function example, where it enables an instantiation of `id` as a higher-order function. Suppose that $(\texttt{Type}, \texttt{Type}, \texttt{Type}) \in R$. This would correspond to the following Pi variable and rewrite rule:

$$\dot{\Pi}_{\texttt{Type},\texttt{Type}} \ : \ (T : U_{\texttt{Type}}) \to (\epsilon_{\texttt{Type}} \ T \to U_{\texttt{Type}}) \to U_{\texttt{Type}}.$$
$$[T, f] \ \epsilon_{\texttt{Type}} \ (\dot{\Pi}_{\texttt{Type},\texttt{Type}} \ T \ f) \ \hookrightarrow \ (t : \epsilon_{\texttt{Type}} \ T) \to \epsilon_{\texttt{Type}} \ (f \ t).$$

If we wish to instantiate `id` as the identity function for the function type $U_{\texttt{Prop}} \to U_{\texttt{Prop}}$, we need to instantiate the type argument of `id` as the encoding of this function type in `Type`:

$$\Delta \vdash^{\hookrightarrow} \texttt{id} \ (\dot{\Pi}_{\texttt{Type},\texttt{Type}} \ \dot{\texttt{Prop}} \ ((\_ : U_{\texttt{Prop}}) \ \texttt{=>} \ \dot{\texttt{Prop}})) : (U_{\texttt{Prop}} \to U_{\texttt{Prop}}) \to (U_{\texttt{Prop}} \to U_{\texttt{Prop}})$$

This allows the application $\texttt{id} \ (\dot{\Pi}_{\texttt{Type},\texttt{Type}} \ \dot{\texttt{Prop}} \ (\lambda(\_ : U_{\texttt{Prop}}).\dot{\texttt{Prop}})) \ (\lambda(\_ : U_{\texttt{Prop}}).\texttt{True})$ to be well-typed and reduce to $\lambda(\_ : U_{\texttt{Prop}}).\texttt{True}$.

Functionality

For our encoding to work, it is crucial that our PTS is "functional", in the sense of there being at most one parent universe for each universe, and at most one universe that every function type occupies given its domain and codomain universes. This can be precisely stated in the following rules:

$$(s_1, s_2) \in A \wedge (s_1, s_3) \in A \implies s_2 = s_3$$
$$(s_1, s_2, s_3) \in R \wedge (s_1, s_2, s_4) \in R \implies s_3 = s_4$$

Without this property, we would have to declare multiple types for the same type variable/Pi variable, which is of course not permitted in Dedukti. As such, the encoding above is only applicable to *functional* pure type systems. Cousineau and Dowek have shown that if a PTS is functional, a PTS-to-$\lambda\Pi/R$ translation based on this encoding respects both the soundness and completeness properties.

The functional PTS requirement is rather restrictive, as not all type systems used by real-world proof assistants can be framed as functional PTSs. For instance, the popular proof assistant Rocq, while having a type theory that is interpretable as a PTS, is not a *functional* PTS. While Rocq is similar to Lean in that it features an infinite universe hierarchy of indexed sorts, its universe hierarchy is cumulative. Cumulativity is particular case of subtyping applied to universes in Rocq, with any universe in Rocq having multiple parent universes – in fact, infinitely many – corresponding to the following axiom set:

$$\{(s_{\ell_1}, s_{\ell_2})\}, \ell_1 \leq \ell_2$$

This axiom set is clearly non-functional, because for any ℓ_1 there are infinitely many ℓ_2 such that $\ell_1 \leq \ell_2$. This means that the PTS encoding described by Cousineau and Dowek cannot be used as a base encoding of Rocq in Dedukti. However, alternative approaches to encoding cumulative universe hierarchies in $\lambda\Pi/R$ have been described, first by Assaf [4], with an extension satisfying completeness later given by Thiré [39], based on the strategy of making the cumulativity relation explicit in the translation.

Fortunately, however, Lean's base type theory *can* be interpreted as a functional PTS, as Lean does not have cumulativity. Recall the axiom and rule sets that we derived for Lean:

$$A := \{(\texttt{Sort}\ \ell, \texttt{Sort}\ (\texttt{s}\ \ell))\},$$
$$R := \{(\texttt{Sort}\ \ell_1, \texttt{Sort}\ \ell_2, \texttt{Sort}\ (\texttt{imax}\ \ell_1\ \ell_2))\},$$

It is easy to see that both sets satisfy the functionality requirement: in the axiom set, the sole parent universe of a universe is the successor of that universe, and in the rule set, the only universe that a function type can reside in is the `imax` of the domain and codomain universe indices.

2.2.4 A Pure Type System Encoding for Lean

As we have established that Lean's type theory can be interpreted as a variant of a functional pure type system, it should be feasible to define a base encoding of Lean within the $\lambda\Pi/R$ theory along the lines of the generic encoding described by Cousineau and Dowek. Let's get into the details of how we can practically implement such an encoding within Dedukti.

Firstly, let's define the symbols related to our encoding of Lean's universe levels in Dedukti. We declare type L type for translated universe levels in our encoding, along with "pseudo-constructors" corresponding to each of the variants of Lean's universe level syntax:

$$L\ \texttt{:}\ \texttt{Type}.$$
$$\texttt{def z}\ \texttt{:}\ L.$$
$$\texttt{def s}\ \texttt{:}\ L \to L.$$
$$\texttt{def max}\ \texttt{:}\ L \to L \to L.$$
$$\texttt{def imax}\ \texttt{:}\ L \to L \to L.$$
$$\texttt{def v}\ \texttt{:}\ \texttt{Nat} \to L \to L.$$

Note that these pseudo-constructors are in fact definable symbols, rather than static ones. This is because they have rewrite rules associated with them that allow us to compute a normal form – these are described in detail in Chapter 3. Also associated with normal form computation is the `inst` symbol, which is used to wrap the translation of universe level parameter instantiations in constant references (see Section 3.3.3):

$$\texttt{inst} \ : \ L \to L.$$

See Section 2.3 for the exact details on how we translate Lean's universe levels.

The PTS encoding presented by Cousineau and Dowek requires the definition of the encoding symbols U_s and ϵ_s for every $s \in S$, which is only practically possible in an actual encoding under the assumption of a finite set S of sorts. Because Lean's universe hierarchy is infinite, there is no hope of explicitly enumerating all of the possible universe symbols in our encoding. Additionally, we have to consider abstract universes arising from Lean's support for prenex universe level polymorphism, where we may have level parameters that arise from a constant declaration's universe context, as in the following universe-polymorphic identity function that is defined on any `Type u`:

```
def id.{u} (T : Type u) : T → T := fun t => t
```

According to Lean's typing, within the universe context of `id`, we have $\Delta \vdash$ `Type u` : `Type (u + 1)`, which needs to be reflected in our translation, where the universe parameter `u` will likely be represented by a bound variable.

This means that that our PTS-based encoding of Lean in Dedukti should necessarily be parametric in our declaration of the meta-type, decoding function, type variable, and Pi variable symbols[5]. We can define the parametric meta-type U and decoding function ϵ as follows:

$$\begin{aligned} U \ &: \ L \to \texttt{Type}. \\ \epsilon \ &: \ (l : L) \to U \ l \to \texttt{Type}. \end{aligned}$$

We can also declare the parametric type variable symbol $\dot{s}$ according to Lean's universe typing relation $\Delta \vdash$ `Sort` ℓ : `Sort (u + 1)`:

$$\dot{s} \ : \ (l : L) \to U \ (\texttt{s} \ l).$$

We can accompany this with a rule that decodes a given universe encoding into its corresponding `Type`-typed meta-type representation:

$$[l] \ \epsilon \ (\texttt{s} \ l) \ (\dot{s} \ l) \ \hookrightarrow \ U \ l.$$

However, note that the pattern `s` l is perhaps too restrictive over which terms the LHS can possibly match on. As mentioned earlier, `s` is a defined symbol in our encoding that we will define rewrite rules on for the purpose of computing universe level normal forms, as described in Chapter 3. In light of this, requiring the LHS to match on the exact pattern `s` l is likely to result in non-confluence in combination with the rules from our universe level encoding. So, we need to find a better way to state this rule.

[5] In this respect, our approach closely resembles that taken by Assaf [5] in an embedding of Rocq's infinite universe hierarchy in Dedukti.

Fortunately, it is in fact not necessary to specify a pattern in this position at all. Dedukti accepts the rewrite rule if we place a fresh pattern variable `x` in the spot of the first argument to ϵ.

$$[l, x]\ \epsilon\ x\ (\dot{s}\ l)\ \hookrightarrow\ U\ l.$$

This allows the rule to match on *any* term in that position.

At first glance, this may seem like a bad idea, since the argument needs to be of a very specific form for the rewrite rule to make sense: it should be equivalent to `s` l. If it is not, then the rewrite should not go through. However, as formalized by Blanqui [9] and Saillard [33], rewrite rules in Dedukti can only apply to already well-typed terms. In order for the LHS to be well-typed, this *must* be the case, as enforced by the dependent type of ϵ. This is verified by Dedukti's subject reduction checking algorithm, which verifies that the rule preserves typing: because the RHS types as `Type`, the LHS must too, and for that to be the case, `x` must have the same normal form as `s` l[6]. In fact, Dedukti allows us to state the rule a bit more compactly:

$$[l]\ \epsilon\ \_\ (\dot{s}\ l)\ \hookrightarrow\ U\ l.$$

Dedukti allows rewrite rules to contain placeholder patterns "`_`", denoting the use of a fresh pattern variable.

For our encoding of Lean's dependent function types, we define a parametric Pi variable symbol $\dot{\Pi}$ as follows:

$$\dot{\Pi}\ :\ (\ell_1 : L) \to (\ell_2 : L) \to (A : U\ \ell_1) \to (\epsilon\ \ell_1\ A \to U\ \ell_2) \to U\ (\texttt{imax}\ \ell_1\ \ell_2).$$

along with its decoding function rewrite rule, that rewrites it to Dedukti's native function types:

$$[\ell_1, \ell_2, A, B]\ \epsilon\ \_\ (\dot{\Pi}\ \ell_1\ \ell_2\ A\ B)\ \hookrightarrow\ (x : (\epsilon\ \ell_1\ A)) \to (\epsilon\ \ell_2\ (B\ x)).$$

2.3 The Syntax-Level Translation

Now, let's describe the exact translation we perform on well-typed Lean terms into Dedukti syntax, assuming the base PTS encoding described above. We will consider in turn each of the syntactic categories comprising the grammar of Lean expressions, defining the semantics of our translation in each case. Our translation will closely mirror that defined by Cousineau and Dowek, but there will also be some Lean-specific aspects to take into consideration. We define our translation on a (mostly) syntactic level as a recursive function on terms taken from our Lean grammar. We will require *some* non-syntactic information in the translation of dependent function type expressions, so, as mentioned earlier, our translation function will implicitly take into account the typing context Δ in order to be able to infer types as needed.

[6]Specifically, Dedukti invokes a unification algorithm which assigns a metavariable corresponding to `x` to the value `lvl.s l`. Knowledge of this assignment can then possibly be used later on in subject reduction checking (though this is not the case in this example).

Lean$^-$: A Restricted Source Theory

Before we can precisely specify the translation semantics, we must first refine the source theory of our translation. Thus far, we have been discussing things in terms of a translation $|\cdot|^{\hookrightarrow}$ from Lean to Dedukti. While it is indeed our end goal to define such a translation, for the sake of consistency of our notation with that of later sections, we must acknowledge the fact that the translation we define below really accepts input terms that are well-typed in a particular sub-theory of Lean, that we refer to as "Lean$^-$" (which uses the same syntax as Lean). We use Lean$^-$ as an intermediate theory between Lean and Dedukti, that is more amenable to a direct syntactic translation. The exact specifics of the Lean$^-$ theory are elaborated in Chapter 4.

Correspondingly, we define a new piece of notation, $|\cdot|^{\hookrightarrow}_{-}$ for our translation from Lean$^-$ to Dedukti. For the rest of this this chapter and Chapter 4, we only concern ourselves with this translation, with our soundness and completeness properties redefined with respect to it:

Theorem 2.3.1 (Lean$^-$-to-Dedukti Translation Soundness)**.** For all terms T such that $\Delta \vdash T : \texttt{Sort}\ \ell$, we have:

$$\Delta \vdash^{-} t : T \implies |\Delta|^{\hookrightarrow}_{-} \vdash^{\hookrightarrow} t : \|T\|^{\hookrightarrow}_{-}$$

Theorem 2.3.2 (Lean$^-$-to-Dedukti Translation Completeness)**.** For all terms T such that $\Delta \vdash T : \texttt{Sort}\ \ell$ we have:

$$(\exists t, |\Delta|^{\hookrightarrow}_{-} \vdash^{\hookrightarrow} t : \|T\|^{\hookrightarrow}_{-}) \implies (\exists t, \Delta \vdash^{-} t : T)$$

Later, in Chapter 5, we will define a preliminary translation $|\cdot|^{-}$, proving soundness and completeness for that preliminary translation, from which soundness and completeness of the composite translation $||\cdot|^{-}|^{\hookrightarrow}_{-}$ immediately follows.

Translating Contexts

Before we define our translation of actual Lean$^-$ terms into Dedukti terms, let's first describe the overall structure of the translation on the level of typing contexts, so that we can effectively reason about the correctness properties that we would like to show. Recall the overall structure of a Lean$^-$ context:

$$\Delta := (\Sigma; (\Gamma_U; \Gamma_B))$$

where Σ is a constant context, Γ_U is a universe level parameter context, Γ_B is a bound variable context. In stating our soundness property, we would like to define a translation $|\Delta|^{\hookrightarrow}_{-}$ on this context to a context of the form corresponding to Dedukti typing contexts:

$$|\Delta|^{\hookrightarrow}_{-} := ((\Sigma_D; \Sigma_R); \Gamma)$$

where Σ_D is a context of type declarations, Σ_R is a context of rewrite rules associated with the types in Σ_D, and Γ is a bound variable context.

We define our context-level translation as the following:

$$|(\Sigma; (\Gamma_U; \Gamma_B))|^{\hookrightarrow}_{-} := ((\Sigma^0_D, |\Sigma|^{\hookrightarrow}_{D}; \Sigma^0_R, |\Sigma|^{\hookrightarrow}_{R}); (|\Gamma_U|^{\hookrightarrow}_{-}, |\Gamma_B|^{\hookrightarrow}_{-}))$$

$|\Sigma|_D^{\hookrightarrow}$ and $|\Sigma|_R^{\hookrightarrow}$ represent the translation of a Lean constant context to a Dedukti type declaration context and rewrite rule context, respectively. Σ_D^0 and Σ_R^0 represent the type declarations and rewrite rules originating from the base PTS encoding, which are prepended to the translated type declaration and rewrite rule contexts. $(|\Gamma_U|_-^{\hookrightarrow}, |\Gamma_B|_-^{\hookrightarrow})$ is the translation of a Lean universe level parameter context and bound variable context to a Dedukti bound variable context, in which we concatenate the translation of the universe level parameter context with the translation of the bound variable context. The particular semantics of the translations $|\Sigma|_D^{\hookrightarrow}$, $|\Sigma|_R^{\hookrightarrow}$, $|\Gamma_U|_-^{\hookrightarrow}$, and $|\Gamma_B|_-^{\hookrightarrow}$ will be described below[7].

Translating Types

In defining our as-type translation, we take after the PTS encoding of Cousineau and Dowek, using the decoding function ϵ applied to the object-level translation of a term representing a Lean type. Specifically, for any type $\Delta \vdash T : \texttt{Sort}\ \ell$, we define our as-type translation as follows:

$$\|T\|_-^{\hookrightarrow} := \epsilon\ |\ell|_L\ |T|_-^{\hookrightarrow}.$$

In the case of T being an inductive type, the as-type translation of T will remain an application of ϵ, with $\epsilon\ |\ell|_L\ |T|_-^{\hookrightarrow}$ being the canonical representation of that inductive type in Dedukti. In the case of T being a universe or function type, however, $|T|_-^{\hookrightarrow}$ will be an application of an encoding-specific symbol $\dot{s}$ or $\dot{\Pi}$, and the associated decoding function rewrite rule will take effect, rewriting the application to an application of U or a Dedukti function type, respectively.

Constant Declarations and Universe Level Parameters

There are various forms of constant declarations in Lean, represented in Lean's metaprogramming framework with the following inductive type:

```
inductive ConstantInfo where
  | axiomInfo    (val : AxiomVal)
  | defnInfo     (val : DefinitionVal)
  | thmInfo      (val : TheoremVal)
  | opaqueInfo   (val : OpaqueVal)
  | quotInfo     (val : QuotVal)
  | inductInfo   (val : InductiveVal)
  | ctorInfo     (val : ConstructorVal)
  | recInfo      (val : RecursorVal)
```

Every constant declaration has an associated type, which we translate to Dedukti as a symbol declaration, whose type is the as-type translation of this type. For instance, consider the following definition in Lean:

```
def z : Nat := Nat.zero
```

[7] In this thesis, the translations $|\Sigma|_D^{\hookrightarrow}$, $|\Sigma|_R^{\hookrightarrow}$ will be specified mostly informally and through the use of examples. A formal description of these translations would require coming up with special notation to indicate every variety of a Lean constant declaration, together with a fully general presentation of their associated reduction rules, which is likely more trouble than it is worth for an initial presentation of our translation. See work by Assaf [5] and Ferey [17] for a more complete formal description of the translation of inductive type declarations.

This translates to Dedukti as the following symbol declaration:

$$\texttt{def z : } \epsilon\ (\texttt{s z})\ \texttt{Nat.}$$

Definitions, theorems, and opaque declarations in Lean also have an associated value expression, corresponding to the definition body/theorem proof, which we represent as the RHS of a rewrite rule defined directly the previously declared symbol. For instance, the translation of the body of `z` becomes the RHS of the following rewrite rule:

$$[]\ z \hookrightarrow \texttt{Nat.z.}$$

Adding rewrite rules for defined symbols in this way implements the equivalent of Lean's "δ-reduction" in our translation, in which constants may be substituted for their values in the course of reduction (see Section 4.2 for more discussion on δ-reduction). Additionally, since Dedukti performs subject-reduction-checking on rewrite rules to ensure that the LHS and RHS have the same types, the well-typedness of these rewrite rules ascertains that the translated values are well-typed according to their translated constant types, corresponding to the top-level typechecking of declared constants performed by the Lean kernel.

Axioms, inductive types, and constructors are simply defined as static symbols, without an associated rewrite rule[8]. For instance, consider the inductive type for natural numbers in Lean:

```
inductive Nat where
| zero : Nat
| succ (n : Nat) : Nat
```

This is translated to the following set of Dedukti symbols:

$$\begin{aligned}
&\texttt{Nat : } U\ (\texttt{s z}).\\
&\texttt{Nat.zero : } \epsilon\ (\texttt{s z})\ \texttt{Nat.}\\
&\texttt{Nat.succ : } \epsilon\ (\texttt{s z})\ \texttt{Nat} \to \epsilon\ (\texttt{s z})\ \texttt{Nat.}
\end{aligned}$$

Inductive type recursors are translated as defined symbols, as they have associated rewrite rules that we translate directly from the "rewrite rules" implemented natively for them by the Lean kernel (see Section 4.2).

However, there is one more important aspect of our translation of constant declarations that we have not yet mentioned: how exactly do we handle universe polymorphism? When we translate a Lean constant declaration, there may be universe level parameters associated with the constant identifier C using the declaration syntax C `{u, v, w, ...}`. This indicates a universe-polymorphic constant, where the universe level parameters must be instantiated wherever the constant is referenced later on (using this same syntax, but replacing the parameter names with concrete universe instantiations).

For instance, recall our universe-polymorphic identity function:

```
def id.{u} (T : Type u) : T → T := fun t => t
```

Any references to this constant would have to instantiate the universe level parameters, for example in the following definition that uses a particular instantiation of `id`:

[8]The only exception is for structure-like inductive types, where the constructor is declared as a defined symbol along with a rewrite rule corresponding to Lean's "struct-η" rule, see Section 4.1.3 for more details.

```
def idRef : Nat := id.{1} Nat Nat.zero
```

When the Lean kernel is typechecking the type and body of the constant declaration of `id`, these universe level parameters constitute the universe level parameter context Γ_U that forms parts of the typing context. The parameters in this context are relevant in the rules for deciding equality between universe levels, which is invoked by the rule for definitional equality of sort and constant expressions (recall the rules [CGR-SORT] and [CGR-CONST], defined in Section 1.2.2).

In our Dedukti translation, we will need to replicate the semantics of Lean's universe level equality checking algorithm, somehow. Such an algorithm will have to be aware of these universe level parameters in some way, in order to be able to properly compare universe level expressions. So we will need some way to simulate this universe level context in Dedukti. There are a couple of ways we could potentially go about this:

- We could define a "global" context of universe level parameters using some canonical representation for them (e.g. natural numbers), for instance, translating the type of `id` as:

  $$\texttt{def id :}\ (T : U\ z_n) \to \epsilon\ z_n\ T \to \epsilon\ z_n\ T.$$

  Here, z_n is the canonical representation of the first universe level parameter, $z_s\ z_n$ is the canonical representation of the second, and so forth. In this case, since we do not account for the level parameter contexts introduced by individual constants, we will have to ensure that our translation only generates such parameter representations within the range of the list of level parameters from the original level parameter context.

- Alternatively, we could simulate a level parameter context using Dedukti's bound variable context, by introducing in the translated symbol type declarations of universe-polymorphic Lean constants a set of additional arguments of type L, one for each universe level parameter. With this approach, the translation of the type of `id` becomes:

  $$\texttt{def id :}\ (l : L) \to (T : U\ l) \to \epsilon\ l\ T \to \epsilon\ l\ T.$$

  Here, instead of using natural numbers to represent universe level parameters in the translation of the definition body, we instead use the variables of Dedukti itself, which are bound by surrounding abstractions, one for each declared universe level parameter:

  $$[]\ \texttt{id} \hookrightarrow\ (l : L)\ (T : U\ l)\ (t : \epsilon\ l\ T)\ \texttt{=>}\ t.$$

These approaches correspond to "deep" and "shallow" encodings of level parameters, respectively. Initially, the latter approach may seem preferable as a more "direct" encoding that respects the original constant's universe level parameter context, with translation correctness in this regard enforced by the theory encoding, rather than by the specifics of the translation algorithm. However, a shallow encoding produces difficulties in our ability to implement a decision procedure for universe level equality in Dedukti based on a rewrite system that derives a normal form. A deep encoding, on the other hand, in addition to being less readily verifiable, complicates the task of universe level parameter instantiation in Dedukti. In the end, we decide to go with a "hybrid" encoding that incorporates aspects of both, see Chapter 3 for more details on the rationale behind this approach.

Our hybrid encoding does explicitly abstract over level parameters in the translated type of constants, as in a shallow encoding, with parameter instantiation corresponding to β-reduction via the application of constants to universe level instantiations. Our final translation of `id` appears as follows:

$$[]\ \texttt{id} \hookrightarrow (l : L)\ (T : U\ (\texttt{v Nat.zero}\ l))\ (t : \epsilon\ (\texttt{v Nat.zero}\ l)\ T)\ \texttt{=>}\ t.$$

The use of the special symbol `v` in the translation of universe level parameter references is described in more detail later in Section 2.3.

Note that L is a constant specific to our encoding – while it has similarities with the type `Lean.Level` from Lean's meta-programming framework (that is not available to ordinary Lean users), it is a different type that is used for different purposes, (in particular normalization, see Chapter 3). Terms of type L can only appear in our translation as the initial arguments to the `Sort` and `Pi` constructors from our PTS encoding, and as the initial level-instantiating arguments of references to universe-polymorphic constants. The type L itself only appears in our translation as the initial argument types of Π/λ binders in the translated types/values of universe-polymorphic constants.

This approach to handling universe polymorphism is reflected in the form of our context-level translation as described above, in which we append together a translation of the universe level parameter context with a translation of the bound variable context to obtain a bound variable context in Dedukti. Precisely, we can state our translation of a universe level parameter context as follows:

$$\begin{aligned} |[]|^{\hookrightarrow}_{-} &:= [] \\ |\Gamma_U, u|^{\hookrightarrow}_{-} &:= |\Gamma_U|^{\hookrightarrow}_{-}, u : L \end{aligned}$$

Constant References

As described above, our hybrid universe encoding turns a Lean constant declaration's universe level parameters into explicit function arguments of type L. Correspondingly, we translate any references to these constants carrying universe parameter instantiations (as a list of levels) into the corresponding Dedukti symbol applied to the translations of each of instantiating levels, in order. Formally, we define the following translation:

$$|C.\{\ell_1, \ldots, \ell_n\}|^{\hookrightarrow}_{-} := C\ (\texttt{inst}\ |\ell_1|_L)\ \ldots\ (\texttt{inst}\ |\ell_n|_L)$$

Where the universe level translation $|\ell|_L$ is defined below. The `inst` wrapper around instantiations of level parameters is necessary as part of our hybrid universe encoding for computing normal forms, see Section 3.3.3 for more details.

Type Universes

Lean's type universe hierarchy is made available to the user through the special `Sort` keyword, where the user can specify a type universe as `Sort` ℓ for some universe level term ℓ. However, just as in the case of universe level parameter instantiation for constant references, this is a specialized syntax that we need to account for in our translation, with the "argument" ℓ coming bundled together with the `Sort` keyword in Lean's syntax (as opposed to being an explicit application).

Similarly to the constant reference case, we translate sort expressions into explicit applications, using the $\dot{s}$ symbol from our PTS encoding applied to the translated universe level:

$$|\texttt{Sort}\ \ell|^{\hookrightarrow}_{\_} := \dot{s}\ |\ell|_L$$

Universe Levels

The actual translation of individual universe levels expressions must be specially handled by our translation, as they are a separate part of Lean's syntax with their own syntactic constructors and special interpretation during typechecking. In particular, the Lean kernel implements a special function for determining whether two universe level expressions should be considered semantically equivalent (that is, equivalent under any possible instantiation of the universe level parameters). The implementation of this check in Lean 4 is based on the partial computation of normal forms (and is currently incomplete), while Lean 3 implemented a complete but less efficient double inequality check algorithm, the semantics of which were described by Carneiro [13].

Recall the grammar for universe level terms:

$$\ell ::= u \mid \texttt{z} \mid \texttt{s}\ \ell \mid \texttt{max}\ \ell_1\ \ell_2 \mid \texttt{imax}\ \ell_1\ \ell_2$$

where $u \in \mathcal{U}$, with $\mathcal{U}$ being a set of universe level parameter symbols.

Our translation is simply a direct mapping from Lean's level terms to a L using these pseudo-constructors:

$$\begin{aligned}
|\texttt{z}|_L &:= \texttt{z} \\
|\texttt{s}\ \ell|_L &:= \texttt{s}\ \ell \\
|\texttt{max}\ \ell_1\ \ell_2|_L &:= \texttt{max}\ |\ell_1|_L\ |\ell_2|_L \\
|\texttt{imax}\ \ell_1\ \ell_2|_L &:= \texttt{imax}\ |\ell_1|_L\ |\ell_2|_L
\end{aligned}$$

The special consideration we have to make is for the translation of universe level parameters, in which we replace a named variable with a `v` construction that takes a unique canonical index as well as a free variable from the context, both of which are needed for the hybrid encoding scheme described in Section 3.3.3. Specifically, given the Lean universe level parameter context Γ_U containing parameter u at position i, we define the following translation:

$$|u|_L := \texttt{v}\ |i|_{\texttt{Nat}}\ u$$

Where the translation $|i|_{\texttt{Nat}}$ represents the translation of the natural number i to the following Peano representation in our Dedukti encoding:

```
Nat : Type.
Nat.zero : Nat.
Nat.succ : Nat → Nat.
```

Functions

Our translation of λ-functions takes after that of Cousineau and Dowek, directly mapping onto Dedukti's own λ-functions via a shallow encoding. Concretely, we define our translation

of Lean's λ-functions as:

$$|\texttt{fun}\ (x : T)\ \texttt{=>}\ b|^{\hookrightarrow}_{\_} := (x : \|T\|^{\hookrightarrow}_{\_})\ \texttt{=>}\ |b|^{\hookrightarrow}_{\_}$$

We make sure to use the as-type translation $\|\cdot\|^{\hookrightarrow}_{\_}$ for the translation of the domain type T, which respects Dedukti's requirement on the typing of lambda domains – namely, that they must live within Dedukti's `Type`.

Function Types

Dependent function types in Lean are identical in form to λ-functions, consisting of a bound variable from a specified domain, along with a term which may reference this bound variable. In the case of a λ-function, this term represents a function body, while in the case of a dependent function type, this term represents a dependent codomain type. Given this close similarity, our first inclination may be to define a similar translation to what we did in the λ-function case, directly mapping Lean's dependent function types onto Dedukti's dependent function types as follows:

$$|(x : A)\ \texttt{->}\ B|^{\hookrightarrow}_{\_} \stackrel{?}{:=} (x : \|A\|^{\hookrightarrow}_{\_})\ \rightarrow\ |B|^{\hookrightarrow}_{\_}$$

However, doing so would violate our soundness property. According to [ALL], the type of a function type in Lean with a domain in $\texttt{Sort}\ \ell$ and codomain in $\texttt{Sort}\ \ell'$ is $\texttt{Sort}\ (\texttt{imax}\ \ell\ \ell')$, whose as-type translation to Dedukti is $U\ (\texttt{imax}\ |\ell|_L\ |\ell|'_L)$. However, the type of function types in Dedukti is always inferred as `Type`. For our translation to be correctly typed, we need to encode it in Dedukti such that that it is typed under $U\ (\texttt{imax}\ |\ell|_L\ |\ell|'_L)$.

This was the exact purpose of the $\dot{\Pi}$ symbol we defined for our encoding. We define our translation such that, for any terms A, B such that $\Delta \vdash^{\_} \texttt{Sort}\ \ell$: and $\Delta, x : A \vdash^{\_} B : \texttt{Sort}\ \ell'$:

$$|(x : A)\ \texttt{->}\ B|^{\hookrightarrow}_{\_} := \dot{\Pi}\ |\ell|_L\ |\ell|'_L\ |A|^{\hookrightarrow}_{\_}\ ((x : \|A\|^{\hookrightarrow}_{\_})\ \texttt{=>}\ |B|^{\hookrightarrow}_{\_})$$

Thanks to the rewrite rule defined on ϵ when applied to $\dot{\Pi}$, the as-type translation of a function type rewrites to the form that we would expect:

$$\Delta \vdash^{\hookrightarrow} \|(x : A)\ \texttt{->}\ B\|^{\hookrightarrow}_{\_} \hookrightarrow_{\beta} (x : \|A\|^{\hookrightarrow}_{\_})\ \rightarrow\ |B|^{\hookrightarrow}_{\_}$$

Note that, unlike the previously defined translations, the translation of function type expressions is not purely syntactic, as we require knowledge of the domain and codomain universe levels ℓ and ℓ' in order to construct our translation[9].

In terms of our context-level translation, we translate bound variables introduced into the typing context by lambdas and dependent function types as follows:

$$|\Gamma_B, x : T|^{\hookrightarrow}_{\_} := |\Gamma_B|^{\hookrightarrow}_{\_}, x : \|T\|^{\hookrightarrow}_{\_}$$

[9] In regards to the actual implementation of our translation, this information is not present in the fields of `Expr.forallE`, which is the constructor corresponding to function types in the Lean metaprogramming framework. Therefore, our translation requires the use of Lean's type inference facilities in order to infer the sorts of domain types to extract the universe level terms from. To have access to these facilities, we implement our translation in Lean's `MetaM` monad, which is designed for writing metaprograms in Lean.

That is, we append a binder using the same identifier[10] to the translated bound variable context, using the as-type translation of the binder's original Lean type as the Dedukti binder type.

Applications

In translating application terms to Dedukti, we follow the strategy of Cousineau and Dowek, using a shallow encoding in which we translate Lean's applications into Dedukti's native application syntax:

$$|f\ a|^{\hookrightarrow}_{\_} := |f|^{\hookrightarrow}_{\_}\ |a|^{\hookrightarrow}_{\_}$$

Since our translation of λ-functions is also shallow, such a translation allows Lean's β-reduction to map directly onto Dedukti's β-reduction. This approach is in contrast to a deep encoding approach to translating applications, which might involve the use of a special binary application symbol in the encoding, along with an associated rewrite rule encode β-reduction. While such an approach has theoretical benefits in certain applications[11], for our purposes it has proven sufficient to take a shallow encoding approach.

Free Variables

As described above, we take a shallow encoding approach our encoding of λ-functions and applications, relying on Dedukti's native β-reduction for reducing β-redexes that would normally reduce in Lean, rather than on the use of special encoding-specific λ-function/application symbols along with an associated rewrite rule.

In accordance with this, we have no special encoding of bound variables in Lean, mapping them directly onto Dedukti's own variable references. Given a free variable context Γ_B, we translate an occurrence of some variable $x \in \Gamma_B$ to the same variable $x \in |\Gamma_B|^{\hookrightarrow}_{\_}$:

$$|x|^{\hookrightarrow}_{\_} := x$$

Let Binders

Recall that Lean features support for let-bindings (a.k.a. local definitions), having the syntax:

$$\texttt{let}\ (x : A)\ :=\ v\ \texttt{in}\ b$$

which declares that within the let binding body b, the variable x of type A has value v.

In order to translate Lean's let-binders to Dedukti, we will likely have to make use of some special encoding or translation, as Dedukti has no native support for let bindings. An easy way to define a correct translation of let binders is to simply expand references to the let-bound variable within the body of the let expression. Formally, this translation would look like:

$$|\texttt{let}\ (x : A)\ :=\ v\ \texttt{in}\ b|^{\hookrightarrow}_{\_} := |b[x/v]|^{\hookrightarrow}_{\_}$$

[10] Recall that the translated Lean universe level parameter context is concatenated with the translated Lean bound variable context in obtaining our Dedukti bound variable context. As such, we have to be sure that there are no conflicts between the variables names in the translated contexts that could create ambiguity regarding whether a Dedukti variable refers to a universe level parameter or a bound variable. In terms of the theory, we resolve this by requiring that the universe level parameter and bound variable name sets $\mathcal{U}$ and $\mathcal{X}$ are disjoint.

[11] See for instance, the work by Felicissimo et. al. [15], where it facilitates a proof of completeness in a possibly non-terminating system.

Note that this is also equivalent to the following translation:

$$|\texttt{let}\ (x : A)\ :=\ v\ \texttt{in}\ b|^{\hookrightarrow}_{\_} := |b|^{\hookrightarrow}_{\_}[x/|v|^{\hookrightarrow}_{\_}]$$

Such an approach would work, however it is not ideal since we duplicate the translated let-bound value expression $|v|^{\hookrightarrow}_{\_}$ wherever the let-bound variable x appears in b. Ideally, we would be able to preserve this sharing to some extent in our output.

In this direction, the closest analog of let-expressions in Dedukti seems to be β-redexes, which are expressions involving a lambda function application head applied to a value (which is immediately reducible via $[\text{BETA}]^D$). Specifically, we could imagine defining the translation:

$$\begin{aligned}|\texttt{let}\ (x : A)\ :=\ v\ \texttt{in}\ b|^{\hookrightarrow}_{\_} &\overset{?}{:=} |(\texttt{fun}\ (x : A)\ \texttt{=>}\ b)\ v|^{\hookrightarrow}_{\_} \\ &= ((x : \|A\|^{\hookrightarrow}_{\_})\ \texttt{=>}\ |b|^{\hookrightarrow}_{\_})\ |v|^{\hookrightarrow}_{\_}\end{aligned}$$

However, such a translation is not guaranteed to be sound, as the abstraction $(\texttt{fun}\ (x : A)\ \texttt{=>}\ b)\ v$ may not be well-typed in Lean$^-$ (and hence will not be well-typed in Dedukti either). The well-typedness of b in Lean$^-$ assumed that x was bound in the typing context with the value v. That is, we have $\Delta, x : A := v \vdash^{-} b : B$ for some term B. For the abstraction $\texttt{fun}\ (x : A)\ \texttt{=>}\ b$ to be well-typed, we require instead that $\Delta, x : A \vdash^{-} b : B$, that is, that b should be well-typed regardless of the value of x, which is a stronger requirement.

The reason that these two conditions are not equivalent is attributable in particular to dependent types, and can be seen concretely in the following example. Suppose that we have the symbol `f` below, which has a dependent function type that accepts a `Nat` as the second argument if the first argument is a `Nat.succ` application, and a `Unit` otherwise:

```
def F : Nat → Type
| Nat.zero => Unit
| Nat.succ _ => Nat

axiom f : (n : Nat) → F n → Type
```

The definition below uses `f` within a let binding, using the bound let variable as the first argument:

```
def letEx (n : Nat) : Type :=
  let x : Nat := Nat.succ n;
    f x Nat.zero
```

If we were to replace this let binding with a β-redex, we would end up with an ill-typed term:

```
def badRedex (n : Nat) : Type :=
  (fun (x : Nat) => f x Nat.zero) (Nat.succ n)
                      --^ Error: application type mismatch
```

The issue is that in abstracting away the particular value of `x`, we have lost information that is necessary for the application of `f` to be well-typed.

So, we can see that β-redexes do not provide a fully general solution to the problem of encoding let bindings within Dedukti. While it may indeed be possible to use a β-redex-based encoding in the majority of cases, where such type dependencies do not appear, in

general we need to think of a way to encode let binding in such a way that we preserve the particular value of the let-bound variable. In this regard, we take after the strategy of Assaf [5] in defining our translation of let binders, by constructing a "closure" on the let binder's bound value that is declared as an auxiliary function definition. This closure is constructed as a λ-function that abstracts over all of the bound variables present in the let binder's bound value, as well as the variables appearing in these variables' types (and the variables appearing in their types, and so on), with the body of the auxiliary function being the translation of the bound value. We then translate the body, replacing any references to the bound variable with an appropriate application of this auxiliary definition to the original variables that were abstracted in constructing the closure.

In this way, the value of the let-bound variable is made available to Dedukti's typechecker via rewriting and β-reduction of the auxiliary definition application. In terms of the example given above, this approach amounts to first defining the following closure definition for the let value:

```
def letAux (n : Nat) : Nat := Nat.succ n
```

We then replace the let binding with its body, substituting the instance of the let-bound variable with an application of `letAux`:

```
def letEx' (n : Nat) : Type :=
  f (letAux n) Nat.zero
```

In general, this approach is clearly not perfect, however, as we would still need to duplicate the auxiliary definition application everywhere the bound let variable appears. However, it seems to be the best that we can hope to do when translating to a system that lacks native support for let bindings.

Structure Projections

Recall that Lean allows for the definition of special structure-like inductive types which are inductive types with a single constructor and no indices. For instance, recall the `Point` struct from earlier, which generates projection functions that can be used to define, for example, a summation function:

```
inductive Point where
| mk : Nat → Nat → Point
def Point.sum (p : Point) : Nat := p.x + p.y
```

Without structures, projections would have to be defined in terms of recursors. For instance, we would have to define `Point.x` an `Point.y` as follows:

```
def Point.x (p : Point) : Nat :=
  Point.rec (fun x _y => x) p
def Point.y (p : Point) : Nat :=
  Point.rec (fun _x y => y) p
```

When applied to an explicit `Point` construction, these functions would have to reduce according to the ordinary recursor reduction rules (explained in more detail in Section 4.2), which is a bit less efficient than simply extracting the fields directly (as would be done by way of projections).

Dedukti, of course, has no notion of inductive types nor structure types, and as such its syntax does not allow for projections. However, we can define explicit projection function symbols corresponding to each of the possible projections of a structure, with corresponding rewrite rules that "simulate" the behavior of Lean's projection reduction. For instance, we might generate the following symbols and rewrite rules corresponding to each of the projections of `Point`:

$$\begin{aligned}
&\texttt{def } \texttt{prj}^0_{\texttt{Point}} \texttt{ : } (s : \texttt{Point}) \to \epsilon\ (\texttt{s z})\ \texttt{Nat}.\\
&\texttt{def } \texttt{prj}^1_{\texttt{Point}} \texttt{ : } (s : \texttt{Point}) \to \epsilon\ (\texttt{s z})\ \texttt{Nat}.\\
&[x, y]\ \texttt{prj}^0_{\texttt{Point}}\ (\texttt{Point.mk}\ x\ y) \hookrightarrow x.\\
&[x, y]\ \texttt{prj}^1_{\texttt{Point}}\ (\texttt{Point.mk}\ x\ y) \hookrightarrow y.
\end{aligned}$$

In general, given a structure type S with universe parameters $u_1 \dots u_k$, type parameters $(p_1 : P_1), \dots, (p_n : P_n)$ and fields $(a_1 : A_1)\ \dots\ (a_m : A_m)$, we generate the following projection symbols and rewrite rules for all $1 \leq i \leq m$:

$$\begin{aligned}
&\texttt{def } \texttt{prj}^i_S \texttt{ : } (u_1 : L) \to \dots \to (u_k : L) \to (p_1 : \|P_1\|^{\hookrightarrow}_{-}) \to \dots \to (p_n : \|P_n\|^{\hookrightarrow}_{-})\\
&\qquad \to (s : \|S\ u_1\ \dots\ u_k\ p_1\ \dots\ p_n\|^{\hookrightarrow}_{-}) \to A_i\ [a_1/s.1,\ \dots,\ a_{i-1}/s.(i-1)].\\
&[\dots]\ \texttt{prj}^i_S\ \ell'_1\ \dots\ \ell'_k\ p'_1\ \dots\ p'_n\ (S.\texttt{mk}\ \ell_1\ \dots\ \ell_k\ p_1\ \dots\ p_n\ f_1\ \dots\ f_n) \hookrightarrow f_i.
\end{aligned}$$

We use this symbol to define our translation of projection applications as follows, for any $s : S\ \ell_1\ \dots\ \ell_k\ p_1\ \dots\ p_n$:

$$|s.i|^{\hookrightarrow}_{-} := \texttt{prj}^i_S\ |\ell_1|_L\ \dots\ |\ell_k|_L\ |p_1|^{\hookrightarrow}_{-}\ \dots\ |p_n|^{\hookrightarrow}_{-}\ |s|^{\hookrightarrow}_{-}$$

Note that this is another instance where our translation is not purely syntactic, as we require information from the type of s (namely, the universe level parameter and parameter instantiations) that are not provided by the projection syntax alone.

Chapter 3

Universe Encoding

In this chapter, we will derive a representation and normalization scheme for Lean's universe levels in Dedukti that will allow us to reflect the rules [CGR-SORT] and [CGR-CONST] onto our Dedukti translation/encoding. In particular, we will have to have our normalization account for the equivalence relation $\ell_1 \approx \ell_2$ between universe levels. Let's start with a recap of Lean's universe level representation and semantics, and try to precisely identify the task at hand.

Recall the interpretation function on Lean's universe level expressions, that is then used to define an equivalence relation on universe levels that figures into the definitional equality rules [CGR-SORT] and [CGR-CONST]:

$$\begin{aligned}
\texttt{eval}_\sigma(u) &:= \sigma(u) \\
\texttt{eval}_\sigma(\texttt{z}) &:= 0 \\
\texttt{eval}_\sigma(\texttt{s}\ \ell) &:= \texttt{eval}_\sigma(\ell) + 1 \\
\texttt{eval}_\sigma(\texttt{max}\ \ell\ \ell') &:= \max(\texttt{eval}_\sigma(\ell), \texttt{eval}_\sigma(\ell')) \\
\texttt{eval}_\sigma(\texttt{imax}\ \ell\ \ell') &:= \begin{cases} 0 & \texttt{eval}_\sigma(\ell') = 0 \\ \max(\texttt{eval}_\sigma(\ell), \texttt{eval}_\sigma(\ell')) & \text{otherwise} \end{cases}
\end{aligned}$$

Our task is to derive a Dedukti encoding (in the form of a rewrite system) that exactly decides the equational theory $\ell_1 \approx \ell_2$. However, how exactly do we go about designing an algorithm that decides this relation in general? Obviously, deciding the relation would be trivial when both terms contain no universe level parameters– in this case, the universal quantification becomes irrelevant and we can simply implement a function that computes $\texttt{eval}_\sigma(\ell)$ and ignores the universe level parameter case, comparing the finally computed values.

With universe level parameters, however, things become more tricky, and we can no longer reduce the problem to checking the syntactic equality of computed natural numbers. At first glance, there is no obvious algorithm deciding $\ell_1 \approx \ell_2$ in such a way that all possible universe level parameter instantiations have been accounted for. One possible way about this is to instead focus on deciding some subsumption relation $\ell_1 \leq \ell_2$, defined as follows:

$$\ell_1 \leq \ell_2 \iff \forall \sigma, \texttt{eval}_\sigma(\ell_1) \leq \texttt{eval}_\sigma(\ell_2)$$

Now, we can observe the implication:

$$\ell_1 \leq \ell_2 \wedge \ell_2 \leq \ell_1 \implies \ell_1 \approx \ell_2$$

which reduces our problem to showing that: $\ell_1 \leq \ell_2$ and $\ell_2 \leq \ell_1$. This approach is indeed feasible, as a decision procedure for $\ell_1 \leq \ell_2$ does exist and was implemented in Lean 3 for deciding universe level equivalence via a double inequality check as described above. However, such an approach is not very applicable in our case, as we must fundamentally rely on syntactic equality of beta-rewrite normal forms to deciding universe level equality – specifically, due to our translation of universe level parameter instantiations as explicit arguments in Dedukti, the universe equivalence check that is normally handled by a specialized algorithm in Lean's kernel becomes an ordinary instance of definitional equality checking between terms in Dedukti.

As such, we need to revisit alternative possible methods for deciding equivalence between Lean's universe level terms, restricting ourselves in particular to methods that are based on the computation of *canonical normal form terms* for representing universe levels. In doing so, we will hopefully be able to arrive at some normal form representation and normalization procedure that is compatible with a Dedukti encoding.

Normalization Criteria

Let's start by trying to design some syntax L_N of normalized terms, along with an accompanying interpretation $\texttt{eval}_\sigma(\cdot)$[1]. This syntax and interpretation should ideally satisfy the following two properties:

1. Every term in L should have some semantically equivalent corresponding term in L_N:

   Theorem 3.0.1 (Sufficiency)**.** For any $\ell \in L$, that there exists some $\ell' \in L_N$ such that, for any variable assignment σ we have $\texttt{eval}_\sigma(\ell) = \texttt{eval}_\sigma(\ell')$.

   This ensures that it should at the very least be *possible* to arrive at an equivalent term in our normal form grammar for every term in our original grammar, following some sequence of transformations.

2. We would like to ensure that our normal form is "minimal" in the sense that any two semantically equivalent normal form terms are also syntactically equivalent:

   Theorem 3.0.2 (Uniqueness)**.** For any $\ell_1, \ell_2 \in L_N$, if $\ell_1 \approx \ell_2$ then $\ell_1 = \ell_2$.

   This is a very useful property for our normal form grammar to have, since it allows us to be sure that *any* semantics-preserving transformation from L to L_N must necessarily arrive at a unique normal form. Precisely, we can state the following lemma:

   Theorem 3.0.3. For any $\ell_1, \ell_2 \in L$, $\ell_1', \ell_2' \in L_N$, if $\ell_1 \approx \ell_1'$, $\ell_2 \approx \ell_2'$, and $\ell_1 \approx \ell_2$, then $\ell_1' = \ell_2'$.

   Proof. From transitivity and symmetry of $\approx$, we have $\ell_1' \approx \ell_2'$, and the conclusion follows from Theorem 3.0.2. □

[1] Rather than defining a new interpretation function for different universe level grammars, we instead take the approach of implicitly extending the domain of our existing interpretation function $\texttt{eval}_\sigma(\cdot)$ to accept terms from newly defined grammars. This is convenient in particular, because it allows us to use the same universe level equivalence operator (as previously defined in terms of $\texttt{eval}_\sigma(\cdot)$) in stating equivalences between universe level terms from different grammars.

We will arrive at our normal form grammar L_N as follows. Starting from the grammar L, we will progressively modify it, with each step guided by an analysis of whether our current grammar satisfies the uniqueness property. If we can find some counterexample – that is, a "redundant pair" of universe level terms that are syntactically distinct yet semantically equivalent – we will use this to guide the definition a new grammar that is just as expressive as the previous one while no longer exhibiting this particular redundancy, in this way narrowing down the classes of equivalent level expressions at each step and hopefully eventually arriving at a normal form grammar L_N for which we can prove the uniqueness property. Each successive modification we make to the grammar will be justified by showing that the new grammar is just as expressive as the previous one, namely by addressing the particular syntactic categories and corresponding interpretations that were removed/added in the grammar modification[2]. Justifying every step in this way, we can be sure that we finally arrive at a final normal form grammar that satisfies the sufficiency property.

3.1 Encoding a Predicative Universe Hierarchy

Let's first consider the case of a level grammar without the `imax` operator, corresponding to the level representation of a fully universe hierarchy in a predicative type system:

$$\begin{aligned} L_P &= \ell \text{ where} \\ \ell &::= u \mid \mathtt{z} \mid \mathtt{s}\ \ell \mid \mathtt{max}\ \ell_1\ \ell_2 \end{aligned}$$

with the following interpretation:

$$\begin{aligned} \mathtt{eval}_\sigma(x) &:= \sigma(x) \\ \mathtt{eval}_\sigma(\mathtt{z}) &:= 0 \\ \mathtt{eval}_\sigma(\mathtt{s}\ \ell) &:= \mathtt{eval}_\sigma(\ell) + 1 \\ \mathtt{eval}_\sigma(\mathtt{max}\ \ell_1\ \ell_2) &:= \max(\mathtt{eval}_\sigma(\ell_1), \mathtt{eval}_\sigma(\ell_2)) \end{aligned}$$

In determining a normal form for this simplified level representation, we will hopefully gather some intuition on how our normal form should look for the impredicative case.

3.1.1 Deriving a Normal Form

Starting from the grammar L_P and its interpretation, we check if our desired uniqueness property holds by searching for a counterexample, in the form of a redundant pair of universe level terms – namely, two syntactically distinct universe level terms that have the same interpretation under all possible substitutions. We can identify one as follows:

$$\mathtt{s}\ (\mathtt{max}\ \ell_1\ \ell_2) \approx \mathtt{max}\ (\mathtt{s}\ \ell_1)\ (\mathtt{s}\ \ell_2)$$

[2]Formally speaking, we prove sufficiency at every step by induction on the size of level expressions, omitting the trivial cases of syntactic categories/interpretations that were preserved from one grammar to the next. For conciseness of presentation, however, we will not describe our proofs in this level of formal detail, instead justifying the transformation by simply showing how any instance of a level term no longer expressible in the original grammar is expressible as some equivalent term in the new grammar (assuming by induction that any level subterms are already equivalently expressible in the new grammar).

This means that, given any term $\ell \in L_P$, we can convert it into an equivalent term $\ell' \in L_P^1$ where a `max` application never appears as an argument to `s`. Specifically, we have restricted L_P to the following grammar[3]:

$$\begin{aligned} L_P^1 &= \ell \text{ where} \\ \ell' &::= u \mid \texttt{z} \mid \texttt{s}\ \ell' \\ \ell &::= \ell' \mid \texttt{max}\ \ell_1\ \ell_2 \end{aligned}$$

From the above equivalence, we can be sure that L_P^1 is sufficient w.r.t. L_P, because the RHS is still expressible in L_P^1.

There are a few more redundancies in our grammar that we have to address. For instance, the interpretation of the `max` operator is both symmetric and associative, yielding the following equivalences:

$$\begin{aligned} \texttt{max}\ (\texttt{max}\ \ell_1\ \ell_2)\ \ell_3 &\approx \texttt{max}\ \ell_1\ (\texttt{max}\ \ell_2\ \ell_3) \\ \texttt{max}\ \ell_1\ \ell_2 &\approx \texttt{max}\ \ell_2\ \ell_1 \end{aligned}$$

Additionally, the grammar allows for redundant `max` applications:

$$\texttt{max}\ \ell\ \ell \approx \ell$$

To eliminate these three redundancies, we can add a new operator to our grammar, `maxS`, that takes as argument a *set* of so-called "sublevels", denoted using the symbol ω, which are level terms that do not include the `max` operator, and whose interpretation is simply the maximum of all of the interpretations of the elements of its set argument:

$$\begin{aligned} L_P^2 &= \ell \text{ where} \\ \omega &::= u \mid \texttt{z} \mid \texttt{s}\ \omega \\ \ell &::= \texttt{maxS}\ \{\omega_1, \omega_2, \ldots\} \end{aligned}$$

We know from the above equivalences that this grammar is sufficient w.r.t. L_P^1 (and thus w.r.t. L_P) as well.

There is still one last redundancy we have to address, however. While we are guaranteed that there are no duplicates in the sublevel set argument to `maxS`, there is still the possibility of sublevels being redundant w.r.t. other sublevels in the set, in terms of certain sublevels being "subsumed" by other sublevels. For instance, we can observe the following equivalences:

$$\begin{aligned} \texttt{maxS}\ \{\texttt{s}\ \texttt{z}, \texttt{z}\} &\approx \texttt{maxS}\ \{\texttt{s}\ \texttt{z}\} \\ \texttt{maxS}\ \{\texttt{s}\ u, u\} &\approx \texttt{maxS}\ \{\texttt{s}\ u\} \\ \texttt{maxS}\ \{\texttt{s}\ u, \texttt{s}\ \texttt{z}\} &\approx \texttt{maxS}\ \{\texttt{s}\ u\} \end{aligned}$$

In general, we can define the "subsumption operator" $\leq$ on sublevels that indicates whether or not the LHS sublevel is exceeded by the RHS sublevel for all possible universe level parameter instantiations.

$$\omega_1 \leq \omega_2 \iff \forall \sigma, \texttt{eval}_\sigma(\omega_1) \leq \texttt{eval}_\sigma(\omega_2)$$

[3]We could also think about applying the equality in the other direction, but this would entail restricting our grammar such that an application of `s` can only ever appear as *at most one* of the arguments to `max`, which is somewhat complex to express and probably not what we want.

We can use this operator to eliminate the above redundancies by adding a "pairwise incomparability" predicate on sublevel sets in our grammar, requiring that no sublevels in a set are subsumed by any other sublevel in the same set:

$$
\begin{aligned}
&L_P^3 = \ell \text{ where} \\
&\quad S ::= \{\omega_1, \omega_2, \dots\} \quad \forall \omega, \omega' \in S,\ \omega \neq \omega' \implies \omega \not\leq \omega' \\
&\quad \omega ::= u \mid \texttt{z} \mid \texttt{s}\ \omega \\
&\quad \ell ::= \texttt{maxS}\ S
\end{aligned}
$$

Lastly, supposing that we interpret $\texttt{eval}_\sigma(\texttt{maxS}\ \{\emptyset\}) := 0$, we have the following redundancy:

$$\texttt{maxS}\ \{\texttt{z}\} \approx \texttt{maxS}\ \{\emptyset\}$$

We can resolve this simply by requiring that any constant sublevels are nonzero[4], arriving at our final grammar L_P^*:

$$
\begin{aligned}
&L_P^* = \{\ell\} \text{ where} \\
&\quad \ell' ::= u \mid \texttt{s z} \mid \texttt{s}\ \ell' \\
&\quad S ::= \{\omega_1, \omega_2, \dots\} \quad \forall \omega, \omega' \in S,\ \omega \neq \omega' \implies \omega \not\leq \omega' \\
&\quad \ell ::= \texttt{maxS}\ S
\end{aligned}
$$

This final grammar closely resembles that of Blanqui [8], which itself derives from a grammar originally introduced by Genestier [18], and it is one that finally allows us to prove the uniqueness property[5].

3.2 Encoding an Impredicative Universe Hierarchy

Introducing the `imax` operator into our level grammar makes things more complicated. `imax` has a more complex semantic than `max`, in particular with its interpretation being asymmetric (and hence non-commutative) w.r.t. it arguments, and non-associative. In the process of refining towards a normal form grammar, we will have to discover new equalities that account for the interaction of `imax` with other symbols. While in the predicative case, we were able to find a normal form grammar that used a subset of the original grammar to represent sublevel terms, in this case, we will see that the presence of `imax` complicates things to the extent of requiring us to construct a new syntax and interpretation for sublevel terms.

3.2.1 Deriving a New Normal Form

As in the predicative case, we will proceed in our derivation of a normal form grammar for impredicative universe level terms by describing redundancies in our grammar in the form of semantically equivalent universe level terms, using them to refine our grammar and hopefully

[4]Alternatively, we can require that the argument of `maxS` is a non-empty set, with `maxS {z}` being the canonical representation of zero. With our approach, we use `maxS` $\emptyset$ instead as the canonical representation of zero.

[5]The actual proof of this property is similar to (and simpler than) that which we will show later for the normal form grammar that we derive for the impredicative case (see Section 3.2.2).

eventually arrive at a normal form grammar respecting the uniqueness property. Much of the derivation presented below, as well as the uniqueness proof found in Section 3.2.2, draws heavily from recently work by Géran [19], who first described and implemented a rewrite system for computing a normal form for universe level terms featuring an `imax` operator, and with whom the author collaborated in adapting this implementation for the translation from Lean to Dedukti. Proofs of the lemmas described below relating to the equivalence of level terms and the sublevel subsumption conditions can also be found in this work.

Extracting `max`

As before, we can pull instances instances of `max` out of `s`:

Lemma. For all $\ell_1, \ell_2 \in L$,

$$\texttt{s}\ (\texttt{max}\ \ell_1\ \ell_2) \approx \texttt{max}\ (\texttt{s}\ \ell_1)\ (\texttt{s}\ \ell_2)$$

To try to arrive at a similar normal form grammar where `max` can only appear nested within another `max`, we follow the intuition to try the same in the `imax` case, extracting `max` when it appears nested within the arguments to `imax`. We have to separately consider each argument, arriving at the following equalities:

Lemma. For all $\ell_1, \ell_2, \ell_3 \in L$,

$$\texttt{imax}\ (\texttt{max}\ \ell_1\ \ell_2)\ \ell_3 \approx \texttt{max}\ (\texttt{imax}\ \ell_1\ \ell_3)\ (\texttt{imax}\ \ell_2\ \ell_3)$$

Lemma. For all $\ell_1, \ell_2, \ell_3 \in L$,

$$\texttt{imax}\ \ell_1\ (\texttt{max}\ \ell_2\ \ell_3) \approx \texttt{max}\ (\texttt{imax}\ \ell_1\ \ell_2)\ (\texttt{imax}\ \ell_1\ \ell_3)$$

We can now proceed as we did in the predicative case, introducing a `maxS` operator and defining our level grammar with a category of sublevel terms ω as follows:

$$\begin{aligned} L^1 &= \ell \text{ where} \\ \omega &::= u \mid \texttt{z} \mid \texttt{s}\ \omega \mid \texttt{imax}\ \omega_1\ \omega_2 \\ \ell &::= \texttt{maxS}\ \{\omega_1, \omega_2, \ldots\} \end{aligned}$$

Restricting the `imax` Grammar

The presence of `imax` in our sublevel terms results in some additional redundancies that we will need to address. Let's analyze the interactions of `imax` with the other symbols of our grammar, and try to derive some equivalences that will allow us to restrict as much as possible the ways in terms involving `imax` can be constructed, while maintaining the same level of expressivity.

Firstly, if `z` is the second argument to `imax`, the `imax` application always evaluates to 0 (regardless of the universe level parameter assignment):

Lemma. For all ω,

$$\texttt{imax}\ \omega\ \texttt{z} \approx \texttt{z}$$

On the other hand, `z` is the first argument to `imax`, the `imax` application always evaluates to its second argument:

Lemma. For all ω,

$$\texttt{imax z } \omega \approx \omega$$

When an `s` application is the second argument to `imax`, it has the same semantics as the `max` operator:

Lemma. For all ω_1, ω_2,

$$\texttt{imax } \omega_1 \ (\texttt{s } \omega_2) \approx \texttt{max } \omega_1 \ (\texttt{s } \omega_2)$$

Consider, however, the case where an `s` application is the *first* argument of `imax`:

$$\texttt{imax } (\texttt{s } \omega_1) \ \omega_2$$

It is not clear what kind of equality can be derived here: ω_2 may still be relevant in evaluating the term to 0, so we can't simplify it to an expression using `max` instead of `imax` as we did before, leaving us seemingly "stuck" with the `imax` application in this case. A similar situation seems to arise when we consider a nested occurrence of `imax` as the first argument:

$$\texttt{imax } (\texttt{imax } \omega_1 \ \omega_2) \ \omega_3$$

which also does not seem to be amenable to any kind of simplification. So, for now, we accept both cases as valid constructions in our new grammar.

It is possible, however, to derive an equivalence when `imax` appears nested as the *second* argument to `imax`:

Lemma. For all $\omega_1, \omega_2, \omega_3$,

$$\texttt{imax } \omega_1 \ (\texttt{imax } \omega_2 \ \omega_3) \approx \texttt{imax } (\texttt{imax } \omega_1 \ \omega_3) \ (\texttt{imax } \omega_2 \ \omega_3)$$

We can also derive the following general equivalence when an `imax` application appears as an argument to `s`:

Lemma. For all ω_1, ω_2,

$$\texttt{s } (\texttt{imax } \omega_1 \ \omega_2) \approx \texttt{max } (\texttt{s } \omega_2) \ (\texttt{imax } (\texttt{s } \omega_1) \ \omega_2)$$

Deriving Custom Sublevels

So far, we have narrowed down our grammar to the following:

$$\begin{aligned} L^2 &= \ell \text{ where} \\ \gamma_c &::= \texttt{s z} \mid \texttt{s } \gamma_c \\ \gamma_v &::= u \mid \texttt{s } \gamma_v \\ \gamma &::= \gamma_c \mid \gamma_v \\ \omega &::= \gamma \mid \texttt{imax } \omega \ u \\ \ell &::= \texttt{maxS } \{\omega_1, \omega_2, \ldots\} \end{aligned}$$

That is, we have been able to restrict our sublevel grammar to terms of the following forms:

- $\mathtt{s}^{k+1}\ \mathtt{z}$: a constant level with value $k+1$.
- $\mathtt{s}^{k}\ u$: a universe level parameter augmented by k.
- $\mathtt{imax}\ (\mathtt{imax} \cdots (\mathtt{imax}\ (\mathtt{s}^{k+1}\ \mathtt{z})\ u) \cdots v)\ w$: A sequence of nested $\mathtt{imax}$ applications with only universe level parameters appearing as secondary arguments, with the innermost nested $\mathtt{imax}$ application taking as its first argument some constant level with value $k+1$.
- $\mathtt{imax}\ (\mathtt{imax} \cdots (\mathtt{imax}\ (\mathtt{s}^{k}\ x)\ u) \cdots v)\ w$: As above, but with the innermost nested $\mathtt{imax}$ application taking as its first argument some universe level parameter augmented by k.

Let's introduce the notation $\mathtt{imaxL}\ \gamma\ \alpha^*$, with α^* being a list of universe level parameters of non-zero length. Interpreting $\mathtt{imaxL}\ \gamma\ [u, \ldots, v, w]$[6] to represent the expression $\mathtt{imax}\ (\mathtt{imax} \cdots (\mathtt{imax}\ \gamma\ u) \cdots v)\ w$, we can state our grammar equivalently as follows:

$$
\begin{aligned}
L^2 &= \ell \text{ where} \\
\gamma_c &::= \mathtt{s}\ \mathtt{z} \mid \mathtt{s}\ \gamma_c \\
\gamma_v &::= u \mid \mathtt{s}\ \gamma_v \\
\gamma &::= \gamma_c \mid \gamma_v \\
\alpha^* &::= u \mid (\alpha^*, u) \\
\omega &::= \gamma \mid \mathtt{imaxL}\ \gamma\ \alpha^* \\
\ell &::= \mathtt{maxS}\ \{\omega_1, \omega_2, \ldots\}
\end{aligned}
$$

Essentially, under some universe level parameter instantiation σ, we interpret $\mathtt{imaxL}\ \gamma\ \alpha^*$ as the maximum of the streak of non-zero-assigned variables in the list α^*, starting from the right, and including γ in the case that all of them have non-zero assignments. If no such streak exists (i.e. if the last universe level parameter is assigned to zero), then the interpretation of the $\mathtt{imaxL}$ term is zero. Precisely defined, we have the following interpretation of $\mathtt{imaxL}$:

$$
\begin{aligned}
\mathtt{eval}_\sigma(\mathtt{imaxL}\ \gamma\ u) &:= \mathtt{eval}_\sigma(\mathtt{imax}\ \gamma\ u) \\
\mathtt{eval}_\sigma(\mathtt{imaxL}\ \gamma\ (\alpha^*, u)) &:= \mathrm{imax}(\mathtt{eval}_\sigma(\mathtt{imax}\ \gamma\ u), \sigma(u))
\end{aligned}
$$

Note that we have snuck an extra restriction into this grammar: as in the predicative case, we assume that we cannot have a sublevel of the form $\mathtt{z}$, fixing the interpretation $\mathtt{eval}_\sigma(\mathtt{maxS}\ \emptyset) := 0$, making $\mathtt{maxS}\ \emptyset$ our canonical representation of zero and eliminating the redundancy $\mathtt{maxS}\ (S \cup \{\mathtt{z}\}) \approx \mathtt{maxS}\ S$ (when $\mathtt{z} \notin S$).

This grammar, however, still does not quite get us all the way there, as we have the equivalence $\mathtt{imaxL}\ \gamma\ [v, w, v] \approx \mathtt{imaxL}\ \gamma\ [w, v]$. The issue is that the consideration of earlier variables in the list is "guarded" by the condition of all later variables being nonzero, making it meaningless for a variable to guard itself. In general, if we ever have duplicate instances of some variable in the list, we can always remove all but the last occurrence: if the variable is assigned to zero, its value is irrelevant to the maximum anyways, and otherwise it would have already been accounted for by a later occurrence. So, we modify the category α^* to enforce a non-duplication list predicate that ensures that this is the case:

$$\alpha^* ::= u \mid (\alpha^*, u) \quad u \notin \alpha^*$$

[6] We use the notation $[u, \ldots, v, w]$ as a shorthand for the list construction $(u, \ldots (v, w))$.

Thus far, we have only addressed redundancy between individual sublevels; however, we should also account for redundant *combinations* of sublevels. We can observe one such redundancy as follows:

$$\texttt{maxS}\ \{\texttt{imaxL}\ u\ [v], \texttt{imaxL}\ v\ [u]\} \approx \texttt{maxS}\ \{u, v\}$$

Here, the LHS sublevels $\texttt{imaxL}\ u\ [v]$ and $\texttt{imaxL}\ v\ [u]$ are incomparable, with neither being subsumed by the other, and so we cannot eliminate either of them from the term. Therefore, there a pairwise incomparability predicate on sublevel sets would not eliminate the possibility of constructing the LHS term and be insufficient in enforcing a unique normal form. And yet, we can still see that we are able to simplify both terms "in parallel", if we consider each in light of the other. Analyzing the problem more precisely, we can discern that it arises in the "overloaded" nature of the elements in the list argument to `imaxL`: these elements both act as guard parameters over previous elements, and also figure into the maximum themselves. Is there a way to separate these responsibilities?

Let's try narrowing things down to an even simpler sublevel grammar, introducing a new sublevel representation $\texttt{imaxS}\ \gamma\ S$, where S is a set of "guard parameters" such that if any of them are set to zero, the sublevel evaluates to zero, otherwise evaluating to γ. Precisely, we define its interpretation as follows:

$$\texttt{eval}_\sigma(\texttt{imaxS}\ \gamma\ S) \coloneqq \begin{cases} 0 & \exists s \in S, \sigma(s) = 0 \\ \texttt{eval}_\sigma(\gamma) & \text{otherwise} \end{cases}$$

Running with the example above, it seems that we can split $\texttt{imaxL}\ u\ [v]$ into two components: $\texttt{imaxS}\ u\ \{v\}$ and $\texttt{imaxS}\ u\ \emptyset$. In this way, we make explicit the distinct semantic aspects of the sublevel $\texttt{imaxL}\ u\ [v]$: namely that the level v is always considered by the outer `maxS`, and the level u is only also considered if v does not evaluate to zero. We can similarly split $\texttt{imaxL}\ v\ [u]$ into the two components $\texttt{imaxS}\ v\ \{u\}$ and $\texttt{imaxS}\ u\ \emptyset$. This leaves us with the term:

$$\texttt{maxS}\ \{\texttt{imaxS}\ u\ \{v\}, \texttt{imaxS}\ v\ \emptyset, \texttt{imaxS}\ v\ \{u\}, \texttt{imaxS}\ u\ \emptyset\}$$

Now, we can see that $\texttt{imaxS}\ u\ \{v\}$ is subsumed by $\texttt{imaxS}\ u\ \emptyset$, and $\texttt{imaxS}\ v\ \{u\}$ is subsumed by $\texttt{imaxS}\ v\ \emptyset$, allowing us to arrive at the simplified equivalent term:

$$\texttt{maxS}\ \{\texttt{imaxS}\ u\ \emptyset, \texttt{imaxS}\ v\ \emptyset\}$$

The question, now, is whether such a decomposition of `imaxL` sublevels to sets of `imaxS` sublevels can be done in general. As a matter of fact, it can, as demonstrated by the following theorem:

Lemma. For all $\ell \in \gamma_c \cup \gamma_v$ and all level parameters $u_1, \ldots, u_n$, we have:

$$\texttt{imaxL}\ \ell\ [u_1, \ldots, u_n] \approx \texttt{maxS}\ \{\texttt{imaxS}\ \ell\ \{u_1, \ldots, u_n\}, \texttt{imaxS}\ u_1\ \{u_2, \ldots, u_n\}, \ldots, \texttt{imaxS}\ u_n\ \{\}\}$$

which can easily be verified by inspection.

Therefore, it is possible for us to restrict our grammar as follows:

$$\begin{aligned} L^3 &= \ell \text{ where} \\ S &::= \{u_1, u_2, \dots\} \\ \gamma_c &::= \mathtt{s\ z} \mid \mathtt{s}\ \gamma_c \\ \gamma_v &::= u \mid \mathtt{s}\ \gamma_v \\ \gamma &::= \gamma_c \mid \gamma_v \\ \omega &::= \mathtt{imaxS}\ \gamma\ S \\ \ell &::= \mathtt{maxS}\ \{\omega_1, \omega_2, \dots\} \end{aligned}$$

Refining the Guard Variable Set

We are now quite close to the final form of our normal form grammar. However there is another redundancy to consider in our level representation, which can be seen in the following equivalence:

$$\mathtt{imaxS}\ u\ \emptyset \approx \mathtt{imaxS}\ u\ \{u\}$$

Here, the presence of u in the guard parameter set does not affect the evaluation of the term. We can also derive the following equivalence:

$$\mathtt{imaxS}\ (\mathtt{s}\ u)\ \emptyset \approx \mathtt{maxS}\ \{\mathtt{imaxS}\ (\mathtt{s}\ u)\ \{u\}, \mathtt{imaxS}\ (\mathtt{s\ z})\ \emptyset\}$$

We require the additional sublevel $\mathtt{imaxS}\ (\mathtt{s\ z})\ \{\}$ in the $\mathtt{maxS}$ argument on the RHS in order to preserve the equality of the respective evaluations in the case that u is assigned to zero. In general, we can derive the following equivalence:

Lemma. For any $u \in \mathcal{U}$ and set $S \subset \mathcal{U}$ such that $u \notin S$,

$$\mathtt{imaxS}\ (\mathtt{s}^k\ u)\ S \approx \mathtt{maxS}\ \{\mathtt{imaxS}\ (\mathtt{s}^k\ u)\ (S \cup \{u\}), \mathtt{imaxS}\ (\mathtt{s}^k\ \mathtt{z})\ S\}$$

What this means for our normal form grammar is that we must assume that whenever we have a term of the form $\mathtt{imaxS}\ (\mathtt{s}^k\ u)\ S$, we have $u \in S$ (as otherwise the above equivalence would apply).

To ease the presentation of our final grammar, as well as the presentation of our comparability predicate below, let's define a couple more pieces of notation. We denote a "variable sublevel" with the notation $\mathrm{V}\ k\ u\ S$, interpreted as follows:

$$\mathtt{eval}_\sigma(\mathrm{V}\ k\ u\ S) := \mathtt{eval}_\sigma(\mathtt{imaxS}\ (\mathtt{s}^k\ u)\ S)$$

That is, $\mathrm{V}\ k\ u\ S$ represents a $\mathtt{imaxS}$ sublevel where the first argument is some variable u augmented by k, guarded by the parameters appearing in S. We denote a "constant sublevel" with, the notation $\mathrm{C}\ k\ S$, interpreted as:

$$\mathtt{eval}_\sigma(\mathrm{C}\ k\ S) := \mathtt{eval}_\sigma(\mathtt{imaxS}\ (\mathtt{s}^k\ \mathtt{z})\ S)$$

$\mathrm{C}\ k\ S$ represents a $\mathtt{imaxS}$ sublevel where the first argument has a constant value evaluating to k, guarded by the universe level parameters appearing in S.

Using this notation, we can describe our new grammar as the following:

$$\begin{aligned} L^4 &= \ell \text{ where} \\ S_p &::= \{u_1, u_2, \dots\} \\ \omega_c &::= \mathrm{C}\ (k+1)\ S_p \\ \omega_v &::= \mathrm{V}\ k\ u\ (S_p \cup \{u\}) \\ \omega &::= \omega_c \mid \omega_v \\ S_\omega &::= \{\omega_1, \omega_2, \dots\} \\ \ell &::= \texttt{maxS}\ S_\omega \end{aligned}$$

with $k \in \mathbb{N}$. Note that by specifying the grammar of constant sublevels as $\mathrm{C}\ (k+1)\ S_p$, we have preserved the requirement from our previous grammar that they must be nonzero.

Deriving Subsumption Conditions

Our final task in constructing our normal form grammar is to add an incomparability predicate on the elements of sublevel sets. As in the predicative case, this is necessary to ensure that we do not allow for sublevels to be present in a sublevel set that are subsumed by other sublevels, as their removal would result in an equivalent term, violating the uniqueness property.

As before, this predicate makes use of a subsumption operator $\leq$ that has the following interpretation:

$$\ell_1 \leq \ell_2 \iff \forall \sigma, \texttt{eval}_\sigma(\ell_1) \leq \texttt{eval}_\sigma(\ell_2),$$

That is, ℓ_1 is subsumed by ℓ_2 when the evaluation of ℓ_1 is less than or equal to the evaluation of ℓ_2 under all possible parameter instantiations σ. We could then enforce the requirement that no terms are subsumed by other terms as a predicate on the definition of the syntactic category of sublevel sets:

$$S_\omega ::= \{\omega_1, \omega_2, \dots\} \quad \forall \omega, \omega' \in S_\omega,\ \omega \neq \omega' \implies \omega \not\leq \omega'$$

For our encoding, it is important that the relation $\ell_1 \leq \ell_2$ corresponds to a function that is amenable to an implementation as a rewrite system, so that we can effectively maintain the incomparability predicate when rewriting level terms to their normal forms. We would like to find a set of necessary and sufficient requirements on ℓ_1 and ℓ_2 that are simple enough to effectively allow us to decide whether $\ell_1 \leq \ell_2$ via a rewrite system. We derive these conditions in the following lemmas:

Lemma. For all $u \in \mathcal{U}$, $S, S' \subset \mathcal{U}$ and $k, k' \in \mathbb{N}$,

$$\mathrm{V}\ k\ u\ S \not\leq \mathrm{C}\ k'\ S'$$

Lemma. For all $S, S' \subset \mathcal{U}$ and $k, k' \in \mathbb{N}$,

$$\mathrm{C}\ k\ S \leq \mathrm{C}\ k'\ S' \iff S' \subseteq S \wedge k \leq k'$$

Lemma. For all $u \in \mathcal{U}$, $S, S' \subset \mathcal{U}$ and $k, k' \in \mathbb{N}$,

$$\mathrm{C}\ k\ S \leq \mathrm{V}\ k'\ u\ S' \iff S' \subseteq S \wedge k \leq k' + 1$$

Lemma. For all $u, u' \in \mathcal{U}$, $S, S' \subset \mathcal{U}$ and $k, k' \in \mathbb{N}$,

$$\mathrm{V}\ k\ u\ S \leq \mathrm{V}\ k'\ u'\ S' \iff S' \subseteq S \wedge u = u' \wedge k \leq k'$$

As we will see in Section 3.3.1, these subsumption conditions easily give way to a rewrite system that enables us to compare sublevels for the purpose of maintaining the incomparability predicate when computing normal forms.

The Final Grammar

At this point, we have arrived at the following normal form grammar for representing universe levels in Lean:

$$\begin{aligned}
&L_N = \ell \text{ where} \\
&S_p ::= \{u_1, u_2, \dots\} \\
&\omega_c ::= \mathrm{C}\ (k+1)\ S_p \\
&\omega_v ::= \mathrm{V}\ k\ u\ (S_p \cup \{u\}) \\
&\omega ::= \omega_c \mid \omega_v \\
&S_\omega ::= \{\omega_1, \omega_2, \dots\} \quad \forall \omega, \omega' \in S_\omega,\ \omega \neq \omega' \implies \omega \not\leq \omega' \\
&\ell ::= \texttt{maxS}\ S_\omega
\end{aligned}$$

By making incremental changes to our grammar and justifying each step, we have shown that this grammar satisfies the sufficiency property. However, for this to be a proper normal form grammar that we can target in our universe level encoding, we need to be sure that it also satisfies the uniqueness property. Upon inspection, it would seem that we can derive no more redundancies in this representation. To be entirely sure of this, however, we would like to prove that it is the case.

3.2.2 Uniqueness of the Normal Form

Having fully defined our normal form grammar L_N, let's now proceed with the proof of Theorem 3.0.2. For convenience, we start by establishing the following notation: for some normal form term $\ell := \texttt{maxS}\ S_\omega$, we use $\omega \in \ell$ to indicate $\omega \in S_\omega$. Our strategy for proving the uniqueness property of our normal form grammar is to prove that for any two equivalent normal form terms, the presence of a particular sublevel in the sublevel set argument to one implies the presence of the same sublevel in the other. Formally, we can state this property as:

$$\forall \ell,\ \ell' \in L_N, \ell \approx \ell' \implies \forall \omega, \omega',\ \omega \in \ell \iff \omega' \in \ell'$$

We start with the case of variable sublevels, first showing a relaxed property that we will later strengthen:

Lemma 3.2.1. For all $\ell, \ell' \in L_N$ such that $\ell \approx \ell'$,

$$\mathrm{V}\ k\ u\ S \in \ell \implies \mathrm{V}\ k\ u\ S' \in \ell' \text{ with } S \subseteq S'$$

Proof. Our strategy for proving this will be to design a particular universe level parameter instantiation that results in an interpretation of ℓ that "elicits" the existence a sublevel in ℓ' with the properties we seek.

Firstly, let's establish the following notation to extract the constant value of a sublevel:

$$\theta : \omega \to \mathbb{N} \text{ where } \theta(\mathrm{V}\ k\ u\ S) := k \text{ and } \theta(\mathrm{C}\ k\ u) := k$$

To start our proof, let's suppose that we have some $\mathrm{V}\ k\ u\ S \in \ell$. Suppose that we define a universe level parameter instantiation σ such that the interpretation of ℓ necessarily arises from the sublevel $\mathrm{V}\ k\ u\ S$:

$$\sigma(v) := \begin{cases} 1 + \max\{\theta(\omega), \omega \in \ell \text{ or } \omega \in \ell'\} & \text{if } v = u \\ 1 & \text{if } v \in S \setminus \{u\} \\ 0 & \text{otherwise} \end{cases}$$

Then, $\texttt{eval}_\sigma(\ell) = \texttt{eval}_\sigma(\mathrm{V}\ k\ u\ S) = k + \sigma(u)$. Since $\ell \approx \ell'$, we also have $\texttt{eval}_\sigma(\ell') = k + \sigma(u)$. For this to be the case, we must have that either:

- There is some $\mathrm{C}\ k'\ S' \in \ell'$ with $k' = k + \sigma(u)$.
- There is some $\mathrm{V}\ k'\ u'\ S' \in \ell'$ with $k' + \sigma(u') = k + \sigma(u)$ and $S' \subseteq S \cup \{u\}$ (as otherwise $\texttt{eval}_\sigma(\mathrm{V}\ k'\ u'\ S') = 0$).

It cannot be the first case because we chose $\sigma(u) > \max\{\theta(\ell), \ell \in \ell \text{ or } \ell \in \ell'\}$.

Therefore, it must be the second case, with $u' = u$ We can therefore deduce that $k' = k$. Additionally, from the fact that our grammar restricts sublevels of the form $\mathrm{V}\ k\ u\ S$ to have $u \in S$, we know that $S \cup \{u\} = S$, so $S' \subseteq S$, giving us our result. □

We can now strengthen this property as follows:

Lemma 3.2.2. For all $\ell, \ell' \in L_N$ such that $\ell \approx \ell'$,

$$\mathrm{V}\ k\ u\ S \in \ell \iff \mathrm{V}\ k\ u\ S \in \ell'$$

Proof. It suffices to show one direction of the proof, as the reverse direction is symmetric. So, let's assume that we have some $\mathrm{V}\ k\ u\ S \in \ell$. Then, by Theorem 3.2.1, we we have that there is some $\mathrm{V}\ k\ u\ S' \in \ell'$ with $S' \subseteq S$. If $S' \subset S$, then by Theorem 3.2.1 again, we have that there is some $\mathrm{V}\ k\ u\ T \in \ell$ in ℓ with $T \subset S' \subset S$, so $T \subset S$. However, this is a contradiction with our incomparability predicate as $\mathrm{V}\ k\ u\ T$ would then be comparable with $\mathrm{V}\ k\ u\ S$. Therefore, it must be the case that $S' = S$, completing the proof. □

The next property to show is that any constant sublevel in ℓ in also necessarily present in ℓ':

Lemma 3.2.3. For all $\ell, \ell' \in L_N$ such that $\ell \approx \ell'$,

$$\mathrm{C}\ k\ S \in \ell \iff \mathrm{C}\ k\ S \in \ell'$$

Proof. We show only the forward direction, with the backward direction following symmetrically. So, let's suppose that we have some sublevel C $k\ S \in \ell$, and proceed by induction on the size of S.

For the base case, suppose $S = \emptyset$. Let σ be an universe level parameter instantiation such that $\sigma(x) = 0$ for all $x \in \mathcal{U}$. Then, we know that $\texttt{eval}_\sigma(\ell) = k$, as otherwise there must exist some sublevel C $k'\ \emptyset \in \ell$ with $k' > k$, which would be comparable with C $k\ S$. Therefore, because $\ell \approx \ell'$, we have $\texttt{eval}_\sigma(\ell') = k$. Because our grammar requires that sublevels of the form C $k\ S$ have $k > 0$, there must also be some sublevel C $k\ S \in \ell'$.

For the inductive step, we pick σ such that:

$$\sigma(u) := \begin{cases} 1 & \text{if } u \in S \\ 0 & \text{otherwise} \end{cases}$$

Then, we know that $\texttt{eval}_\sigma(\ell) = k$, as otherwise there must exist some sublevel C $k'\ S' \in \ell$ with $S' \subseteq S$ and $k' > k$, or V $k'\ u\ S' \in \ell$ with $S' \subseteq S$ and $k' + 1 > k$, both of which would be comparable with C $k\ S$.

Therefore, because $k > 0$, we have that either:

- There is some C $k'\ S' \in \ell'$ with $S' \subseteq S$ and $k' = k$.
- There is some V $k'\ u\ S' \in \ell'$ with $S' \subseteq S$ and $k' + \sigma(u) = k$.

In the second case, because $S' \subseteq S$ and $u \in S'$, we have $\sigma(u) = 1$. Therefore, $k' = k - 1$, and by Theorem 3.2.2 we have V $(k-1)\ u\ S' \in \ell$, which is a contradiction as this sublevel is comparable with C $k\ S$.

In the first case, suppose $S' \subset S$. Then, we can apply the inductive hypothesis to obtain that there is some C $k\ T \in \ell$ with $T \subset S$. However, this is a contradiction as such a sublevel would be comparable with C $k\ S$. Therefore, $S' = S$, giving us our result. □

3.3 Implementation as a Rewrite System

While we have shown our normal form grammar to be theoretically sound for our purposes, let's move on to the task of designing a practical rewrite system that implements it, effectively computing a β-rewrite normal form for level-typed terms that respects the restrictions imposed by the grammar L_N.

3.3.1 Base Encoding

To get ourselves started with the task of deriving a rewrite rule encoding for computing universe level normal forms, let's assume the following set of top-level symbols for constructing universe levels:

$$\begin{aligned} &L\ \texttt{:}\ \texttt{Type}. \\ &\texttt{def z :}\ L. \\ &\texttt{def s :}\ L \to L. \\ &\texttt{def max :}\ L \to L \to L. \\ &\texttt{def imax :}\ L \to L \to L. \\ &\texttt{def v :}\ \texttt{Nat} \to L. \end{aligned}$$

This slightly differs from the final set of symbols that was first presented in Section 2.2.4, in that it lacks the `inst` symbol (to be introduced later), and variables are represented by a single natural number via the constructor `v`, which is characteristic of a deep encoding. The need for the `inst` symbol and the extra `v` argument will become apparent when we address the instantiation issue, as described later in Section 3.3.3. Note that all of these top-level constructors are in fact defined symbols, as we will be defining rewrite rules on them to enable the computation of their equivalent normal form representations.

Booleans

To define a few relations and branching operations that will be used by our universe level encoding, we will use the following simple representation of booleans:

```
Bool : Type.
false : Bool.
true : Bool.
```

Natural Numbers

Recall the Peano representation of natural numbers in our encoding:

```
Nat : Type.
Nat.zero : Nat.
Nat.succ : Nat → Nat.
```

To represent the universe level parameter set arguments to our sublevels, we use ordered lists of natural numbers, represented with the following type and constructors:

$$
\begin{aligned}
&S_{\texttt{Nat}} \texttt{ : Type.}\\
&\texttt{nil}_{\texttt{Nat}} \texttt{ : } S_{\texttt{Nat}}\texttt{.}\\
&\texttt{cons}_{\texttt{Nat}} \texttt{ : Nat} \to S_{\texttt{Nat}} \to S_{\texttt{Nat}}\texttt{.}
\end{aligned}
$$

Note that although we will refer to $S_{\texttt{Nat}}$ as a "set", it is isomorphic to an ordinary list of natural numbers, with nothing to ensure that it behaves like a set – namely, there is no guarantee that it does not contain duplicate elements, and there is no way to ascertain that Dedukti will consider two `Set`s with the same elements but in different orders to be equivalent to one another.

For this, we need to maintain certain invariant properties about our lists, using special symbols to construct them with associated rewrite rules that ensure they do not contain duplicates and are always sorted in increasing order. In doing so, we can check whether or not two level parameter set encodings are equivalent by simply checking syntactic equality

of their normal forms. We start by defining a comparison function on natural numbers:

$$\begin{aligned}
&\text{CMP} \ \texttt{:}\ \texttt{Type}.\\
&\texttt{lt} \ \texttt{:}\ \text{CMP}.\\
&\texttt{eq} \ \texttt{:}\ \text{CMP}.\\
&\texttt{gt} \ \texttt{:}\ \text{CMP}.
\end{aligned}$$

$$\begin{aligned}
&\texttt{def}\ \texttt{cmp}_{\texttt{Nat}} \ \texttt{:}\ \texttt{Nat} \to \texttt{Nat} \to \text{CMP}.\\
&[]\ \texttt{cmp}_{\texttt{Nat}}\ \texttt{Nat.zero}\ \texttt{Nat.zero} \hookrightarrow \texttt{eq}.\\
&[]\ \texttt{cmp}_{\texttt{Nat}}\ \texttt{Nat.zero}\ (\texttt{Nat.succ}\ _) \hookrightarrow \texttt{lt}.\\
&[]\ \texttt{cmp}_{\texttt{Nat}}\ (\texttt{Nat.succ}\ _)\ \texttt{Nat.zero} \hookrightarrow \texttt{gt}.\\
&[n, m]\ \texttt{cmp}_{\texttt{Nat}}\ (\texttt{Nat.succ}\ n)\ (\texttt{Nat.succ}\ m) \hookrightarrow \texttt{cmp}_{\texttt{Nat}}\ n\ m.
\end{aligned}$$

Note that these operators are linear in the size of the natural numbers involved: this is of course far from ideal, but for practical purposes it shouldn't matter very much because there are often only a few universe levels in the context at a time when typechecking Lean definitions, which will carry over into our translation, which assigns indices to level parameters starting from zero.

To help us maintain that $S_{\texttt{Nat}}$ are always ordered and without duplicates, we define an $\texttt{add}_{\texttt{Nat}}$ operator that adds a number to a set, avoiding duplication and maintaining the ordering of the underlying list representation. This makes use of a $\texttt{cases}_{S_{\texttt{Nat}}}$ operator that branches based on the cases of a natural number being less than, equal to, or greater than another natural number:

$$\begin{aligned}
&\texttt{def}\ \texttt{cases}_{S_{\texttt{Nat}}} \ \texttt{:}\ \text{CMP} \to S_{\texttt{Nat}} \to S_{\texttt{Nat}} \to S_{\texttt{Nat}} \to S_{\texttt{Nat}}.\\
&[a]\ \texttt{cases}_{S_{\texttt{Nat}}}\ \texttt{lt}\ a\ _\ _ \hookrightarrow a.\\
&[b]\ \texttt{cases}_{S_{\texttt{Nat}}}\ \texttt{eq}\ _\ b\ _ \hookrightarrow b.\\
&[c]\ \texttt{cases}_{S_{\texttt{Nat}}}\ \texttt{gt}\ _\ _\ c \hookrightarrow c.
\end{aligned}$$

$$\begin{aligned}
&\texttt{def}\ \texttt{add}_{\texttt{Nat}} \ \texttt{:}\ \texttt{Nat} \to S_{\texttt{Nat}} \to S_{\texttt{Nat}}.\\
&[n]\ \texttt{add}_{\texttt{Nat}}\ n\ \texttt{nil}_{\texttt{Nat}} \hookrightarrow \texttt{cons}_{\texttt{Nat}}\ n\ \texttt{nil}_{\texttt{Nat}}.\\
&[n, l, m]\ \texttt{add}_{\texttt{Nat}}\ m\ (\texttt{cons}_{\texttt{Nat}}\ n\ l) \hookrightarrow\\
&\quad \texttt{cases}_{S_{\texttt{Nat}}}\ (\texttt{cmp}_{\texttt{Nat}}\ n\ m)\\
&\qquad (\texttt{cons}_{\texttt{Nat}}\ n\ (\texttt{add}_{\texttt{Nat}}\ m\ l))\\
&\qquad (\texttt{cons}_{\texttt{Nat}}\ n\ l)\\
&\qquad (\texttt{cons}_{\texttt{Nat}}\ m\ (\texttt{cons}_{\texttt{Nat}}\ l\ n)).
\end{aligned}$$

We will ensure that we only ever use this operator (rather than using $\texttt{cons}_{\texttt{Nat}}$ directly) to construct $S_{\texttt{Nat}}$ terms. $\texttt{add}_{\texttt{Nat}}$ implements linear search to maintain the sorted order, which again is not ideal but perhaps acceptable at the small scales on which it will be applied.

We also define a few more operations on $S_{\texttt{Nat}}$, that we will find useful in subsumption-checking:

- A "set merge" operation on $S_{\texttt{Nat}}$ that allows us to simulate taking the union of two natural number sets by adding all of the elements of one to the other:

$$\begin{aligned}
&\texttt{def}\ \cup_{\texttt{Nat}}\ \texttt{:}\ S_{\texttt{Nat}} \to S_{\texttt{Nat}} \to S_{\texttt{Nat}}.\\
&[l]\ \cup_{\texttt{Nat}}\ \texttt{nil}_{\texttt{Nat}}\ l \hookrightarrow l.\\
&[l_1, l_2, n]\ \cup_{\texttt{Nat}}\ (\texttt{cons}_{\texttt{Nat}}\ l_1\ n)\ l_2 \hookrightarrow \cup_{\texttt{Nat}}\ l_1\ (\texttt{add}_{\texttt{Nat}}\ n\ l_2).
\end{aligned}$$

- An operator that checks whether or not one set is a subset of another one, using a membership operator:

$$\begin{aligned}
&\texttt{def}\ \texttt{cases}_{\texttt{Bool}}\ \texttt{:}\ \text{CMP} \to \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool} \to \texttt{Bool}.\\
&\ldots\quad \text{(rewrite rules for } \texttt{cases}_{\texttt{Bool}}\text{)}
\end{aligned}$$

$$\begin{aligned}
&\texttt{def}\ \in_{\texttt{Nat}}\ \texttt{:}\ \texttt{Nat} \to S_{\texttt{Nat}} \to \texttt{Bool}.\\
&[]\ \in_{\texttt{Nat}}\ _\ \texttt{nil}_{\texttt{Nat}} \hookrightarrow \texttt{false}.\\
&[n, m, l]\ \in_{\texttt{Nat}}\ n\ (\texttt{cons}_{\texttt{Nat}}\ m\ l) \hookrightarrow\\
&\quad \texttt{cases}_{\texttt{Bool}}\ (\texttt{cmp}_{\texttt{Nat}}\ n\ m)\ \texttt{false}\ \texttt{true}\ (\in_{\texttt{Nat}}\ n\ l).
\end{aligned}$$

$$\begin{aligned}
&\texttt{def}\ \subseteq_{\texttt{Nat}}\ \texttt{:}\ S_{\texttt{Nat}} \to S_{\texttt{Nat}} \to \texttt{Bool}.\\
&[]\ \subseteq_{\texttt{Nat}}\ \texttt{nil}_{\texttt{Nat}}\ _ \hookrightarrow \texttt{true}.\\
&[n, l, l']\ \subseteq_{\texttt{Nat}}\ (\texttt{cons}_{\texttt{Nat}}\ n\ l)\ l' \hookrightarrow\\
&\quad \texttt{and}\ (\in_{\texttt{Nat}}\ n\ l')\ (\subseteq_{\texttt{Nat}}\ l\ l').
\end{aligned}$$

- An operator that checks whether or not two lists are equal in terms of both being subsets of one another:

$$\begin{aligned}
&\texttt{def}\ \texttt{Eq}_{\texttt{Nat}}\ \texttt{:}\ S_{\texttt{Nat}} \to S_{\texttt{Nat}} \to \texttt{Bool}.\\
&[l_1, l_2]\ \texttt{Eq}_{\texttt{Nat}}\ l_1\ l_2 \hookrightarrow \texttt{and}\ (\subseteq_{\texttt{Nat}}\ l_1\ l_2)\ (\subseteq_{\texttt{Nat}}\ l_2\ l_1).
\end{aligned}$$

Lastly, we define a lexicographic order on $S_{\texttt{Nat}}$ via the operator $\texttt{LT}_{S_{\texttt{Nat}}}$:

$$\begin{aligned}
&\texttt{def}\ \texttt{LTE}_{S_{\texttt{Nat}}}\ \texttt{:}\ S_{\texttt{Nat}} \to S_{\texttt{Nat}} \to \texttt{Bool}.\\
&[]\ \texttt{LTE}_{S_{\texttt{Nat}}}\ \texttt{nil}_{\texttt{Nat}}\ _ \hookrightarrow \texttt{true}.\\
&[]\ \texttt{LTE}_{S_{\texttt{Nat}}}\ (\texttt{cons}_{\texttt{Nat}}\ _\ _)\ \texttt{nil}_{\texttt{Nat}} \hookrightarrow \texttt{false}.\\
&[n, m, l_1, l_2]\ \texttt{LTE}_{S_{\texttt{Nat}}}\ (\texttt{cons}_{\texttt{Nat}}\ n\ l_1)\ (\texttt{cons}_{\texttt{Nat}}\ m\ l_2) \hookrightarrow\\
&\quad \texttt{cases}_{\texttt{Bool}}\ (\texttt{cmp}_{\texttt{Nat}}\ n\ m)\ \texttt{true}\ (\texttt{LTE}_{S_{\texttt{Nat}}}\ l_1\ l_2)\ \texttt{false}.
\end{aligned}$$

$$\begin{aligned}
&\texttt{def}\ \texttt{LT}_{S_{\texttt{Nat}}}\ \texttt{:}\ S_{\texttt{Nat}} \to S_{\texttt{Nat}} \to \texttt{Bool}.\\
&[l_1, l_2]\ \texttt{LT}_{S_{\texttt{Nat}}}\ l_1\ l_2 \hookrightarrow \texttt{and}\ (\texttt{LTE}_{S_{\texttt{Nat}}}\ l_1\ l_2)\ (\texttt{not}\ (\texttt{Eq}_{\texttt{Nat}}\ l_1\ l_2)).
\end{aligned}$$

This ordering will be used to help us establish a total order on sublevels, which will be used in our representation of sublevel sets described below.

Sublevels

We represent sublevels with the static constant symbols `V` and `C`:

$$\begin{aligned}&\Omega \ \texttt{:}\ \texttt{Type}.\\ &\mathrm{V} \ \texttt{:}\ \texttt{Nat} \to \texttt{Nat} \to S_{\texttt{Nat}} \to \Omega.\\ &\mathrm{C} \ \texttt{:}\ \texttt{Nat} \to S_{\texttt{Nat}} \to \Omega.\end{aligned}$$

As the `maxS` operator takes a sublevel set as its argument, we need to define sets of sublevels, in a similar fashion to what we did with sets of `Nat`s:

$$\begin{aligned}&S_\Omega \ \texttt{:}\ \texttt{Type}.\\ &\texttt{nil}_\Omega \ \texttt{:}\ S_\Omega.\\ &\texttt{cons}_\Omega \ \texttt{:}\ \Omega \to S_\Omega \to S_\Omega.\end{aligned}$$

For convenience, we also define a constructor for normal form level terms defined using singleton sublevel sets:

$$\begin{aligned}&\texttt{sgl} \ \texttt{:}\ \Omega \to L.\\ &[s]\ \texttt{sgl}\ s \ \hookrightarrow\ \texttt{maxS}\ (\texttt{cons}_\Omega\ s\ \texttt{nil}_\Omega).\end{aligned}$$

We similarly define an $\texttt{add}_\Omega$ symbol that adds elements to sublevel lists, preventing duplication and maintaining a sorted order. For this, we define a lexicographic comparator $\texttt{cmp}_\Omega$ on sublevels, which uses the previously defined $\texttt{LTE}_{S_{\texttt{Nat}}}$ symbol for comparing the $S_{\texttt{Nat}}$ arguments to the sublevel constructors:

$$\begin{aligned}&\texttt{def}\ \texttt{cmp}_\Omega \ \texttt{:}\ \Omega \to \Omega \to \mathrm{CMP}.\\ &\ldots \quad \text{(rewrite rules for } \texttt{cmp}_\Omega\text{)}\\ &\texttt{def}\ \texttt{add}_\Omega \ \texttt{:}\ S_\Omega \to \Omega \to S_\Omega.\\ &\ldots \quad \text{(rewrite rules for } \texttt{add}_\Omega\text{)}\end{aligned}$$

However, w.r.t. the lists representing sublevel sets, ensuring that they are always sorted via some canonical order is not the only invariant that we would like to maintain. We also need to ensure that they respect the pairwise incomparability predicate we enforce on our normal form grammar. So, we start by defining a subsumption relation $\leq_\Omega$ between sublevels

directly based on the previously derived subsumption conditions:

$$
\begin{array}{l}
\texttt{def}\ \leq_{\texttt{Nat}} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Bool}. \\
[n, m]\ \leq_{\texttt{Nat}}\ n\ m\ \hookrightarrow\ \texttt{cases}_{\texttt{Bool}}\ (\texttt{cmp}_{\texttt{Nat}}\ n\ m)\ \texttt{true}\ \texttt{true}\ \texttt{false}.
\end{array}
$$

$$
\begin{array}{l}
\texttt{def}\ =_{\texttt{Nat}} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Bool}. \\
[n, m]\ =_{\texttt{Nat}}\ n\ m\ \hookrightarrow\ \texttt{cases}_{\texttt{Bool}}\ (\texttt{cmp}_{\texttt{Nat}}\ n\ m)\ \texttt{false}\ \texttt{true}\ \texttt{false}.
\end{array}
$$

$$
\begin{array}{l}
\texttt{def}\ \leq_{\Omega} : \Omega \to \Omega \to \texttt{Bool}. \\
[]\ \leq_{\Omega}\ (\mathrm{V}\ \_\ \_\ \_)\ (\mathrm{C}\ \_\ \_)\ \hookrightarrow \\
\quad \texttt{false}. \\
[l_1, l_2, n, m]\ \leq_{\Omega}\ (\mathrm{C}\ n\ l_1)\ (\mathrm{C}\ m\ l_2)\ \hookrightarrow \\
\quad \texttt{and}\ (\subseteq_{\texttt{Nat}}\ l_2\ l_1)\ (\leq_{\texttt{Nat}}\ n\ m). \\
[l_1, l_2, n, m]\ \leq_{\Omega}\ (\mathrm{C}\ n\ l_1)\ (\mathrm{V}\ m\ \_\ l_2)\ \hookrightarrow \\
\quad \texttt{and}\ (\subseteq_{\texttt{Nat}}\ l_2\ l_1)\ (\leq_{\texttt{Nat}}\ n\ (\texttt{Nat.succ}\ m)). \\
[l_1, l_2, n, m, x, y]\ \leq_{\Omega}\ (\mathrm{V}\ n\ x\ l_1)\ (\mathrm{V}\ m\ y\ l_2)\ \hookrightarrow \\
\quad \texttt{and}\ (\subseteq_{\texttt{Nat}}\ l_2\ l_1)\ (\texttt{and}\ (=_{\texttt{Nat}}\ x\ y)\ (\leq_{\texttt{Nat}}\ n\ (\texttt{Nat.succ}\ m))).
\end{array}
$$

Then, we define a new set insertion function `maxadd` that maintains the pairwise incomparability predicate by not inserting any sublevels that are subsumed by some other sublevel in the set, and replacing any sublevels that are subsumed by the newly inserted sublevel:

$$
\begin{array}{l}
\texttt{def}\ \texttt{ite}_{S_\Omega}\ : \texttt{Bool} \to S_\Omega \to S_\Omega \to S_\Omega. \\
[t]\ \texttt{ite}_{S_\Omega}\ \texttt{true}\ t\ \_\ \hookrightarrow\ t. \\
[f]\ \texttt{ite}_{S_\Omega}\ \texttt{false}\ \_\ f\ \hookrightarrow\ f.
\end{array}
$$

$$
\begin{array}{l}
\texttt{def}\ \texttt{subs\_add}_{\Omega}\ :\ S_\Omega \to \Omega \to S_\Omega. \\
[\omega]\ \texttt{subs\_add}_{\Omega}\ \texttt{nil}_{\Omega}\ \omega\ \hookrightarrow\ \texttt{cons}_{\Omega}\ \omega\ \texttt{nil}_{\Omega}. \\
[\omega, \omega', l]\ \texttt{subs\_add}_{\Omega}\ (\texttt{cons}_{\Omega}\ \omega'\ l)\ \omega\ \hookrightarrow \\
\quad \texttt{ite}_{S_\Omega}\ (\leq_{\Omega}\ \omega\ \omega') \\
\qquad (\texttt{cons}_{\Omega}\ \omega'\ l) \\
\qquad (\texttt{ite}_{S_\Omega}\ (\leq_{\Omega}\ \omega'\ \omega) \\
\qquad\quad (\texttt{subs\_add}_{\Omega}\ l\ \omega) \\
\qquad\quad (\texttt{add}_{\Omega}\ (\texttt{subs\_add}_{\Omega}\ l\ \omega)\ \omega')).
\end{array}
$$

Now, if we are careful to only ever use this function when adding a sublevel to a set, we are guaranteed to only ever construct sublevels sets that satisfy the pairwise incomparability predicate.

3.3.2 Deriving a Normalizing Rewrite System

To finally construct our normal form terms, we define a `maxS` function taking a sublevel set as an argument and producing a normal-form level term:

$$\texttt{maxS} \ : \ S_\Omega \to L.$$

We now need to define the rewrite rules for our top-level universe level term constructors that enable us to arrive at an equivalent term in our normal form grammar. To do this, we might consider attempting to directly convert into rewrite rules each of the equivalences that we derived in Section 3.2 as we progressively modified our level grammar. However, these equivalences were really only useful in helping us arrive at our normal form grammar, and proving the sufficiency property; they are not well-suited to be converted directly into rewrite rules for computing normal form terms.

Specifically, they would not observe the confluence property. For instance, if we were to take this approach, we would define the following rewrite rule, corresponding to the equality we derived to pull `max` out of `s` applications:

$$[a, b]\ \texttt{s}\ (\texttt{max}\ a\ b) \ \hookrightarrow\ \texttt{max}\ (\texttt{s}\ a)\ (\texttt{s}\ b). \quad (r_1)$$

At the same time, however, we would also need to define a rewrite rule on `max` that yields a normal form term assuming that both arguments have been already normalized. We can define this by "eliminating" on the second argument, defining the $\texttt{nil}_\Omega$ case as follows:

$$[l_1]\ \texttt{max}\ (\texttt{maxS}\ l_1)\ (\texttt{maxS}\ \texttt{nil}_\Omega) \ \hookrightarrow\ \texttt{maxS}\ l_1. \quad (r_2)$$

This gives rise to the following critical pair:

$$\begin{aligned}
&\texttt{s}\ (\texttt{max}\ (\texttt{maxS}\ l_1)\ (\texttt{maxS}\ \texttt{nil}_\Omega)) \ \overset{r_1}{\hookrightarrow}\ \texttt{max}\ (\texttt{s}\ (\texttt{maxS}\ l_1))\ (\texttt{s}\ (\texttt{maxS}\ \texttt{nil}_\Omega)).\\
&\texttt{s}\ (\texttt{max}\ (\texttt{maxS}\ l_1)\ (\texttt{maxS}\ \texttt{nil}_\Omega)) \ \overset{r_2}{\hookrightarrow}\ \texttt{s}\ (\texttt{maxS}\ l_1).
\end{aligned}$$

These terms cannot be joined without some rewrite rule matching on `s` (`maxS` ...).

In order for our rules to be confluent, they must match on terms of type L on the basis of the normal form representation: namely, they must always expect level terms to be applications of `maxS` (in the style of the rule defined on `max` above, but defined on *all* top-level constructors). Therefore, we have to derive a *new* set of equivalences to base our rewrite rules on. These equivalences must account for the interactions of the top-level L constructors `imax`, `max`, and `s` with normal-form subterms headed by `maxS`. To this end, we define the following composite $L + L_N$ grammar (with the expected interpretation):

$$\begin{aligned}
&L^* = \ell \text{ where}\\
&S_p ::= \{u_1, u_2, \dots\}\\
&\omega_c ::= \mathrm{C}\ (k+1)\ S_p\\
&\omega_v ::= \mathrm{V}\ k\ u\ (S_p \cup \{u\})\\
&\omega ::= \omega_c \mid \omega_v\\
&S_\omega ::= \{\omega_1, \omega_2, \dots\} \quad \forall \omega, \omega' \in S_\omega,\ \omega \neq \omega' \implies \omega \not\leq \omega'\\
&\ell_n ::= \texttt{maxS}\ S_\omega\\
&\ell_c ::= \texttt{imax}\ \ell_n\ \ell'_n \mid \texttt{max}\ \ell_n\ \ell'_n \mid \texttt{s}\ \ell_n \mid \texttt{z} \mid \texttt{v}\ u\\
&\ell ::= \ell_n \mid \ell_c
\end{aligned}$$

Note that `imax`, `max`, and `s` with `maxS` take terms from the category ℓ_n – corresponding to terms of the normal form grammar L_N – rather than ℓ. This is in line with the kind of equalities we would like to derive to help us design our rewrite rules. When matching, these rules must assume that that any level arguments are already in normal form, as otherwise they could result in non-confluence (as demonstrated above).

So, our goal is to derive equalities that effectively eliminate the category l_I from the grammar L^* above. These equalities will hopefully all correspond to easily implementable rewrite rules, allowing us to define a rewrite system that transforms terms from the grammar L directly to the grammar L_N.

To ease the presentation of the derived equivalences on this grammar, we introduce the symbol `maxL`, taking a list of level terms from the grammar L^* as an argument, with the following interpretation:

$$\begin{aligned}\texttt{eval}_\sigma(\texttt{maxL}\ []) &:= 0\\ \texttt{eval}_\sigma(\texttt{maxL}\ (\ell :: l)) &:= \max(\texttt{eval}_\sigma(\ell), \texttt{eval}_\sigma(\texttt{maxL}\ l))\end{aligned}$$

which gives us the following equivalence:

Lemma 3.3.1. For all $\ell_1, \ldots, \ell_n$, we have:

$$\texttt{maxL}\ [\ell_1, \ldots, \ell_n] \approx \texttt{max}\ \ell_1\ (\texttt{max}\ \ell_2\ (\ldots(\texttt{max}\ \ell_{n-1}\ \ell_n)))$$

Eliminating `imax`

Let's start by eliminating `imax` from our grammar. We start by considering the most general case, when both arguments to `imax` are `maxS` applications to sets of arbitrary size. We can derive the following equivalence:

Lemma. For all $\omega_1, \ldots, \omega_n$ and ℓ,

$$\texttt{imax}\ (\texttt{maxS}\ \{\omega_1, \ldots, \omega_n\})\ \ell \approx \texttt{maxL}\ [\texttt{imax}\ (\texttt{maxS}\ \{\omega_1\})\ \ell, \ldots, \texttt{imax}\ (\texttt{maxS}\ \{\omega_n\})\ \ell]$$

It also turns out that we can make a similar simplification to the second argument:

Lemma. For all $\omega_1, \ldots, \omega_n$ and ℓ,

$$\texttt{imax}\ \ell\ (\texttt{maxS}\ \{\omega_1, \ldots, \omega_n\}) \approx \texttt{maxL}\ [\texttt{imax}\ \ell\ (\texttt{maxS}\ \{\omega_1\}), \ldots, \texttt{imax}\ \ell\ (\texttt{maxS}\ \{\omega_n\})]$$

Taken together, these equalities mean that we can now assume that both arguments to `imax` are `maxS` applications to a singleton sublevel set. They are implemented by the following rewrite rules:

$$\begin{aligned}&\texttt{def imax\_aux}_\beta\ :\ \Omega \to \Omega \to L.\\ &\texttt{def imax\_aux}_\alpha\ :\ \Omega \to L \to L.\\ &[\omega]\ \texttt{imax\_aux}_\alpha\ \omega\ (\texttt{maxS nil}_\Omega) \hookrightarrow \texttt{maxS nil}_\Omega.\\ &[\omega, \omega', l]\ \texttt{imax\_aux}_\alpha\ \omega\ (\texttt{maxS}\ (\texttt{cons}_\Omega\ \omega'\ l)) \hookrightarrow\\ &\quad \texttt{max}\ (\texttt{imax\_aux}_\beta\ \omega\ \omega')\ (\texttt{imax\_aux}_\alpha\ \omega\ l).\\ \\ &[\ell]\ \texttt{imax}\ (\texttt{maxS nil}_\Omega)\ \ell \hookrightarrow \ell.\\ &[\omega, l, \ell]\ \texttt{imax}\ (\texttt{maxS}\ (\texttt{cons}_\Omega\ \omega\ l))\ \ell \hookrightarrow\\ &\quad \texttt{max}\ (\texttt{imax\_aux}_\alpha\ \omega\ \ell)\ (\texttt{imax}\ (\texttt{maxS}\ l)\ \ell).\end{aligned}$$

Note that rather than using a symbol corresponding to `maxL` directly, we make use of Theorem 3.3.1 instead, rewriting to a sequence of nested applications of `lvl.max` (because in the event that `q` is not `sublvl.nil`, the rule will apply again to the second argument to `lvl.max` on the the RHS). We have to define two intermediate symbols, `imax_auxL` and `imax_aux`, in order to ensure termination. This is necessary because, for instance, the following rewrite rule results in non-termination:

$$[\omega, l, \ell]\ \texttt{imax}\ (\texttt{maxS}\ (\texttt{cons}_\Omega\ \omega\ l))\ \ell \hookrightarrow \\ \texttt{max}\ (\texttt{imax}\ (\texttt{maxS}\ (\texttt{cons}_\Omega\ \omega\ \texttt{nil}_\Omega))\ \ell)\ (\texttt{imax}\ (\texttt{maxS}\ l)\ \ell).$$

While this rule correctly implements the equality derived above, it would be able to infinitely apply to itself, as the LHS matches the first argument to `lvl.max` on the RHS. To finally eliminate `imax` from our grammar, we are now left to consider how to deal with the case of `imax` being applied to two `maxS` applications to singleton sublevel sets – that is, terms of the form:

$$\texttt{imax}\ (\texttt{maxS}\ \{\omega_1\})\ (\texttt{maxS}\ \{\omega_2\})$$

In our encoding, we represent terms of this form with the $\texttt{imax\_aux}_\beta$ operator, which we declared above as the following symbol:

$$\texttt{def}\ \texttt{imax\_aux}_\beta\ :\ \Omega \to \Omega \to L.$$

Our goal is to derive rewrite rules for `imax_aux` that allow for the computation of a normal form. We consider in turn each possible pair of constructions on the sublevels ℓ_1 and ℓ_2, deriving equalities that eliminate the use of the `imax` symbol.

Lemma. For all $k, k' \in \mathbb{N}$, $u, v \in \mathcal{U}$ and $S, S' \subset \mathcal{U}$,

$$\texttt{imax}\ (\texttt{maxS}\ \{\text{V}\ k\ u\ S\})\ (\texttt{maxS}\ \{\text{V}\ k'\ v\ S'\}) \approx \texttt{max}\ (\texttt{maxS}\ \{\text{V}\ k\ u\ (S \cup S')\})\ (\texttt{maxS}\ \{\text{V}\ k'\ v\ S'\})$$

Lemma. For all $k, k' \in \mathbb{N}$, $u \in \mathcal{U}$ and $S, S' \subset \mathcal{U}$,

$$\texttt{imax}\ (\texttt{maxS}\ \{\text{V}\ k\ u\ S\})\ (\texttt{maxS}\ \{\text{C}\ k'\ S'\}) \approx \texttt{max}\ (\texttt{maxS}\ \{\text{V}\ k\ u\ (S \cup S')\})\ (\texttt{maxS}\ \{\text{C}\ k'\ S'\})$$

Lemma. For all $k, k' \in \mathbb{N}$ and $S, S' \subset \mathcal{U}$,

$$\texttt{imax}\ (\texttt{maxS}\ \{\text{C}\ k\ S\})\ (\texttt{maxS}\ \{\text{C}\ k'\ S'\}) \approx \texttt{max}\ (\texttt{maxS}\ \{\text{C}\ k\ (S \cup S')\})\ (\texttt{maxS}\ \{\text{C}\ k'\ S'\})$$

Lemma. For all $k, k' \in \mathbb{N}$, $u \in \mathcal{U}$, and $S, S' \subset \mathcal{U}$,

$$\texttt{imax}\ (\texttt{maxS}\ \{\text{C}\ k\ S\})\ (\texttt{maxS}\ \{\text{V}\ k'\ u\ S'\}) \approx \texttt{max}\ (\texttt{maxS}\ \{\text{C}\ k\ (S \cup S')\})\ (\texttt{maxS}\ \{\text{V}\ k'\ u\ S'\})$$

These lemmas correspond to the following set of rewrite rules:

$$[l_1, l_2, n, m, u, v]\ \texttt{imax\_aux}_\beta\ (\text{V}\ l_1\ u\ n)\ (\text{V}\ l_2\ v\ m) \hookrightarrow \\ \texttt{max}\ (\texttt{sgl}\ (\text{V}\ (\cup_{\texttt{Nat}}\ l_1\ l_2)\ u\ n))\ (\texttt{sgl}\ (\text{V}\ l_2\ v\ m)).$$
$$[l_1, l_2, n, m, u]\ \texttt{imax\_aux}_\beta\ (\text{V}\ l_1\ u\ n)\ (\text{C}\ l_2\ m) \hookrightarrow \\ \texttt{max}\ (\texttt{sgl}\ (\text{V}\ (\cup_{\texttt{Nat}}\ l_1\ l_2)\ u\ n))\ (\texttt{sgl}\ (\text{C}\ l_2\ m)).$$
$$[l_1, l_2, n, m]\ \texttt{imax\_aux}_\beta\ (\text{C}\ l_1\ n))\ (\text{C}\ l_2\ m) \hookrightarrow \\ \texttt{max}\ (\texttt{sgl}\ (\text{C}\ (\cup_{\texttt{Nat}}\ l_1\ l_2)\ n))\ (\texttt{sgl}\ (\text{C}\ l_2\ m)).$$
$$[l_1, l_2, n, m, x]\ \texttt{imax\_aux}_\beta\ (\text{C}\ l_1\ n))\ (\text{V}\ l_2\ x\ m) \hookrightarrow \\ \texttt{max}\ (\texttt{sgl}\ (\text{C}\ (\cup_{\texttt{Nat}}\ l_1\ l_2)\ n))\ (\texttt{sgl}\ (\text{V}\ l_2\ x\ m)).$$

Eliminating `s`, `z`, `v`, and `max`

Let's now look at how to eliminate the `s` symbol from our grammar. We can observe the following equivalence:

Lemma. For all $\omega_1, \ldots, \omega_n$,

$$\texttt{s}\ (\texttt{maxS}\ \{\omega_1, \ldots, \omega_n\}) \approx \texttt{maxL}\ [\texttt{s}\ (\texttt{maxS}\ \{\omega_1\}), \ldots, \texttt{s}\ (\texttt{maxS}\ \{\omega_n\}), \texttt{s}\ \texttt{z}]$$

So, we can restrict `s` to apply to a `maxS` with a singleton sublevel set – that is, terms of the following form:

$$\texttt{s}\ (\texttt{maxS}\ \{\omega\})$$

We can derive the following equalities for both possible constructions of sublevel ℓ:

Lemma. For all $k \in \mathbb{N}$, $u \in \mathcal{U}$ and $S \subset \mathcal{U}$,

$$\texttt{s}\ (\texttt{maxS}\ \{\text{V}\ k\ u\ S\}) \approx \texttt{max}\ (\texttt{maxS}\ \{\text{V}\ (k+1)\ u\ S\})\ (\texttt{maxS}\ \{\text{C}\ 1\ \emptyset\})$$

Lemma. For all $k \in \mathbb{N}$ and $S \subset \mathcal{U}$,

$$\texttt{s}\ (\texttt{maxS}\ \{\text{C}\ k\ S\}) \approx \texttt{max}\ (\texttt{maxS}\ \{\text{C}\ (k+1)\ S\})\ (\texttt{maxS}\ \{\text{C}\ 1\ \emptyset\})$$

In both cases, the additional term $\texttt{maxS}\ \{\text{C}\ 1\ \emptyset\}$ is needed to preserve the interpretation when some element of S is set to zero, in which case the value of the term is exactly one. Implementing this as a set of rewrite rules, we obtain:

$$\begin{aligned}
&\texttt{def s_aux} : S_\Omega \to L.\\
&[]\ \texttt{s_aux}\ \texttt{nil}_\Omega \hookrightarrow \texttt{sgl}\ (\text{C}\ \texttt{nil}_{\texttt{Nat}}\ (\texttt{Nat.succ}\ \texttt{Nat.zero})).\\
&[l, k, l']\ \texttt{s_aux}\ (\texttt{cons}_\Omega\ (\text{C}\ k\ l)\ l') \hookrightarrow\\
&\quad \texttt{max}\ (\texttt{sgl}\ (\text{C}\ (\texttt{Nat.succ}\ k)\ l))\ (\texttt{s_aux}\ l').\\
&[l, x, k, l']\ \texttt{s_aux}\ (\texttt{cons}_\Omega\ (\text{V}\ k\ x\ l)\ l') \hookrightarrow\\
&\quad \texttt{max}\ (\texttt{sgl}\ (\text{V}\ (\texttt{Nat.succ}\ k)\ x\ l))\ (\texttt{s_aux}\ l').
\end{aligned}$$

Again, we have applied Theorem 3.3.1 in deriving our rewrite rules from the above equations.

Eliminating `z` is fairly straightforward, as our interpretation $\texttt{eval}_\sigma(\texttt{maxS}\ \emptyset) \coloneqq 0$ gives us the equivalence $\texttt{z} \approx \texttt{maxS}\ \emptyset$, which becomes the following rewrite rule:

$$[]\ \texttt{z} \hookrightarrow \texttt{maxS}\ \texttt{nil}_\Omega.$$

We can eliminate level variables by observing the equivalence $u \approx \texttt{maxS}\ \{\text{V}\ 0\ u\ \{u\}\}$ for all $u \in \mathcal{U}$, which is implemented as the rewrite rule:

$$[u]\ \texttt{v}\ u \hookrightarrow \texttt{sgl}\ (\text{V}\ \texttt{Nat.zero}\ u\ (\texttt{cons}_{\texttt{Nat}}\ u\ \texttt{nil}_{\texttt{Nat}})).$$

Finally, let's eliminate the `max` symbol from our grammar. Let's suppose the existence of some special union operator $\cup^*$ that eliminates subsumed terms from the union of two sublevel sets. Precisely, for all sublevel sets S, S', the following conditions apply:

$$\begin{aligned}
\forall \omega, \omega' \in S \cup^* S', \omega \neq \omega' &\implies \omega \not\leq \omega'\\
\texttt{max}\ (\texttt{maxS}\ S)\ (\texttt{maxS}\ S') &\approx \texttt{maxS}\ (S \cup^* S')
\end{aligned}$$

By Theorem 3.0.2, $S \cup^* S'$ must be the unique sublevel set equivalent to $\texttt{max}\ (\texttt{maxS}\ S)\ (\texttt{maxS}\ S')$ satisfying the incomparability predicate, so it remains to implement a set of rewrite rules computing $S \cup^* S'$. We can do this by adding the elements of S to those of S' one by one, skipping any elements from S that are subsumed by some element of S'. We can do this with the help of the $\texttt{subs\_add}_\Omega$ function described earlier:

$$[l]\ \texttt{max}\ (\texttt{maxS}\ l)\ (\texttt{maxS}\ \texttt{nil}_\Omega) \hookrightarrow \texttt{maxS}\ l.$$
$$[\omega, l_1, l_2]\ \texttt{max}\ (\texttt{maxS}\ l_1)\ (\texttt{maxS}\ (\texttt{cons}_\Omega\ \omega\ l_2)) \hookrightarrow$$
$$\texttt{max}\ (\texttt{maxS}\ (\texttt{subs\_add}_\Omega\ l_1\ \omega))\ (\texttt{maxS}\ l_2).$$

Having eliminated `imax`, `s`, `z`, `v`, and `max` from the grammar L^*, we no longer need the ℓ_c syntactic category, and the grammar reduces to an equivalent of L_N. Each of the rewrite rules we have designed preserve the semantic interpretation of universe level terms, while allowing us to arrive at terms of a particular normal form grammar where syntactic equality exactly corresponds to semantic equivalence. The comparison of normal forms computed by our encoding's rewrite rules effectively decides equality between universe level expressions directly translated from Lean via top-level constructors corresponding to Lean's native pseudo-constructors for universe level terms, thus completing our deep encoding of universe level equality in Dedukti.

Instantiation?

While the deep encoding described above is quite convenient for a representation of sets of universe level parameters, it doesn't really allow for any form of instantiation. For instance, consider the following universe-polymorphic definition:

```
def poly.{u} : Sort (u + 1) := Sort u
```

This would translate as:

$$\texttt{def poly} : (u : L) \to U\ (\texttt{s}\ (\texttt{v Nat.zero})).$$
$$[]\ \texttt{poly} \hookrightarrow (u : L) \Rightarrow (\dot{s}\ (\texttt{v Nat.zero})).$$

By itself, this translation is well-typed in Dedukti. However, notice that we completely ignore the universe level parameter argument that was included by our translation of constant declarations (as described in Section 2.3) to allow for the possibility for instantiating universe level parameters via β-reduction. This is problematic, because if we try to use this constant elsewhere, say, in the context of a constant that does not use level parameters, it becomes ill-typed.

For instance, consider the Lean definition below:

```
def polyInst : Sort 1 := poly.{0}
```

This would have the following Dedukti translation:

$$\texttt{def polyInst} : U\ (\texttt{s z}).$$
$$[]\ \texttt{polyInst} \hookrightarrow \texttt{poly z}.$$

where the level-instantiated constant reference `poly.{0}` becomes the explicit application `poly z`. However, this term is ill-typed, on account of the dependency of the type of `poly`

on its universe level parameter `u` being lost in the Dedukti translation: the inferred type of `poly z` is U (`s` (`v Nat.zero`)) while the expected type is U (`s` (`z`)), and the two do not normalize to the same form. So, how can we modify our universe level encoding to enable the kind of β-reduction-based universe level parameter instantiation that we were originally planning for when we initially defined our translation of constant declarations?

A Shallow Encoding?

Something we could attempt here is to rely on Dedukti's own bound variables to represent universe level parameters, rather than using a natural number encoding. This approach can be thought of as a "shallow encoding", characterized by the universe level parameter constructor, which now takes a term of type L as its argument:

$$\texttt{def v} \ \texttt{:} \ L \to L.$$

with our translation of universe level parameter instances applying this symbol to the bound variable representing the parameter, rather than a natural number encoding of the parameter's canonical index.

Correspondingly, we would then change the types of V and C to take level sets, rather than natural number sets, for the guard parameter set arguments:

$$\begin{aligned} \Omega \ &\texttt{:} \ \texttt{Type}. \\ \mathrm{V} \ &\texttt{:} \ \texttt{Nat} \to \texttt{Nat} \to S_L \to \Omega. \\ \mathrm{C} \ &\texttt{:} \ \texttt{Nat} \to S_L \to \Omega. \end{aligned}$$

The rewrite rule for `v`, for instance, would then become:

$$[\ell] \ \texttt{v} \ \ell \ \hookrightarrow \ \texttt{sgl} \ (\mathrm{V} \ \texttt{Nat.zero} \ \ell \ (\texttt{cons}_L \ \ell \ \texttt{nil}_L)).$$

Under such an encoding, our translation of `poly` becomes:

$$\begin{aligned} &\texttt{def poly} \ \texttt{:} \ (u : L) \to U \ (\texttt{s} \ (\texttt{v} \ u)). \\ &\quad [] \ \texttt{poly} \ \hookrightarrow \ (u : L) \texttt{=>} \ (\dot{s} \ (\texttt{v} \ u)). \end{aligned}$$

from which it seems at least feasible to possibly implement some form of instantiation, as the universe level parameter variable `u` *is* now used in the type and body of the translated constant declaration.

However, when it comes to actually expressing these universe level parameter sets, all of the machinery we have previously developed for maintaining ordered lists of natural numbers to represent sets are not adaptable to creating ordered lists bound variables of type L representing universe level parameters, specifically because there is no way to define a total ordering on bound variables in Dedukti. What's more, there is nothing in the type of the argument to `v` above to suggest that its argument is necessarily a bound variable to begin with, as opposed to an arbitrary term of type L.

So, our best bet here is to rely on a certain special feature of Dedukti: its ability to define special binary symbols that act modulo associativity and commutativity (A/C) – that is, the order or their arguments is irrelevant when the kernel decides whether two terms

constructed using these symbols are equal, or when performing pattern matching. We declare A/C symbols using the following syntax:

$$\texttt{defac}\ f\ [T].$$

which declares that f is an A/C operator with type $T \to T \to T$. We could use this to define a list-like type representing sets of terms of type L as follows:

$$\begin{aligned} &S_L\ \texttt{:}\ \texttt{Type}.\\ &\texttt{nil}_L\ \texttt{:}\ S_L.\\ &\texttt{singleton}_L\ \texttt{:}\ L \to S_L.\\ &\texttt{defac union}_L\ [S_L]. \end{aligned}$$

Because $\texttt{union}_L$ is marked as A/C, Dedukti will effectively disregard the order of the elements appearing in a construction of S_L of size greater than one. For convenience, we can also define a symbol that mimics a normal list construct operator, but which is actually defined using $\texttt{union}_L$:

$$\begin{aligned} &\texttt{def cons}_L\ \texttt{:}\ L \to S_L \to S_L.\\ &[\ell, S]\ \texttt{cons}_L\ \ell\ S \hookrightarrow \texttt{union}_L\ (\texttt{singleton}_L\ \ell)\ S. \end{aligned}$$

Now, via the A/C $\texttt{union}_L$ symbol, Dedukti is able to use A/C equivalence to identify the symbols t_1 and t_2 below:

$$\begin{aligned} &x\ \texttt{:}\ L.\\ &y\ \texttt{:}\ L.\\ &\texttt{def}\ t_1\ \texttt{:}\ L.\\ &[]\ t_1 \hookrightarrow \texttt{cons}_L\ x\ (\texttt{cons}_L\ y\ \texttt{nil}_L).\\ &\texttt{def}\ t_2\ \texttt{:}\ L.\\ &[]\ t_2 \hookrightarrow \texttt{cons}_L\ y\ (\texttt{cons}_L\ x\ \texttt{nil}_L). \end{aligned}$$

Additionally, Dedukti allows us to perform *matching* modulo A/C, enabling, for instance, the definition of a duplicate-removal rule that allows the symbols t_3 and t_4 below to be identified:

$$\begin{aligned} &[S, S']\ \texttt{union}_L\ S\ (\texttt{union}_L\ S\ S') \hookrightarrow \texttt{union}_L\ S\ S'.\\ &\texttt{def}\ t_3\ \texttt{:}\ L.\\ &[]\ t_3 \hookrightarrow \texttt{cons}_L\ x\ (\texttt{cons}_L\ y\ (\texttt{cons}_L\ x\ \texttt{nil}_L)).\\ &\texttt{def}\ t_4\ \texttt{:}\ L.\\ &[]\ t_4 \hookrightarrow \texttt{cons}_L\ x\ (\texttt{cons}_L\ y\ \texttt{nil}_L). \end{aligned}$$

Here, the fact that the second occurrence of x is not directly adjacent to the first occurrence does not prevent the rule defined on $\texttt{union}_L$ from applying, thanks to matching modulo A/C.

A/C equality and matching are in fact sufficient for our purposes here, allowing us to recover a set-like representation for both level and sublevel sets, defining $\texttt{union}_\Omega$ and $\texttt{singleton}_\Omega$ symbols for constructing sublevel sets analogous to $\texttt{union}_L$ and $\texttt{singleton}_L$

defined above. But, how exactly do we go about instantiation now, which was our original motivation for doing a shallow encoding? Returning to our example from before, the translation of the `polyInst` declaration remains the same as shown above. When typechecking its δ-reduction rewrite rule, however, things get ugly pretty quickly: using the typing rule [ALL], we instantiate `u` in the type declaration of `poly` with `z`, leaving us with a term of type U (`s` (`v z`)), with the argument normalizing[7] to a universe level term of the form:

$$\begin{aligned}&\texttt{maxS}(\texttt{cons}_\Omega\\&\quad \text{V } \texttt{Nat.zero } (\texttt{maxS nil}_\Omega)\ (\texttt{cons}_L\ (\texttt{maxS nil}_\Omega)\ \texttt{nil}_L)\\&\quad (\texttt{cons}_\Omega\ (\text{C } (\texttt{Nat.succ Nat.zero})\ \texttt{nil}_L)\ \texttt{nil}_\Omega))\end{aligned}$$

where the normal form of `z` has been instantiated within the universe level parameters that would have appeared in the arguments to V prior to instantiation. This is distinct from the normal form of `s z`:

$$\begin{aligned}&\texttt{maxS}(\texttt{cons}_\Omega\\&\quad (\text{C } (\texttt{Nat.succ Nat.zero})\ \texttt{nil}_L)\ \texttt{nil}_\Omega)\end{aligned}$$

In order to be able to unify these two terms, we would have to define some complex rewrite rules to pull the instantiated variable out of the normal form, essentially undoing the normalization in the process, obtaining some term isomorphic to `s z`, and performing normalization again on this term. This is probably much more trouble than it is worth. On top of this, modulo A/C equality/matching as implemented in Dedukti is not well-optimized and quite expensive for typechecking, and we have observed it to scale very poorly in practice.

So, to conclude, while a shallow encoding may initially seem to be quite a natural approach to enabling universe level parameter instantiation, it is not a very practical option, and implementing a confluent rewrite system for it is in fact quite a bit more complex than it might seem at first glance. It would be preferable to find another way to address the instantiation issue.

3.3.3 A Hybrid Encoding

Our deep and shallow encoding approaches have mutual benefits and drawbacks that arise fundamentally from our representation of universe level parameters in the encoding. This may make us wonder: could it be possible to combine these two approaches into one, affording convenient instantiation while also avoiding the drawbacks associated with the use A/C symbols? One obvious thing to try is to combine the use of canonical indices and bound variable references in our universe level parameter representation:

$$\texttt{def v : Nat} \to L \to L.$$

We can keep our sublevels as they were in the deep encoding (using ordered lists of natural numbers to represent sets), along with a similar normalizing rewrite rule for var:

$$[u]\ \texttt{v}\ u\ \texttt{\_} \hookrightarrow \texttt{sgl}\ (\text{V } \texttt{Nat.zero}\ u\ (\texttt{cons}_{\texttt{Nat}}\ u\ \texttt{nil}_{\texttt{Nat}})).$$

The constructor `v` now carries two arguments that are used for two separate purposes: a canonical index argument of type `Nat` that is used to establish an order on universe level

[7] For simplicity, we do not normalize $\texttt{cons}_\Omega$ applications to applications of the A/C $\texttt{union}_\Omega$ symbol.

parameters (that is only useful within the context of typechecking the constant that a level term *originally* appears in), along with an argument of type L to be instantiated with a bound variable representing a universe level parameter that can used for instantiation purposes.

Now, `poly` translates to:

$$\begin{aligned}&\texttt{def poly} : (u : L) \to U\ (\texttt{s}\ (\texttt{v Nat.zero}\ u)).\\ &\quad [] \texttt{ poly} \hookrightarrow (u : L) \Rightarrow (\dot{s}\ (\texttt{v Nat.zero}\ u)).\end{aligned}$$

which is well-typed in Dedukti. However, the translation of `polyInst` is still ill-typed. The issue is that applying the rewrite rule on `v` effectively ignores the second bound variable argument, which is instantiated with a level expression that we need to replace the `v` application with in order to normalize to a form that is identical to what we saw with our deep encoding. What we need is a way to preemptively replace an application of `v` with its second argument whenever this argument represents an explicitly instantiated universe level parameter (as opposed to a simple bound variable, as it would appear in a translated constant symbol's type declaration and definition). We can achieve this with a new symbol `inst` to mark universe level parameter instantiations, along with a new rewrite rule on `v`:

$$\begin{aligned}&\texttt{inst} : L \to L.\\ &[\ell]\ \texttt{v \_}\ (\texttt{inst}\ \ell) \hookrightarrow \ell.\end{aligned}$$

Now, we must be careful in our translation to ensure the invariant that any universe level parameter-instantiating levels are wrapped with `inst`. This is achieved by our translation of constants references, as we described it in Section 2.3:

$$|C.\{\ell_1, \ldots, \ell_n\}|^{\hookrightarrow}_{-} := C\ (\texttt{inst}\ |\ell_1|_L)\ \ldots\ (\texttt{inst}\ |\ell_n|_L)$$

With this approach, the translation of `polyInst` now becomes:

$$\begin{aligned}&\texttt{def polyInst} : U\ (\texttt{s z}).\\ &[] \texttt{ polyInst} \hookrightarrow \texttt{poly}\ (\texttt{inst z}).\end{aligned}$$

However, there's still one more issue here that causes this translation to be ill-typed in Dedukti. The new rewrite rule on `v` is non-confluent with the previous one, arising from the following critical pair:

$$\begin{aligned}&\texttt{v}\ u\ (\texttt{inst}\ \ell) \hookrightarrow \ell.\\ &\texttt{v}\ u\ (\texttt{inst}\ \ell) \hookrightarrow \texttt{sgl}\ (\texttt{V Nat.zero}\ u\ (\texttt{cons}_{\texttt{Nat}}\ u\ \texttt{nil}_{\texttt{Nat}})).\end{aligned}$$

To fix this, we need some way to tell Dedukti to "prefer" to instantiate a level variable whenever possible. We can achieve this by making these two rewrite rules *sequential*, which is achieved in Dedukti by placing them one after the other, with the first rule lacking a trailing period:

$$\begin{aligned}&[\ell]\ \texttt{v \_}\ (\texttt{inst}\ \ell) \hookrightarrow \ell\\ &[u]\ \texttt{v}\ u\ \texttt{\_} \hookrightarrow \texttt{sgl}\ (\texttt{V Nat.zero}\ u\ (\texttt{cons}_{\texttt{Nat}}\ u\ \texttt{nil}_{\texttt{Nat}})).\end{aligned}$$

Dedukti will attempt sequential rewrite rules in the order that they are declared, in this case causing it to always attempt to replace a `v` application with its instantiating level argument before normalizing it in the style of a deep encoding. This approach has proven to work well in solving our instantiation issue, while maintaining the ability to efficiently represent universe level parameter sets in our encoding without resorting to the use of Dedukti's expensive A/C symbols.

Chapter 4

Encoding Lean's Definitional Equalities

In the previous chapter, we designed an encoding for Lean's universe levels that allowed us to transfer Lean's notion of equality between universe level representations to our Dedukti translation. This encoding allowed us to decide a particular equational theory that is defined on a very specific meta-theoretical construct. When it comes to general object-level terms in Lean, however, there are additional notions of equality that we will need to account for in our translation. In this chapter, we address the encoding of Lean's *definitional equalities* in Dedukti: recall from Section 1.2.1 that definitional equality in Lean consists of an equational relation on terms that is "built-in" to Lean's type theory, that Lean's kernel makes use of in identifying types during typechecking. In designing our translation from Lean to Dedukti, we will have to address these definitional equalities, ensuring in particular that our soundness property holds, where we must ensure that our translation preserves the well-typedness of our input, where the well-typedness of a term in Lean may depend on the use of certain Lean-specific definitional equalities.

4.1 Deriving Definitional Equality Encodings

In order to show that our soundness property (Theorem 2.3.1) holds, we must show that typing is preserved by our translation in the case of the source term having been typed by [CONV]. We can state this as the following theorem:

Theorem 4.1.1. If $\Delta \vdash^{\Gamma} A, B : \texttt{Sort}\ \ell$, $\Delta \vdash^{\Gamma} t : A$, and $\Delta \vdash^{\Gamma} A \equiv B$, then $|\Delta|^{\hookrightarrow}_{\_} \vdash^{\hookrightarrow} |t|^{\hookrightarrow}_{\_} : |B|^{\hookrightarrow}_{\_}$.

Recall that Dedukti features an analogous rule to [CONV] in its own type theory:

$$\frac{\Delta \vdash^{\hookrightarrow} T, U : \texttt{Type} \quad \Delta \vdash^{\hookrightarrow} T \equiv U \quad \Delta \vdash^{\hookrightarrow} t : T}{\Delta \vdash^{\hookrightarrow} t : U}\ [\text{CONV}]$$

As we have by induction that $\Delta \vdash^{\hookrightarrow} \|A\|^{\hookrightarrow}_{\_}, \|B\|^{\hookrightarrow}_{\_} : \texttt{Type}$ and $\Delta \vdash^{\hookrightarrow} |t|^{\hookrightarrow}_{\_} : \|A\|^{\hookrightarrow}_{\_}$, it remains to show that $\Delta \vdash^{\hookrightarrow} \|A\|^{\hookrightarrow}_{\_} \equiv \|B\|^{\hookrightarrow}_{\_}$. For this, it suffices to show that definitional equality of types is preserved by the term-level translation, which can be stated as the following theorem:

Theorem 4.1.2. For all terms A, B such that $\Delta \vdash^{\Gamma} A, B : \texttt{Sort}\ \ell$ and $\Delta \vdash^{\Gamma} A \equiv B$, we have $|\Delta|^{\hookrightarrow}_{\_} \vdash^{\hookrightarrow} |A|^{\hookrightarrow}_{\_} \equiv |B|^{\hookrightarrow}_{\_}$.

This property would follow immediately from the fact that our translation preserves the definitional equality of arbitrarily typed terms, which can be expressed by the following theorem concerning the soundness of the translation w.r.t. the respective definitional equality judgments:

Theorem 4.1.3 (Defeq-Soundness)**.** For all terms t, s such that $\Delta \vdash^{-} t, s : T$ and $\Delta \vdash^{-} t \equiv s$, we have $|\Delta|^{\hookrightarrow}_{-} \vdash^{\hookrightarrow} |t|^{\hookrightarrow}_{-} \equiv |s|^{\hookrightarrow}_{-}$.

Lean's definitional equalities are utilized implicitly by typechecking – there is nothing in the structure of proof terms themselves that directly indicate what particular definitional equalities are relevant to their typing. The situation is similar in Dedukti, in that the rules of β-reduction, rewriting, and syntactic equality are applied implicitly by Dedukti's typechecker. In defining our encoding, we will have to reconcile the differences in the two systems' definitional equality judgments – in particular, Lean includes a number definitional equalities that are not present in Dedukti, whilst Dedukti features rewriting.

If we wish to establish a "one-to-one" translation from Lean syntax to Dedukti syntax (modulo our theory encoding), we will necessarily have to make use of Dedukti's rewrite rules in order to compensate for Dedukti's lack of equivalent judgments in its own theory. As such, our translation will have to produce rewrite rules in the course of translation, extending the set of rewrite rules Σ_R with new ones adapted from particular definitional equality rules found in Lean. In this way, we make the use of these rules "semi-explicit" – they are not reflected in the translation of individual subterms, but can they still be found explicitly represented in our initial encoding and the translation of the input Lean^{-} constant context.

However, we will also have to face the fact that Dedukti's rewrite rules are not exactly sufficient to cover every possible definitional equality implicitly used by Lean's kernel. When rewriting falls short, we will have no choice but to make these rules explicit in the translation itself, by "annotating" subterms in such a way that the use of these definitional equalities in conversion becomes explicit. As we will see in Chapter 5, this can be done via the use of Lean's `cast` operator (an explicit form of conversion) and the equality types `Eq` and `HEq`. Encoding Lean-specific definitional equalities directly as rewrite rules works best in the case that the definitional equality in question has a certain "directionality" that can allows it to be interpreted as a special kind of reduction. When it comes to rules that are of a more "undirected" nature, however, we will see that this approach does not work as well, so the use of explicit type conversions may become necessary.

However, making type conversions and the associated definitional equalities explicit in the translation comes at the cost of larger and less readable output, and will therefore be a last-resort approach that we will only attempt if a direct encoding does not seem possible. It is in attempting to establish Theorem 4.1.3 that we will refine our theory Lean^{-} to some "maximal translatable sub-theory" that can be directly translated to Dedukti via a one-to-one syntactic translation of subterms, along with a context-level translation encoding various definitional equalities via rewrite rules. This refinement process may require possibly dropping definitional equalities from Lean^{-}'s definitional equality judgment if they do not seem amenable to a Dedukti encoding. As we assess the possibility of designing Dedukti encodings for each of Lean's definitional equality rules, we will see there are in fact a few cases where we do end up needing to do this.

To prove Theorem 4.1.3, we will consider each of Lean's definitional equalities in turn, showing for every judgment with a conclusion of the form $\Delta \vdash^{-} a \equiv b$ that its premises imply $|\Delta|^{\hookrightarrow}_{-} \vdash^{\hookrightarrow} |a|^{\hookrightarrow}_{-} \equiv |b|^{\hookrightarrow}_{-}$. Our proof will be within the context of Theorem 4.1.3 and Theo-

rem 2.3.1, where we proceed by induction on the size of typing derivations and definitional equality derivations. Therefore, by the inductive hypothesis, for any typing premise of the form $\Delta \vdash^{-} t : T$ we can assume $|\Delta|^{\hookrightarrow} \vdash^{\hookrightarrow} |t|^{\hookrightarrow} : \|T\|^{\hookrightarrow 1}$, and similarly for any definitional equality premise of the form $\Delta \vdash^{-} a \equiv b$ we can assume $|\Delta|^{\hookrightarrow}_{-} \vdash^{\hookrightarrow} |a|^{\hookrightarrow}_{-} \equiv |b|^{\hookrightarrow}_{-}$.

4.1.1 Congruence Identities

Recall that Lean features a set of definitional equality rules referred to as "congruence identities" that allow two terms of the same root syntactic form to be identified based on the definitional equality of their corresponding subterms, corresponding to each of the syntactic variants of lean expressions. We would like to show that definitional equality is preserved by our translation in the case of definitional equality in Lean by each of these congruence identities. Unfortunately, however, Dedukti has no similar congruence identity rules in its own theory that we can easily map onto in our proof, having only the following singular rule based on the syntactic equality of reduced forms:

$$\frac{\Delta \vdash^{\hookrightarrow} t \hookrightarrow^{*}_{\beta} u_1 \quad \Delta \vdash^{\hookrightarrow} s \hookrightarrow^{*}_{\beta} u_2 \quad \Delta \vdash^{\hookrightarrow} u_1 =_{\mathrm{S}} u_2}{\Delta \vdash^{\hookrightarrow} t \equiv s} \text{ [DEQ]}$$

To try to close the gap between the theories, let's start by trying to derive a similar set of rules in Dedukti that closely correspond to a subset of Lean's congruence identities.

Dedukti's Congruence Identities

It is in fact possible to show that the following set of rules can be derived for Dedukti's definitional equality judgment:

$$\frac{\Delta \vdash^{\hookrightarrow} f \equiv f' \quad \Delta \vdash^{\hookrightarrow} a \equiv a'}{\Delta \vdash^{\hookrightarrow} f\ a \equiv f'\ a'} \text{ [CGR-APP-DK]} \qquad \frac{\Delta \vdash^{\hookrightarrow} A \equiv A' \quad \Delta, x : A \vdash^{\hookrightarrow} e \equiv e'}{\Delta \vdash (x : A) \texttt{=>} e \equiv (x : A') \texttt{=>} e'} \text{ [CGR-LAM-DK]}$$

$$\frac{\Delta \vdash^{\hookrightarrow} A \equiv B \quad \Delta, x : A \vdash^{\hookrightarrow} D \equiv D'}{\Delta \vdash^{\hookrightarrow} (x : A) \ \to \ D \equiv (x : B) \ \to \ D'} \text{ [CGR-ALL]}$$

These rules are analogous to Lean's congruence identities [CGR-APP], [CGR-LAM], and [CGR-ALL]. [CGR-APP] can be shown to follow thanks to the confluence requirement on the Dedukti typing context. Because rewrite rules cannot apply to λ-functions or function types in Dedukti, [CGR-LAM-DK] and [CGR-ALL-DK] both follow easily since replacing each of the corresponding subterms in the LHS and RHS with their common normal forms (derived from the inductive hypothesis applied to the premises) results in unique and syntactically identical normal forms for the left- and right-hand sides.

We will use the lemmas [CGR-APP-DK] and [CGR-LAM-DK] to prove Defeq-Soundness in the case of definitional equality by each of Lean's congruence identities[2]. W.r.t our translation, [CGR-APP-DK] also gives us the following property, which will be useful to us:

Lemma 4.1.4. For all terms T, S such that $\Delta \vdash^{-} T, S : \texttt{Sort}\ \ell$, if $|\Delta|^{\hookrightarrow}_{-} \vdash^{\hookrightarrow} |T|^{\hookrightarrow}_{-} \equiv |S|^{\hookrightarrow}_{-}$, then $|\Delta|^{\hookrightarrow}_{-} \vdash^{\hookrightarrow} \|T\|^{\hookrightarrow}_{-} \equiv \|S\|^{\hookrightarrow}_{-}$.

Proof. Our as-type translation is defined such that $\|T\|^{\hookrightarrow}_{-} := \epsilon\ |\ell|_L\ |T|^{\hookrightarrow}_{-}$ and $\|S\|^{\hookrightarrow}_{-} := \epsilon\ |\ell|_L\ |S|^{\hookrightarrow}_{-}$, so it remains to show that $\Delta \vdash^{\hookrightarrow} \epsilon\ |\ell|_L\ |T|^{\hookrightarrow}_{-} \equiv \epsilon\ |\ell|_L\ |S|^{\hookrightarrow}_{-}$, which follows from the assumption $|\Delta|^{\hookrightarrow}_{-} \vdash^{\hookrightarrow} |T|^{\hookrightarrow}_{-} \equiv |S|^{\hookrightarrow}_{-}$ and [CGR-APP-DK]. □

[1] This is justified because Theorem 4.1.3 is a sub-proof of the inductive proof of Theorem 2.3.1.

[2] As we will see, we do not actually need to use [CGR-ALL-DK] on account of our PTS encoding of function types via the symbol $\dot{\Pi}$.

Application, Function, and Function Type Congruence

Let's start with the case of application congruence via the rule [CGR-APP], recalled below:

$$\frac{\Delta \vdash f \equiv f' \quad \Delta \vdash a \equiv a'}{\Delta \vdash f\ a \equiv f'\ a'}\ \text{[CGR-APP]}$$

Because Lean applications have a shallow encoding in Dedukti, [CGR-APP] in Lean corresponds directly to [CGR-APP-DK]:

Lemma 4.1.5. If $\Delta \vdash^{-} f \equiv g$ and $\Delta \vdash^{-} a \equiv b$, then $|\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |f\ a|_{-}^{\hookrightarrow} \equiv |g\ b|_{-}^{\hookrightarrow}$.

Proof. By the inductive hypothesis, we have $|\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |f|_{-}^{\hookrightarrow} \equiv |g|_{-}^{\hookrightarrow}$ and $|\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |a|_{-}^{\hookrightarrow} \equiv |b|_{-}^{\hookrightarrow}$. Therefore, because our translation is defined such that $|f\ a|_{-}^{\hookrightarrow} = |f|_{-}^{\hookrightarrow}\ |a|_{-}^{\hookrightarrow}$ and $|g\ b|_{-}^{\hookrightarrow} = |g|_{-}^{\hookrightarrow}\ |b|_{-}^{\hookrightarrow}$, we have $\Delta \vdash^{\hookrightarrow} |f|_{-}^{\hookrightarrow}\ |a|_{-}^{\hookrightarrow} \equiv |g|_{-}^{\hookrightarrow}\ |b|_{-}^{\hookrightarrow}$ following from [CGR-APP-DK]. □

Lambdas

Recall the rule for λ-function congruence, [CGR-LAM]:

$$\frac{\Delta \vdash A \equiv A' \quad \Delta, x : A \vdash e \equiv e'}{\Delta \vdash \texttt{fun}\ (x : A)\ \texttt{=>}\ e \equiv \texttt{fun}\ (x : A')\ \texttt{=>}\ e'}\ \text{[CGR-LAM]}$$

[CGR-LAM] corresponds directly to the property [CGR-LAM-DK] in Dedukti, since (like with applications) Lean function terms are translated directly to Dedukti function terms via a shallow encoding.

Lemma 4.1.6. If $\Delta \vdash^{-} A \equiv B$ and $\Delta, x : A \vdash^{-} t \equiv s$, then $|\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |\texttt{fun}\ (x : A)\ \texttt{=>}\ t|_{-}^{\hookrightarrow} \equiv |\texttt{fun}\ (x : B)\ \texttt{=>}\ s|_{-}^{\hookrightarrow}$.

Proof. By the inductive hypothesis, we have $|\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |A|_{-}^{\hookrightarrow} \equiv |B|_{-}^{\hookrightarrow}$ and $|\Delta, x : A|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |t|_{-}^{\hookrightarrow} \equiv |s|_{-}^{\hookrightarrow}$. By Theorem 4.1.4, we also have $|\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} \|A\|_{-}^{\hookrightarrow} \equiv \|B\|_{-}^{\hookrightarrow}$.
Therefore, because our translation is defined such that

$$|\texttt{fun}\ (x : A)\ \texttt{=>}\ t|_{-}^{\hookrightarrow} = (x : \|A\|_{-}^{\hookrightarrow})\ \texttt{=>}\ |t|_{-}^{\hookrightarrow}$$

and

$$|\texttt{fun}\ (x : B)\ \texttt{=>}\ s|_{-}^{\hookrightarrow} = (x : \|B\|_{-}^{\hookrightarrow})\ \texttt{=>}\ |s|_{-}^{\hookrightarrow},$$

our conclusion follows from [CGR-LAM-DK]. □

Function Types

The proof is a bit different for the case of function type congruence via the rule [CGR-ALL], recalled below:

$$\frac{\Delta \vdash A \equiv A' \quad \Delta, x : A \vdash B \equiv B'}{\Delta \vdash (x : A) \to B \equiv (x : A') \to B'}\ \text{[CGR-ALL]}$$

While the structure of the rule is quite similar to that of the rule [CGR-LAM], recall from Section 2.3 that the actual encoding of Lean's dependent function types in Dedukti is based on an encoding that uses the special symbol $\dot{\Pi}$. Therefore, function type congruence in Lean^{-} becomes an instance of application congruence in Dedukti (in fact, we do not actually need to use the property [CGR-ALL-DK]).

Projections

Let's now consider the case of projection congruence via the rule [CGR-PROJ]:

$$\frac{\Delta \vdash t \equiv s}{\Delta \vdash t.i \equiv s.i} \text{ [CGR-PROJ]}$$

As we encode projections in Dedukti using special projection functions, this reduces to an instance of application congruence in Dedukti.

Lemma 4.1.7. Suppose S is a structure-like inductive type with k universe level parameters, n parameters and m fields. For all terms s, s' such that $\Delta \vdash^{-} s, s' : S\ \ell_1\ \ldots\ \ell_k\ p_1\ \ldots\ p_n$ and $\Delta \vdash^{-} s \equiv s'$, we have for all $1 \leq i \leq m$ that $|\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |s.i|_{-}^{\hookrightarrow} \equiv |s'.i|_{-}^{\hookrightarrow}$.

Proof. Recall from Section 2.3 that the projection $s.i$ would be translated to Dedukti as follows:

$$|s.i|_{-}^{\hookrightarrow} = \texttt{prj}_S^i\ |\ell_1|_L\ \ldots\ |\ell_k|_L\ |p_1|_{-}^{\hookrightarrow}\ \ldots\ |p_n|_{-}^{\hookrightarrow}\ |s|_{-}^{\hookrightarrow}$$

Therefore, we are tasked with proving the following:

$$|\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} \texttt{prj}_S^i\ |\ell_1|_L\ \ldots\ |\ell_k|_L\ |p_1|_{-}^{\hookrightarrow}\ \ldots\ |p_n|_{-}^{\hookrightarrow}\ |s|_{-}^{\hookrightarrow} \equiv \texttt{prj}_S^i\ |\ell_1|_L\ \ldots\ |\ell_k|_L\ |p_1|_{-}^{\hookrightarrow}\ \ldots\ |p_n|_{-}^{\hookrightarrow}\ |s'|_{-}^{\hookrightarrow}$$

which follows from the inductive hypothesis and [CGR-APP-DK]. □

Sorts

Recall that Lean's definitional equality judgment includes the following rule for comparing sorts:

$$\frac{\ell \approx \ell'}{\Delta \vdash \texttt{Sort}\ \ell \equiv \texttt{Sort}\ \ell'} \text{ [CGR-SORT]}$$

Recall also from Section 2.3 that our translation of Lean's sorts to Dedukti involves the use of a special symbol $\dot{s}$, which is applied to a translation of the sort's universe level:

$$|\texttt{Sort}\ \ell|_{-}^{\hookrightarrow} := \dot{s}\ |\ell|_L$$

As such, [CGR-SORT] in Lean becomes an instance of application congruence in Dedukti, with Dedukti being able to identify universe level translations thanks to our universe normalization encoding and Theorem 3.0.3.

Constants

Recall that Lean includes a similar definitional equality for checking the equality of level-instantiated constant references. If a constant symbol C takes n level parameters, we have the following definitional equality:

$$\frac{\ell_1 \approx \ell'_1 \quad \cdots \quad \ell_n \approx \ell'_n}{\Delta \vdash C.\{\ell_1, \ldots, \ell_n\} \equiv C.\{\ell'_1, \ldots, \ell'_n\}} \text{ [CGR-CONST]}$$

Recall also our translation of constant references that was defined in Section 2.3:

$$|C.\{\ell_1, \ldots, \ell_n\}|_{-}^{\hookrightarrow} := C\ (\texttt{inst}\ |\ell_1|_L)\ \ldots\ (\texttt{inst}\ |\ell_n|_L)$$

Here, similarly to the case of [CGR-SORT], the equality check performed between each pair of instantiated levels becomes a comparison of the normal forms of the translations of these levels in Dedukti, which is assured to succeed via our universe normalization encoding and Theorem 3.0.3

4.1.2 Proof Irrelevance

Recall the rule for proof irrelevance in Lean:

$$\frac{\Delta \vdash P : \texttt{Prop} \quad \Delta \vdash p, q : P}{\Delta \vdash p \equiv q} \text{ [PI]}$$

As Dedukti does not have any built-in notion of propositional types, it also does not provide support for any notion of proof irrelevance, or anything similar to it, in terms of a definitional equality rule that identifies terms based on their typing. So, we will have to try to find a way to explicitly encode [PI] within Dedukti, if possible.

Encoding Proof Irrelevance via Erasure

Unlike the congruence rules, where definitional equality depended on the definitional equality of subterms (or level expressions), definitional equality by proof irrelevance depends on the definitional equality of the *types* of the terms being compared. In Lean's kernel, it is implemented by checking the definitional equality of the inferred propositional types of both proof terms being compared.

This creates a particular difficulty for our translation, since it is not possible to define rewrite rules in Dedukti that are conditional on typing. In order to have such an encoding possibly work, we will need to provide this typing information ourselves, as part of the translation, say, by defining a tuple constructor `Erase` for proof terms that carries a proof along with its type:

$$\texttt{def Erase} : (P : U\ \texttt{z}) \to \epsilon\ \texttt{z}\ P \to \epsilon\ \texttt{z}\ P.$$

We could then define our translation such that, for all proof terms p such that $\Delta \vdash p : P$ and $\Delta \vdash p : \texttt{Prop}$:

$$|p|^{\hookrightarrow}_{\_} \stackrel{?}{:=} \texttt{Erase}\ |P|^{\hookrightarrow}_{\_}\ |p|^{\hookrightarrow 0}_{\_},$$

with the translation $|\cdot|^{\hookrightarrow 0}_{\_}$ defined mutually with $|\cdot|^{\hookrightarrow}_{\_}$, the only difference being that it does not apply this transformation to its input (to avoid non-termination).

For two distinct proof terms $\Delta \vdash p : P$ and $\Delta \vdash q : Q$, we would now need to reduce the problem of comparing $\texttt{Erase}\ |P|^{\hookrightarrow}_{\_}\ |p|^{\hookrightarrow}_{\_}$ and $\texttt{Erase}\ |Q|^{\hookrightarrow}_{\_}\ |q|^{\hookrightarrow}_{\_}$ to a comparison of their translated proof types $|P|^{\hookrightarrow}_{\_}$ and $|Q|^{\hookrightarrow}_{\_}$. We could do this by defining a new symbol `Erased`, together with a rewrite rule on `Erase` that performs proof erasure:

$$\texttt{def Erased} : (P : U\ \texttt{z}) \to \epsilon\ \texttt{z}\ P.$$
$$[P]\ \texttt{Erase}\ P\ \_ \hookrightarrow \texttt{Erased}\ P.$$

In this way, `Erased` P takes the place of `Erase` $P\ p$ as the "normal form" of a proof term p with type P. By [CGR-APP-DK] and the inductive hypothesis, which gives us $\Delta \vdash^{\hookrightarrow} |P|^{\hookrightarrow}_{\_} \equiv |Q|^{\hookrightarrow}_{\_}$, we can then show that we satisfy Defeq-Soundness in the case of definitional equality by [PI].

However, the use of erasure creates issues for the definition of rewrite rules relating to recursor reduction that we need to address. Consider the following inductive proposition:

```
inductive T (f : Bool → Prop) : f Bool.true → Type where
| mk (p : f Bool.true) : T f p
```

This generates a recursor with the type:

```
-- T.rec.{u} {f : Bool → Prop} {p : f Bool.true} {motive : T f p → Sort u}
-- (mk : motive (T.mk p)) (t : T f p) : motive t
#check T.rec
```

that reduces as follows:

```
example : @T.rec f p mtv m (@T.mk f p) = m := rfl
```

Converting this into a recursor reduction rule into a rewrite rule along the lines of what is described below in Section 4.2, we would require that the major premise is a constructor application:

$$[\texttt{f}, \texttt{p}, \texttt{mtv}, \texttt{m}]\ \texttt{T.rec f p mtv m \_ (T.mk f p)} \hookrightarrow \texttt{m}.$$

However, with our erasure scheme, this rule is in fact ill-typed. The type of `m` is `mtv (T.mk f (Erased (f Bool.true)))` (according to the translated type of `T.rec`), and yet the type of the recursor application is `mtv (T.mk f p)`, so Dedukti cannot verify subject reduction. We know due to our translation that that any proof term must reduce to an application of `Erased`, but the unification algorithm that Dedukti's subject reduction checker uses has no awareness of this invariant.

To fix this, we might think to make the erasure explicit in LHS of the rewrite rule as follows:

$$[\texttt{f}, \texttt{p}, \texttt{mtv}, \texttt{m}]\ \texttt{T.rec f p mtv m \_ (T.mk f (Erased (f Bool.true)))} \hookrightarrow \texttt{m}.$$

However, this still does not quite cut it – the expression `f Bool.true` in the LHS violates a restriction that Dedukti places on rewrite patterns, namely that they must be "Miller patterns". In Miller patterns, higher-order pattern variables (such as `f` above) must only be applied to a list of distinct pattern variables, and the pattern `f Bool.true` does not obey this. Indeed, such a rule *should* not go through as it results in non-confluence. If `f` is a particular defined symbol – for instance, the constant function `(_ : Bool) => True` – the reduction of the application `f Bool.true` may occur before this rule is considered, at which point it can no longer apply.

To fix this, we can observe that since the argument to `Erased` appears directly in its type, its value can be resolved by unification during subject reduction checking. So, simply replacing it with "`_`" in the pattern gets around the non-Miller pattern issue, and Dedukti's unifier is able to assign the value of the placeholder to `f Bool.true`:

$$[\texttt{f}, \texttt{p}, \texttt{mtv}, \texttt{m}]\ \texttt{T.rec f p mtv m \_ (T.mk f (Erased \_))} \hookrightarrow \texttt{m}.$$

So, with this approach it would seem to be be necessary to encode every instance of a proof in the LHS of rewrite rules as `Erased _`, with the hope that the placholder value will always be able to be determined when necessary.

While seemingly within the realm of possibility, this erasure approach to encoding proof irrelevance comes at the cost of an important translation property: completeness. The typing of the `Erased` operator is problematic, causing us to lose our completeness property, as we can construct a proof of `False` in our encoding as $\texttt{Erased}\ |\texttt{False}|^{\hookrightarrow}_{\_}$ (with `False` presumably not being provable in Lean). If we were to include such a rule in our encoding, we would have to take extra care to ensure that it is not exploited in any way when translating terms to Dedukti and exporting proofs to other systems, and it would erode some of our confidence in the translated proofs. Besides, we would have to assume the equivalent of the rewrite rule on `Erase` in our target theory, which already somewhat amounts to assuming that the target theory has some degree of proof irrelevance, and this may very well not be the case.

So, in light of these complications, for the moment let's take a more conservative approach and suppose that [PI] is not present in Lean$^-$, deferring it to be handled in a preliminary translation from Lean to Lean$^-$.

4.1.3 η Rules

Lean's type theory includes some special definitional equality rules known as "η" rules, which extend its definitional equality judgment in certain practical ways to allow for easier formalization. While

all of the other definitional equalities can be seen as forming the basis of Lean's definitional equality judgment, defining "base equalities" that would otherwise be impossible to show, η rules can be thought of as more "derivative": they can be shown as provable consequents of other definitional equalities or axioms.

Unit η

Recall Lean's definitional equality rule for unit-η:

$$\frac{U \text{ unit-like} \quad \Delta \vdash t, s : U}{\Delta \vdash t \equiv s} \text{ [UNIT]}$$

This rule bears close resemblance to the rule [PI], requiring a particular typing condition on the subject terms. As such, it also does not readily give way to a Dedukti encoding, with all of our previous attempts at designing an encoding of [PI] also failing to provide a satisfactory encoding of [UNIT] in Dedukti for similar reasons. Therefore, we also assume [UNIT] to not be present in the theory Lean$^-$, relying instead on its elimination via a preliminary translation from Lean to Lean$^-$.

Struct-η

Recall the simplified struct-η rule that was presented in Section 1.2.2:

$$\frac{S \text{ struct-like} \quad \Delta \vdash t : S\ p_1\ \ldots\ p_n}{\Delta \vdash t \equiv S.\texttt{mk}\ p_1\ \ldots\ p_n\ t.1\ \ldots\ t.m} \text{ [ETA-S']}$$

Let's start by trying to encode this simpler rule, and see to what extent an encoding that we devise for it can be adapted for the more general struct-η rule. Such an equality is characteristic of what is known as "surjective pairing" which is a fairly well-studied topic, with results demonstrating that it is not possible to design a confluent rewrite system that captures surjective pairing in an untyped system. However, since Dedukti is typed, it should be possible to design a confluent rewrite system encoding Lean's struct-η.

We could first think about attempting to encode [ETA-S'] directly as a rewrite rule. Could we, perhaps, define a rewrite rule that applies struct-η-expansion to any term typed as a struct? That is, one that essentially applies the conclusion judgment of [ETA-S'] from left to right as a rewrite rule? Unfortunately, this idea immediately runs into some problems.

Firstly, what exactly would the LHS of this rule have to look like? We have to be sure that it is able to match on every possible instance of a struct-typed term. This effectively means that it would have to match even in the case that the term is simply a bound variable, carrying no syntactic information to indicate that it is of struct type. Consequently, the LHS of our rewrite rule would have to simply be a pattern variable that places no syntactic constraints on the matched term. In terms of our `Point` example, this rewrite rule might look something like:

$$[t]\ t \hookrightarrow \texttt{Point.mk}\ (\texttt{prj}^1_S\ t)\ (\texttt{prj}^2_S\ t).$$

We would correspondingly have to also declare `Point.mk` as a defined symbol, rather than a static symbol (as we would normally do when declaring translated constructor types). However, such a rule violates one of the basic restrictions Dedukti places on rewrite rules, as it lacks a defined symbol as the application head of the LHS, and as such it is rejected by Dedukti's syntax parser.

Additionally, even if Dedukti allowed for such a rule to be stated, it does not satisfy subject reduction, as we do not know the type of t. While the type of t could theoretically be determined from the RHS, subject reduction in Dedukti works in the other direction, ensuring that the RHS term is well-typed according to pattern variable typings derived from the LHS alone. If the subject reduction checker could determine pattern variable typings that are only derivable from the RHS,

this would necessarily require that the LHS is only matched on *conditionally* when these typings are observed to actually apply, as otherwise the rewritten term will not be well-typed. As mentioned earlier, Dedukti does not offer support for conditional rewrite rules of this form – and besides, even if it did, this rule would immediately result in non-termination, as the only restriction on the LHS is its typing, which does not change as a result of rewriting, and so this rule could be infinitely applied. We would need to additionally constrain the rule such that it does not match on applications of `Point.mk`, and Dedukti also does not provide any way to specify "exceptions" to the applicability of rewrite rules. So, we clearly need to think of another way to encode this rule within Dedukti.

Another option is to define the rewrite rule in the opposite direction – that is, we can declare `Point.mk` as a defined symbol and add the following rewrite rule to our encoding:

$$[t]\ \texttt{Point.mk}\ (\texttt{prj}^1_{\texttt{Point}}\ t)\ (\texttt{prj}^2_{\texttt{Point}}\ t)\ \hookrightarrow\ t.$$

Now, Dedukti's subject reduction checker can determine that $t : \texttt{Point}$ from the LHS, which has type `Point` as well, so the RHS validates as having the expected type of `Point`. Unfortunately, however, using this rewrite rule results in non-confluence when combined with our standard encoding of reductions rules in Dedukti (see Section 4.2). The problem arises from the possibility of erasing explicit constructor application heads that are necessary to match on in recursor rewrite rules. For instance, our standard reduction encoding for the `Point.rec` recursor would look like:

$$[\texttt{m}, \texttt{x}, \texttt{y}]\ \texttt{Point.rec}\ \_\ \texttt{m}\ (\texttt{Point.mk}\ \texttt{x}\ \texttt{y})\ \hookrightarrow\ \texttt{m}\ \texttt{x}\ \texttt{y}.$$

Here, matching on an explicit constructor application is necessary to extract the field values needed for the reduction. In combination with the struct-η rewrite rule, however, this results in the following critical pair:

$$\texttt{Point.rec}\ \texttt{mtv}\ \texttt{m}\ (\texttt{Point.mk}\ (\texttt{prj}^1_{\texttt{Point}}\ t)\ (\texttt{prj}^2_{\texttt{Point}}\ t)) \overset{\texttt{Point.mk}}{\hookrightarrow} \texttt{Point.rec}\ \texttt{mtv}\ \texttt{m}\ t.$$

$$\texttt{Point.rec}\ \texttt{mtv}\ \texttt{m}\ (\texttt{Point.mk}\ (\texttt{prj}^1_{\texttt{Point}}\ t)\ (\texttt{prj}^2_{\texttt{Point}}\ t)) \overset{\texttt{Point.rec}}{\hookrightarrow} \texttt{m}\ (\texttt{prj}^1_{\texttt{Point}}\ t)\ (\texttt{prj}^2_{\texttt{Point}}\ t).$$

As we will see in Section 4.2, it is in fact possible to address this issue by defining a special rule for recursor reduction on structs, rather than using the default encoding strategy for Lean's recursor rules.

So, in general, we define our encoding of struct-η as follows. Given some struct-like type S with n parameters and m fields, we declare the constructor as a defined symbol and add the following rewrite rule:

$$[t]\ S.\texttt{mk}\ (\_)_1\ \ldots\ (\_)_n\ (\texttt{prj}^1_S\ t)\ \ldots\ (\texttt{prj}^m_S\ t)\ \hookrightarrow\ t.$$

Struct-η: The Actual Rule

While the rule [ETA-S'] presented above served well for a first introduction to struct-η in Lean that helped us design an initial Dedukti encoding, recall that the actual rule implemented by the kernel is a bit less directly adapted from struct-η-expansion, taking the form:

$$\frac{S \text{ struct-like} \quad \Delta \vdash t : S\ p_1\ \ldots\ p_n \quad \Delta \vdash t.1 \equiv a_1 \quad \ldots \quad \Delta \vdash t.m \equiv a_m}{\Delta \vdash t \equiv S.\texttt{mk}\ p_1\ \ldots\ p_n\ a_1\ \ldots\ a_m}\ \text{[ETA-S]}$$

This rule is a bit more general than [ETA-S'], requiring that each of the constructor field arguments are only definitionally equal to (rather than syntactically equal to) a corresponding projection of t.

Despite this difference, we can in fact show that the rewrite rule derived for [ETA-S'] still ensures Defeq-Soundness in the case of typing by [ETA-S]. For this, we need to use the following property, stating that any two terms having definitionally equal Dedukti translations also have reducts whose translations are definitionally equal:

Theorem 4.1.8. If $\Delta \vdash^{-} t \rightsquigarrow^{*} t'$ and $\Delta \vdash^{-} s \rightsquigarrow^{*} s'$, then

$$|\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |t|_{-}^{\hookrightarrow} \equiv |s|_{-}^{\hookrightarrow} \implies |\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |t'|_{-}^{\hookrightarrow} \equiv |s'|_{-}^{\hookrightarrow}.$$

This property follows from Reduction-Soundness, which is described below in Section 4.2.

Theorem 4.1.9. Suppose S is a struct-like inductive type with n parameters and m fields and constructor `mk`. For all terms t such that $\Delta \vdash^{-} t : S\ p_1\ \ldots\ p_n$ and $\Delta \vdash^{-} t.1 \equiv a_1, \ldots, \Delta \vdash^{-} t.n \equiv a_n$, we have:

$$|\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |t|_{-}^{\hookrightarrow} \equiv |S.\texttt{mk}\ p_1\ \ldots\ p_n\ a_1\ \ldots\ a_m|_{-}^{\hookrightarrow}$$

Proof. Let t' and a_i' be the weak-head normal forms of t and the a_i. We then have that $\Delta \vdash^{-} t.i \rightsquigarrow^{*} t'.i$ and $\Delta \vdash^{-} a_i \rightsquigarrow^{*} a_i'$ for all $1 \leq i \leq n$. Via [RED] and the premises, it follows that $\Delta \vdash^{-} t'.1 \equiv a_1', \ldots, \Delta \vdash^{-} t'.n \equiv a_n'$ From the inductive hypothesis, we have $|\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |t.1|_{-}^{\hookrightarrow} \equiv |a_1|_{-}^{\hookrightarrow}, \ldots, |\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |t.n|_{-}^{\hookrightarrow} \equiv |a_n|_{-}^{\hookrightarrow}$, and so from Theorem 4.1.8 it then follows that $|\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |t'.1|_{-}^{\hookrightarrow} \equiv |a_1'|_{-}^{\hookrightarrow}, \ldots, |\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |t'.n|_{-}^{\hookrightarrow} \equiv |a_n'|_{-}^{\hookrightarrow}$, resulting in the same set of assumptions that we originally had, but with t and the a_i replaced with the WHNF terms t' and a_i'.

So, without loss of generality, we can suppose that t and all of the a_i are in weak-head normal form. If t is a constructor application $S.\texttt{mk}\ p_1\ \ldots\ p_n\ b_1\ \ldots\ b_m$, then [RED], [RPROJ], and the premises give us $\Delta \vdash^{-} b_1 \equiv a_1, \ldots, \Delta \vdash^{-} b_n \equiv a_n$, and thus $|\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |b_1|_{-}^{\hookrightarrow} \equiv |a_1|_{-}^{\hookrightarrow}, \ldots, |\Delta|_{-}^{\hookrightarrow} \vdash^{\hookrightarrow} |b_n|_{-}^{\hookrightarrow} \equiv |a_n|_{-}^{\hookrightarrow}$ by the inductive hypothesis and Theorem 4.1.8, with the result following by application congruence.

Otherwise, suppose that t is not a constructor application. As t is in WHNF and not a constructor application, we know that each projection $t.i$ cannot reduce via [RPROJ], and so for all i, $\Delta \vdash^{\hookrightarrow} |t.i|_{-}^{\hookrightarrow} \hookrightarrow_{\beta}^{*\mathrm{N}} \texttt{prj}_S^i\ t'$ for some term t' such that $\Delta \vdash^{\hookrightarrow} |t|_{-}^{\hookrightarrow} \hookrightarrow_{\beta}^{*\mathrm{N}} t'$. By the inductive hypothesis, it follows that $\Delta \vdash^{\hookrightarrow} a_i \hookrightarrow_{\beta}^{*\mathrm{N}} \texttt{prj}_S^i\ t'$ for all i. Therefore, we have:

$$\Delta \vdash^{\hookrightarrow} |S.\texttt{mk}\ p_1\ \ldots\ p_n\ a_1\ \ldots\ a_m|_{-}^{\hookrightarrow} \hookrightarrow_{\beta}^{*} S.\texttt{mk}\ |p_1|_{-}^{\hookrightarrow}\ \ldots\ |p_n|_{-}^{\hookrightarrow}\ (\texttt{prj}_S^1\ t')\ \ldots\ (\texttt{prj}_S^m\ t')$$

Thus, the LHS of the struct-η rewrite rule is able to match, giving us

$$\Delta \vdash^{\hookrightarrow} |S.\texttt{mk}\ p_1\ \ldots\ p_n\ a_1\ \ldots\ a_m|_{-}^{\hookrightarrow} \hookrightarrow_{\beta}^{*} t'$$

and hence our result. □

Function η

Recall the rule for function-η in Lean:

$$\frac{}{\Delta \vdash (\texttt{fun}\ (x : A)\ \texttt{=>}\ e\ x) \equiv e}\ \text{[FUN-ETA]}$$

Unfortunately, [FUN-ETA] does not seem to encodable as a rewrite rule in Dedukti. Attempting to define a rule that directly applies η-expansion runs into similar problems that we observed in the struct-η case:

$$[f]\ f \hookrightarrow (x : A)\ \texttt{=>}\ f\ x.$$

Namely, such rule would have to match conditionally on typing, which Dedukti does not support, and even if it did, the rule would be non-terminating. So, we might consider encoding the equality as a rewrite rule in the reverse direction, defining a rewrite rule that performs "de-η-expansion" on function terms:

$$[f]\ (x : A)\ \texttt{=>}\ f\ x \hookrightarrow f.$$

However, such a rewrite rule would not be permitted by Dedukti, as this is not compatible with Dedukti's matching algorithm, which requires rewrite rules to match on defined function heads. As

we perform a shallow translation of Lean functions to Dedukti function via our PTS encoding, such a rule would have be defined with a λ-function on the LHS, which is not allowed in Dedukti. In addition, the LHS a non-Miller pattern, as it applies the pattern variable f to the non-pattern-variable x.

So, any possible encoding of function-η will have to rely on our translation itself. One approach we might take is to explicitly η-expansion any terms of function type in our translation output. However, as described by Genestier [18] this is in fact not sufficient, due to dependent types. Instead, Genestier introduces the use of a special type annotation symbol that allows for the application of rewrite rules simulating function-η. In our case, this might look something like the following:

$$\texttt{def Typ :}\ (\ell : L) \to (T : U\ \ell)\ \to \epsilon\ \ell\ T \to \epsilon\ \ell\ T.$$

This symbol would have the following rewrite rules defined on it, effecting η-expansion in the case of it being applied to a term of function type, and otherwise simply reducing to the original term. This can be expressed in the following pair of sequential rewrite rules:

$$[\ell_1, \ell_2, A, B, f]\ \texttt{Typ \_}\ (\dot{\Pi}\ \ell_1\ \ell_2\ A\ B)\ f\ \hookrightarrow\ (x : \epsilon\ \ell_1\ A)\ \texttt{=>}\ \texttt{Typ}\ \ell_2\ (B\ (\texttt{Typ}\ \ell_1\ A\ x)\ (f\ (\texttt{Typ}\ \ell_1\ A\ x)))$$
$$[t]\ \texttt{Typ \_ \_}\ t\ \hookrightarrow\ t.$$

While this does make for a sound encoding, it is not a very practical approach, because the use of `Typ` to annotate every output term with its type makes the translation output much larger and more complex.

Alternatively, recall from Section 1.3.2 that Dedukti provides native support function-η in the following rule:

$$\frac{\Delta \vdash^{\hookrightarrow} f : (x : A)\ \to\ B}{\Delta \vdash^{\hookrightarrow} ((x : A)\ \texttt{=>}\ f\ x) =_{\mathrm{S}} f}\ \text{[FUN-ETA]}$$

This is allowed because function-η is a very common conversion present in many proof assistants, and it is reasonable to expect many target systems to already support function-η, making an explicit encoding in Dedukti unnecessary. The use of this identity in Dedukti's typecheker can be enabled by passing the command line option "`-eta`" to `dkcheck`. So, assuming that we are okay with restricting export of our translated proofs to systems already supporting function-η, we can suppose that the target Dedukti type theory features the rule [FUN-ETA], and we do not have to design any special encoding of function-η in Dedukti.

4.2 Deriving Reduction Rule Encodings

Recall the rule for definitional equality via reduction in Lean:

$$\frac{\Delta \vdash t \rightsquigarrow^* t' \quad \Delta \vdash t' \equiv s}{\Delta \vdash t \equiv s}\ \text{[RED]}$$

This rule is a bit more involved than the previously considered definitional equality, as it makes use of the separately defined reduction relation $\Delta \vdash t \rightsquigarrow^* t'$.

Reduction-Soundness

So, let's consider how to show Defeq-Soundness in the case of a definitional equality arising in Lean$^-$ via [RED]. Analogously to our definition of Defeq-Soundness, we can define a "Reduction-Soundness" property as follows:

Theorem 4.2.1. If $\Delta \vdash^{-} t \rightsquigarrow s$, then $\Delta \vdash^{\hookrightarrow} |t|^{\hookrightarrow}_{-} \hookrightarrow^*_{\beta} |s|^{\hookrightarrow}_{-}$.

That is, every single-step head-reduction in Lean$^-$ corresponds to a valid reduction between the translated terms in Dedukti. From this, we can prove the following via induction on the size of Lean$^-$ reduction sequences and the transitivity of $\hookrightarrow_\beta^*$:

Theorem 4.2.2 (Reduction-Soundness)**.** If $\Delta \vdash^- t \rightsquigarrow^* s$, then $\Delta \vdash^{\hookrightarrow} |t|_-^{\hookrightarrow} \hookrightarrow_\beta^* |s|_-^{\hookrightarrow}$.

This property allows us to show Defeq-Soundness in the case of [RED] as follows:

Theorem 4.2.3. If $\Delta \vdash^- t \rightsquigarrow^* s$ and $\Delta \vdash^- t \equiv t'$, then $|\Delta|_-^{\hookrightarrow} \vdash^{\hookrightarrow} |t'|_-^{\hookrightarrow} \equiv |s|_-^{\hookrightarrow}$.

Proof. We have from Theorem 4.2.2 that $\Delta \vdash^{\hookrightarrow} |t|_-^{\hookrightarrow} \hookrightarrow_\beta^* |s|_-^{\hookrightarrow}$. Let s' be the unique normal form of $|s|_-^{\hookrightarrow}$, that is, $\Delta \vdash^{\hookrightarrow} |s|_-^{\hookrightarrow} \hookrightarrow_\beta^{*\mathrm{N}} s'$. Then, we have $\Delta \vdash^{\hookrightarrow} |t|_-^{\hookrightarrow} \hookrightarrow_\beta^{*\mathrm{N}} s'$. By the inductive hypothesis, we have $|\Delta|_-^{\hookrightarrow} \vdash^{\hookrightarrow} |t|_-^{\hookrightarrow} \equiv |t'|_-^{\hookrightarrow}$, and so it follows that $\Delta \vdash^{\hookrightarrow} |t'|_-^{\hookrightarrow} \hookrightarrow_\beta^{*\mathrm{N}} s'$, giving us our result. □

So, our task is to define our translation encoding such that every reduction rule in Lean corresponds to a sequence of reductions in Dedukti. At first glance, this may seem like a trivial task: we could simply directly translate the left- and right-hand sides of each one of Lean's reduction rules into the left- and right-hand sides of a corresponding rewrite rule in Dedukti. However, Dedukti's restrictions on how rewrite rules must be written may not allow for such an encoding in every case. Additionally, Lean places conditions on how certain reduction rules can apply (in the form of typing premises on reduction rules) that are not necessarily syntactically enforceable using a Dedukti rewrite rule, particularly when these conditions rely on typing information. As we will see, this latter discrepancy in particular causes difficulties in our ability to encode certain rules in Dedukti. As was the case when encoding definitional equality rules, we will have the leeway to eliminate from our theory any reduction rules that prove difficult to express as rewrite rules in Dedukti, deferring their handling to a preliminary translation step.

β-Reduction

Recall Lean's β-reduction rule:

$$\frac{}{\Delta \vdash (\texttt{fun}\ (x : A)\ \texttt{=>}\ e)\ a \rightsquigarrow e[x/a]}\ [\text{BETA}]$$

As our PTS encoding does a shallow translation of both λ-functions and application terms, we immediately have Reduction-Soundness in the case of [BETA] via Dedukti's own [BETA] rule.

δ-Reduction

Recall Lean's rule for δ-reduction:

$$\frac{\Delta = (\Sigma; \Gamma) \quad \Sigma \text{ contains } C.\{u_1, \ldots, u_n\}, \text{ defined with value } v}{\Delta \vdash C.\{l_1, \ldots, l_n\} \rightsquigarrow v[[u_1/l_1, \ldots, u_n/l_n]]}\ [\text{DELTA}]$$

As was described in Section 2.3, our translation of constant declarations introduces additional abstractions for each of the universe level parameters, with the references to these parameters within the definition translating to bound variable references. Correspondingly, later references to these constants translate to applications of the corresponding Dedukti constant symbols to the translations of the level parameter instantiations. As such, we do not need to make any special consideration of Lean's universe level parameter instantiation in our translation as this simply becomes an instance of β-reduction in Dedukti.

Projection Reduction

Recall the rule for structure projection reduction in Lean:

$$\frac{S \text{ structure-like}}{\Delta \vdash (S.\texttt{mk}\ p_1\ \ldots\ p_n\ a_1\ \ldots\ a_i\ \ldots\ a_m).i \rightsquigarrow a_i} \text{ [RPROJ]}$$

While is feasible to eliminate [RPROJ] via a pre-translation step in which we turn every instance of a projection into an explicitly constructed recursor application corresponding to the projection, this approach result in exactly the same kind of inefficiencies in typechecking the output of our translation that Lean had sought to to avoid in introducing projections.

Instead, we directly transcribe [RPROJ] into a Dedukti rewrite rule, generating the following rules on our projection symbols $\texttt{prj}_S^i$:

$$\forall 1 \leq i \leq n,\ [\ldots]\ \texttt{prj}_S^i\ \ell'_1\ \ldots\ \ell'_k\ p'_1\ \ldots\ p'_n\ (S.\texttt{mk}\ \ell_1\ \ldots\ \ell_k\ p_1\ \ldots\ p_n\ f_1\ \ldots\ f_n) \hookrightarrow f_i.$$

from which Reduction-Soundness in the case of [RPROJ] can be shown to follow.

Recursor Reduction

Recall that the recursor rules follow a particular pattern determined by the inductive type and its constructors, a general form of which is formally outlined by Carneiro [13] in the presentation of the "ι rule" for recursor reduction. The definition of this rule uses some extra notation for expressing inductive types in Lean's type theory, that we will not use here. As it turns out, we do not need to make direct use of a generic ι rule in defining our translation, because of the fact that the rules already come pre-computed for us when the Lean kernel generates the recursor for an inductive type, which is very convenient for our translation as it means that we do not need to generate these rules ourselves.

Specifically, in Lean's metaprogramming framework, recursor rules are stored in the following datatype:

```
structure RecursorVal extends ConstantVal where
  all : List Name
  numParams : Nat
  numIndices : Nat
  numMotives : Nat
  numMinors : Nat
  rules : List RecursorRule
  k : Bool
  isUnsafe : Bool
```

The field `rules` contains a list of `RecursorRule` objects that contain information on the recursor rules for each of the inductive type constructors:

```
structure RecursorRule where
  ctor : Name -- ctor this rule is for
  nfields : Nat -- number of non-param args
  rhs : Expr -- reduction function
```

The field `rhs` contains a λ-function corresponding to the recursor reduction rule for the constructor `ctor`. Recall in particular the example `Vec` inductive type and its associated recursor:

```
inductive Vec (T : Type) : Nat → Type where
| nil  : Vec T Nat.zero
```

```
| cons : (n : Nat) → Vec T n → T → Vec T (Nat.succ n)
#check Vec.rec
recursor Vec.rec.{u} : {T : Type} → {motive : (a : Nat) → Vec T a → Sort u} →
  (nil : motive Nat.zero Vec.nil) →
  (cons : (n : Nat) → (v : Vec T n) → (t : T) →
    motive n v → motive n.succ (Vec.cons n v t)) →
  {n : Nat} → (t : Vec T n) → motive n t
```

The kernel will generate the following two reduction rules corresponding to each of the constructors of `Vec`, which it will populate the `RecursorRule.rhs` field with:

```
-- `RecursorRule.rhs` field for `Vec.nil'
fun T motive nil cons => nil
-- `RecursorRule.rhs` field for `Vec.cons`
fun T motive nil cons (n : Nat) (v : Vec n) (t : T)
  => cons v t (@Vec.rec motive nil cons v)
```

When a recursor is reduced, these functions are applied to the corresponding arguments from the recursor application, plus the extracted field values from the major premise constructor application.

For our translation encoding, we can interpret the bodies of these functions as the right-hand-sides of a rewrite rule that takes a corresponding recursor application on the LHS, generating in the case of `Vec.rec` the following rewrite rules:

```
[...] Vec.rec u T mtv nil cons Nat.zero (Vec.nil _ _) ↪ nil.
[...] Vec.rec u T mtv nil cons (Nat.succ n) (Vec.cons _ _ v t)
      ↪ cons n v t (Vec.rec u T mtv nil cons n v).
```

Note that we have to take special care in translating the universe level parameter `u` in this rule. Unlike in normal translation (e.g. while translating the body of a defined constant symbol), where our translation would output the level expression `lvl.var 0 v` according to our universe encoding, we clearly do not want to do this in the LHS of the above rule. We want to be able to match on *any* particular instantiation of the universe level parameter. Therefore, we simply translate the universe level variable as a pattern variable, with the corresponding pattern variable being used on the RHS.

However, observe that the rewrite rules above introduce some complexity in the LHS pattern in the recursor index arguments `Nat.z` and `Nat.succ n`. While in this example, this is quite innocuous, in general doing so can lead to non-confluence as recursor indices can be arbitrarily complex functions of the parameters and constructor fields.

For instance, suppose that we have the following inductive type:

```
inductive Vec' (T : Type) : Nat → Type where
| nil : Vec' T Nat.zero
| sgl : T → Vec' T (Nat.succ (Nat.zero))
| app : (n m : Nat) → Vec T n → Vec T m → Vec' T (Nat.add n m)
```

Here, the index argument is a particular function of constructor fields in the `Vec.app` case. Naively, we would translate the recursor reduction rule for `Vec.app` construction as follows:

```
[...] Vec'.rec u T mtv nil sgl app (Nat.add n m) (Vec'.app _ _ v v')
      ↪ app n m v v' (Vec'.rec u T mtv nil sgl app n v) (Vec'.rec u T mtv nil sgl app m v').
```

However, this rule result in non-confluence, as we have the following critical pair:

```
Vec'.rec u T mtv nil sgl app (Nat.add Nat.zero Nat.zero) (Vec'.app _ _ Vec'.nil Vec'.nil)
      -Vec'.rec-> app n m v v' (Vec'.rec u T mtv nil app n v) (Vec'.rec u T mtv nil app m v').
Vec'.rec u T mtv nil sgl app (Nat.add Nat.zero Nat.zero) (Vec'.app _ _ Vec'.nil Vec'.nil)
      -Nat.add-> Vec'.rec u T mtv nil sgl app Nat.zero (Vec'.app _ _ Vec'.nil Vec'.nil).
```

In the latter rewrite, we apply reduction to reduce `Nat.add Nat.zero Nat.zero` before attempting to apply the recursor rewrite rule. The resultant recursor application cannot reduce, despite the fact that the recursor application is well-typed with an explicit construction in the major premise. since `Nat.zero` does not match the pattern `Nat.add n m` expected by the rule for `Vec'.rec`.

There is an easy way around this issue, however. Because the index arguments can always be inferred from the particular construction in the major premise, we can in fact omit the index from the LHS pattern, replacing it with a wildcard pattern and obtaining the following recursor reduction rewrite rule for `Vec'.app` instead:

```
[...] Vec'.rec u T mtv nil sgl app _ (Vec'.app _ _ v v')
    ↪ app n m v v' (Vec'.rec u T mtv nil app n v) (Vec'.rec u T mtv nil app m v').
```

With this new encoding, we would similarly translate the rules of `Vec` as:

```
[...] Vec.rec u T mtv nil cons _ (Vec.nil _ _) ↪ nil.
[...] Vec.rec u T mtv nil cons _ (Vec.cons _ _ v t)
    ↪ cons n v t (Vec.rec u T mtv nil cons n v).
```

Recall that Lean also allows for the definition of more complex inductive types, such as mutual inductive types, where one inductive type is recursively defined with respect to another. Consider again the mutual `Even` / `Odd` inductive types, with the following recursors and associated recursor reduction rules:

```
mutual
inductive Even : Nat → Type where
| zero : Even Nat.zero
| succ : {n : Nat} → Odd n → Even (Nat.succ n)
inductive Odd : Nat → Type where
| succ : {n : Nat} → Even n → Odd (Nat.succ n)
end
-- Even.rec :
--   {mtv_e : (a : Nat) → Even a → Sort u} → {mtv_o : (a : Nat) → Odd a → Sort u} →
--   mtv_e Nat.zero Even.zero →
--   ({n : Nat} → (a : Odd n) → mtv_o n a → mtv_e (Nat.succ n) (Even.succ n)) →
--   ({n : Nat} → (a : Even n) → mtv_e n a → mtv_o (Nat.succ n) (Odd.succ n)) →
--   {a : Nat} → (t : Even a) → mtv_e a t
#print Even.rec
-- Odd.rec :
--   {mtv_e : (a : Nat) → Even a → Sort u} → {mtv_o : (a : Nat) → Odd a → Sort u} →
--   mtv_e Nat.zero Even.zero →
--   ((n : Nat) → (a : Odd n) → mtv_o n a → mtv_e (Nat.succ n) (Even.succ n)) →
--   ((n : Nat) → (a : Even n) → mtv_e n a → mtv_o (Nat.succ n) (Odd.succ n)) →
--   {a : Nat} → (t : Odd a) → mtv_o a t
```

```
#print Odd.rec

-- recursor rules
example : Even.rec z succ_e succ_o Even.zero = z := rfl
example : Even.rec z succ_e succ_o (Even.succ n)
           = succ_e n (Odd.rec z succ_e succ_o n) := rfl
example : Odd.rec z succ_e succ_o (Odd.succ n)
           = succ_o n (Even.rec z succ_e succ_o n) := rfl
```

The generation of these rules is a bit more complex than the generation of the rules for normal inductive types. Fortunately, however, we again do not have to account for the particular rules governing the generation of recursor rules for mutual inductive types, as these are precomputed for us just the same in the `RecursorRule.rhs` field, which we use to directly encode a corresponding Dedukti rewrite rule. The only special consideration we have to make in implementing our translation is that these rewrite rules are added only after we have declared *all* of the recursor types in the mutual inductive block, which is necessary due to their mutual references.

K-Like Reduction

Recall the rule for K-like reduction in Lean:

$$\frac{K \text{ K-like} \quad \Delta \vdash K.\texttt{mk}\ p_1\ \ldots\ p_n : \texttt{K}\ p_1\ \ldots\ p_n\ i_1\ \ldots\ i_m \quad \Delta \vdash t : K.\texttt{mk}\ p_1\ \ldots\ p_n\ i_1\ \ldots\ i_m}{\Delta \vdash t \rightsquigarrow K.\texttt{mk}\ p_1\ \ldots\ p_n} \text{ [KLR]}$$

For similar reasons as were described in Section 4.1.3 in the context of struct-η, we cannot hope to directly turn this reduction rule into a Dedukti rewrite rule. The LHS would have to be a variable that we match on based on its type, which is something that Dedukti is not capable of.

In order to get around this limitation, we might reconsider some aspects of our previous idea regarding proof erasure from Section 4.1.2, making use of an `Prf` symbol that annotates proof terms with their type (analogous to the `Erase` symbol):

$$\texttt{def Prf}\ :\ (P : U\ \texttt{z})\ \to \epsilon\ P \to \epsilon\ P.$$

We can then be sure to have the translation always apply a `Prf` annotation to every translated proof term. We would also want to define rewrite rules removing this annotation when we encounter a constructor of an inductive predicate, in order to allow the recursor reduction rewrite rules described in Section 4.2 to go through. For instance, in the case of the translation of the propositional `Or` type in Lean:

```
inductive Or (P Q : Prop) : Prop where
| inl (h : P) : Or P Q
| inr (h : Q) : Or P Q
```

we would define the following rewrite rules:

$$[P, Q, p]\ \texttt{Prf}\ (\texttt{Or}\ P\ Q)\ (\texttt{Or.inl}\ p) \hookrightarrow \texttt{Or.inl}\ p.$$
$$[P, Q, q]\ \texttt{Prf}\ (\texttt{Or}\ P\ Q)\ (\texttt{Or.inr}\ q) \hookrightarrow \texttt{Or.inr}\ q.$$

Now, to encode the above rule, we could also define a rewrite rule on `Prf` taking terms of a K-like inductive type with the expected indices to the unique constructor. For instance, for the type `Eq`:

$$[A, a]\ \texttt{Prf}\ (\texttt{Eq}\ A\ a\ a)\ \_ \hookrightarrow \texttt{Eq.refl}\ A\ a.$$

While this approach works perfectly fine for this example, some difficulty arises when we think about the more general case.

Suppose we have the following K-like inductive type:

```
inductive K (n : Nat) : Nat → Prop
| mk : K n (Nat.add n n)
-- K.rec.{u} : {n : Nat} → {motive : (a : Nat) → K n a → Sort u} →
--             motive (n.add n) (K.mk n) → {a : Nat} → (t : K n a) → motive a t
#print K.rec
```

K-like reduction enables the recursor reduction to go through in the case of a generic parameter `n`:

```
theorem KLR_ex (k : K n (Nat.add n n)) : K.rec true k = true := rfl
```

It also enables the reduction in the case of a particular instantiation of the parameter `n`:

```
theorem KLR_ex_inst (k : K 1 2) : K.rec true k = true := rfl
```

However, how exactly do we go about defining a rewrite rule `Prf` in this case? Our first attempt may be to encode it directly as follows:

$$[n]\ \texttt{Prf}\ (\texttt{K}\ n\ (\texttt{Nat.add}\ n\ n))\ \_\ \hookrightarrow\ \texttt{K.mk}\ n.$$

However, the fact that the index of the output type of the unique constructor is a particular function of the parameters is problematic. Specifically, here it results in non-confluence, as we have the following critical pair:

$$\texttt{Prf}\ (\texttt{K Nat.zero}\ (\texttt{Nat.add Nat.zero Nat.zero}))\ \_ \overset{\texttt{Prf}}{\hookrightarrow} \texttt{K.mk Nat.zero}.$$
$$\texttt{Prf}\ (\texttt{K Nat.zero}\ (\texttt{Nat.add Nat.zero Nat.zero}))\ \_ \overset{\texttt{Nat.add}}{\hookrightarrow} \texttt{Prf}\ (\texttt{K Nat.zero Nat.zero})\ \_.$$

Unfortunately, this critical pair in unjoinable, because in the second term where we immediately reduce the `Nat.add` application, the rewrite rule for `Prf` cannot apply, as it requires to match on a `Nat.add` application in the second argument to `K`. It would seem that for this strategy to work, we would necessarily have to define a *conditional* rewrite rule along the lines of the following:

$$[n, m]\ \texttt{Prf}\ (\texttt{K}\ n\ m)\ \_\ \hookrightarrow\ \texttt{K.mk}\ n \mid \text{IF}\ m \equiv \texttt{Nat.add}\ n\ n.$$

This hypothetical rewrite rule would only apply if Dedukti is able to unify m with `Nat.add` n n, thus ensuring subject reduction without the syntactic restriction of a `Nat.add` application on the LHS.

However, Dedukti offers no way to define such conditional rewrite rules. Even if we could define such a rule in Dedukti, it would complicate proof export as we would have to assume that any target system also implements a similar conditional rewrite rule. What's more, conditional rewriting theory is not nearly as developed as standard rewriting theory, so it would be harder to show theoretical results about our translation, like termination and confluence (for which some automated tools exist, but are implemented for stardard rewriting theory).

Recall, however, that [KLR] is only relevant for recursor reduction with K-like inductive types, so it is in fact equivalent to suppose the following reduction rule on recursor applications of K-like inductive types:

$$\frac{K \text{ K-like} \quad \Delta \vdash K.\texttt{mk}\ p_1\ \ldots\ p_n : K\ p_1\ \ldots\ p_n\ i_1\ \ldots\ i_m \quad \Delta \vdash t : \texttt{K}\ p_1\ \ldots\ p_n\ i_1\ \ldots\ i_m}{\Delta \vdash K.\texttt{rec}\ p_1\ \ldots\ p_n\ \_\ f\ i_1\ \ldots\ i_m\ t \rightsquigarrow f}\ \text{[KLR-REC]}$$

So perhaps, rather than taking the approach of proof annotations, could we instead attempt to encode the typing requirement directly into the recursor reduction rule instead? In the case of `K`, we might end up with a rule like this:

$$[\ldots]\ \texttt{K.rec}\ n\ \_\ f\ (\texttt{Nat.add}\ n\ n)\ \_\ \hookrightarrow\ f(\texttt{K.mk}\ n).$$

Unfortunately, however, we have to again match on $\texttt{Nat.add}\ n\ n$ in the LHS to ensure we respect the requirements of [KLR-REC], which leads to the same non-confluence issues as earlier.

The central issue we are contending with here is the need to turn a typing requirement into a syntactic one, something that seemingly cannot be avoided in the absence of conditional rewriting. Therefore, as we did in the case of proof irrelevance, we choose to exclude the K-like reduction rule from the Lean$^-$ theory, leaving it to also be eliminated via a pre-translation step (see Chapter 5).

Struct-Like Reduction

Recall rule for struct-like reduction in Lean:

$$\frac{S\ \text{struct-like} \quad \Delta \vdash t : S\ p_1\ \ldots\ p_n}{\Delta \vdash t \rightsquigarrow S.\texttt{mk}\ p_1\ \ldots\ p_n\ t.1\ \ldots\ t.m}\ \text{[R-ETA-S]}$$

Compared to [KLR], one key difference to note is that the rule does not place any particular requirements on the type of the subject term t, other than that it must be a structure instance. As structures lack indices, the RHS construction always has the same type as the LHS, so this rule is guaranteed to statisfy subject-reduction without any extra premise conditions. This gives us some hope that it may in fact be possible to encode [R-ETA-S] as a rewrite rule in Dedukti. We again cannot expect to be able to directly encode such a reduction rule as a Dedukti rewrite rule, on account of the lack of a defined head symbol on the LHS. However, some of our scrapped ideas from our attempt at encoding [KLR] may now prove to be useful.

Recall that as with [KLR], this rule is only really relevant for the reduction of recursor applications, making it sufficient for us to satisfy Reduction-Soundness w.r.t. the following reduction rule:

$$\frac{S\ \text{struct-like} \quad \Delta \vdash t : S\ p_1\ \ldots\ p_n}{\Delta \vdash S.\texttt{rec}\ p_1\ \ldots\ p_n\ \_\ f\ t \rightsquigarrow f\ t.1\ \ldots\ t.m}\ \text{[R-ETA-S-REC]}$$

We can again attempt to define a rewrite rule on the recursor application itself based on the above reduction step. In the case of [KLR], we were limited by the functional dependence of the index arguments on the parameters, which resulted in patterns on the LHS that could result in non-confluence. This isn't an issue the case of structure types, which lack indices. Therefore, we can declare the following rewrite rule as a direct encoding of the above rule:

$$[f, t]\ S.\texttt{rec}\ p_1\ \ldots\ p_n\ \_\ f\ t\ \hookrightarrow\ f\ (\texttt{prj}^1_S\ t)\ \ldots\ (\texttt{prj}^m_S\ t).$$

Note that this rule makes the originally defined rewrite rule for recursor reduction on S redundant:

$$[f, t]\ S.\texttt{rec}\ p_1\ \ldots\ p_n\ \_\ f\ (S.\texttt{mk}\ a_1\ \ldots\ a_m)\ \hookrightarrow\ f\ a_1\ \ldots\ a_m.$$

This is because, in the event that we have a constructor application $S.\texttt{mk}\ a_1\ \ldots\ a_m$ as a major premise, the new rewrite rule encoding [R-ETA-S-REC] is able to match, rewriting to $f\ (\texttt{prj}^1_S\ (S.\texttt{mk}\ a_1\ \ldots\ a_m))$ with the rewrite rules defined on struct projections then able to take effect to finally rewrite to $f\ a_1\ \ldots\ a_m$, the equivalent of the RHS of the original recursor reduction rewrite rule. As such, our translation can override the usual generation of recursor rewrite rules on structure types to instead generate rules based on [R-ETA-S-REC], as described above.

Generating the rewrite rules in this way also recovers confluence with the rewrite rule for struct-η (recall the non-confluence issue with our struct-η encoding described earlier in Section 4.1.3). This is because it does not require an explicit construction in the major premise, so the fact that the struct-η rewrite rule possibly rewrites constructor applications to non-constructor applications is no longer a problem.

Recall the reduction rules associated with Lean's quotient types:

$$\frac{\Delta = (\Sigma; \Gamma) \quad \Sigma \text{ quotient-declared}}{\Delta \vdash \texttt{@Quot.ind}\ A\ r\ B\ p\ (\texttt{@Quot.mk}\ A\ r\ a) \rightsquigarrow p\ a} \text{ [QIND]}$$

$$\frac{\Delta = (\Sigma; \Gamma) \quad \Sigma \text{ quotient-declared}}{\Delta \vdash \texttt{@Quot.lift}\ A\ r\ B\ f\ h\ (\texttt{@Quot.mk}\ A\ r\ a) \rightsquigarrow f\ a} \text{ [QLIFT]}$$

Both rules can be easily translated to Dedukti rewrite rules that closely resemble those that would be generated for recursor reduction with ordinary inductive types. In terms of the implementation, we generate these rules, and the associated symbol declarations for quotient types, when we encounter a quotient declaration (the constructor `Lean.ConstantInfo.quotInfo` in the metaprogramming framework). The rules generated for `Quot.ind` and `Quot.lift` are as follows:

$$[p, a]\ \texttt{Quot.ind}\ \_\ \_\ \_\ \_\ p\ (\texttt{Quot.mk}\ \_\ \_\ \_\ a) \hookrightarrow p\ a.$$
$$[f, a]\ \texttt{Quot.lift}\ \_\ \_\ \_\ \_\ f\ \_\ (\texttt{Quot.mk}\ \_\ \_\ \_\ a) \hookrightarrow f\ a.$$

Chapter 5

Designing a Preliminary Translation

At this point, we have fully specified the theory Lean$^-$, a maximal sub-theory of Lean that is amenable to direct translation to Dedukti (given an adequate encoding of Lean$^-$ in Dedukti). In Lean$^-$, we have done away with the rules [PI], [KLR], and [UNIT], which we have observed to cause particular difficulties for defining a Dedukti encoding. We have been able to define the translation $|\cdot|^{\hookrightarrow}_{-}$, which respects the desired properties of soundness and completeness.

This translation can likely already take us quite far in terms of translating substantial parts of Lean's standard library, and even some portions of Mathlib. But we will be limited to translating very low-level concepts; in particular, we will be limited to constructive logic, as Lean's proof of the excluded middle property makes essential use of proof irrelevance[1]. The use of proof irrelevance and K-like reduction pervades many other parts of the standard library as well (unit-η is also used in a number of places, but not nearly to a similar extent). Therefore, if we are aiming to translate the full transitive closure of the dependencies of every constant (without introducing any new axioms in our translation), many higher-level formalizations in the standard library and Mathlib would be out of reach, because it is very likely that they would depend on some constant that uses an eliminated feature.

So, to have a full translation, we are left with the question of whether it is possible to define a *preliminary translation* from Lean to Lean$^-$, which we can compose with our previously defined translation from Lean$^-$ to Dedukti to have a complete translation from Lean to Dedukti. Let's use the notation $|\cdot|^-$ for our hypothetical preliminary translation from Lean to Lean$^-$, using the same notation for context translation. If such a translation is possible, respecting the properties of soundness and completeness, we can achieve a full translation to Dedukti by defining it as the following composition:

$$|t|^{\hookrightarrow} := ||t|^-|^{\hookrightarrow}_-$$

We also aim to define the translation $|\cdot|^-$ such that Lean types are translated to Lean$^-$ types, meaning that our type-level translation would be defined as:

$$\|t\|^{\hookrightarrow} := \||t|^-\|^{\hookrightarrow}_-$$

Soundness and completeness of this composite translation would follow from the soundness and completeness of the individual translations $|\cdot|^-$, $|\cdot|^{\hookrightarrow}_-$, and $\|\cdot\|^{\hookrightarrow}_-$. So, the question is: does such a translation $|\cdot|^-$ exist, and if so, what does it look like and how can we implement it?

[1]The proof, which can be found in the standard library file `Classical.lean`, is based on Diaconescu's theorem, which utilizes the axioms of choice and propositional extensionality, and the theorem of functional extensionality (whose proof makes use of quotient constructions).

5.1 Theoretical Background

We want our translation $|\cdot|^-$ to respect the following soundness property:

Theorem 5.1.1 (Lean-to-Lean$^-$ Soundness)**.** For all contexts Δ and terms t, T such that $\Delta \vdash t : T$, we have $|\Delta|^- \vdash^- |t|^- : |T|^-$.

Note that this soundness property looks a bit different from the soundness properties that we expressed w.r.t. the translations $|\cdot|^{\hookrightarrow}$ (Property 2.1.4) and $|\cdot|^{\hookrightarrow}_{\Sigma}$ (Theorem 2.3.1), because we do not make use of a separate type-level translation from Lean to Lean$^-$ (as we have done before in defining the type-level translations $\|\cdot\|^{\hookrightarrow}_{\Sigma}$ and $\|\cdot\|^{\hookrightarrow}$). Specifically, the above statement implicitly requires that that $|\Delta|^- \vdash^- |T|^- : \texttt{Sort}\ \ell$ (necessary for the conclusion $|\Delta|^- \vdash^- |t|^- : |T|^-$ to be well-typed), which we assume to follow from $\Delta \vdash T : \texttt{Sort}\ \ell$.

We also seek to satisfy the following completeness property:

Theorem 5.1.2 (Completeness)**.** For all contexts Δ and terms T such that $\Delta \vdash T : \texttt{Sort u}$, if there exists some term t such that $|\Delta|^- \vdash^- t : |T|^-$, then there exists some term t' such that $\Delta \vdash t' : T$.

Observing these desired properties, we can speculate a bit on what their proofs might look like, and in doing so perhaps derive some intuition regarding the possible form of our translation $|\cdot|^-$. Recall that with our Lean$^-$-to-Dedukti translation, its correctness heavily relied on the translation of the typing context, where we were able to encode a number of the particular definitional equalities from Lean$^-$ as rewrite rules in Dedukti. This allowed the original Lean$^-$ typing derivation to have a corresponding derivation in Dedukti, with the term-level translation being essentially a one-to-one mapping from Lean$^-$ syntax to Dedukti syntax.

In contrast, the task of translating from Lean to Lean$^-$ involves translating from a larger theory to a strictly smaller one. Lean$^-$ offers no additional features over Lean that we can take advantage of in our translation. As such, a lot more of the "burden" of maintaining these translation properties will likely lie on the translation of the terms themselves, rather than some particular encoding. In a similar way to how implicit definitional equalities in Lean$^-$ became explicit in the form of rewrite rules when translating the typing context to Dedukti, we can expect that the implicit rules of [PI], [KLR], and [UNIT] will become explicit in translating from Lean to Lean$^-$, but in the term-level translation, rather than in the context-level translation.

5.1.1 Comparing Theories

Before we start trying to actually define our translation, let's take a closer look at our theories and try to establish some relative properties about them. Doing so could be a useful "sanity check", helping to ensure that our translation goal is feasible in the first place, and possibly helping us gain some additional intuition on what our translation should look like.

Firstly, we know that it is the case that Lean$^-$ $\subset$ Lean – this follows from the fact that Lean$^-$'s definitional equality relation uses a subset of the definitional equality rules in Lean, where the missing rules cannot be reconstructed from the other rules (Lean's theory contains no redundant rules). However, this does not tell us a whole lot about the feasibility of a translation – roughly speaking, the fact that a term is no longer typeable in the smaller theory does not mean that there isn't an "equivalent" term in a smaller theory with similar properties, that we can try to align our translation towards. Indeed, the subset relation works in the opposite direction to what we would like to show: it says that any term typeable in Lean$^-$ is also typeable in Lean, whereas we would like to say something about the inhabitance of types in Lean$^-$ given the inhabitance of corresponding types in Lean. So, let's try to more deeply analyze our respective theories to better develop some notion of translation feasibility.

Conservativity of Lean over Lean$^-$

Intuitively, for Lean$^-$ to be a good target for translation from Lean, Lean$^-$ should not be strictly less expressive than Lean in terms of the provable propositions. Any proposition provable in Lean should also be provable in Lean$^-$ (the opposite always holds since Lean$^-$ $\subseteq$ Lean) – this needs to be the case if we want to have any hope in defining a correct translation. More generally, the above should apply to any type, replacing the notion of provability with inhabitance.

When we talk about types, though, it's important to make a distinction between types only expressible in Lean and types expressible in both Lean and Lean$^-$. Some types are expressible exclusively within Lean; for instance, the type `T Qq` below requires the use of proof irrelevance in order to be well-typed in Lean:

```
variable (P : Prop) (p q : P) (Q : P -> Prop) (Qq : Q q) (T : Q p -> Type)
axiom trickyType : T Qq -- need `Q q` to be defeq to `Q p` for `T Qq : Type`
```

It's important to consider these types as well when considering the relative expressivities of our theories. But, how exactly can we express the inhabitance of `T Qq` in Lean$^-$ when it is not even well-typed in Lean$^-$ to begin with?

For the sake of simplicitly, let's brush aside this question for the moment and only consider the relative expressivity of the theories w.r.t. types that are *already* also well-typed in Lean$^-$. Let's state a preliminary requirement: if we have some type T that is well-typed in Lean$^-$, T being inhabited in Lean should imply that it is also inhabited in Lean$^-$. This corresponds to the well-known condition of "conservativity" of one theory over another, and can be stated explicitly as follows:

Conjecture (conservativity)**.** We say that Lean is "conservative over" Lean$^-$ if for all contexts Δ and terms T such that $\Delta \vdash^- T :$ `Sort u` and $\vdash^-$ Δ well-formed, if there exists some term t such that $\Delta \vdash t : T$, then there exists some term t' such that $\Delta \vdash^- t' : T$.

Conservativity is a relatively standard notion in logic. It is typically used in reference to theories that can be seen as extensions of other theories, where showing conservativity of the extended theory over the original one can serve as a justification of the fact that the extension is "safe" – in particular, that it does not introduce any possible inconsistencies by expanding the class of provable propositions to include the proposition `False`.

So, we would like to demonstrate that Lean is a conservative extension of Lean$^-$. If this turns out not to be the case, then we will need to adjust our translation in order to somehow "compensate" for the inherent lack of expressivity in Lean$^-$. Let's start by analyzing the specific definitional equalities that we have eliminated from Lean, and try to see where we might run into trouble.

Suppose we have some term $\Delta \vdash^- T :$ `Type` and $\Delta \vdash t : T$, but with $\Delta \nvdash^- t : T$. This would require that one of the eliminated definitional equalities – [PI], [KLR], or [UNIT] – was used in an "essential" way in the typing of t in Lean. If we can find minimal examples of terms t and T that satisfy these conditions, we can assess the inhabitance of T in Lean$^-$ to determine whether Lean is truly conservative over Lean$^-$[2].

Useful here is the equality inductive type, defined in Lean as the type `Eq`:

```
inductive Eq {A : Sort u} (a : A) : A → Prop where
| refl : Eq a a
```

The constructor of the equality type, `Eq.refl`, takes a parameter `a : A` that enforces an identical inductive type index in the output type `Eq a a`. This corresponds to a notion of equality since

[2]Note that this will check a *necessary* condition for conservativity, *not* a sufficient one. It may be the case that a minimal term using an eliminated definitional equality in Lean can be translated another term of an equivalent type in Lean$^-$, but with more complex terms using this same definitional equality still remaining unstranslatable.

it requires that the LHS and the RHS of the equality type are syntactically identical for any base construction.

This constructor is particularly useful to us here because it can be used as a quick check of whether or not the kernel considers two terms to be definitionally equal. Specifically, we have the following property:

Lemma 5.1.3. For all contexts Δ and terms a, b, T such that $\Delta \vdash T : \texttt{Sort u}$, $\Delta \vdash a : T$ and $\Delta \vdash b : T$,

$$\Delta \vdash a \equiv b \iff \Delta \vdash \texttt{Eq.refl}\ a : \texttt{Eq}\ a\ b.$$

Proof. In the forward direction, suppose $\Delta \vdash a \equiv b$. Then, we have the following derivation:

$$\dfrac{\Delta \vdash \texttt{Eq.refl}\ a : \texttt{Eq}\ a\ a \qquad \dfrac{\Delta \vdash \texttt{Eq}\ a \equiv \texttt{Eq}\ a \quad \Delta \vdash a \equiv b}{\Delta \vdash \texttt{Eq}\ a\ a \equiv \texttt{Eq}\ a\ b}}{\Delta \vdash \texttt{Eq.refl}\ a : \texttt{Eq}\ a\ b}$$

In the reverse direction, suppose $\Delta \vdash \texttt{Eq.refl}\ a : \texttt{Eq}\ a\ b$. Such a typing necessarily involves a use of conversion, as the type of $\texttt{Eq.refl}\ a$ is normally $\texttt{Eq}\ a\ a$. To derive a definitional equality between $\texttt{Eq}\ a\ a$, $\texttt{Eq}\ a\ b$, we would necessarily need to use application congruence (as Eq is not a defined symbol), thus requiring a derivation of $\Delta \vdash a \equiv b$. □

Now, in order to get our minimal terms, we can use Eq.refl to prove a theorem that directly encodes an eliminated definitional equality as a propositional one in the theorem statement. Let's start with unit-η, which can be encoded as a propositional equality between two terms of the same unit type:

```
inductive Unit : Type where
| mk : Unit
-- (`rfl` abbreviates `refl _`)
theorem unit_eta (a b : Unit) : Eq a b := rfl
```

Such a reflexive proof would not be valid in Lean$^-$, because without [UNIT] there is no way to establish a definitional equality between a and b. However, this theorem is still provable in Lean$^-$, where it follows from elimination on the equality type:

```
theorem unit_eta' (a b : Unit) : Eq a b :=
  -- Eq a b
  Unit.rec (motive := fun x => Eq a x) (
    -- Eq a Unit.mk
    Unit.rec (motive := fun x => Eq x Unit.mk)
      -- Eq Unit.mk Unit.mk
      rfl
      a)
    b
```

Therefore, the lack of [UNIT] in Lean$^-$ does not present any immediate concerns regarding the conservativity of Lean over Lean$^-$.

The situation is similar for K-like reduction. Consider again the example K-like inductive type K:

```
inductive K : Bool → Prop where
| mk : K Bool.true
```

We can characterize the rule [KLR] w.r.t. `K` with the following theorem:

```
theorem klr (k : K Bool.true) : Eq k K.mk := rfl
```

Again, this proof is no longer valid in Lean$^-$, which lacks [KLR], so we would like to see if it is still at least provable as a propositional equality. While K-like inductive types are not always technically unit types (as unit types cannot carry indices), we can use the eliminator in much the same way as we did for unit types, when we have a term that is well-typed as the unique constructor of the K-like inductive type. Together with the use of the equality inductive type `Eq`k its heterogeneous counterpart `HEq`, and the equality substitution principle `Eq.subst`, we can prove this theorem in Lean$^-$ through elimination on K as follows;

```
theorem klr' (k : K Bool.true) : Eq k K.mk
  eq_of_heq (K.rec (motive := fun _ k' => HEq k' K.mk) HEq.rfl k)
```

Notably, this proof also does not require the use of [PI]. Recall, however, that the only place where the use of [KLR] really matters is in recursor reduction on K-like inductive types. Let's also verify that recursor reduction involving [KLR] remains provable as a propositional equality in Lean$^-$. In Lean, we can characterize this with the following theorem:

```
theorem klr_rec {motive : (a : Bool) → K a → Sort u}
   (m : motive Bool.true K.mk) (k : K Bool.true) :
   Eq (K.rec (motive := motive) m k) m := rfl
```

In Lean$^-$, using the theorem `klr'`, we can also prove this property:

```
theorem klr_rec' {motive : (a : Bool) → K a → Sort u}
   (m : motive Bool.true K.mk) (k : K Bool.true) :
   Eq (K.rec (motive := motive) m k) m :=
  have : Eq K.mk k :=
    Eq.symm (klr' k)
  have : HEq (K.rec (motive := motive) m k) m :=
    Eq.subst
      (motive := fun k' => HEq (K.rec (motive := motive) m k') m)
      this HEq.rfl
  eq_of_heq this
```

Since the use of [KLR] in recursor reduction is provable as a propositional equality in Lean$^-$, we also do not have any immediate reason to suspect that the lack of [KLR] in Lean$^-$ leads to any possible problems in showing the conservativity of Lean over Lean$^-$.

Let's now consider the case of general proof irrelevance. We can encode this as an equality in a similar fashion, asserting the equality of two elements of the same propositional type. Thanks to [PI], this can also be proven reflexively in Lean:

```
theorem prfIrrelThm (P : Prop) (p q : P) : Eq p q := rfl
```

This proof is again no longer valid in Lean$^-$, which lacks [PI]. What's more, in this case, we cannot attempt to do any kind of elimination as we did in the unit-η case: this definition is polymorphic in the propositional type `P`, which we know nothing about other than the fact that it lives in `Prop`.

In fact, it would seem to be the case that it is in fact *not* possible to prove the equality between two proofs of the same propositional type in a theory that does not include [PI]. [PI] was, indeed, the only aspect of Lean accounting for any base notion of proof irrelevance – it is not assumed anywhere else in the theory, except in the [KLR] rule, which doesn't help us here either – `P` is not necessarily a K-like inductive type, and [KLR] is not present in Lean$^-$ anyways. So, it seems to

be the case that Lean is *not a conservative extension* of Lean$^-$: the type of `prfIrrelThm` above is well-typed in Lean$^-$, but not inhabited[3], meaning that we have lost some degree of expressivity in eliminating the proof irrelevance rule from our theory.

The Context-Level Translation

So, we are faced with the sad reality that Lean is not conservative over Lean$^-$. How can we adjust our translation to address this? Well, fortunately, the form of our translation also allows us to adjust the typing context Δ, which means that we can probably introduce axioms to the constant context Γ as needed to recover some expressivity in our terms. Obviously, however, we don't want to go *too* far in doing this, e.g. introducing axioms that make it possible to prove `False` – ensuring that our translation also respects the completeness property should help prevent this.

Firstly, let's define our translation of constant contexts $|\Sigma|^-$ and local contexts $|\Gamma|^-$ as usual, simply applying the object-level translation to the types and values found in the constant and bound variable contexts (with the universe level parameter context remaining unchanged). We will want to determine the set of prefix constants Σ' to define a context-level translation of the following form:

$$|(\Sigma; (\Gamma_U; \Gamma_B))|^- := (\Sigma', |\Sigma|^-; (\Gamma_U; |\Gamma_B|^-)).$$

We want to choose Σ' such that this typing context, in light of Lean$^-$'s type theory, is "sufficiently expressive" to enable us to possibly define a term-level translation respecting our desired properties.

In particular, as we have seen above, the absence of [PI] in Lean$^-$ results in some possible difficulties in deriving a sound and complete translation, so we will probably want to add an axiom to Σ' that enables us to prove `prfIrrelThm`. This axiom should essentially mirror the proof irrelevance rule [PI], converting its premises to hypotheses and its conclusion to the output type. The output type should also be a propositional equality, rather than a definitional one[4]. This actually exactly corresponds to the type of the theorem `prfIrrelThm` above, which we purposely chose to be a minimal propositional representation of [PI]. In our theory Lean$^-$, we will have to turn it into an axiom, as it is no longer provable without [PI][5]:

```
axiom prfIrrel (P : Prop) (p q : P) : Eq p q
```

We can now tentatively define $\Sigma' = [\,\texttt{prfIrrel}\,]$. We can be reasonably confident that introducing this axiom in our translation does not cause any inconsistencies that would be problematic for showing soundness and completeness: since `prfIrrel` was already a provable proposition in Lean, we know that assuming it as an axiom in Lean$^-$ results in a theory that is at least as consistent as Lean.

The question remains of how we could actually *utilize* this axiom to effect our translation, and whether the axiom `prfIrrel` alone is truly sufficient for a valid translation. For now, however, we will be content in the knowledge that with this axiom included by our translation, we at least stand some chance of being able to define a correct translation.

---

[3] That is, without assuming the axiom `propext` (and all of its dependencies) in our typing context – recall from Section 1.2.2 that propositional extensionality allows us to prove proof irrelevance as a propositional equality in the absence of [PI].

[4] Lean provides no way to designate new definitional equalities via definitions, so propositional equality is the best representation we have.

[5] Alternatively, because `prfIrrel` is derivable from `propext` (as shown in Section 1.2.2), we could instead assume the presence of `propext` axiom and all of its dependencies in our smaller theory. For simplicity, however, we chose to take `prfIrrel` to be an axiom for the remainder of this work.

Lean$^-$: An "Adequate" Target Subtheory?

Analyzing the conservativity of Lean over Lean$^-$ has proven to be useful in helping us refine part of our translation. However, we are still not entirely sure of the feasibility of defining a translation. In fact, we have been doing things somewhat backwards w.r.t. convention: the conservativity property is often shown to establish that a particular extension of a theory does not allow you to prove things that you would not have been able to in the original theory, often with the goal of showing that the extended theory maintains any consistency properties that can be shown to hold in the original theory. We have kind of been trying to use it in the reverse direction, to make sure that a smaller theory can inhabit every type that is expressible an extended theory. We showed that this was in fact not the case, and showed a way to extend our translation of the constant context with an axiom to hopefully get it "up to par" with the expressibility of the extended theory without making any actual changes to the theory itself.

Indeed, as we hinted at earlier, conservativity does not fully capture a notion of translation feasibility as we would like to express it – one of the premises of conservativity is that the inhabited type is already a valid type in Lean$^-$, when of course that is not always the case (as demonstrated in the example `trickyType` above). When it comes to defining a practical translation, we need to consider these types as well. Recall in particular the soundness and completeness properties mentioned earlier – analyzing whether or not we can possibly satisfy these properties in a translation to our target theory can help us possibly refine the theory further, and also get at some requirements that may help guide us in defining our translation.

Precisely, suppose we have some $\Delta \vdash T : \texttt{Sort}\ \ell$ and $\Delta \vdash t : T$ with $\Delta \not\vdash^- T : \texttt{Sort}\ \ell$. Going along the lines of standard conservativity, we would like to conclude that there is some $\Delta \vdash^- t' : T$, but this is not possible here: the derivation cannot exist because T is not a type in Lean$^-$. So, we need to say something a bit weaker. Rather than asserting that we have some inhabitant of T itself, we would perhaps like to instead talk about the inhabitance of some alternative type T' and context Δ' that does satisfy $(\texttt{prfIrrel}, \Delta') \vdash^- T' : \texttt{Sort}\ \ell$, and which is related to T in some principled way. One obvious possible target relation here could be propositional equality: T' should, at the very least, be *propositionally equal* to T in Lean (it cannot possibly be in Lean$^-$, where the equality type cannot be stated as T is not typable). Additionally, the context Δ should be propositionally equal to the context Δ' in the sense of all of the types and values found in Δ being provably equal to the corresponding types and values found in Δ'. Given a sequence of proof terms $\boldsymbol{p}$, we express the fact that these are well-typed as equality proofs between the corresponding types in Δ and Δ' as follows:

$$\Delta \vdash^- \boldsymbol{p} : \Delta =_T \Delta'$$

In full, we would like to show the following "adequacy" property:

Conjecture (Adequacy-α)**.** For all typing contexts Δ and terms T such that $\Delta \vdash T : \texttt{Sort}\ \ell$, there exists some term T' and context Δ' such that $(\texttt{prfIrrel}, \Delta') \vdash^- T' : \texttt{Sort}\ \ell$ and some terms $p, \boldsymbol{p}$ such that $\Delta \vdash p : T = T'$ and $\Delta \vdash^- \boldsymbol{p} : \Delta =_T \Delta'$. Additionally, if there exists some term t such that $\Delta \vdash t : T$ then there exists some term t' such that $(\texttt{prfIrrel}, \Delta') \vdash^- t' : T'$.

This is closer to the notion that we want here because it leaves the door open for a complete translation. This can be shown in the following lemma:

Lemma. For all typing contexts Δ, Δ' and terms T, T', and $p, \boldsymbol{p}$, such that $\Delta \vdash T : \texttt{Sort}\ \ell$, $(\texttt{prfIrrel}, \Delta') \vdash^- T' : \texttt{Sort}\ \ell$, $\Delta \vdash p : T = T'$, and $\Delta \vdash^- \boldsymbol{p} : \Delta =_T \Delta'$ (that is, Adequacy-α holds), if there exists some term t' such that $(\texttt{prfIrrel}, \Delta') \vdash^- t' : T'$, then there exists some $\Delta \vdash t : T$.

Proof. Let t'_{rfl} be t' with all occurrences of `prfIrrel` replaced with the term `fun _ _ _ => rfl`. Because Lean$^-$'s typing is a subset of Lean's, we know that $\Delta \vdash t'_{\text{rfl}} : T'$. Then, we have $\Delta \vdash$ `cast` $T'\ T\ p\ t'' : T$. □

In the above proof, we make use of the `cast` operation, which has the following type signature:

```
def cast {A B : Sort u} (h : A = B) (a : A) : B := ...
```

In the literature, `cast` is often referred to as a "type transport" operator, which effects a kind of "explicit conversion" between two provably equal types. As we will see, the use of the `cast` operation will be crucial in defining our translation from Lean to Lean$^-$.

If our translation $|T|^-$ on any type T correctly outputs a term T' satisfying the conditions described by the Adequacy-α property, then the translation respects the completeness property. Similarly, the feasibility of soundness in a translation can be shown, assuming that the translation $|t|^-$ of some inhabitant $\Delta \vdash t : T$ also correctly produces a term t' respecting $(\texttt{prfIrrel}, \Delta') \vdash^- t' : T$, corresponding to Adequacy-α's inhabitation clause.

This notion, while useful for ensuring translatability of proof terms under equivalent propositional types in the target theory, is perhaps not as strong as we would like, however. The requirement of propositional equality between T and T' in the source theory roughly ensures that we can define a translation that preserves the "meaning" (i.e. semantic interpretation) of the type T, but what about terms that are not types? How, for instance, can we be sure that the term `Nat.succ Nat.zero` preserves its interpretation as the natural number one when we translate it to Lean$^-$?

If we are only concerned about non-type terms that correspond to proofs, there isn't anything we need to pay particular attention to here. Due to Lean's small elimination principle, it is not important to preserve the particular structural makeup of proof terms in a translation – a proof can always be substituted with any other proof with no effect on any subsequent computations. However, this is of course not the case for non-proof terms, which can be compared on the basis of their internal makeup, and eliminated upon to define transformations based on the actual "contents" of the term (in the case of the term being reducible to a constructor application). For these reasons, we would also like the semantic content of such terms to be preserved by our translation.

Therefore, we would like to generalize the Adequacy-α property even further – specifically, we want to be able to propositionally equate not just types with their hypothetical images in our target theory, but terms as well. However, while we were able to use standard propositional equality in the Adequacy-α property to state the equality between the types T and T', it doesn't quite make the cut for the property we would like to state here. Recall the type of the equality inductive type constructor:

```
-- Eq.{u} {A : Sort u} : A → A → Prop
#check Eq
```

To state an equality, both the LHS and RHS must have definitionally equal types. The only reason we were able to use `Eq` in our statement of Adequacy-α was because the type `Sort` ℓ happens to already be well-typed in Lean$^-$, and so the equality type construction T `=` T' was well-typed (the LHS and RHS both have type `Sort` ℓ). However, for an arbitrary well-typed term t such that $\Delta \vdash t : T$, we can only surmise the existence of a term t' and type T' such that $\Delta \vdash t : T$ and T `=` T'. We would like to also assert equality between t and t', but they have the types T and T', which are not necessarily definitionally equal in Lean, so t `=` t' may not be well-typed.

It would seem that we need to work with a more general notion of equality, which allows the LHS and RHS types to differ. This exactly corresponds to Lean's definition of the heterogeneous equality type `HEq`:

```
inductive HEq : {A : Sort u} → A → {B : Sort u} → B → Prop where
| refl (a : A) : HEq a a
```

We will use the notation `a == b` for `HEq a b`. As one would expect, its constructor is limited to inhabiting `HEq` type constructions with definitionally equal LHS and RHS types and values.

Using `HEq`, we can now formulate a more general adequacy requirement. Firstly, let's establish some new notation for indicating provably equal typing contexts: given two typing contexts Δ and Δ' and a sequence of proof terms $\boldsymbol{p}$, we express the fact that these are well-typed as *heterogeneous* equality proofs between the corresponding types *and values* in the contexts as follows:

$$\Delta \vdash \boldsymbol{p} : \Delta == \Delta'$$

We can now state our more complete adequacy requirement as follows:

Conjecture (Adequacy)**.** We say that Lean$^-$ is an "adequate subtheory" of Lean if, for all typing contexts Δ and terms t, T such that $\Delta \vdash T : \texttt{Sort}\ \ell$ and $\Delta \vdash t : T$, there exists terms t', T' and context Δ' such that $(\texttt{prfIrrel}, \Delta') \vdash T' : \texttt{Sort}\ \ell$ and $(\texttt{prfIrrel}, \Delta') \vdash t' : T'$ and terms $p, q, \boldsymbol{p}$ such that $\Delta \vdash p : T\ \texttt{=}\ T'$, $\Delta \vdash q : t\ \texttt{==}\ t'$, and $\Delta \vdash \boldsymbol{p} : \Delta == \Delta'$.

Note that Adequacy implies Adequacy-α. If T is inhabited, Adequacy gives us the existence in Lean$^-$ of an inhabited and propositionally equal T'. Otherwise, we can assign $T := \texttt{Sort}\ \ell$ and $t := T$, and the existence of a propositionally equal T' follows from Adequacy, where the heterogeneous equality proof $\Delta \vdash q : T\ \texttt{==}\ T'$ can used to derive a proof of the homogeneous equality $T\ \texttt{=}\ T'$ as the LHS and RHS have the same types.

Therefore, if it is the case Lean$^-$ is an adequate subtheory of Lean, we have good reason to believe that it would be possible to define a sound and complete translation from Lean to Lean$^-$. While formalizing this property was interesting, at this point we might not be able to get much more out of analyzing whether it actually holds at a more abstract level. Let's get started with actually trying to define what our translation might look like. If we can successfully define a sound translation that maintains heterogeneous equality between the input and output terms (in Lean's theory), this would amount to a constructive proof of Adequacy, ensuring in turn that the translation satisfies the completeness property. As we will see, it is indeed possible to do such a translation, and, in fact, the translation we derive allows us to always have a *definitional* equality in Lean between the original and translated terms.

5.1.2 An Intuitive Translation Sketch

Subterm Translation

In deriving a translation respecting the soundness property, we again need to pay special attention to the conversion rule, proving soundness in the case of typing by [CONV]. In the previous chapter, we came up with the notion of "Defeq-Soundness", which we showed to be sufficient for showing this property, making for a particularly straightforward translation, given an adequate encoding of Lean$^-$'s definitional equalities as a Dedukti term rewrite system. Can we show a similar property in defining our translation from Lean to Lean$^-$?

Let's first look at exactly how far we would need to go for the equivalent of definitional equality-soundness to hold in our translation, investigating the problematic case of proof irrelevance. Consider the following theorem in Lean:

```
variable (P : Prop) (p : P) (q : P) (T : P → Prop)
theorem prfIrrelEx (t : T p) : T q := t
```

Here, we need to have a definitional equality between `T p` and `T q` in order for `t : T p` to be well-typed as `T q`. Since `T` is an irreducible function head, this boils down to the definitional equality between `p` and `q`, which holds in Lean via [PI]. For Defeq-Soundness, we would need the translation of `p` to be definitionally equal to the translation of `q` in Lean$^-$. Since [PI] is not present in Lean$^-$, this would seemingly take nothing less than *erasing* the proofs themselves. Specifically, we could imagine introducing a symbol like this:

```
axiom sorryAx (A : Sort u) : A
```

which enables any type to be inhabited. We could then "erase" the proofs by substituting them for `sorryAx P`, making them syntactically equal:

```
theorem prfIrrelExSorry (t : T (sorryAx P)) : T (sorryAx P) := t
```

Defining a translation that erases proof terms in this way might seem to work for our purposes, however the inclusion of such an axiom our in constant context via the translation obviously renders the context inconsistent, as `sorryAx` can be used to provide a proof of `False`. And being able to prove `False` spells doom for our ability to show that our translation is complete, since we would presumably want to translate `False` as `False`, and we would expect that `False` is not inhabited in Lean. Not to mention, such a translation loses any information on the structure of the original proof, eliding one of the principal motivations for our translation to Dedukti – the translation of proofs to other proof assistants for cross-checking.

So, it seems that we may need to take a step back and try a different strategy. It seems that we can no longer solely rely on instances of [CONV] to be preserved in the derivation of our translated terms if they make use of definitional equalities that no longer apply in Lean$^-$. As in the above case, when adding an extra axiom introduced inconsistencies, any attempt at altering our translation to ensure that [CONV] judgments are preserved will likely be destructive in some sense. So, our translated terms will likely have to take another derivation path. Whenever we have an instance of [CONV] in the original typing that doesn't hold in the smaller theory, we can imagine translating the subject term such that it already has the desired type without needing to invoke the problematic conversion.

How exactly can we go about this? Well, something we could try to do is to make the conversion explicit in the translated term itself, so that the desired type results directly from type inference without requiring conversion. Since the "explicit" variant of definitional equality is propositional equality, the `Eq`/`HEq` types will probably figure into our translation here. We just need a way to turn the propositional equality of types into an explicit type conversion, which exactly corresponds to the `cast` (i.e. type transport) operation that we introduced earlier.

Using this cast symbol in the translation of `prfIrrelEx`, we could obtain the following translation:

```
theorem prfIrrelExTrans (t : T p) : T q :=
  cast (T p) (T q)
    (congrArg T (prfIrrel p q))
    t
```

The use of proof irrelevance is made explicit in the proof provided to `cast`, which uses the `prfIrrel` axiom. In general, if we have a use of [CONV] in the original theory, this could signal to apply a cast to the subject term using some term p proving equality between the translations of the inferred and expected types in Lean$^-$, essentially effecting a transformation of the derivation tree as follows:

$$\dfrac{\dfrac{\dfrac{\Delta \vdash A, B : \texttt{Sort}\ \ell \quad \Delta \vdash A \equiv B \quad \Delta \vdash t : A}{\Delta \vdash t : B}}{\vdots\ |\cdot|^-}}{\dfrac{\dfrac{\dfrac{\dots}{|\Delta|^- \vdash \texttt{cast}\ |A|^-\ |B|^- : |A|^- = |B|^- \to |A|^- \to |B|^-} \quad |\Delta|^- \vdash p : |A|^- = |B|^-}{|\Delta|^- \vdash \texttt{cast}\ |A|^-\ |B|^-\ p : |A|^- \to |B|^-} \quad |\Delta|^- \vdash |t|^- : |A|^-}{|\Delta|^- \vdash \texttt{cast}\ |A|^-\ |B|^-\ p\ |t|^- : |B|^-}}$$

Defining our translation in this way is a marked departure from how we have defined translations previously, in that it depends on the actual typing derivation of a term, rather than just its syntactic structure. When defining our translation from Lean$^-$ to Dedukti, we were able to define a translation that was essentially a transformation of Lean syntax into Dedukti syntax (assuming a particular type theory encoding of Lean$^-$ in Dedukti). This translation only depended on the structure of the term itself[6], with the typing derivation only being relevant in the proofs of soundness and completeness of the translation – in particular, it would be feasible to apply the translation to certain non well-typed terms. In this case, however, knowing the details of a typing derivation is essential to our translation task, with the translation only being defined on well-typed terms. We do not know from the term t by itself that it must have a type cast applied to it in order for it to be well-typed within some larger typing context.The fact that we have to apply a cast around $|t|^-$ is a result of the use of [CONV] in the typing derivation, which in turn is a result of the typing of the term in which t originally appears (e.g. if t is the argument of a function with domain type B).

Assuming that we can always provide the type equality proof p that is required by `cast`, this would seem to be a valid translation strategy, in which we make implicit uses of type conversion in typechecking *explicit* in the translation. In effect, we convert instances of [CONV] in the Lean typing derivation to a series of [APP] instances in a Lean$^-$ typing derivation via a translation of the subject term, as shown in the derivation transformation above. There is one subtlety to consider here, however: as reflected in the type of p, the types A and B may themselves require translation if they are ill-typed in the target theory, and in general our translation will produce a cast of t from $|A|^-$ to $|B|^-$, with the resultant term having type $|B|^-$, rather than the type B. This is probably not an issue, however, since we can presume (formally by induction) that whatever term context originally required t to have type B would now require type $|t|^-$ to have type $|B|^-$ once this context is itself translated to Lean$^-$.

However, we likely do not wish to eliminate every instance of [CONV] in typing, but rather limit this translation to places where it is actually needed: wherever the definitional equality between A and B no longer holds in Lean$^-$. That is, when we have $\Delta \vdash A \equiv B$ appearing in an instance of [CONV] in the original Lean typing derivation, but $|\Delta|^- \nvdash |A|^- \equiv |B|^-$. Additionally, we may want retain uses of [CONV] in the type equality proof argument to `cast`, where we may want to use `Eq.refl` to provide proofs of equality between syntactically distinct subterms that remain definitionally equal in Lean$^-$. Truly eliminating all instances of [CONV] via our translation would also eliminate all uses of definitional equality in Lean$^-$, which would essentially amount to a translation to weak type theory (WTT) where the kernel can only identify terms on the basis of syntactic equality. Translating to WTT would be far in excess of what we need to do, as it would also require providing propositional equality axioms/lemmas for every other definitional equality in Lean (even very "basic" ones such as β-reduction and δ-reduction) to be able to build the needed `cast` equality proofs, whereas we only care about making explicit a select few definitional equalities.

We can observe a particular characteristic of this proposed translation: the output terms are essentially the same as the input terms, except that they have been "decorated" by casts between propositionally equal types. This gives us some hope that such a translation respects completeness by maintaining propositional equality between the input and output terms. Hopefully, we can heterogeneously equate cast subterms with the original subterms, and stitch everything together with congruence lemmas for the top-level proof of propositional equivalence between the original and translated terms.

[6]The only practical exception to this is the need to infer the universe level of binder domains when translating λ-expressions and Π-types, as required by our PTS encoding. Here, you could say that the typing derivation of such terms becomes relevant to our translation via the typing rules [LAM] and [ALL], which require binder types to be well-typed as some `Sort` ℓ. However, we can also easily imagine an alternate syntax for Lean where binder domains are annotated with universe levels, in which case we truly would have a purely syntactic translation.

Generating Equality Proof Terms

So, how exactly do we go about producing the type equality proofs required by the type casts injected by our translation? Thus far, we have defined our translation in terms of the type inference derivations alone, as this was sufficient for us to be able to sketch how we should translate subterms by applying type casts to them to make uses of type conversion explicit. To generate well-typed Lean$^-$ type equality proofs between terms that were originally definitionally equal in Lean, we might take a similarly inspired approach, making explicit the implicit judgments used in definitional equality derivations.

The first question that may arise here is: what equality representation should we use for our generated proofs? At first glance, the normal homogeneous equality type `Eq` seems like an obvious choice: at the application of `cast`, we provide a proof of equality between two types, which must inhabit the same type universe. Therefore, since the LHS and RHS have the same type (some `Sort` ℓ), it suffices to use the `Eq` type at the application of the `cast` operation. Indeed, it would not even make sense to consider using `HEq` here, as it is not possible to define a "heterogeneous" cast operation between two types that inhabit possibly different sorts.

However, we need to also consider the typing of equality subproofs of the main equality proof, which may relate non-type terms. For instance, recall the example `prfIrrelExTrans`. The "input" to our translation is the following definitional equality derivation[7]:

$$\dfrac{\Delta \vdash \texttt{t} : \texttt{T p} \qquad \dfrac{\Delta \vdash \texttt{T} \equiv \texttt{T} \qquad \dfrac{\Delta \vdash \texttt{P} : \texttt{Prop} \quad \Delta \vdash \texttt{p}, \texttt{q} : \texttt{P}}{\Delta \vdash \texttt{p} \equiv \texttt{q}}}{\Delta \vdash \texttt{T p} \equiv \texttt{T q}}}{\Delta \vdash \texttt{t} : \texttt{T q}}$$

Converting the derivation of $\Delta \vdash \texttt{T p} \equiv \texttt{T q}$ into an explicit proof, we obtained the expression `congrArg T (prfIrrel p q)`, where the subproof `prfIrrel p q` comes from the derivation of $\Delta \vdash p \equiv q$, where `p` and `q` are proof terms, not types.

Since `p` and `q` happen to have the same type here, we were able to use homogeneous equality again. However, in general, we may have to produce proofs of equality between terms that do not necessarily have the same type. Taking the previous example a step further, we can make the type of our predicate more complex, requiring a second argument whose type is dependent on the second argument:

```
variable (P : Prop) (Q : P → Prop) (p : P) (q : P) (Qp : Q p) (Qq : Q q)
  (U : (p : P) → Q p → Prop)
theorem prfIrrelExHeq (t : U p Qp) : U q Qq := t
```

The typing of `t` as `U p Qp` has a more complex derivation, featuring a nested use of proof irrelevance

[7] Alternatively, we could say that the input is a well-typed term t with the given typing derivation (if we want to think of our translation as a partial function on terms, rather than a total function on derivations).

in identifying `Qp` and `Qq`:

$$\dfrac{\Delta \vdash \texttt{Q p : Prop} \quad \Delta \vdash \texttt{Qp : Q p} \qquad \dfrac{\Delta \vdash \texttt{Qq : Q q} \qquad \dfrac{\Delta \vdash \texttt{Q} \equiv \texttt{Q} \qquad \dfrac{\Delta \vdash \texttt{P : Prop} \quad \Delta \vdash \texttt{p, q : P}}{\Delta \vdash \texttt{p} \equiv \texttt{q}}}{\Delta \vdash \texttt{Q p} \equiv \texttt{Q q}}}{\Delta \vdash \texttt{Qq : Q p}}}{\Delta \vdash \texttt{Qp} \equiv \texttt{Qq}}$$

This figures into a larger derivation identifying `U p Qp` and `U q Qq`:

$$\dfrac{\dfrac{\Delta \vdash \texttt{U} \equiv \texttt{U} \qquad \dfrac{\Delta \vdash \texttt{P : Prop} \quad \Delta \vdash \texttt{p, q : P}}{\Delta \vdash \texttt{p} \equiv \texttt{q}}}{\Delta \vdash \texttt{U p} \equiv \texttt{U q}} \qquad \dfrac{\ldots}{\Delta \vdash \texttt{Qp} \equiv \texttt{Qq}}}{\Delta \vdash \texttt{U p Qp} \equiv \texttt{U q Qq}}$$

In this instance of [APP], both the function heads `U p` and `U q` and arguments `Qp` and `Qq` have different types that cannot be identified in Lean$^-$: $\Delta \nvdash$ `Q p -> Prop` $\equiv$ `Q q -> Prop` since $\Delta \nvdash$ `Q p` $\equiv$ `Q q`. Therefore, in order to express a propositional equality between them, we will necessarily have to use heterogeneous equality: our translation will need to generate Lean$^-$ proofs p_1 and p_2 typed as $\Delta \vdash p_1$: `U p == U q` and $\Delta \vdash p_2$: `Qp == Qq`, and we will need to combine them into our final proof using the following heterogeneous version of the application congruence lemma:

```
theorem appHEqAB {A B : Sort u} {U : Sort v}
  (hAB : A = B)
  {f : A → U} {g : B → U} {a : A} {b : B}
  (hfg : f == g) (hab : a == b)
  : f a == g b := ...
```

The proof term p_2 will also have to use a different version of the proof irrelevance lemma, generalized to heterogeneous equality:

```
theorem prfIrrelHEqPQ {P Q : Prop} (hPQ : P = Q)
   (p : P) (q : Q) : p == q := by
  subst hPQ
  exact prfIrrelHEq _ _
```

Note in particular that this theorem requires us to provide a proof of equality between the LHS and RHS propositional types. In this case, such a proof would be constructed from the derivation of $\Delta \vdash$ `Q p` $\equiv$ `Q q` in the conversion step prior to the outer application of proof irrelevance, resulting in a subproof that we then incorporate into the full type equality proof, giving us the following translation:

```
theorem prfIrrelExHeqTrans (t : U p Qp) : U q Qq := cast
  (eq_of_heq
    -- U p Qp == U q Qq
    appHEqABUV
      -- Q p = Q q
```

```
      (congrArg Q (prfIrrel p q))
      -- U p == U q
      (congrArg U (prfIrrel p q))
      -- Qp == Qq
      (prfIrrelHEqPQ
        -- Q p = Q q
        (congrArg Q (prfIrrel p q))
        Qp Qq))
  t
```

5.1.3 A More General Translation Framework

We now have a solid intuition on how our translation will work: we will turn implicit uses of type conversion and definitional equality judgments in typing derivations into explicit uses of type transport accompanied by generated propositional equality proofs in our final translated term. But many details remain unclear, particularly regarding precisely how definitional equality derivations will be converted into heterogeneous equality proofs. Before getting too far into figuring that out for ourselves, however, let's first take a step back and try to see if we can at all generalize the task we are trying to achieve, in case there is any existing work in a similar direction that we can take advantage of.

Extensional Type Theory

Let's start by asking the question: what exactly is Lean's theory relative to Lean$^-$? Well, we can think of it as a theory where certain propositional equalities have been "promoted" to definitional ones: in particular, those corresponding to unit-η (provable via elimination on unit types), proof irrelevance (when it is assumed in the typing context), and K-like reduction (which is actually promoted to a reduction rule). What if we were to take this approach to its limit, treating *every* propositional equality as definitional? Doing so can be represented by the following rule, which is referred to as "equality reflection":

$$\frac{\Delta \vdash^-_e A : \texttt{Sort}\ \ell \qquad \Delta \vdash^-_e t, s : A \qquad \Delta \vdash^-_e \_ : t = s}{\Delta \vdash^-_e t \equiv s}\ \text{[EQ-REFL]}$$

Equality reflection is characteristic of extensional type theory (ETT), which is distinguished from intensional type theory (ITT) by the presence of the rule [EQ-REFL] promoting all propositional equalities to definitional ones. Turning *every* provable propositional equality into a definitional one is quite hard ask for a typechecker kernel – as a matter of fact, including [EQ-REFL] renders typechecking undecidable for sufficiently expressive type systems[8]. Therefore, no practical proof assistant actually implements equality reflection in its fully general form. However, ETT is still theoretically interesting as an "ideal target" that many theorem provers aspire to. For instance, the proof assistants Andromeda [7], F* [37], and Nuprl [2] have limited extensional theories that restrict the equivalent of [EQ-REFL] to some subset of provable propositional equalities.

Lean^-_e: An Extensional Theory

Now, suppose that we add the equality reflection rule to our theory to obtain a new theory "Lean^-_e", using the notation $\Delta \vdash^-_e t : T$ for the typing judgment and $\Delta \vdash^-_e a \equiv b$ for the definitional equality judgment. How does Lean^-_e relate to Lean$^-$ and Lean? Well, firstly, we can observe that Lean$^- \subset$ Lean^-_e. Every well-typed Lean$^-$ term is also well-typed in Lean^-_e, since Lean^-_e's type

[8] One can imagine encoding the question of whether a particular program halts as a propositional equality.

theory is a strict extension of Lean$^-$'s type theory. Being an extensional theory, Lean^-_e can also type more terms than Lean$^-$ can.

However, what exactly is Lean^-_e's relation to Lean itself? Lean's type theory contains certain rules in its definitional equality judgment that are not present in Lean^-_e. However, the presence of equality reflection in Lean^-_e greatly expands the space of definitional equalities, possibly to the extent of making up for the lack of the eliminated rules, in which case we would also have that Lean $\subset$ Lean^-_e. If this is the case, we may have found an interesting possible way to frame our translation: since every well-typed Lean term is also a well-typed Lean^-_e term, we can focus instead on the task of translating from Lean^-_e to Lean$^-$, which may be "easier" since we only concern ourselves with the elimination of the single rule [EQ-REFL] that is well-known to type theory, rather than multiple Lean-specific rules. So, let's try to verify that this is really the case. Formally, we wish to prove the following theorem:

Theorem 5.1.4. For all contexts Δ and all terms t, T such that $\Delta \vdash t : T$, we have $(\texttt{prfIrrel}, \Delta) \vdash^-_e t : T$.

Note that in the above theorem, we assume the presence of the axiom `prfIrrel` in the typing context of the Lean^-_e typing judgment. This is necessary because without it, proof irrelevance would no longer be definitional, as the proof irrelevance property is not actually provable without an axiom equivalent to `prfIrrel`. That is, without including `prfIrrel` (or `propext`) in the Lean^-_e typing context, we would be able to find a counterexample to the above theorem, as otherwise the theorem `prfIrrelThm` would be provable by `rfl` in Lean, but not in Lean^-_e.

Since the differences between the theories are all contained within the definitional equality relation, to prove this theorem it suffices to show that every definitional equality in Lean is also one in Lean^-_e. We can precisely state this as the theorem:

Theorem 5.1.5. For all contexts Δ and all terms t, s, if $\Delta \vdash t \equiv s$, then we have $(\texttt{prfIrrel}, \Delta) \vdash^-_e t \equiv s$.

To prove this, we proceed by induction on the size of the derivations. This means that for any subsequent proofs shown here, we can assume by the inductive hypothesis that any premise definitional equality assumed to hold in Lean must also hold in Lean$^-$.

For our proof, we have to consider in turn each of the possible rules that can appear as the root of a definitional equality derivation in Lean, and show that each one corresponds to some derivation in Lean^-_e. In the case of the root rule having only definitional equality or typing premises and being one of the rules that was retained in Lean^-_e, we can immediately show that the corresponding definitional equality is preserved in Lean^-_e using the same rule and the inductive hypothesis for the premises. This covers all of the definitional equality rules except for [RED], [UNIT], and [PI].

For the rules of [PI] and [UNIT], while both of them have been removed from Lean^-_e, they can still be encoded as propositional equalities: we assume the presence of the axiom `prfIrrel` in our context, and we have shown previously how a propositional equality corresponding to [UNIT] can be proven by elimination for all unit-like inductive types. Therefore, we can use them together with [EQ-REFL] in Lean^-_e to derive the same conclusion from the same premises.

Let's now consider the rule [RED], recalled below:

$$\frac{\Delta \vdash t \rightsquigarrow^* t' \quad \Delta \vdash t' \equiv s}{\Delta \vdash t \equiv s} \text{ [RED]}$$

This rule makes use of the head reduction relation $\rightsquigarrow$, which is specific to definitional equality and not covered by the induction hypothesis. Since Lean^-_e also features a reduction relation, we may think to proceed with the proof similarly to what we did for definitional equality, that is, by showing that any Lean reduction is also a Lean^-_e reduction:

Conjecture 5.1.1. For all contexts Δ and all terms t, s, if $\Delta \vdash t \rightsquigarrow s$, then we have $(\texttt{prfIrrel}, \Delta) \vdash^{-}_{e} t \rightsquigarrow s$.

However, this property does not actually hold. Since we have removed [KLR], any reduction involving [KLR] will no longer hold in Lean^{-}_{e}. If we want to prove this property, we would have to extend Lean^{-}_{e}'s reduction relation with an extensional reduction rule along the lines of:

$$\frac{\Delta \vdash^{-}_{e} A : \texttt{Sort u} \qquad \Delta \vdash^{-}_{e} t, s : A \qquad \Delta \vdash^{-}_{e} \_ : t = s}{\Delta \vdash t \rightsquigarrow s} \text{ [REFL-RED]}$$

Then, we would be able to simply replace instances of [KLR] with uses of this rule, using `prfIrrel` to construct the proof term.

However, we do not actually need to add such a rule to our theory. Doing so is redundant and does not actually add any extra expressivity. To see why, recall the reason why Lean includes a special reduction relation in the first place – to compensate for a lack of general transitivity in its definitional equality judgment, which is something that carries over into Lean^{-}. However, in adding [EQ-REFL] to the theory Lean^{-}_{e}, we also add general transitivity to it. This is because propositional equality itself is transitive, as expressed in the equality transitivity lemma:

```
theorem Eq.trans {A : Sort u} {a b c : A} (h1 : a = b) (h2 : b = c) : a = c := ...
```

This lemma allows us to show transitivity of definitional equality in Lean^{-}_{e} as follows:

Lemma 5.1.6. For all contexts Δ and all terms t, s, u, T such that $\Delta \vdash^{-}_{e} t, s, u : T$, if $\Delta \vdash^{-}_{e} t \equiv s$ and $\Delta \vdash^{-}_{e} s \equiv u$, then we have $\Delta \vdash^{-}_{e} t \equiv u$.

Proof. Because $\Delta \vdash^{-}_{e} t \equiv s$, we have by [CONV] and [CGR-APP] that $\Delta \vdash \texttt{Eq.refl t} : t = s$. Similarly, because $\Delta \vdash^{-}_{e} s \equiv u$, we have $\Delta \vdash \texttt{Eq.refl s} : s = u$. Therefore, it follows that $\Delta \vdash \texttt{Eq.trans (Eq.refl t) (Eq.refl s)} : t = u$, and the result follows by [EQ-REFL]. □

This means that, for the purpose of our proof, we may not actually need Conjecture 5.1.1 to hold. As long as every instance of $\Delta \vdash t \rightsquigarrow s$ in Lean corresponds to an instance of $\Delta \vdash^{-}_{e} t \equiv s$ in Lean^{-}_{e}, may be able to use Theorem 5.1.6 to stitch together the derivation we need.

Formally, we can show that any individual reduction step corresponds to a definitional equality in Lean^{-}_{e}:

Lemma 5.1.7. For all contexts Δ and all terms t, s, if $\Delta \vdash t \rightsquigarrow s$, then $(\texttt{prfIrrel}, \Delta) \vdash^{-}_{e} t \equiv s$.

Proof. This must be shown on a case-by-case basis for each possible Lean reduction rule. Let's first the consider the case of a reduction rule in Lean that was preserved in Lean^{-}_{e}, with a conclusion judgment $\Delta \vdash t \rightsquigarrow s$. If the rule has no premises of the form $\Delta \vdash^{-}_{e} a \rightsquigarrow^{*} b$, then we know by the inductive hypothesis that all of the premises are preserved as their corresponding Lean^{-}_{e} judgments, and so the conclusion follows by [RED].

On the other hand, suppose the rule has premises of the form $\Delta \vdash^{-}_{e} a_1 \rightsquigarrow^{*} b_1, \ldots, \Delta \vdash^{-}_{e} a_n \rightsquigarrow^{*} b_n$, with the a_i being a subterms of t – that is, $t = C[a_1, \ldots, a_n]$ for some term C with n subterm holes. Then, we know by the inductive hypothesis that that $(\texttt{prfIrrel}, \Delta) \vdash^{-}_{e} a_1 \equiv b_1, \ldots, (\texttt{prfIrrel}, \Delta) \vdash^{-}_{e} a_n \equiv b_n$. Therefore, by subterm congruence, we have $(\texttt{prfIrrel}, \Delta) \vdash^{-}_{e} t \equiv C[b_1, \ldots, b_n]$. We also know by applying the same reduction rule that $\Delta \vdash^{-}_{e} C[b_1, \ldots, b_n] \rightsquigarrow s$, where each of the reduction premises are satisfied reflexively, with the others following by induction. So, $(\texttt{prfIrrel}, \Delta) \vdash^{-}_{e} C[b_1, \ldots, b_n] \equiv s$ follows from [RED], with our conclusion following from Theorem 5.1.6.

The above reasoning applies to all of Lean's reduction rules except for [KLR], which is not present in Lean^{-}_{e}. For [KLR], the definitional equality follows as a particular case of proof irrelevance, (which we have shown to hold definitionally in Lean^{-}_{e} with `prfIrrel` in our typing context). □

This allows us to show that any transitive chain of reduction steps corresponds to a definitional equality in Lean_e^-:

Lemma 5.1.8. For all contexts Δ and all terms t, s, if $\Delta \vdash t \rightsquigarrow^* s$, then $(\texttt{prfIrrel}, \Delta) \vdash_e^- t \equiv s$.

Proof. We proceed by induction on the length of the reduction sequence. If the reduction is a single step, then $\Delta \vdash t \rightsquigarrow s$ and we can use Theorem 5.1.7 to show our result. Otherwise, there is some term u such that $\Delta \vdash t \rightsquigarrow u$ and $\Delta \vdash u \rightsquigarrow^* s$. We have $\Delta \vdash t \equiv u$ by the Theorem 5.1.7, and $\Delta \vdash u \equiv s$ by the inductive hypothesis, so our result follows by Theorem 5.1.6. □

Finally, we can show the desired property:

Lemma 5.1.9. For all contexts Δ and all terms t, s, if $\Delta \vdash t \rightsquigarrow^* s$ and $\Delta \vdash s \equiv u$, and $(\texttt{prfIrrel}, \Delta) \vdash_e^- t \equiv u$.

Proof. From the inductive hypothesis, we have $(\texttt{prfIrrel}, \Delta) \vdash_e^- s \equiv u$, and from Theorem 5.1.8 we have $(\texttt{prfIrrel}, \Delta) \vdash_e^- t \equiv s$, and so the result follows from Theorem 5.1.6. □

The premises of this lemma corresponds to those of the rule [RED], so this shows that definitional equality is preserved in Lean_e^- in the case of typing in Lean by [RED], thereby completing our proof of Theorem 5.1.5.

The three theories relevant to our translation that we have introduced thus far are summarized in the following table:

Theory	Rules	Assumed Axioms	$\subset$
Lean^- ($\vdash^-$)		`prfIrrel`	Lean
Lean ($\vdash$)	[PI], [KLR]		Lean_e^-
Lean_e^- ($\vdash_e^-$)	[EQ-REFL]	`prfIrrel`	

Repurposing an ETT-to-ITT translation?

We have now established that Lean $\subset$ Lean_e^-: any typing valid in Lean is also valid in Lean_e^-, and, Lean_e^- additionally allows any propositional equality to become definitional via [EQ-REFL]. Relative to Lean_e^-, Lean can be interpreted as a more restricted extension of Lean^-, where only certain propositional equalities are promoted: specifically, those of proof irrelevance (if we assume the presence of the axiom `prfIrrel` in the typing context) and unit-η, as well as certain special cases of transitivity corresponding to Lean reduction derivations involving [KLR]. Therefore, the task of translating from Lean_e^- to Lean^- must be "at least as hard" as translating from Lean. Also, a translation from Lean_e^- to Lean^- is sufficient for a translation from Lean to Lean^-, since for any Lean derivation we can construct a corresponding Lean_e^- derivation, which we can then use as input to this translation.

Translating from Lean_e^- to Lean can be interpreted as a specific case of the more general task of translating from extensional to intensional type theory, which is a fairly well-studied topic. Such a translation is in fact possible, with a formally verified implementation in Rocq by Winterhalter et. al. in `ett-to-itt` [43, 44], which builds on previous work by Oury [29] and Hofmann [23], with the first result showing conservativity of ETT over ITT demonstrated by Hofmann [22].

This translation places certain restrictions on the target intensional theory, namely that it exhibits propositional uniqueness of identity proofs (UIP) and function extensionality. Lean^- satisfies UIP thanks to `prfIrrel`:

```
theorem UIP {A : Sort u} (x y : A) (p q : x = y) : p = q := prfIrrel p q
```

Lean^- also satisfies function extensionality with the theorem `funext` from the Lean standard library, where it is proven through the use of quotient types:

```
-- (module `Init.Core`)
theorem funext {A : Sort u} {B : A → Sort v} {f g : (x : A) → B x}
    (h : (x : A) → f x = g x) : f = g := ...
```

Restrictions are also placed on the source extensional theory by requiring an ETT syntax with domain- and codomain-annotated lambda and application constructors, which Lean does not have. We skirt this requirement through the use of an extra premise in our application congruence lemma (see Section 6.2).

Somewhat unsurprisingly, many characteristics of the translation defined in `ett-to-itt` are closely in line with the intuitive translation that we derived above. The overall design of the translation is the same, in that it consists of applying casts to subterms to make implicit uses of conversion explicit, with type equality proofs being similarly generated from the definitional equality derivations used by these conversion steps. The `ett-to-itt` translation also principally works with heterogeneous equality, though the formulation it uses is somewhat different in that it also carries a proof of equality between the LHS and RHS types, which, if defined in Lean, might look something like:

```
def HEqP {A : Sort u} (a : A) {B : Sort u} (b : B) : Prop :=
  Exists fun (p : Eq A B) => cast p a = b
```

While such a formulation makes for more convenient correctness proofs, it is less convenient for an actual implementation, so we instead choose to return to the "John Major equality" used by Oury [29], which is a more compact and equivalent formulation already defined in the Lean standard library in the `HEq` type introduced earlier (corresponding to `JMeq` in the Rocq standard library).

However, this is done mainly as a formalization convenience, and it is in fact an equivalent type to the non-proof-carrying `HEq` type that we propose to use. Another difference is that the translation is defined w.r.t. minimal extensional and intensional theories, whereas we are concerned with the particular theories Lean and Lean$^-$. However, we suspect that the results from `ett-to-itt` can be adapted to our case (we work towards showing this in Section 5.1.3).

Practically speaking, our translation does not need to implement a full ETT-to-ITT translation. We only care about translating terms that are already typeable in Lean, so proof irrelevance, unit-η, and K-like reduction are the only implicit definitional equality judgments we need to make explicit. Nevertheless, this does not afford us any real simplifications in the translation algorithm. Proof irrelevance may be used during typechecking to the same extent as the equality reflection rules in an extensional theory, as there are no syntactic restrictions on where proofs can appear in terms. In particular, they can appear within types, leading to some fairly complex translations (as we have seen previously in Section 5.1.2 in our attempt to translate `prfIrrelExHeq`).

Interesting to us is the definition by Winterhalter et. al. of a "similarity relation", which they denote with the operator "$\sim$". We say that $t \sim s$ if t and s differ only in having certain subterms "decorated" by type casts between provably equal types. Winterhalter et. al. prove a certain "fundamental lemma" related to this which states that any two terms that are well-typed in the intensional theory and related by the similarity relation are also provably equal in the intensional theory. With respect to our theories, we can state this as the following conjecture:

Conjecture 5.1.2 (Fundamental Lemma)**.** For all contexts Δ and all terms t, s, if t and s are well-typed in Lean$^-$ and $t \sim s$, then there exists some term p such that $\Delta \vdash^- p : t == s$.

This conjecture will be useful to us when we try to extend the results from `ett-to-itt` to our theories in Section 5.1.3.

So, let's officially propose a translation $|\cdot|^-$ from Lean to Lean$^-$ based on an ETT-to-ITT translation. Recall that our composite translation from Lean to Dedukti was defined as $|t|^{\hookrightarrow} := ||t|^-|^{\hookrightarrow}_{\sqsubset}$ on all well-typed terms t. The form of the function $|t|^-$, however, is a bit contrary to

what we would expect for an ETT-to-ITT-based translation, as it takes terms as input, rather than typing derivations. To address this discrepancy, we can introduce a hypothetical function $\mathrm{D}(\cdot)$ that returns (some representation of) a typing derivation given any well-typed input term. Then, we could define $|\cdot|^-_{\mathrm{D}}$ as our ETT-to-ITT-based translation from typing derivations to terms, with the full translation from Lean to Lean$^-$ being the composition of these two functions:

$$|t|^- := |\mathrm{D}(t)|^-_{\mathrm{D}}$$

The soundness of such a translation would follow from the soundness of the translation $|\cdot|^-_{\mathrm{D}}$, defined w.r.t. the subject term of the root node of the input derivation tree.

The translation $|\cdot|^-_{\mathrm{D}}$ itself can be considered a composition of a translation $\mathrm{D}^-_e(\cdot)$ from Lean derivations to Lean^-_e derivations[9] with a translation $|\cdot|^-_{\mathrm{D}^-_e}$ from Lean^-_e derivations to Lean$^-$ terms:

$$|d|^-_{\mathrm{D}} := |\mathrm{D}^-_e(d)|^-_{\mathrm{D}^-_e}$$

The soundness of this translation can then, in turn, be reduced to the soundness of the translation $|\cdot|^-_{\mathrm{D}^-_e}$, which should largely follow from the results by Winterhalter et. al. – however, there are certain discrepancies between our theories and the ETT and ITT theories used in `ett-to-itt` that we need to resolve in order us to be able to fully extend these results to our own translation.

Is ETT-to-ITT Enough?

One major remaining question here is: are the results by Winterhalter et. al. immediately applicable to the kind of translation we are trying to achieve? Can we define a similar translation whose correctness follows directly from the correctness proofs formalized in `ett-to-itt`? It would seem that the answer is no – this work defined the translation w.r.t. a very specific extensional and intensional theory, rather than in a possibly more modular way that could be applicable to a more abstract class of theories obeying certain minimal properties. This is because the concrete theories used in the formalization were specifically chosen to be minimal for a convenient first formal proof of the feasibility of a translation from ETT to ITT.

However, in general, given two theories that are parallel extensions of these minimal theories (and thus differ from one another only in the presence of an equality reflection rule in the extensional theory), it may in certain cases be possible to adapt these results without too much trouble. Specifically, if these theories only include additional typing rules over those in the minimal theories, we could define the translation such that it performs no transformation of the subject term in the case of typing by this new rule, with the typing premises following by induction.

If the theories include new definitional equality rules, however, there may be a bit more work work to do, in both defining the translation and proving its correctness. We need to show that, for any additional definitional equality rules, it is still possible to compose a heterogeneous equality proof between the LHS and RHS, assuming by induction that this is possible for any definitional equality judgments in the premises (in the same manner as was done by Winterhalter et al).

The Lean^-_e/Lean$^-$ theories are essentially the same as the minimal ETT/ITT theories used in `ett-to-itt`, except for Lean's use of the additional definitional equality rule [RED]. So, we have to show that it is possible to construct a proof of heterogeneous equality between two terms in the case that they are identified by [RED]. Similarly to the proof of Theorem 5.1.8, this boils down to showing that it holds in the case of a single-step reduction:

Conjecture 5.1.3. For all contexts Δ and all terms t, s, if $\Delta \vdash^-_e t \rightsquigarrow s$, then there exists some term p such that $\Delta \vdash^- p : |t|^- \texttt{ == } |s|^-$.

[9] Such a function could be extracted from the constructive proof of Theorem 5.1.4 that we provide above in showing that Lean $\subset$ Lean^-_e.

Like in the proof of Theorem 5.1.7, to prove this property we have to consider each possible single-step reduction case in turn and show that we can construct such a proof p. As such, the proof of this property is likely to be fairly long and involved, especially in light of certain special translation considerations such as recursor application alignment (described in Section 6.2), so for the moment it is left as a conjecture. However, let's take a look at what the proof looks like for the reduction rule [CTX] in the case of application function head reduction.

Lemma 5.1.10. For all contexts Δ and all terms f, f', e, if $\Delta \vdash^{-}_{e} f \rightsquigarrow f'$ and $\Delta \vdash^{-} p : |f|^{-}$ `==` $|f'|^{-}$, then there exists some term p' such that $\Delta \vdash^{-} p' : |f\ e|^{-}$ `==` $|f'\ e|^{-}$.

Proof. We know from the structure of our translation that there must be some e'_1, e'_2 such that $e \sim e'_1$ and $e \sim e'_2$, with $|f\ e|^{-} = |f|^{-}\ e'_1$ and $|f'\ e|^{-} = |f'|^{-}\ e'_2$. Since $e'_1 \sim e'_2$ by transitivity of $\sim$, have by Conjecture 5.1.2 that there is some $\Delta \vdash^{-}_{e} p_1 : e'_1$ `==` e'_2. We also know by the inductive hypothesis that there is some $\Delta \vdash^{-}_{e} p_2 : |f|^{-}$ `==` $|f'|^{-}$. So, we can use an application congruence lemma together with p_1 and p_2 to obtain a term $\Delta \vdash^{-} p' : |f|^{-}\ e'_1$ `==` $|f'|^{-}\ e'_2$. □

Revisiting Completeness

Assuming that Conjecture 5.1.3 holds, the proof of the soundness property formalized in `ett-to-itt` should be adaptable to showing the soundness of the $|\cdot|^{-}_{\mathrm{D}^{-}_{e}}$ translation, and hence the soundness of the $|\cdot|^{-}$ translation. Showing completeness of this translation is less straightforward, however. Recall the completeness conjecture:

Conjecture (Completeness)**.** For all contexts Δ and terms T such that $\Delta \vdash T :$ `Sort u`, if there exists some term t such that $|\Delta|^{-} \vdash^{-} t : |T|^{-}$, then there exists some term t' such that $\Delta \vdash t' : T$.

To show that our translation is complete, we start with the assumption that the translation of some term $\Delta \vdash T :$ `Sort` ℓ is inhabited in Lean^{-} by some term t, and we want to show that T is also inhabited in Lean. Our first idea may be to attempt something similar to what we did in the case of soundness, that is, show that the translation is complete w.r.t. Lean$^{-}_{e}$ as the source theory, and hopefully have this transfer over to completeness w.r.t. Lean as the source theory.

However, this doesn't quite work: completeness w.r.t. Lean$^{-}_{e}$ would tell us that there exists some t' such that $\Delta \vdash^{-}_{e} t' : T$, but it does not necessarily follow that $\Delta \vdash^{-} t' : T$: this is precisely the *opposite* direction of implication that we were dealing with in the case of soundness, and in general we do not have that a well-typed Lean$^{-}_{e}$ term is also a well-typed Lean term, as Lean$^{-}_{e}$ is more expressive that Lean.

However, that isn't to say that this approach doesn't work: since we assume that T is Lean-typeable, it may be the case that t' must have some derivation that is "compatible" with Lean, in that it does not use any equality reflection steps that do not already correspond to valid definitional equalities in Lean. In that case, we can perhaps replace these steps with uses of these definitional equalities, recovering a Lean derivation from the Lean$^{-}_{e}$ one. However, all of this may be somewhat complex to formalize, so let's try a different approach.

Let's see if we can skip the middle ground of Lean$^{-}_{e}$ in showing this property. Something we do know is that since Lean$^{-} \subset$ Lean, if $\Delta \vdash^{-} t : |T|^{-}$, then $\Delta \vdash t : |T|^{-}$. However, is it also the case that $\Delta \vdash t : T$? Well, it would be if $\Delta \vdash T \equiv |T|^{-}$, which would allow us to apply [CONV]. We can show that this is indeed the case, using the lemma below.

Lemma 5.1.11. For any terms t, s such that $\Delta \vdash t \equiv s$, $\Delta \vdash |t|^{-} \equiv |s|^{-}$.

Proof. We proceed by induction on the derivation of $\Delta \vdash t \equiv s$. Thanks to [RED], it suffices to show that $\Delta \vdash |t|^{-} \rightsquigarrow^{*} t$ and $\Delta \vdash t \equiv |s|^{-}$, and to show $\Delta \vdash t \equiv |s|^{-}$, it suffices to show that $\Delta \vdash |s|^{-} \rightsquigarrow^{*} s$ (via [RED] and [SYMM]). We need only show one of $\Delta \vdash |t|^{-} \rightsquigarrow^{*} t$ or $\Delta \vdash |s|^{-} \rightsquigarrow^{*} s$,

with the other following symmetrically. We can show that $\Delta \vdash |t|^- \rightsquigarrow^* t$ by demonstrating that every subterm of $|t|^-$ that is surrounded by a cast introduced by the translation can be reduced in Lean such that the cast is eliminated.

We know that for any subterm t' that has been surrounded by a cast between two types A and B, i.e. `cast` $A\ B\ p\ t'$, where we originally had that $\Delta \vdash A \equiv B$. By the inductive hypothesis, then, we have $\Delta \vdash |A|^- \equiv |B|^-$, and $\Delta \vdash$ `cast` $A\ B\ p\ t' \rightsquigarrow^* t'$ follows from [KLR][10]. In this way, we can eliminate all of the casts from the translated term via subterm reduction, giving us our result. □

A corollary of the above is that $\Delta \vdash |T|^- \rightsquigarrow^* T$, using a similar proof to the inductive step (replacing uses of the inductive hypothesis with Theorem 5.1.11), which gives us via [RED] that $\Delta \vdash |T|^- \equiv T$, and hence our completeness result.

[10] This is because an application of the `cast` operation in Lean reduces to an `Eq.rec` application with the provided type equality proof as the major premise and the cast term as the minor premise. In this case, the type equality proof p is between the definitionally equal types A and B, so [KLR] applies and reduces the recursor application to the minor premise t'.

Chapter 6

Lean4Less: Implementation Details

We have now contextualized our task of translating from Lean to Lean$^-$ as a special case of a translation from extensional to intensional type theory, essentially interpreting Lean as a restricted extensional version of Lean$^-$. The constructive proof by Winterhalter et. al. that we extended upon to define a translation from Lean^-_e to Lean$^-$ was useful in showing that a translation is theoretically possible – however, for the purpose of verifying proofs with a smaller trusted kernel and eventually exporting proofs from Lean to other proof assistants, we would like to have an actual translation implemented that can be integrated within a larger translation pipeline (for instance, to translate from Lean to Dedukti). So, how do we actually implement such a preliminary translation, and how do we make it practical? Let's start by trying to figure out what the general structure of our implementation is going to look like, in terms of its expected input/output and our overall translation strategy.

6.1 Implementation Framework

Adapting `ett-to-itt`?

Our first thought may be to simply adapt the work by Winterhalter et. al. which was formalized in Rocq in the `ett-to-itt` repository [44]. Rocq provides some facilities for extracting OCaml code from constructive proofs, which we could possibly use to extract a translation implementation from the formalization. However, this is likely not a great idea, as the translation defined therein was designed for the purpose of demonstrating that such a translation is possible and formally verifiable, with minimal considerations made for its practicality. It includes several particularities for the purpose of making the formal proof easier, often at the cost of making the translation itself less practical (for instance, its choice of a heterogeneous equality representation that carries a proof of equality between the LHS and RHS types).

One of the principal difficulties that we expect to encounter in adapting such an extracted implementation for our purposes, however, is that the formally defined translation expects to take typing derivations as input. It is often the case that typing derivations are very large and difficult to work with, and this runs contrary to the standard kind of translation we want to do, in which we take terms as input and produce terms as output. Typing derivations are not usually handled explicitly in this way – they are often only implicitly accounted for in implementing typechecker kernels or meta-programs, and as such most proof assistants (including Lean) provide no way to access them directly, nor even any way to represent them to begin with.

6.1.1 Adapting a Lean Kernel

So, without easy access to the derivations that we would need as input for a formally verified translation, is there possibly another way to implement our translation? We would like our translation to take terms as input, but, as we have made clear, operating just on the terms at a syntactic level simply does not suffice for the kind of derivation-aware translation that we need. Ultimately, we would like to implement a translation that is only defined on well-typed Lean terms, since well-typed terms will always have some kind of derivation that we will hopefully be able to "access" somehow in order to construct our translation.

In this respect, a promising approach is to base our translation on a typechecker kernel implementation. Proof assistant typechecker kernels are essentially programs that attempt to decide the well-typedness of terms according to some theoretical typing judgment by effecting a kind of "search" over possible typing derivations. This typing judgment may be explicitly laid out beforehand, guiding the implementation of the kernel (as is the case for proof assistants like Andromeda [7], which are designed around particular type theories), or it may be left more implicit in the typechecker implementation itself (as was the case for Lean, initially). As such, the steps of the execution of a typechecker kernel in deciding the well-typedness of a term can be seen as a "trace" of the implicit typing derivation tree of that term, as these individual steps can be correlated with uses of specific rules from the proof assistant's underlying type theory.

For instance, consider the case of the typing of a λ-function. Recall that Lean's typing rule for λ-functions is:

$$\frac{\Delta \vdash A : \texttt{Sort}\ \ell \quad \Delta, x : A \vdash e : B}{\Delta, x : A \vdash \texttt{fun}\ (x : A)\ \texttt{=>}\ e : (x : A) \to B}\ [\text{LAM}]$$

This rule corresponds to the following subroutine in Lean's kernel[1]:

```
def inferLambda (e : Expr) : RecM Expr := match e with
  | .lam name domType body bi => do
    -- infer the type of `domType`, ensuring that it is some `Sort l`
    -- (throwing an exception otherwise)
    ensureSort domType
    -- add a new free variable to the typechecking context, with the fresh name `id`
    let id ← mkFreshId
    withLCtx ((← getLCtx).mkLocalDecl id name domType bi) do
      let bodyType ← inferType (body.instantiate1 (.fvar id))
      return (← getLCtx).mkForall (#[.fvar id]) bodyType
  | e => throw <| .other "expected lambda function"
```

This function receives as input a term that is expected to be a λ-function, and starts by ensuring the domain type lives withing some `Sort` ℓ by a call to `ensureSort`, which corresponds to the first premise of the rule [LAM], It then descends into the binder, adding a fresh variable to the context corresponding to the domain type, which is then used to instantiate the body of the function, whose type is then inferred, with a call to `inferType` corresponding to the second premise. Finally, we use this inferred type to construct the return value as a function type expression.

So how, exactly, can we possibly make use of the correspondence between typechecker execution traces and typing derivation trees in our translation? There are two possible avenues we can consider.

Generating Typing Derivations?

The first possible approach is to modify a typechecker kernel to produce some literal representation of an input term's typing derivation, and have the main translation operate on this explicit Lean

[1] Note that we have simplified the function somewhat for presentation purposes.

typing derivation as its input. This would involve constructing some representation of a typing derivation as an explicit data structure, with constructors corresponding to each of the typing rules. We could then modify `inferLam` to capture the typing derivation of the input λ-function expression in a construction of this derivation object that is built in parallel to normal typechecking. At a higher level, we could then define the main phase of our translation as one that operates on these derivation objects. Firstly, we would translate the derivation itself from a Lean derivation to a Lean^{-}_{e} derivation, along the lines of what was described in the constructive proof of Theorem 5.1.4. Then, we would implement a translation from Lean^{-}_{e} derivations to Lean^{-} terms that is somewhat in line with the kind of program that could be extracted from `ett-to-itt`.

Doing our translation in this way has the benefit of clearly delineating the different translation steps w.r.t. the different theories involved, which would result in an implementation that is easy to formally verify. Additionally, the typing derivations produced by our modified typechecker could also possibly be useful outside of just the context of proof translation.

Translating in Parallel to Typechecking?

Alternatively, we could structure our translation as one that uses a modified kernel to directly produce the translated terms, rather than using some intermediate derivation representation. Since all the information needed to produce typing derivations is already present at typechecking runtime, we could perhaps instead produce our translation directly in-place during the process of normal typechecking, having reconstructed the final translated term by the time that typechecking completes. With this approach, type inference would correspond to translation, and definitional equality checking would correspond to equality proof generation, (in line with the way our translation is originally defined on typing/defeq derivations).

For instance, in this case our modified `inferLam` subroutine would look something like:

```
def inferLambda (e : Expr) : RecM (Expr × Expr) := match e with
  | .lam name domType body bi => do
    let (_, domType') ← inferType domType
    -- `p` is a proof that the inferred type of `domType'` is equal to some `Sort u`
    let p ← ensureSort domType'
    let domTypeCast' ← applyCast p domType'

    let id ← mkFreshId
    withLCtx ((← getLCtx).mkLocalDecl id name domTypeCast' bi) do
      let (bodyType, body') ← inferType (body.instantiate1 (.fvar id))
      return
        ((← getLCtx).mkForall (#[.fvar id]) bodType,
         (← getLCtx).mkLambda (#[.fvar id]) bod')
  | e => throw <| .other "expected lambda function"
```

This function operates in much the same way as the original one, however it has an extra return value of type `Expr` that represents the translated input term, with the return type of the top-level `inferType` function being similarly modified. We start by calling `inferType` on the domain type of λ-function, extracting its Lean^{-} translation from the second return value. Then, we ensure that the type of the translated domain type is some `Sort` ℓ, with `ensureSort` now returning a proof term corresponding to this equality. This proof term is used to apply a cast around the translated domain type[2], ensuring that it types as `Sort` ℓ in Lean^{-} as well. Finally, we descend into the

[2]Note that in our actual implementation, rather than always applying casts, we attempt to only apply them where necessary (that is, where expected types and inferred types are not already definitionally equal in Lean^{-}; see Section 6.3 for more details).

binder, translating the body expression and inferring its type, which we use to construct our return value, consisting of both the inferred function type and the translation of the λ-function to Lean$^-$.

This approach is perhaps somewhat less elegant and straightforward to verify than the one using explicit derivations. In particular, as we are translating directly from a Lean derivation, rather than a Lean$_e^-$ one, such an implementation will not line up neatly with the `ett-to-itt` formalization, as we will have to account for the derivation-level translation from Lean to Lean$_e^-$ simultaneously with the translation from Lean$_e^-$ derivations to Lean$^-$ terms. However, this is easier to implement as a first prototype translation, as we do not have to concern ourselves with multiple translation steps and a separate intermediate representation of typing derivations. So, this is the approach that we have decided to go with for our preliminary translation from Lean to Lean$^-$.

Lean4Lean

The specific kernel implementation that we choose to go with for our translation is taken from "Lean4Lean" [12], which is a project to write a formally verified kernel implementation and accompanying metatheory for Lean along the lines of the MetaRocq project [35]. The Lean4Lean kernel implementation is essentially a direct port of the original Lean kernel's C++ code into Lean[3]. The implementation makes use of Lean's monadic `do` notation to imitate an effectful, imperative program with global state that directly corresponds to the original C++ kernel implementation.

The original motivations for implementing Lean4Lean's typechecker in Lean itself were to make it possible for the implementation to be formally verified w.r.t. Lean's metatheory. In the context of possibly adapting it for proof translation, however, this fact is especially convenient for our purposes. In general, we would like to implement our translation in Lean, as doing so has a number of benefits. Firstly, as Lean is a partly bootstrapped language, many of its higher-level features are implemented exclusively within Lean, which use a number of helper functions for traversing and constructing expressions, manipulating free/bound variables, modifying the typechecking environment, etc., that will be useful to us in our own implementation. Also, Lean's orientation towards formal proof and typechecking affords us certain "soft" guarantees in the correctness of our implementation. Specifically, its strict typing, higher-order functional programming style, termination checking and monadic facilities for manipulating global state and context provide more confidence during development by eliminating certain classes of errors. An implementation in Lean also leaves the door open for an eventually fully verified translation, on account of Lean's capabilities as a general theorem prover. Formally verifying a translation could also have some important meta-theoretical implications, (as explained in Section 7.2.2).

6.2 Implementation Details

Our translation from Lean to Lean$^-$, which we call "Lean4Less"[4], is adapted from Lean4Lean's typechecker kernel implementation (as was motivated above), which implements a bidirectional typechecking algorithm using the following three primary mutually recursive functions[5]:

```
-- type inference
def inferType (e : Expr) : RecM Expr := ...
-- definitional equality check
def isDefEq (t s : Expr) : RecM Bool := ...
-- weak-head normalization
def whnf (e : Expr) : RecM Expr := ...
```

[3] In fact, the kernel code snippets used above were taken directly from Lean4Lean's kernel implementation

[4] `https://github.com/Deducteam/Lean4Less`

[5] These can be found in the file containing the kernel's main typechecking subroutines, `TypeChecker.lean`.

- `inferType` is a type inference function that checks that `e` is well-typed, throwing an error if it is not, and returning its inferred type if it is.

  Whether or not it returns successfully effectively decides Lean's typing judgment. Specifically, it has the specification that under some typing context Δ, for all terms `t`, `inferType t` returns successfully if and only if there exists some `T` such that $\Delta \vdash$ `t : T`.

- `isDefEq` returns whether or not the well-typed terms `t` and `s` are definitionally equal according to Lean's definitional equality judgment. Notationally, this means that under some typing context Δ, for all terms `t` and `s`, `isDefEq t s` returns `true` if and only if $\Delta \vdash t \equiv s$.

- `whnf` reduces an expression to its weak-head normal form (WHNF). It is a subroutine of `isDefEq`, where terms must sometimes be (partly) reduced to determine if they are definitionally equal. Under some typing context Δ, if `whnf t` successfully returns some term `s`, this means that the reduction relation $\Delta \vdash t \rightsquigarrow^* s$ holds.

These functions live inside of the potentially non-terminating[6] `RecM` monad, which enables stateful side-effects such as updating a cache of previously inferred types and incrementing a unique name generator (used for the naming of free variables), and which provides a monadic context that keeps track of the current constant declaration environment, free variable context, and universe level parameters.

These three functions correspond to the three relations efining Lean's type theory, which appear in the Lean typing derivations that we have defined our translation on. As such, we would expect each of these functions in our modified typechecker to additionally output terms that correspond to the translation output semantics that we have established for the typing relation that each function decides. Therefore, we can imagine modifying each of the functions as follows[7]:

```
def inferType (e : Expr) : RecM (Expr × Expr) := ...
                                    -- ^ translated `e`
def isDefEq (t s : Expr) : RecM (Bool × Option Expr) := ...
                                    -- ^ proof of `t == s`
def whnf (e : Expr) : RecM (Expr × Expr) := ...
                                  -- ^ proof of `e == whnf e`
```

The `inferType` function now also returns a translated version of the input expression that has essentially been "injected" with type transports where required by typing constraints (see below) – note that the first return value is the original inferred type return value, and we maintain that this inferred type is Lean$^-$-typeable (that is, it is a translation to Lean$^-$ of the type that would have normally been inferred by Lean4Lean's `inferType` function). Formally, this semantics can be presented with the following lemma:

Lemma. If $\Delta \vdash e : T$, then `inferType` $e = (T', e')$, with $\Delta \vdash^- e' : T'$, $\Delta \vdash e \equiv e'$ and $\Delta \vdash T \equiv T'$.

The `isDefEq` function now also returns a generated proof of equality between the input terms, which are expected to already be Lean$^-$-typeable. It has the following formal specification:

[6] Lean's type theory is known to be non-terminating as a consequence of K-like reduction, proof irrelevance, and impredicativity, with a general result having been demonstrated by Coquand and Abel [1], which has been verified empirically by Carneiro [12].

[7] These are not quite the actual return values of our functions; we have simplified things here for the initial presentation. In light of certain output optimization considerations, we actually only *optionally* return additional `Expr` values. See Section 6.3 for more details.

Lemma. If $\Delta \overset{\hookrightarrow}{\vdash} t : T$, $\Delta \overset{\hookrightarrow}{\vdash} s : S$, and $\Delta \vdash t \equiv s$, then `isDefEq` t s $=$ `(true, Option.some` p`)`, with $\Delta \overset{-}{\vdash} p : t$ `==` s.

Lastly, the `whnf` function also returns a proof of equality between the Lean$^-$-typeable input term and its weak head normal form, with the following specification:

Lemma. If $\Delta \overset{\hookrightarrow}{\vdash} t : T$, then `whnf` $t = (t', p)$, with $\Delta \vdash t \rightsquigarrow^* t'$ and $\Delta \overset{-}{\vdash} p : t$ `==` t'.

As required by the general ETT-to-ITT translation, both `isDefEq` and `whnf` return *heterogeneous* equality proofs using the type `HEq`. Note that `whnf` must also return a heterogeneous equality proof because the inferred type of the input term may change during reduction[8].

The proof term returned from `isDefEq` or `whnf` can be interpreted as a "trace" of the typechecker's steps in deciding definitional equality/performing WHNF reduction. For instance, if the typechecker determines that the applications `f a` and `f b` are definitionally equal, where proof irrelevance was used at some point when comparing `a` and `b` to produce a proof term `p : a = b`, Lean4Less will construct a proof using Lean's `congrArg` lemma in order to produce the proof term `congrArg f a b p : f a = f b`[9]. This corresponds to our definition of the theoretical ETT-to-ITT translation in the case of definitional equality by [CGR-APP].

The Congruence Lemmas

The translation defined by Winterhalter et. al. in `ett-to-itt` makes use of a set of "congruence lemmas" that are used to compose equality proofs between terms from the equality proofs of their corresponding subterms in the function type, λ-function, and application cases. For the purposes of our translation, we require a similar set of lemmas, expressed in Lean as follows:

```
theorem forallHEqABUV' {A B : Sort u} {U : A → Sort v} {V : B → Sort v}
  (hAB : A == B) (hUV : (a : A) → (b : B) → a == b → U a == V b)
  : ((a : A) → U a) ((b : B) → V b) := ...
theorem lambdaHEqABUV' {A B : Sort u} {U : A → Sort v} {V : B → Sort v}
  (f : (a : A) → U a) (g : (b : B) → V b)
  (hAB : A == B) (hfg : (a : A) → (b : B) → a == b → f a == g b)
  : (fun a => f a) == (fun b => g b) := ... -- (uses funext)
theorem appHEqABUV' {A B : Sort u} {U : A → Sort v} {V : B → Sort v}
  (hAB : A == B) (hUV : (a : A) → (b : B) → a == b → U a == V b)
  {f : (a : A) → U a} {g : (b : B) → V b} {a : A} {b : B}
  (hfg : f == g) (hab : a == b)
  : f a == g b := ...
```

Note that `appHEqABUV'` contains the additional hypothesis `hUV` that allows us to equate `U` and `V` in its proof, which was *not* required by the equivalent application congruence lemma used in `ett-to-itt`. Including this hypothesis enables us to prove the lemma without the presence of domain- and codomain-annotated λ-function and application constructors, which was a requirement on the source ETT syntax imposed by [43] in order to be able to prove a version of this lemma

[8] For instance, if we have types A, B such that $\Delta \vdash A \equiv B$ but $\Delta \not\vdash A \equiv B$, and terms p, t such that $\Delta \overset{-}{\vdash} p : A$ `=` B and $\Delta \overset{-}{\vdash} t : A$, then, thanks to K-like reduction, we have $\Delta \vdash$ `@cast` $A\ B\ p\ t \rightsquigarrow^* t$, with $\Delta \overset{-}{\vdash}$ `@cast` $A\ B\ p\ t : B$, so to state an equality between `@cast` $A\ B\ p\ t$ and t in Lean$^-$, we must use heterogeneous equality.

[9] In reality, our implementation always generates heterogeneous equality proofs, using a heterogeneous version of `congrArg`, though it should be technically feasible to implement an optimization that generates homogeneous equality proofs instead wherever possible.

that does not carry this hypothesis[10]. While it may seem feasible to derive this hypothesis from the equality of the types of `f` and `g` implied by `hfg`, this is in fact not possible in Lean without the addition of a "forall-η" axiom with the following signature:

```
-- (not used by our translation)
axiom forallEta : ((a : A) → U a) = ((a : A) → V a) → U = V
```

Assuming such an axiom breaks some theoretical properties of Lean, in particular its interpretation under a cardinality model where all types of equal size are considered equal[11].

We also need the proof irrelevance axiom and its derivable extension to heterogeneous equality in the case of provably equal propositional types. For convenience, we also add a heterogeneous cast function:

```
axiom prfIrrel {P : Prop} (p q : P) : p = q
theorem prfIrrelHEq {P : Prop} (p q : P) : p == q := ...
theorem prfIrrelHEqPQ {P Q : Prop} (hPQ : P == Q)
   (p : P) (q : Q) : p == q := ...
def castHEq {A B : Sort u} (h : A == B) (a : A) : B :=
  cast (eq_of_heq h) a
```

These constants, along with all of their dependencies, need to be enumerated to our translation to be added to the translation output environment first, since any later definitions may reference them as a result of translation. Importantly, they must already be well-typed in Lean$^-$ and should not require translation themselves, since this would result in cyclic self-references.

Inserting Type Transports

The modified `inferType` function translates the input term by "injecting" type casts around subterms whose inferred type is not definitionally equal to its expected type in Lean$^-$. Specifically, this can occur wherever uses of the [CONV] typing rule are implicitly invoked by the kernel. Such conversions checks correspond to calls to `isDefEq` that check that two type expressions are definitionally equal, ultimately arising from Lean's particular typing rules that originate from either from either user-provided annotations or typing restrictions imposed by certain type inference rules. The type casts require proofs of equality between the expected and inferred types, which are computed in the second return value of `isDefEq` (more details on the computation of these equality proofs are provided later).

Let's take a closer look at the particular typing rules that can generate a `cast` application in our translation output. Recall first the rules for the typing of λ-functions and function types:

$$\frac{\Delta \vdash A : \texttt{Sort}\ \ell \quad \Delta, x : A \vdash e : B}{\Delta, x : A \vdash \texttt{fun}\ (x : A)\ \texttt{=>}\ e : (x : A) \to B}\ [\text{LAM}] \qquad \frac{\Delta \vdash A : \texttt{Sort}\ \ell \quad \Delta, x : A \vdash B : \texttt{Sort}\ \ell'}{\Delta \vdash (x : A) \to B : \texttt{Sort}\ (\texttt{imax}\ \ell\ \ell')}\ [\text{ALL}]$$

These rules require the domain types (and codomain type, in the case of [ALL]) to be sorts. Together with [CONV], this weakens to the assumption that the binder types have types that are definitionally

[10] For a verified translation, using this hypothesis requires a proof that it can always be inhabited, which has not been shown by Winterhalter et. al. However, we have not encountered any problems proving this hypothesis on-the-fly as a part of our translation.

[11] If we assume this axiom, we can show a counterexample to the cardinality model as follows: Let `A := Fin 2`, and let `U := fun x => if x = 0 then Bool else Unit` and `V := fun x => if x = 0 then Unit else Bool`. Then, we have the function type cardinalities `|(a : A) → U a| = |(a : A) → V a| = 2`, allowing us to derive `U = V` from `forallEta`. By application congruence `U 0 = V 0`, which contradicts that `|U 0| = 2` $\neq$ `|V 0| = 1`.

equal to some `Sort` ℓ. For instance, recall that we can use [ALL] and [CONV] to derive the following equivalent rule:

$$\frac{\Delta \vdash A : T \quad \Delta \vdash T \equiv \texttt{Sort}\ \ell \quad \Delta, x : A \vdash B : U \quad \Delta, x : A \vdash U \equiv \texttt{Sort}\ \ell'}{\Delta \vdash (x : A) \to B : \texttt{Sort}\ (\texttt{imax}\ \ell\ \ell')} \text{ [ALL']}$$

from which it is more evident how definitional equality may figure into the judgment of the well-typedness of a function type expression. This rule is much closer to how the kernel function `inferForall` is actually implemented:

```
def inferForall (e : Expr) : RecM Expr := match e with
  | .forallE name domType codomType bi => do
    let domTypeSort ← inferType domType -- premise 1
    ensureSort domType -- premise 2
    let (Expr.sort l1) := domTypeSort | unreachable!
    let id ← mkFreshId
    withLCtx ((← getLCtx).mkLocalDecl id name domType bi) do
      let codomType := codomType.instantiate1 (.fvar id)
      let codomTypeSort ← inferType codomType.instantiate1 -- premise 3
      ensureSort codomType -- premise 4
      let (Expr.sort l2) := codomTypeSort | unreachable!
      return .sort <| mkLevelIMax s.sortLevel!
  | e => throw <| .other "expected lambda function"
```

W.r.t. the translation, each of the definitional equality premises corresponds to a place where a `cast` application may be inserted around a subterm. Specifically, we will insert a cast around the subterm typed as the LHS of the premise (according to a preceding premise) to ensure it types as the RHS instead. In the case of [ALL'], if it turns out that the translation of T is not definitionally equal to some `Sort` ℓ in Lean$^-$, then our translation will have to include a cast around A, transporting its type from T to `Sort` ℓ.

Also recall the rule for the typing of applications:

$$\frac{\Delta \vdash f : (x : A) \to B \quad \Delta \vdash e : A}{\Delta \vdash f\ e : B[e/x]} \text{ [APP]}$$

Here, the duplication of the type A in the premises effectively requires that the type of e is definitionally equal to the domain type of the function f. Additionally, we require that the function f has a type that is equivalent to some function type. Again via [CONV], this rule can be equivalently expressed as:

$$\frac{\Delta \vdash f : T \quad \Delta \vdash T \equiv (x : A) \to B \quad \Delta \vdash e : U \quad \Delta \vdash U \equiv A}{\Delta \vdash f\ e : B[e/x]} \text{ [APP']}$$

whose premises are again much closer to the conditions that are actually checked by the kernel. As it relates to our translation, the premises of the rule [APP'] imply that both the function f and argument a might possibly have a `cast` applied to them: in the case of the function f, we may need to explicitly transport its type to be a function type, and in the case of the argument a, we may need to explicitly transport its type to match the domain type of f.

Lastly, recall our rule for typing `let` binders:

$$\frac{\Delta \vdash A : \texttt{Sort}\ \ell \quad \Delta \vdash e : A \quad \Delta, x : A \coloneqq e \vdash b : B}{\Delta \vdash \texttt{let}\ (x : A)\ \coloneqq\ e\ \texttt{in}\ b : B\ [x/e]} \text{ [LET]}$$

Thanks to [CONV], we can equivalently express this rule as follows:

$$\frac{\Delta \vdash A : T \quad \Delta \vdash T \equiv \texttt{Sort}\ \ell \quad \Delta \vdash e : U \quad \Delta \vdash U \equiv A \quad \Delta, x : A := e \vdash b : B}{\Delta \vdash \texttt{let}\ (x : A)\ :=\ e\ \texttt{in}\ b : B\ [x/e]}\ [\text{LET'}]$$

In terms of our translation, we may apply a cast around either the type A to transport it to some $\texttt{Sort}\ \ell$, or we may apply a cast around e so that it matches its annotated type A. A similar translation also applies w.r.t. our translation of Lean constant contexts, where declared constant type signatures may be cast to ensure that they are of some $\texttt{Sort}\ \ell$, and the bodies of defined constants (that is, Lean definitions and theorems) may also be cast to match the annotated constant type signature.

Proof/Translated Term Representations

To build equality proofs as part of the definitional equality-checking routines, we use a custom data structure `EExpr` to represent our proof, defined in the file `EExpr.lean`:

```
/-- Structured data representing equality proof expressions. -/
inductive EExpr where
-- prfIrrel axiom
| prfIrrel : PIData EExpr → EExpr
-- congruence lemmas
| lam      : LamData EExpr → EExpr
| forallE  : ForallData EExpr → EExpr
| app      : AppData EExpr → EExpr
-- free variable equality assumption
| fvar     : FVarDataE → EExpr
-- HEq.trans
| trans    : TransData EExpr → EExpr
-- HEq.refl
| refl     : ReflData → EExpr
-- reversed proof
| rev      : EExpr → EExpr
```

The inductive type has constructors for equality proofs using the proof irrelevance axiom, each of the congruence lemmas, free variable equality assumptions (bound by certain congruence lemma arguments), transitivity for chaining proofs together (needed for composing proofs corresponding to reduction sequences), reflexivity proofs (which must be used sometimes), as well as a constructor that flags equality proofs for reversal.

Using such a purpose-built representation for proof terms has two main advantages. Firstly, it centralizes the production of proof terms via the `EExpr.toExpr` function, allowing us to use the object representation when constructing proofs inside of the modified definitional equality subroutines, deferring the actual construction of the `Lean.Expr` proof term to the topmost call producing the typecast, which is much more convenient than building the `Lean.Expr` proofs in-place. Secondly, it enables certain post-hoc optimizations, such as lambda-casting (see Section 6.3) that operate on the structure of the proof *after* it has been produced.

We also use the "pseudo-refinement type" `PExpr` to represent already-translated expressions, taking `Expr` to be the type of expressions that have yet to be translated. Using this "safety wrapper" can help avoid certain errors. For instance, it is important for the correctness of the translation that the terms passed as input to `isDefEq` and `whnf`, and the output inferred type of `inferType`

and its translated argument are all well-typed in Lean$^-$. As such, these functions have types that look closer to the following[12]:

```
def inferType (e : Expr) : RecM (PExpr × PExpr) := ...
def isDefEq (t s : PExpr) : RecM (Bool × Option EExpr) := ...
def whnf (e : PExpr) : RecM (PExpr × EExpr) := ...
```

- `inferType` expects as input a term `e` that is Lean-typeable, and thus has type `Expr`. It outputs an inferred type and translated version of `e`, both of which are Lean$^-$-typeable, which indicated by the fact that they have type `PExpr`.
- `isDefEq` expects two Lean$^-$-typeable terms `t` and `s` as input, as enforced by the type `PExpr`. It outputs a judgment of definitional equality, as well as an `EExpr` representing a Lean$^-$-typeable proof of heterogeneous equality between `t` and `s`.
- `whnf` expects a Lean$^-$-typeable term `e` as input, again enforced by the type `PExpr`. It outputs a Lean$^-$-typeable weak-head normal form of `e`, as well as an `EExpr` representing a Lean$^-$-typeable proof of heterogeneous equality between `e` and its weak-head normal form.

Producing Equality Proofs

As we have mentioned, the modified `isDefEq` function produces a proof of heterogeneous equality between the LHS and RHS terms that is well-typed in Lean$^-$. In doing so, it calls a number of subroutines that correspond to the various rules of Lean's definitional equality judgment. Considering the particular rules we are eliminating via our translation, most of these functions – for instance those corresponding to congruence identities – are not very "interesting" , in that they just combine and propagate equality proofs that are produced by their own function calls, and do not actually produce "base level" proofs that generate proof terms directly corresponding to the eliminated definitional equalities. The three functions that do generate such "base proof terms" are `isDefEqProofIrrel`, `toCtorWhenK`, and `isDefEqFVar`; our modifications to these functions are described below.

Firstly, we adapt the kernel function `isDefEqProofIrrel`, which checks whether two proof terms are equal by proof irrelevance by checking that their propositional types are definitional equality:

```
def isDefEqProofIrrel (t s : Expr) : RecM (Option Bool) := do
  let tType ← inferType t
  if !(← isProp tType) then
    -- [PI] is not applicable
    return Option.none
  let eql ← isDefEq tType (← inferType s)
  return (Option.some eql)
```

We modify this function to generate a proof of equality between the proof terms using the `prfIrrel` axiom as follows:

```
def isDefEqProofIrrel (t s : PExpr) : RecM (Option (Bool × Option EExpr)) := do
  let (tType, _) ← inferType t
  let (sType, _) ← inferType s
  if !(← isProp tType) then
```

[12]Note that these are, again, not the final type signatures for these functions (which can be found in Section 6.3).

```
    -- [PI] is not applicable
    return Option.none
  let (eql, p?) ← isDefEq tType (← inferType s)
  if not eql then
    return (Option.some (false, Option.none))
  let some p := p? | unreachable!
  let mut p := default
  if ← isDefEqLeanM t s (recDepth := 15) then
     p := ... -- construct a proof of `t = s` using `prfIrrel`
  else
     p := ... -- construct a proof of `t == s` using `prfIrrelHEq`
  return (Option.some (true, Option.some p))
```

If the propositional types are not already Lean$^-$-defeq, we will need to use the proof irrelevance lemma `prfIrrelHEqPQ`, which takes this explicit proof of equality between the LHS and RHS propositional types, and produces a heterogeneous equality proof. Otherwise, we can return a proof using `prfIrrelHEq`, which assumes that the LHS and RHS propositions are the same (but which still uses heterogeneous equality for compatibility with the rest of the translation). There is also a check that avoids producing an equality proof (that is, returning `none`) in the case that the proof terms can be shown to be equal in the Lean$^-$ kernel after a small number of steps.

We generate a similar proof in the case of K-like reduction via the function `toCtorWhenK` (which is called by the recursor reduction function `inductiveReduceRec`):

```
def toCtorWhenK (rval : RecursorVal) (e : Expr) : m Expr := do
  assert! rval.k -- ensure this is a K-like type

  -- apply the K-like type's unique constructor to obtain `ctorApp`
  let type ← inferType e
  let (ctorApp, ctorName) ← mkNullaryCtor type rval.numParams

  -- make sure that the indices of the inferred types match
  unless ← isDefEq type (← inferType ctorApp) do return (e, false)

  return (ctorApp, true)
```

Given as input a recursor major premise `e` of a K-like inductive type `toCtorWhenK` effectively "rewrites" `e` to the unique constructor application implied by the inferred K-like type of `e`, which is then substituted in for `e` in the term being reduced in order to continue the reduction.

Our modification of this function generates a proof of equality between the input `e` and the output constructor application by returning a proof obtained by calling the modified `isDefEqProofIrrel` function:

```
def toCtorWhenK (rval : RecursorVal) (e : Expr) : m (Expr × Option EExpr) := do
  assert! rval.k

  let (type, _) ← inferType e
  let (ctorApp, ctorName) ← mkNullaryCtor type rval.numParams

  unless ← isDefEq type (← inferType ctorApp) do return (e, false)

  let (true, Option.some p) ← isDefEqProofIrrel e ctorApp | unreachable!

  return (ctorApp, Option.some p)
```

Another place where we may generate base equality proof terms is in equating pairs of free variables introduced by the variable-binding proof arguments of certain congruence lemmas: specifically, the argument `hUV` in `forallHEqABUV'` and `appHEqABUV'`, having the type

```
hUV : (a : A) → (b : B) → a == b → U a == V b,
```

and the argument `hfg` in `lambdaHEqABUV'`, having the type

```
hfg : (a : A) → (b : B) → a == b → f a == g b.
```

When generating the proof terms for these arguments, we introduce two separate variables `a : A` and `b : B` into our context, one for each domain type. Additionally, we introduce into our countext a variable representing an equality proof between `a` and `b`, `hab : HEq a b`, which we may need to use to construct the proof. We "register" the variables as being provably equal in the translation's monadic context:

```
structure TypeChecker.Context : Type where
  ...
  -- stores fvar triples as the map (x : A), (y : B) -> (hxy : x == y)
  eqFVars : Std.HashMap (FVarId × FVarId) FVarId := {}
  ...
```

This corresponds to the triple-valued context computed by the "`Pack`" function of Winterhalter et al. [43]. We add a free variable-specific equality check, `isDefEqFVar`, that returns an equality proof using the relevant variable equality hypothesis:

```
def isDefEqFVar (idt ids : FVarId) : RecM (Option (Bool × (Option EExpr))) := do
  if let some d := (← readThe Context).eqFVars.get? (idt, ids) then
    return (Option.some (.true, some (EExpr.fvar d)))
  else if let some d := (← readThe Context).eqFVars.get? (ids, idt) then
    return (Option.some (.true, some (EExpr.rev (EExpr.fvar d))))
  return Option.none
```

Recursor Application Alignment

There is one particularly tricky aspect that we will have address in our translation relating to the reduction of recursor applications in Lean. In the process of reducing a recursor application, the kernel will first attempt to simplify the major premise to a constructor application so that it can extract the necessary fields to perform the reduction by applying a constructor-specific recursor reduction rule. It first uses `whnf` to reduce the major premise to its weak-head normal form, followed by `toCtorWhenK` to rewrite it as its unique constructor application if its type is that of a K-like inductive with valid indices. In the process of doing so, the type of the major premise may change such that it is no longer Lean^{-}-defeq to its original inferred type.

For instance, as we mentioned in Section 1.2.3, the [KLR] enables a cast application of the form `@cast A B p t` between Lean-defeq types `A` and `B` to reduce to the argument t via K-like reduction, which has the type `A`, rather than type `B`. This behavior carries over into our translation, which can create some difficulties involving recursor reduction. Consider the following example:

```
axiom P : Prop
inductive T : (p : P) → Type where
| mk (p : P) : T p
-- only needed so that T isn't a struct (and doesn't use struct-like reduction)
```

```
| extra (p : P) : T p

-- T.rec.{u} : {p : P} → {motive : T p → Sort u} →
--   (mk : motive (@T.mk p)) → (extra : motive (T.extra p)) → (t : T p)

def castEx (p q : P) :
  @T.rec p (fun _ => Bool) true true
    (@cast (T q) (T p) (congrArg T (L4L.prfIrrel q p)) (T.mk q))
  = true
   := rfl
```

In the type signature of `castEx`, both the LHS and RHS of the equality are already Lean$^-$-typeable. However, LHS and RHS are not Lean$^-$-defeq (as the major premise of `T.rec` requires K-like reduction in order to reduce to `T.mk q`), so the proof by `rfl` is not correctly typed and requires translation.

For this translation, Lean4Less will need to produce a Lean$^-$-typeable proof of equality between the LHS and RHS, which it will do by reducing the LHS via `whnf`, and in the process of doing so, generate a proof that the LHS is equal to its Lean-reduct of `true`. When reducing the recursor application, however, it first needs to reduce the `cast` application in the major premise, which results in the term `T.mk q : T q`. This is a problem, because it no longer has the expected type of `T p`. Substituting in this new major premise, we obtain the recursor application `@T.rec p (fun _ => Bool) true true (T.mk q)`, which is no longer well-typed in Lean$^-$ as we expect the major premise to be of type `T p`, not `T q`. Consequently, we cannot construct a proof about the equality of the old recursor application with the new one.

To recover a well-typed recursor application, we must do some post-hoc alignment of its initial arguments to accept the new major premise type. We start by replacing its parameters with those of the new major premise type (in this case, this consists of replacing the parameter `p` with `q`). We then cast any dependent motives/minor premises as needed so they remain well-typed[13], replace the old major premise with the new one, and lastly (if the resultant motive type is a function type) cast as necessary any remaining dependent arguments following the major premise [14]. We finally construct a proof of equality between the old and new recursor applications, which is transitively chained onto the returned equality proof. In terms of the above example, this amounts to producing a proof of the following equality:

```
def castEx' (p q : P) :
  @T.rec p (fun _ => Bool) true true
    (@cast (T q) (T p) (congrArg T (L4L.prfIrrel q p)) (T.mk q))
  =
  @T.rec q (fun _ => Bool) true true (T.mk q)
   := ...
```

where the RHS reduces immediately to the final Lean-reduct of `true`.

It is also important to note that the casting of minor premises, which may happen as a result of recursor application alignment, can break the normal reduction of recursors, which consists of the direct application of minor premises to fields extracted from the major premise (if there are any). Applying a cast around a λ-function that would normally be immediately reducible upon application via β-reduction can result in costly extra proof generation step to show that the cast reduces to a λ-function, and in some cases we have even observed this to lead to non-termination.

[13] In the implementation, this procedure can be found in the function `replaceParams`.

[14] This corresponds to the function `replaceFType` in the implementation.

In this respect an optimization for λ-casting that we later describe (see Section 6.3) seems to also be relevant for translation correctness.

Similar considerations arise when eliminating on quotient types, when the `Quot.lift` and `Quot.ind` quotient elimination operations are used [15]. Here, we also compute the weak head normal form of the "major premise" quotient argument before applying the reduction, in order to be able to extract the representative term from the quotient construction. As in the case of recursor reduction, performing this normalization on the quotient argument may result in a quotient type with different parameters, in which case it becomes necessary to post-align the quotient eliminator application with the new type and cast any parameters (using the same helper function `replaceParams`), transitively chaining a proof of equality between the old and new applications onto the returned proof.

6.3 Optimizations

Output and runtime optimizations are particularly important for a tool like Lean4Less, to be able to scale up the translation to large libraries and to have a reasonably sized output that avoids redundancy. Additionally, it is important to have an efficient implementation that enables the translation to complete within a reasonable amount of time without excessive memory requirements. By virtue of being based on an efficient typechecker implementation, Lean4Less already enjoys many output and runtime optimizations that transfer over from the kernel. For instance:

- Lean uses "lazy δ-reduction" in its `isDefEq` check, avoiding the expansion of equal δ-expandable constant function application heads where possible, opting to first perform a comparison on each pair of arguments. This translates to an output optimization in which we can also avoid expanding these constants in the output when generating equality proofs.
- Lean's proof irrelevance check is placed very early on in the `isDefEq` check, ensuring that we do not needlessly compare proof subterms if we already know that the proof types are equal (thus making the proofs definitionally equal by proof irrelevance). This also becomes an output optimization, because we can immediately output an equality proof using the `prfIrrel` axiom, rather than possibly producing a larger proof resulting from a more detailed comparison of subterms (in the case that the proofs can be shown equal without applying [PI]).
- Lean's kernel makes use of a cache for recording previously computed weak-head normal forms. Lean4Less adapts this cache to store an equality proof in addition to the weak-head normal form itself, and can be queried to avoid unnecessary computations. This translates into an output optimization since these redundant proofs will also share object pointers in the `.olean` output.

Lean4Less also implements some optimizations of its own, described below.

Congruence Lemma Variants

When generating an equality proof, we may use the congruence lemmas presented in Section 6.2 in a variety of common ways that make certain proof arguments redundant. For example, when producing an application congruence proof, we may use reflection proofs for the `hAB` argument to `appHEqABUV'` when the function domain types are already Lean$^-$-defeq, or we may ignore the bound variables introduced within `hUV` arguments when the codomains `U` and `V` are not dependent. In

[15] This is implemented in the function `quotientReduceRec`.

such cases, we can use specialized variants of our congruence lemmas that allow us to ignore these arguments. For instance, the following variant of application congruence assumes Lean⁻-defeq domain types and non-dependent (but non-Lean⁻-defeq) codomain types:

```
theorem appHEqUV {A : Sort u} {U V : Sort v}
  (hUV : HEq U V)
  {f : (a : A) → U} {g : (b : A) → V} {a b : A}
  (hfg : HEq f g) (hab : HEq a b)
  : HEq (f a) (g b) := ...
```

In the case that the arguments are also Lean⁻-defeq, we can simplify the lemma further:

```
theorem appFunHEqUV {A : Sort u} {U V : Sort v}
  (hUV : HEq U V)
  {f : (a : A) → U} {g : (b : A) → V} (a : A)
  (hfg : HEq f g)
  : HEq (f a) (g a) := ...
```

We use similar simplified variants of the λ-function and function type congruence lemmas where possible.

Avoiding Redundant Casts/Proof Terms

A property that our translation should ideally respect is that we only introduce applications of `cast` in places where truly needed. That is, we should only wrap subterms with an explicit type transport if it is strictly necessary to do so to ensure that they are well-typed in the context in which they appear. Of course, this will only be the case if one of he eliminated definitional equality rules [PI], [KLR], or [UNIT] is used in some "essential" way in a definitional equality judgment, where removing these rules from our theory without casting the subterm would make the larger term in which the subterm appears non-well-typed (or result in a definition whose body's inferred type does not match the annotated type signature).

One approach we could take here is to simply run the Lean kernel's definitional equality check whenever we encounter a situation where we might have to insert a cast in our translation. If the original kernel ever uses an eliminated definitional equality, it can set a flag that it returns to the calling function. If the flag it set, we will then call our modified `isDefEq` function to generate the type equality proof and apply a cast to the subterm in question. Alternatively, we could take the following approach: since our modified `isDefEq` function is based on the original kernel implementation, we could instead call it immediately, and have it *optionally* return a proof of equality between the LHS and RHS as its second return value, with the status optional value being either `Option.some _` or `Option.none` essentially taking the place of the flag in the previous approach. With this approach, the type signatures of our `isDefEq` and `whnf` functions become:

```
def isDefEq (t s : PExpr) : RecM (Bool × Option EExpr) := ...
                                    -- ^ proof of `t == s`
def whnf (e : PExpr) : RecM (PExpr × Option EExpr) := ...
                                  -- ^ proof of `e == (whnf e)`
```

The invariant that we might maintain would be that we only return an `Option.some` value from a definitional equality subroutine call if the two terms are definitionally equal (according to Lean) and we actually make use of one of the eliminated definitional equalities when comparing them. A return of `Option.none` indicates that the terms are already Lean⁻-defeq. In the case of `isDefEq` being called as a type conversion check, this means that no typecast is required. In the case of `isDefEq` or `whnf` being called to construct part of a larger type equality proof, this indicates that

a proof using `HEq.refl _` suffices for an equality proof between the subterms, and also informs which congruence lemma variant to use.

However, the invariant stated above is perhaps not exactly what we want – the fact that a particular definitional equality was used at some point does not mean that it was strictly *necessary* to use it in order for the overall identity to hold. For instance, consider the definition below:

```
variable (P : Prop) (p : P) (q : P) (T : P → Prop)
theorem redundantCastEx (t : T p) : T ((fun x => x) p) := t
```

This proof is already well-typed in Lean$^-$, because we already have a definitional equality between `p` and `(fun x => x) p` without using proof irrelevance, thanks to β-reduction. With the above strategy, however, we would translate this as:

```
theorem redundantCastExTrans (t : T p) : T ((fun x => x) p) :=
  cast (T p) (T ((fun x => x) p))
    (congrArg T (prfIrrel p ((fun x => x) p)))
    t
```

Here, we have unnecessarily introduced a cast around the definition body, with a type equality proof using `prfIrrel` to show the equality between the already-definitionally equal `p` and `(fun x => x) p`.

Ideally, our `isDefEq` function would only return an `Option.some` value if the definitional equality does not already hold in Lean$^-$– in other words, if the use of an eliminated definitional equality was essential in its derivation. Precisely, we can state our ideal invariant as:

$$\texttt{isDefEq}\ t\ s = (\texttt{true}, \texttt{Option.some}\ p) \iff \Delta \vdash t \equiv s \wedge \Delta \nvdash^{-} t \equiv s,$$

given some typing context Δ. This is a stronger invariant than our previous one (which we do not quite have the notation to express symbolically). Satisfying it would require exhaustively searching the space of possible Lean$^-$ definitional equality derivations, which, if the terms being compared are in fact not Lean$^-$-defeq, can be quite expensive. With respect to a typechecker implementation (that would practically implement this search), this is tantamount to invoking a kernel's worst-case runtime, which may involve completely reducing both terms to their normal forms and comparing them subterm-by-subterm.

Such an exhaustive search can prove to be prohibitively expensive in the worst case, which is something that we have verified experimentally. So, we will necessarily have to make a trade-off here. The approach that we have gone with is to have an initial check in `isDefEq` which checks whether two terms are equal in Lean$^-$ within a limited recursion depth, returning `(true, none)` if so. With this recursion depth limit set to 15 (as opposed to the default value of 1000), we have been able to verify that a great number of unnecessary casts/equality proof terms can be avoided with minimal runtime/memory cost to the execution of the translation.

Domain Variable Sharing

As explained earlier, when generating the proof terms for variable-binding arguments of certain congruence lemmas – specifically, the `hUV` parameter to `forallHEqABUV'` and `appHEqABUV'`, and the `hfg` parameter to `lambdaHEqABUV'` – we introduce *two* separate variables, `a : A` and `b : B` into our typechecking context, one for each domain type. This partly undoes an optimization from the original typechecker implementation, where upon recursion into a binder term during definitional equality-checking, a *single* variable is added to the context to simultaneously represent both domains and substituted into the λ-function body/function type codomain before a recursive call. Then, within some later nested definitional equality check, checking equality in the kernel

between the variables instantiated on both sides is simply a matter of checking that they are syntactically equal, as they are literally the same variable. This is a sound optimization because both binding domains are checked to be Lean-defeq just before descending into the binder. However, this optimization does not carry over into the modified version of this procedure in the Lean4Less implementation, where the domain types may not be definitionally equal in Lean$^-$. Therefore, in the general case, we have to introduce two separate variables, keeping track of a propositional equality between them with another variable which originates from the type of the `hUV` / `hfg` arguments to the congruence lemmas, using the additional congruence-checking function `isDefEqFVar`, as was described in Section 6.2.

However, we *do* retain this optimization in the special case that the domain types are Lean-defeq, because it translates into an output optimization that allows us to avoid some unnecessary proof generation. This case corresponds to the use of the following simplified congruence lemma variants:

```
theorem forallHEqUV' {A : Sort u} {U V : A → Sort v}
  (hUV : (a : A) → HEq (U a) (V a))
  : HEq ((a : A) → U a) ((b : A) → V b) := ...

theorem lambdaHEqUV' {A : Sort u} {U V : A → Sort v}
  {f : (a : A) → U a} {g : (b : A) → V b} (hfg : (a : A) → HEq (f a) (g a))
  : HEq (fun a => f a) (fun b => g b) := ...

theorem appHEqUV' {A : Sort u} {U V : A → Sort v} (hUV : (a : A) → HEq (U a) (V a))
  {f : (a : A) → U a} {g : (b : A) → V b} {a : A} {b : A}
  (hfg : HEq f g) (hab : HEq a b)
  : HEq (f a) (g b) := ...
```

In contrast with the fully general congruence lemmas these lemmas only to introduce a single variable for the domain types of the `hUV` and `hfg` proof arguments. This optimization not only saves us from including redundant proofs at the level of the application of these lemmas themselves, but also precludes the generation of large, unnecessary proof arguments as a result of nested calls to `isDefEq` that use the needlessy introduced proof hypothesis. For instance, a translation that does not include this optimization may unnecessarily generate the following equality proof:

```
axiom P : Prop
axiom Q : P → Prop
theorem lamEqEx : ((p : P) → Q p) = ((p : P) → Q p) := rfl
theorem lamEqExBadTrans : ((p : P) → Q p) = ((p : P) → Q p) :=
  L4L.castHEq
    (L4L.appArgHEq' (Eq ((p : P) → Q p))
      (L4L.forallHEqABUV' HEq.rfl (fun a b hab => L4L.appArgHEq' Q hab)))
    rfl                          --^ hUV argument
```

Determining that the proof in the body of the `hUV` argument can in fact be shown via `HEq.refl` would then require a post-hoc investigation of the computed proof term to discover that it does not contain any uses of `prfIrrel`. With this optimization, however, we introduce a single variable for the `p` binder, which avoids the construction of a proof term entirely (and thus avoiding the `cast` application as well).

λ-Casting

When we apply a cast to a λ-function, the equality proof that it uses must necessarily be an application of a `forallHEq` congruence lemma, as in the following example:

```
axiom G : P → Prop
inductive H : (p : P) → G p → Type where
| mk (p : P) (g : G p) : H p g

def pushTest : (g : G q) → H q g := fun (g : G p) => H.mk p g

def pushTestTrans1 : (g : G q) → H q g :=
  L4L.castHEq
    -- proof of (g : G p) → H p g == (g : G q) → H q g
    (L4L.forallHEqABUV' (L4L.appArgHEq' G (L4L.prfIrrelHEq p q))
      fun (gp : G p) (gq : G q) (a : HEq gp gq) =>
        L4L.appHEqABUV' (L4L.appArgHEq' G (L4L.prfIrrelHEq p q)) ...
          (L4L.appArgHEq' H (L4L.prfIrrelHEq p q)) a)
    fun (g : G p) => H.mk p g
```

where we cast the entirety of the function with a single equality proof between the inferred and expected function types, generating subproofs showing equality between the domain types `G q` and `G p` and codomain types `H q g` and `H p g`. In doing so, however, we have lost the ability to reduce applications of this function in Lean$^-$– while the application `pushTest g` is β-reducible for some `g : G q` (following the δ-expansion of `pushTest`), `pushTestTrans1 g` is not, as the λ-function term has been surrounded by a cast which cannot be eliminated on account of the lack of K-like reduction. This is not a problem for our translation, however, as the cast is between Lean-defeq types, and so by K-like reduction we can generate a proof of propositional equality as needed between the application of the cast λ-function and its Lean$^-$-typeable reduct:

```
@cast ((x : G p) → H p x) ((x : G q) → H q x) _ (fun (x : G p) => H.mk p x) g
-- reduces to -->
H.mk p (@cast (G q) (G p) _ g)
```

(the `cast` around `g` here appears as a result of recursor application alignment, see Section 6.2). However, this is an instance of "transport hell", as we will need to generate a large proof to additionally eliminate the `cast` surrounding the function. It would be convenient to instead directly encode this reduced form within the body of the λ-function, recovering β-reducibility of the application in Lean$^-$.

We can accomplish this by "pushing" the cast into the λ-function itself, changing the domain type and casting the body and bound variable within it as needed to produce a Lean$^-$ term that has the expected type:

```
def pushTestTrans2 : (g : G q) → H q g :=
  fun (g : G q) =>
    @cast
      (H p (@cast (G q) (G p) _ g))
      (H q g)
      _
      (H.mk p (@cast (G q) (G p) _ g))
```

In general, given a λ-function `fun` $(x : A)$ `=>` t of type $(x : A)$ `→` B and a generated equality proof p such that $\Delta \vdash^{-} p : (x : A) \to B = (x : A') \to B'$, applying a cast to `fun` $(x : A)$ `=>` t using p effects the following transformation:

$$\texttt{fun}\ (x : A)\ \texttt{=>}\ t \hookrightarrow \texttt{fun}\ (x : A')\ \texttt{=>}\ \texttt{@cast}\ B[x/c(x)]\ B'\ p_1\ (t[x/c(x)])$$

where we define $c(x) \coloneqq$ `@cast` $A'\ A\ p_2\ x$, with the proofs p_1 and p_2 being extracted from the proof p, with $\Delta, x : A' \vdash p_1 : B[x/c(x)]$ `=` B' and $\Delta \vdash p_2 : A$ `=` A'.

Implementation-wise, when we apply a cast to a λ-function, we make use of the structured `EExpr` data to guide the above transformation. Given an `EExpr` proof of equality between the inferred and expected function types, representing a nested sequence of `forallHEq` lemma applications, we recurse into the λ-functions in parallel to the structure of this proof. Every λ-function binder is changed to bind to the corresponding RHS domain type, and all of the bound references within the body are instantiated with a cast of the variable to the original LHS domain type in order to maintain the well-typedness of the function body. After processing all of the binders, we then cast the entire function body if necessary to match the RHS codomain type. This optimization is implemented in Lean4Less in the "`smartCast`" function.

Application Pre-Abstraction

One further optimization that we make is in avoiding the generation of long chains of application congruence lemma applications in the construction of equality proofs, which can result from a translation that too closely follows the original sequence of recursive definitional equality subroutine calls. For instance, consider the definition below:

```
axiom (P : Prop) (p q : P)
axiom A : P → Nat → Nat → Nat → Nat → Nat → Nat → Type
axiom Aq : A q 0 0 0 0 0 0

def absEx : A p 0 0 0 0 0 0 := Aq
```

Translating this to Lean$^-$ requires us to generate a proof of equality between the applications `A q 0 0 0 0 0 0` and `A p 0 0 0 0 0 0`. A naïve implementation may compute the following proof:

```
example : Eq (A q 0 0 0 0 0 0) (A p 0 0 0 0 0 0) :=
  (L4L.appFunHEq (A q 0 0 0 0 0) (A p 0 0 0 0 0) 0
    (L4L.appFunHEq (A q 0 0 0 0) (A p 0 0 0 0) 0
      (L4L.appFunHEq (A q 0 0 0) (A p 0 0 0) 0
        (L4L.appFunHEq (A q 0 0) (A p 0 0) 0
          (L4L.appFunHEq (A q 0) (A p 0) 0
            (L4L.appFunHEq (A q) (A p) 0
              (L4L.appArgHEq A q p (L4L.prfIrrel q p))))))))
```

where we propagate the base proof of equality between `q` and `p` with a chain of application congruences, one for each remaining argument. This is a natural first implementation of the translation that follows the steps taken by the original kernel's check for definitional equality of applications, which iteratively checks the definitional equality of each pair of arguments in turn[16]. However, notice that the latter arguments are already all Lean$^-$-defeq, which makes the six additional uses of application congruence seem extraneous, as the size of the generated proof should ideally only depend on the number of arguments that are not already Lean$^-$-defeq. Indeed, we can abstract both applications to the function `fun x : P => A x 0 0 0 0 0 0`, applied to `q` and `p` respectively, computing instead the equality between these β-equivalent forms This results in a much shorter proof:

```
example : Eq (A q 0 0 0 0 0 0) (A p 0 0 0 0 0 0) :=
L4L.appArgHEq (fun (a : P) => A a 0 0 0 0 0 0) (L4L.prfIrrel q p)
```

[16] This is the function `isDefEqApp` in the Lean4Lean kernel.

In terms of the implementation, this optimization would suggest constructing a proof only after we finish iterating over all of the pairs of arguments and registering which ones are not already Lean$^-$-defeq, building an appropriate "pre-abstraction" and applying this to the abstracted arguments to obtain a new LHS and RHS to construct a proof of equality between. However, a number of additional complexities arise here. Firstly, consider the following definition:

```
inductive I : Type where
| left  : P → I
| right : P → I

def ITB : I → Type
| .left _  => Unit
| .right _ => Bool

axiom B : (i : I) → Nat → Nat → Nat → ITB i → Nat → Nat → Nat → Type
axiom Bq : B (.left q) 0 0 0 () 0 0 0
def absDemoB : B (.left p) 0 0 0 () 0 0 0 := Bq
```

where the application head `B` has a dependent type, and we must construct a proof of equality between `B (.left q) 0 0 0 () 0 0 0` and `B (.left p) 0 0 0 () 0 0 0`. Here, the type of the fifth argument depends on the first one, and so both arguments must be abstracted simultaneously for the application in the body of the abstraction to be well-typed. In the implementation, this means marking an argument pair for abstraction if their types reference prior ones that have already been abstracted, regardless of whether or not they are already Lean$^-$-defeq. This results in the following proof:

```
example : B (I.left q) 0 0 0 () 0 0 0 = B (I.left p) 0 0 0 () 0 0 0 :=
  L4L.appFunHEq ()
    (L4L.appArgHEq' (fun (i : I) (a : ITB i) => B i 0 0 0 a 0 0 0)
      (I.left q) (I.left p)
      (L4L.appArgHEq I.left (L4L.prfIrrel P q p)))
```

Note however that, in this particular case, we can avoid the extra abstraction if we perform a "deeper" abstraction on the application, leaving the first arguments' constructor intact, which enables us to avoid the additional abstraction:

```
example : B (I.left q) 0 0 0 () 0 0 0 = B (I.left p) 0 0 0 () 0 0 0 :=
  L4L.appArgHEq' (fun (x : P) => B (I.left x) 0 0 0 () 0 0 0)
    q p (L4L.prfIrrel P q p)
```

For this reason, we implement a "deep" application abstraction that is able to recursively abstract within application arguments[17]

An additional complexity arises in the case of a function of dependent arity. Consider the following definition:

```
def ITC : I → Type
| .left _  => Nat → Nat → Nat → Type
| .right _ => Bool

axiom C : (i : I) → Nat → Nat → Nat → ITC i
```

[17]This optimization can also likely be generalized to non-application cases; we have yet to explore this possibility in more detail.

```
axiom Cq : C (.left q) 0 0 0 0 0 0

def absDemoC : C (.left p) 0 0 0 0 0 0 := Cq
```

where the arity of the function `C` depends on the value of its first argument. Here, imagining for a moment that we have not implemented the deep application pre-abstraction mentioned above, we would attempt to abstract the `.left _` arguments as a whole, resulting in the head function `fun (x : I) => C x 0 0 0 0 0 0`, which is an ill-typed expression since the type of `C x 0 0 0` is irreducible at `ITC x`. To resolve this, we immediately generate a proof of equality between the partial applications thus far (`C (.left q) 0 0 0` and `C (.left p) 0 0 0`), perform an abstraction on these partial applications, and obtain a new head function `fun (f : Nat → Nat → Nat → Prop) => f 0 0 0`. In essence, we have abstracted the applications at "two levels", obtaining the new left- and right-hand sides:

```
-- LHS
(fun (f : Nat → Nat → Nat → Prop) => f 0 0 0) ((fun x : I => C x 0 0 0) (.left q))
-- RHS
(fun (f : Nat → Nat → Nat → Prop) => f 0 0 0) ((fun x : I => C x 0 0 0) (.left p))
```

allowing us to isolate as much as possible the base difference between the terms `.left q` and `.left p`. Here again, we in fact can (and do) avoid the extra abstraction entirely by performing a deeper abstraction on `q` and `p` themselves, but this kind of optimization is still useful in more complex cases.

There is one last additional complexity that arises here, when the types of the abstracted partial applications are not Lean$^-$-defeq, which occurs in the following example:

```
axiom Q : Nat → P → Prop
axiom Qp : Q 0 p
axiom Qq : Q 0 q

def ITD : I → Type
| .left x  => (n : Nat) → Q n x → Nat → Prop
| .right _ => Bool

axiom D : (i : I) → Nat → Nat → Nat → ITD i
axiom Dq : D (.left q) 0 0 0 0 Qq 0
theorem absDemoD : D (.left p) 0 0 0 0 Qp 0 := Dq
```

In particular, this arises when a dependent domain type references a previously abstracted argument. When this happens, we must also abstract these differing types in the type of the abstracted application, taking into account the fact that these types may depend on previous binders in the function type expression. This results in the following top-level applications:

```
-- LHS
(fun (aT : Nat → Prop) (f : (n : Nat) → aT n → Nat → Prop) (a : aT 0) => f 0 a 0)
  (fun n : Nat => Q n q)
  ((fun x : I => D x 0 0 0) (.left q))
  Qq
-- RHS
(fun (aT : Nat → Prop) (f : (n : Nat) → aT n → Nat → Prop) (a : aT 0) => f 0 a 0)
  (fun n : Nat => Q n p)
  ((fun x : I => D x 0 0 0) (.left p))
  Qp
```

This generates another proof obligation between the abstracted domain types `(fun n : Nat => Q n q)` and `(fun n : Nat => Q n p)`. A key invariant that we maintain here is that we are sure to always abstract sufficiently so that the LHS and RHS abstracted application heads are always Lean^{-}-defeq.

Instantiation Avoidance

Two meta-functions that are particularly likely to contribute to large translation output sizes are `Lean.Expr.instantiate1`, which instantiates the innermost bound variable (of De Bruijn index 0), and `Lean.Expr.replaceFVar`, which instantiates a specified free variable. Instantiating variables should be avoided as much as possible because doing so can lead to the proliferation of several similar terms that differ only at the instantiated locales, but nevertheless cannot share object pointers in the output representation (because they are not exactly the same). This is relevant especially in the application congruence case, where the computed equality proof terms reference domain and codomain types that are progressively instantiated as we tack on more arguments in our proof, a pattern which is illustrated by the example below.

Let us briefly adopt the notation `E[n0, n1, ... nm]` to signify the instantiation of bound variables `0` to `m` in `E` with the expressions `n0` to `nm`. Suppose that we are generating a proof of equivalence between the applications `f a b c` and `f' a' b' c'`, where both `f` and `f'` have the following type:

```
f, f' : (x : X) -> (y : Y[x]) -> (z : Z[y]) -> F[x, y, z].
```

To generate this proof, we must use application congruence lemmas, which take as arguments the domain and (possibly dependent) codomain types, which we will refer to with the symbols `A` and `U`, respectively. The proof will be generated in left-to-right order of the arguments, generating the following sequence of domains and codomains:

```
f a = f b
  --> A := X
  --> U := fun (x : X) => (y : Y[x]) -> (z : Z[y]) -> F[x, y, z]
f a b
  --> A := Y[a]
  --> U := fun (y : Y[a]) => (z : Z[y]) -> F[a, y, z]
f a b c
  --> A := Z[b]
  --> U := fun (z : Z[b]) => F[a, b, z]
```

Here, we have instantiated `Y`, `Z`, and `F` in several different ways, where none of these distinct instantiations can share a common object pointer in the output. In our translation, we have implemented an optimization in which we generate the following abstracted helper types as follows as an initial step in our proof computation, generating `let`-bindings in the output proof term as follows:

```
let F' := fun (x : X) (y : Y[x]) (z : Z[y]) => F[x, y, z]
let Y' := fun (x : X) => Y[x]
let Z' := fun (x : X) (y : Y[x]) => Z[y]
```

Using these abstracted types allows us to obtain a much higher degree of sharing in our output:

```
(fun (x : X) (y : Y[x]) (z : Z[y]) => f x y z) a
  --> A := X
  --> U := fun (x : X) => (y : Y' x) -> (z : Z' y) -> F' x y z
(fun (x : X) (y : Y[x]) (z : Z[y]) => f x y z) a b
```

```
  --> A := Y' a
  --> U := fun (y : Y' a) => (z : Z' y) -> F' a y z
(fun (x : X) (y : Y[x]) (z : Z[y]) => f x y z) a b c
  --> A := Z' b
  --> U := fun (z : Z' b) => F' a b z
```

Note that while the type `Z[y]` only depends on `y`, in the definition of `Z'` we must abstract both `x` and `y` on account of the dependence of the type of `y` on `x` (in general, we abstract the full transitive closure of domain dependencies when generating domain type abstractions).

Chapter 7

Results, Prospects and Conclusion

In this final chapter, we will give an overview of the current capabilities and limitations of our translation, providing some preliminary data on its performance and success in translating formal mathematics libraries and highlighting certain pain points and possible areas for improvements. We will also describe some prospects for future work related to this research topic, from both a practical and theoretical perspective.

7.1 Translation Results and Limitations

7.1.1 Lean4Less Translation: Results

From our preliminary experiments, our implementation of Lean4Less has proven to be a practical translation that can be successfully applied at moderate scales. Our overall translation and verification workflow is visualized in Figure 7.1. Starting from an E that we obtain from a set of `.olean` files, combined with our preliminary translation constants that we obtain from the file `PatchTheorems.lean`, we pass this as input to the Lean4Less translation, obtaining an output environment E^P, which we can then export to be used by other systems (e.g. proof translation tools). For verification, we typecheck E^P using a modified fork of Lean4Lean representing a Lean$^-$ kernel, which is identical the Lean kernel but which is lacking the special checks for proof irrelevance and K-like reduction in the `isDefEq` routine that decides definitional equality between terms.

We have tested the Lean translation on the Lean standard library and various lower-level Mathlib modules, verifying our output in the manner described above, and have already had success in translating significant subsets of Mathlib to Lean$^-$, for instance Lean's real numbers library `Mathlib.Data.Real.Basic`, containing several thousands of lines of code and thousands of uses of proof irrelevance and K-like reduction.

We benchmark our translation on `Std`, the Lean core standard libary, and on the mathlib library `Mathlib.Algebra.Order.Field.Rat`, with the versions of both libraries using Lean toolchain `v4.16.0-rc2`. We report below on some measures relating to the translation of these modules on a machine with an Intel Xeon 8-core CPU @ 2.20GHz and 32 GB RAM:

Module	Total Constants	Constants Using [PI]/[KLR] (% of total)	Input/Output Environment Size (Overhead)	Translation Runtime	Input/Output Typechecking Runtime (Overhead)[1]
`Std`	29859	1736/134 (6.3%)	226MB/261MB (15.5%)	18m02s	2m19s/3m9s (36.0%)
`Algebra.Order.Field.Rat`	113899	2965/237 (2.8%)	1485MB/1501MB (1.1%)	32m16s	5m11s/5m44s (10.6%)

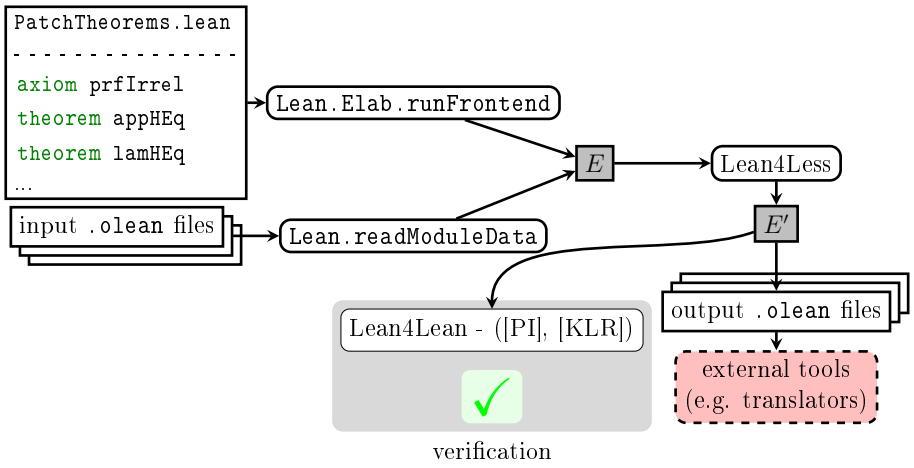

Figure 7.1: Lean4Less translation and verification workflow.

The standard library translation overhead of 15.5% is not very excessive relative to the 6% of total constants using proof irrelevance/K-like reduction, and we observe an even more modest translation overhead when translating an actual `Mathlib` module. In both cases, however, this is somewhat disproportionate to the amount of extra typechecking runtime overhead translation incurs. It is not clear how much of this overhead is truly unavoidable, but more work can certainly to be done to optimize the output size.

We can see above that translation takes significantly longer than typechecking, and we have found that the translation tends to get "stuck" for significant amounts of time translating certain constants, sometimes taking longer than ten minutes to translate a single definition. Such long-running translations also consume significant amounts of memory, which has proven to be a prohibitive factor in attempting to translate larger mathlib libraries. Further investigation is needed here. Such slowdowns may be related to general scaling problems that are closely tied to output inefficiencies, and may be resolved through the implementation of further output optimizations – for instance, the generation of auxiliary helper definitions and the more efficient use of caching.

7.1.2 Lean2dk: Preliminary Translation Results and Limitations

We have implemented our translation from Lean to Dedukti in a tool which we call "`Lean2dk`"[2]. Lean2dk uses Lean4Less as an external dependency, which it executes as an initial translation step to translate from Lean to Lean$^-$ prior to the translation from Lean to Dedukti. The final translated Dedukti environment is output in text format to a set of `.dk` files whose file structure mirrors that of the input `.olean` iles.

Lean2dk implements the PTS-based translation described in Chapter 2[3], and includes a set of Dedukti files for both the PTS encoding [4] and the universe level encoding[5], which are imported by the output `.dk` files containing the translated Lean input modules. The translation is implemented in Lean itself, giving us access to Lean's standard library which provides many useful facilities for working with Lean terms (like helper functions for recursing into binder expressions, instantiating

[1]When run with the Lean and Lean$^-$ kernels, respectively (i.e. Lean4Lean with and without proof irrelevance/K-like reduction).

[2]`https://github.com/Deducteam/lean2dk`

[3]This translation implementation can mostly be found in the file `Trans.lean`.

[4]See the file `enc.dk`.

[5]See the files `bool.dk`, `nat.dk`, `lvl.dk`, `normalize.dk`, and `sublvl.dk`, in the `dk/enc` directory.

bound variables within terms, etc.). Additionally, the translation makes us of Lean's "`MetaM`" monad, which was designed for Lean's metaprogramming framework. Relevant to us is `MetaM`'s type inference function "`inferType`", which is used in our translation of Lean function type and projection expressions (as was described in Section 2.3 and Section 2.3).

Unfortunately, at the time of writing this thesis, we have not had time to thoroughly explore and profile the efficacy of our translation; however, we have had some initial success in translating lower-level modules of the standard library (like its classical logic library `Init.Classical`). The translation unfortunately scales quite poorly to higher-level standard library modules, resulting in translated output Dedukti files that are 2-3 orders of magnitude larger than the original input Lean `.olean` iles. We suspect that this excessively large overhead is mostly attributable to the fact that we output our translation in the form of a `.dk` text file, as opposed to the highly optimized `.olean` nput format, which, among other things, implements sharing between common expresion subterms through object pointer sharing. This sharing is largely lost to our translation, where identical subterms are simply repeated in the output. While it may be possible to "simulate" this form of sharing to some extent by generalizing the auxiliary function closure generation used for our translation of `let` bindings (as was described in Section 2.3), this possible approach has not yet been fully investigated.

7.1.3 Lean4Less Translation: Caveats and Limitations

While we have had considerable success in utilizing the Lean4Less translation to translate substantial real-world mathematical libraries, there are a number of caveats and limitations to consider that make the translation less practical than we might like. Below, we characterize a few known issues with our translation that may be addressable to varying degrees.

Transport Hell

There are certain cases when terms representing expected types may themselves require translation, which must then be reconciled with inferred types, leaving us in a situation of "transport hell". This arises in the following example:

```
inductive K : Prop where
| mk : K

def F : Bool → Type
| true => Bool
| _ => Unit

structure S : Type where
b : Bool
f : F b

def projTest {B : Bool → Type} (s : B true)
   : B (@K.rec (fun _ => S) (S.mk true true) k).2 := s
```

For the annotated output type `B (@K.rec ...).2` to be well-typed, the kernel must reduce the `K.rec` application inside of the projection to `S.mk true true` in order to ensure that the projection is of type `Bool`, which requires the use of K-like reduction. This means that the translation must apply a cast around the entire projection to explicitly transport its type to `Bool`. Additionally, the translation must cast the definition body `s` to match this translated annotated type signature, which results in a large proof term as we must eliminate the cast placed around the projection in

addition to reducing the `K.rec` application. In this simple example, we can avoid such a translation altogether by simply rewriting the output type annotation to `B true`, but in general this kind of simplification may not be very obvious or practical (e.g. when the term is hidden behind a constant/reduction sequence).

We broadly refer to this kind of situation as "transport hell", in which an expected type contains translated subterms that must be simplified in the course of generating an equality proof between the expected and inferred types. Transport hell seems to be an inherent consequence of translation that unfortunately cannot be avoided, though it may help somewhat to make use of a general lemma for eliminating previously introduced casts:

```
def castRedHEq {A B : Sort u} (h : HEq A B) (a : A) : HEq (castHEq h a) a := ...
```

Then, in the modified function for computing weak-head normal forms, we can enable immediately "rewriting" any instances of `castHEq h a` to `a`, using `castRedHEq` in the returned equality proof. Implementing this would save us from unnecessarily computing a proof corresponding to the reduction of applications of `castHEq` every time that one is encountered by the modified `whnf` function.

Scaling Difficulties

As mentioned above, our translation runs into difficulty at scale when attempting to translate larger libraries from Mathlib, which is seemingly tied to excess memory usage. The first question we may ask is whether these scaling difficulties are inherent to the kind of translation we are trying to achieve, or if they can be perhaps addressed by aggressively optimizing the translation output. To what extent might our existing optimizations have already helped with scaling, and how much more can we do?

While many of the optimizations described above are useful in reducing the final output size, it is less clear whether they are actually useful in easing the total memory burden of translation and the typechecking of the translated output. In many cases, translation does not introduce anything "new" into the output that wasn't already encountered during normal typechecking. For instance, the translation of the proof of the lemma `prfIrrelExHeq` from Section 5.1.2, while quite large relative to the original proof, does not actually introduce any additional expressions that were not already encountered by the kernel during normal typechecking[6], and so do not necessarily add anything onto the total amount of memory/runtime needed to typecheck the translated term relative to the original one. This may call into question all of our attempts to optimize the output that we have described above. We may be saving somewhat on the total amount of disk space our finally translated output requires, but do our optimizations really help us scale more effectively?

We believe that our optimizations *are* useful, even in terms of scaling, as we can in fact characterize a situation in which we might end up with output terms that are much larger than anything originally encountered by the kernel when typechecking the original input terms. This can arise in particular on account of constant dependencies that may be expanded during typechecking.

For instance, suppose we have the definitions A and B, where B references A. Let A' be the translation of A to Lean$^-$, and suppose it contains some large, unwieldy subterm t that takes a particularly long time to typecheck. Suppose that B also contains another large subterm s. Now, in the course of typechecking B, suppose that the definition of A is expanded, and becomes explicit in the translation B' (this may or may not happen, possibly depending on optimizations), which then contains both t and s. This situation is already not ideal – now, to typecheck B', we have to typecheck t as well, which the original kernel did not necessarily need to do (A' may have just been

[6] That is, other than the uses of cast the proof irrelevance axiom, and some congruence lemmas, but these scale proportionally to the size of the term being translated and do not make up crucial difference.

expanded in the course of reduction, without requiring that t in particular was typechecked at any point).

However, things get even more problematic when we add yet another layer to this scenario. Suppose we have another definition C, which references B, and in the course of typechecking C, the definition of B is also expanded and becomes explicit in the translation C'. Suppose also that the kernel did not happen to need to expand A after expanding B. Then, the translation C' would include s from B' as well as t from A', as an artefact of the fact that B needed to expand A in its typing – this is despite the fact that originally, the kernel never even *encountered* t when typechecking C. Now, typechecking C' incurs the cost of typechecking *both* t and s, whereas typechecking C did not require typechecking either of them.

We can see now that in general, as we translate larger libraries with deeper constant dependencies, this effect can accumulate, which may result in scaling difficulties. In light of this, the optimizations we have implemented so far may actually be important to some extent for scaling, as they reduce the size of definitions that may be referenced by other definitions and accumulate up the dependency hierarchy during the course of translation in the manner described above. However, is there perhaps a way to fundamentally address this issue of transitive dependency expansion? A concrete example of the problem, as well as a possible way to address it, are given in Section 7.2.1.

GCD as a Primitive Operation

Lean implements a number of primitive arithmetical operations in its typechecking kernel for common natural number operations, e.g. addition, subtraction, multiplication, comparison, etc., by allowing WHNF reduction to reduce such operations using an external arbitrary precision arithmetic implementation, rather than using the kernel's normal reduction routines. This is an important optimization for allowing us to work with arbitrarily large natural numbers, which are relevant to Lean's code generation procedures (where we may want to represent, say, the largest 32 bit unsigned integer). To ensure that it is sound to use these optimized operations, we must verify that their corresponding Lean definitions satisfy the base definitional equalities that we would expect of them. For example, we must have that `x + 0 = x` and `x + (succ y) = succ (x + y)` hold definitionally in Lean.

If we want to take advantage of these optimizations in the Lean$^-$ kernel as well, we must ensure that these same definitional equalities hold just as well in Lean$^-$. All of them in fact do, with the exception of the `Nat.gcd` operation for computing the greatest common denominator between two natural numbers. Here, the equality `(gcd (succ y) x) = (gcd (mod x (succ y)) (succ y))` fails to hold definitionally in Lean$^-$. This seems to be attributable to the fact that the implementation uses well-founded recursion, rather than structural recursion, invoking a use of K-like reduction at some point during reduction. While it should be possible define the `Nat.gcd` via structural recursion (as shown in [26]), enabling the equality to hold definitionally in Lean$^-$, the current definition of `Nat.gcd` is based on a more standard definition that uses well-founded recursion.

For this reason, we do not currently enable `Nat.gcd` as a primitive operation in the Lean$^-$ kernel, as doing so may be unsound. As such, we also have to abort the translation of any constants involving `Nat.gcd` computations on large numbers, as well as any of their dependent constants. If we can patch/overhaul the definition of `Nat.gcd` so that its characteristic equations hold definitionally in Lean$^-$, we will be able to restore it as a primitive operation in the Lean$^-$ kernel and re-enable the translation of these constants.

7.2 Translation Prospects

7.2.1 Addressing Lean4Less Scaling Difficulties with Auxiliary Constants

Let's try to come up with a concrete example of how the translation scaling issue relating to dependency expansion described above may arise, and how we might be able to possibly address it. Consider the following definitions:

```
inductive E where
| a : E
| b : E

inductive T : Nat → Nat → Prop where
| mk : (n : Nat) → T n n

#print T.rec
/-
T.rec.{u} : {n : Nat} →
  {M : (m : Nat) → T n m → Sort u} →
    M n (T.mk n) → {m : Nat} → (t : T n m) →
      M m t
-/

-- some very large, long-to-typecheck term
macro x : Nat := ...

abbrev C := fun m => (n : Nat) → T n (n + m)

-- (must be marked as noncomputable because uses `T.rec`;
-- this is irrelevant to typechecking)
noncomputable def f (m : Nat) (c : C m) : E → E
| .a => @T.rec x (fun _ _ => E) .a (x + m) (c x)
| .b => .b
```

where `x` is some large term that takes a long time to typecheck (note that it is a macro, not a definition, so the elaborator directly inlines it wherever it appears). Now, suppose that we have the following definition, which invokes WHNF computation and K-like reduction through `f`:

```
noncomputable def fp (c : C 0) : f 0 c .a = .a := rfl
```

When reducing the LHS, we apply δ-expansion and β-reduction expand to reduce `f 0 c .a` to the application `@T.rec x ... (c x)`, with the major premise not being a constructor application, thus invoking K-like reduction. Because of this, while the definition of `fp` makes no reference itself to `x`, its translation to Lean$^-$ does:

```
noncomputable def fpTrans1 (c : C 0) : f 0 c .a = .a :=
  L4L.castHEq
    (L4L.appArgHEq' (Eq (f 0 c .a))
      -- proof that `@T.rec x (fun _ _ => E) .a (x + 0) (c x) = .a`
      -- (i.e., `f 0 c .a = .a`)
      (L4L.appArgHEq' (@T.rec _ (fun _ _ => E) .a _) (L4L.prfIrrelHEq (c x) (.mk x))))
    rfl
```

At first, this translation explicitly including `x` may not seem that problematic, since normally typechecking `fp` would involve δ-expanding `f`, so `x` would be pulled into memory anyways (even though its actual value is not relevant to reduction here). But the key difference here is that in the course of typechecking `fp` with the kernel, we do not actually need to do the expensive step of typechecking `x` – it briefly appears and is compared syntactically with itself when deciding whether to apply K-like reduction, but it is not actually typechecked at any point. On the other hand, our translation explicitly includes `x` in the translated definition body of `fp`, and so it *must* be typechecked when typechecking `fpTrans1`. This is an instance of the general translation scaling problem that we got at earlier – large terms can propagate to dependent constants in the process of translation, become variously instantiated and accumulate, resulting in progressively larger output terms as we translate constants that are higher up the import hierarchy.

Ideally, we would like to contain uses of `x` to a generalized "reduction proof fragment" that is particular to the function `f`, and can be utilized by any translation that generates a proof involving the reduction of an application of `f`. We can do the same for `T.rec`, generating the following auxiliary lemmas:

```
theorem T.rec_aux {n : Nat}
  {M : (m : Nat) → T n m → Sort u}
    (mtv : M n (T.mk n)) {m : Nat} (t : T n m) (p_nm : n == m) :
      (@T.rec n M mtv m t) == mtv
  := ... -- (generated proof using `p_nm`)

theorem f_aux (m : Nat) (c : C m) (e : E)
   (p_m : m == 0) (p_e : e == .a) : f m c e == .a :=
   -- (proof using `p_m`, `p_e`, and T.rec_aux)
   ...
```

With these in hand, we can achieve a more compact translation of `fp` that does not reference `x`:

```
noncomputable def fpTrans2 (c : C 0) : f 0 c .a = .a :=
  L4L.castHEq
    (L4L.appArgHEq' (Eq (f 0 c .a))
      (f_aux 0 c .a rfl rfl))
    rfl
```

What's more, any other definitional equality making use of a reduction involving `f` can also use the lemma `f_aux` in producing a corresponding Lean$^-$ equality proof, instantiated with the appropriate arguments, eliminating redundant branches of proof in the translation output.

The type signatures of the auxiliary functions `T.rec_aux` and `f_aux` above mostly follow those of the functions they are derived from, outputting instead an equality proof between the original application and a particular reduced form, and additionally carrying the `p_*` "weakest precondition" (WP) arguments, denoting the minimum criteria on the application arguments for the reduction equality specified in the output type to hold[7].

We can also consider an alternate formulation where these WPs are instead integrated directly within the auxiliary definition's type signature:

[7]This example has been simplified for presentation purposes, and does not capture the full range of complexity in WP generation. In general, WPs represent requirements on the shape of the WHNF of their left-hand side, which must have some inductive type, with the right-hand side being some (potentially nested) constructor application. The RHS may introduce new variables representing (nested) constructor arguments and the LHS may apply argument subexpressions to one another. We must also handle the case of types dependent on arguments with WP conditions. See this file for a more comprehensive example: `https://github.com/Deducteam/Lean4Less/blob/auxdefs-example/Lean4Less/Fixtures/AuxDefs.lean`

```
def T.rec_aux' {n : Nat}
  {M : (m : Nat) → T n m → Sort u}
    (mtv : M n (T.mk n)) (t : T n n):
      @T.rec n M mtv n t == mtv
  := ...

noncomputable def f_aux' (c : C 0) : f 0 c .a == .a := ...

noncomputable def fpTrans2' (c : C 0) : f 0 c .a = .a :=
  L4L.castHEq
    (L4L.appArgHEq' (Eq (f 0 c .a))
      (f_aux' c))
    rfl
```

This "baked-in" WP format has the benefit of smaller, easier-to-construct proofs from the auxiliary lemmas which have smaller type signatures, which we can use in the (common) case when the WP conditions are already satisfied definitionally.

Two key questions still remain: how do we compute the bodies of these functions, and how do we compute their WP conditions? We suspect the best approach will be to compute them on an "on-demand" basis, in parallel to WHNF computations. WP generation may be achieved by some form of "tracing" on the arguments of δ-expandable function heads, keeping track of how they are reduced and eliminated upon through recursor reduction. The exact specifics of how this is to be done, however, remain to be investigated further.

7.2.2 Implications of a Verified Translation for Lean's Metatheory

Although Lean is technically not a conservative extension of Lean$^-$, we can show an adjacent property that is quite similar to conservativity, but which assumes the presence on `prfIrrel` in the Lean$^-$ typing context:

Theorem 7.2.1. For all terms T such that $\Delta \vdash T : \texttt{Sort}\ \ell$, if there is some term t such that $\Delta \vdash t : T$, then there is some term t' such that $(\texttt{prfIrrel}, \Delta) \vdash^- t' : T$.

Proof. By Theorem 5.1.1, we have that $(\texttt{prfIrrel}, \Delta) \vdash^- |T|^- : \texttt{Sort}\ \ell$ and $(\texttt{prfIrrel}, \Delta) \vdash^- |t|^- : |T|^-$. Additionally, since T is well-typed in Lean$^-$ and $T \sim |T|^-$, we have by Conjecture 5.1.2 that there exists some term p such that $(\texttt{prfIrrel}, \Delta) \vdash^- p : |T|^-\ \texttt{==}\ T$. Therefore, we have $(\texttt{prfIrrel}, \Delta) \vdash^- \texttt{cast}\ |T|^-\ T\ p\ |t|^- : T$. □

In light of this, we may wonder about the extent to which Lean$^-$ can be used as a "proxy metatheory" for Lean for the purpose of simplifying meta-theoretical analyses. A particularly interesting meta-property we would like to verify is the consistency property, which can be stated as follows:

Conjecture 7.2.1. There is no proof of the proposition `False` in a typing context consisting solely of the axiom `prfIrrel`, i.e.:

$$\neg\exists p, \texttt{prfIrrel} \vdash^- p : \texttt{False}$$

With the proposition `False` expressed in Lean as an inductive type with no constructors:

```
inductive False : Prop
```

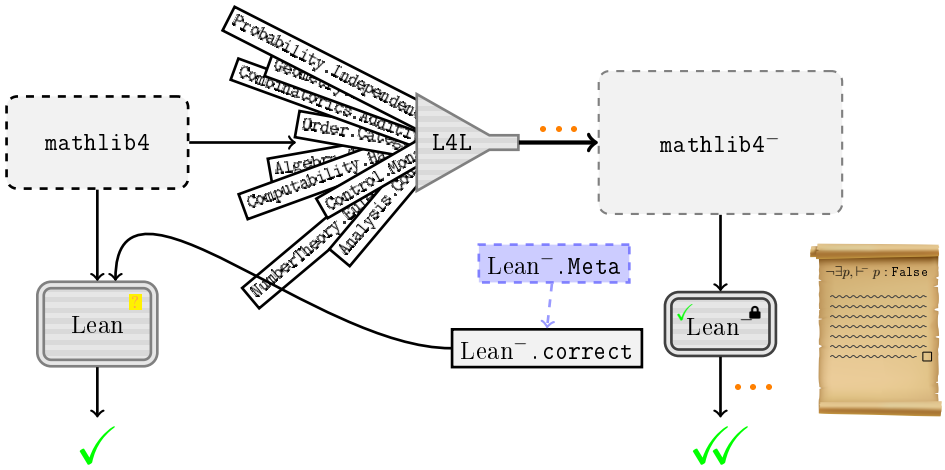

Figure 7.2: Translating entire libraries to be checked by a verified Lean$^-$ kernel.

In fact, such a proof cannot be formally verified in Lean itself due to Gödel's Incompleteness Theorem [21], which identifies a class of meta-properties (including consistency) which are not provable in any sufficiently expressive system. Rather, it would have to either be shown informally or in a different (more expressive) type system. Carneiro's Lean4Lean project [12] contains some meta-theoretical formalizations towards a proof of consistency of Lean's type theory, but efforts in this direction face significant difficulties that are particularly attributable to features such as definitional proof irrelevance and K-like reduction, as well as the more recent features of struct-η and struct-like reduction (whose elimination should also be under the scope of the Lean4Less translation). So, certain important meta-properties of Lean may be significantly easier to show in the smaller theory of Lean$^-$ where these problematic features have been removed. Additionally, previous examples of non-termination and undecidability of typechecking shown by Carneiro [12, 13] depend on the use of definitional proof irrelevance and K-like-reduction. These features do not exist in Lean$^-$, so it is an open question whether or not the same issues affect the smaller theory of Lean$^-$. We conjecture that both decidability of typechecking and termination may in fact hold – if still not entirely, then perhaps at least with much weaker assumptions – without K- and struct-like reduction.

Assuming we are able to show that Lean$^-$ is consistent, Lean$^-$ then becomes an ideal target for us to translate Lean formalizations to, in order to typecheck terms with this smaller, provably safe kernel. In this respect, it will also be interesting to formally prove that the Lean$^-$ kernel implementation is correct according to the Lean$^-$ type theory. To have this extra degree of assurance for large-scale formalizations such as Mathlib, it may at first seem necessary to hyper-optimize the Lean4Less translation so that it can effectively scale up to Mathlib-size formal libraries. This hypothetical process is visualized in Figure 7.2. As mentioned earlier, however, it is not certain that such scaling is even theoretically possible to begin with, as some scaling issues may be more or less inherent to the translation. However, if our only objective is to improve the confidence that we have in mathematics that has been formalized in Lean, we likely do not need to actually go this far, particularly if we are able to formally prove the correctness of our translation and the Lean kernel itself w.r.t. the Lean and Lean$^-$ metatheories. If we can show Theorem 7.2.1, we would know that any axiom-free proof of the proposition `False` in Lean would translate into a proof of `False` in Lean$^-$, and so, proving the consistency of Lean$^-$– assuming that the axiom `prfIrrel` is in the typing context, as stated in Conjecture 7.2.1 – would imply the consistency of Lean, as being able to prove `False` in Lean would imply that such a proof can then be translated to Lean$^-$, which is in contradiction with Conjecture 7.2.1. In combination with a proof of correctness of the Lean kernel w.r.t. the Lean metatheory, this would imply the consistency of the Lean kernel. Proving

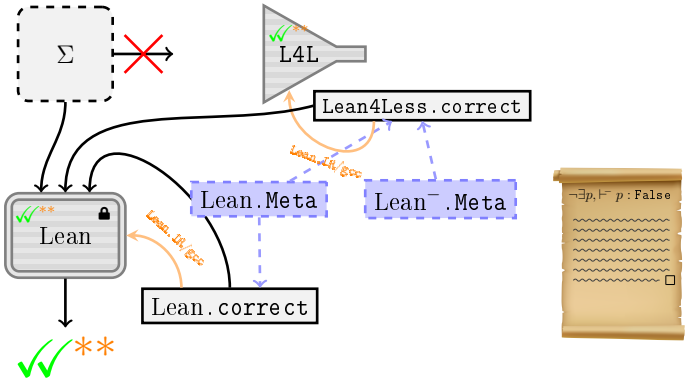

Figure 7.3: Verifying the correctness and consistency of the Lean kernel.

Theorem 7.2.1 formally in Lean could be done constructively via a proof of correctness of the translation implemented by Lean4Less. If we can formally show that Lean4Less's implementation ensures the property that on all well-typed, terminating Lean input environments, the translation terminates and produces well-typed, semantically equivalent Lean$^-$ output environments, we could then formally prove the equivalent of Theorem 5.1.1, Conjecture 5.1.2, and finally Theorem 7.2.1 in Lean.

Doing this kind of formalization would effectively justify the continued use of the Lean kernel, allowing us to be almost just as confident in using the original Lean kernel to typecheck large formalizations as we would be if we were to translate the entire libraries to be typechecked with the Lean$^-$ kernel. This strategy is visualized in Figure 7.3.

Attempting a proof of translation correctness could also be quite interesting in relation to identifying potential consistency bugs in Lean's kernel implementation. The ability to eliminate a certain definitional equality from Lean testifies to the fact that its inclusion as a convenience for formalization does not expand the class of provable propositions in any meaningful way, so in particular there is no possibility of it introducing consistency issues. If we encounter any significant difficulties in trying to construct this proof, it may in fact be the case that Conjecture 7.2.1 is not provable w.r.t. the type theory that is actually implemented by the Lean kernel, pointing to a possible issue in the kernel implementation of one of the eliminated definitional equalities that renders Lean inconsistent.

However, this approach carries a couple caveats (as indicated by the asterisks in the Figure 7.3). Firstly, it requires some additional trust in the stack of tooling used to compile the kernel program-down into machine-executable bytecode. Without the use of a formally verified compilation stack, we cannot be entirely sure that the operational semantics of the Lean kernel, which was originally expressed and proven correct as a Lean program, have been preserved through its conversion first to Lean's IR, then C code, and then to executable bytecode.

Secondly, this approach is somewhat unsatisfactory, because part of the proof of the Lean kernel's consistency (that is, the proofs of correctness of the translation and the Lean kernel) would be checked by the `same` kernel, rather than the Lean$^-$ kernel, whose underlying theory we have directly shown consistency for. However, there is a way for us to address this issue. If we could formally show the correctness of the Lean$^-$ kernel as well, we could use Lean4Less to translate this proof, as well as the other two program proofs, to be checked by the verified Lean$^-$ kernel. This can be visualized as follows:

With this approach, we would still want our translation to be practical to *some* extent, but only insofar as it is capable of translating these select few proofs to Lean$^-$.

Additionally, in the opposite direction, we can think about *extending* Lean's type theory and

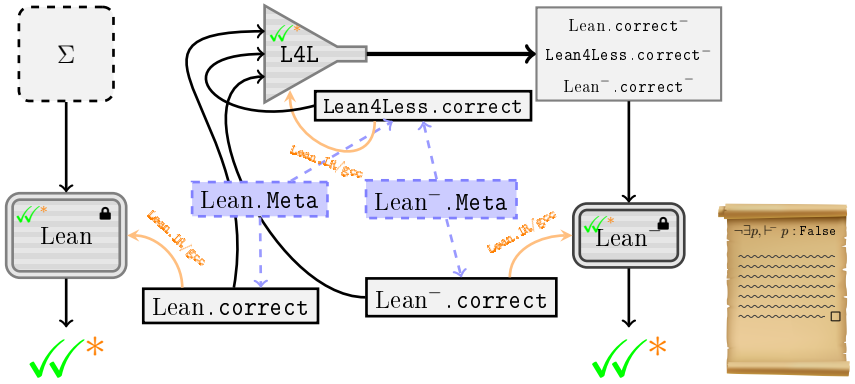

Figure 7.4: Verifying the correctness and consistency of the Lean kernel against the Lean$^-$ kernel.

typechecker kernel in certain conservative ways to obtain some hypothetical theory Lean$^+$, and proving the correctness of a translation from Lean$^+$ to Lean. Such a translation could then be composed with the translation from Lean to Lean$^-$ for a provably correct translation from Lean$^+$ to Lean$^-$, with this proof also translated to Lean$^-$ via the translation from Lean$^-$ to Lean[8], giving us the confidence to reason henceforth in this more powerful theory (again, at the cost of some extra trust in the stack of tooling that generates the machine code from the formal representation of the implementation of the Lean$^+$ kernel in Lean).

Along these lines, we can also consider formally verifying a translation from Lean$^-$ to Dedukti, using Dedukti instead as the base theory for reasoning about consistency results. Because Dedukti's type theory is even more minimal than Lean$^-$'s, showing the consistency of our Lean encoding in Dedukti may be simpler than reasoning about Lean$^-$ itself, with Lean's special reduction rules (e.g. those for recursor reduction) becoming explicit Dedukti rewrite rules. Additionally, existing tooling that has been developed for Dedukti, such as confluence checkers, may be helpful in showing consistency results.

7.2.3 Extensionality in Lean

Given that Lean4Less's implementation is motivated by a general ETT-to-ITT translation, an interesting prospect is the possibility of adapting it for the purpose of translating Lean terms from some more powerful theory that has been extended with additional definitional equalities back to the original theory. Indeed, as Lean4Less is implemented in Lean, it could be modified to make it capable of eliminating general, user-defined equalities beyond those already defined in Lean. That is, it could accept input terms from some hypothetical user-defined extensional theory "Lean$_{e*}$", which is Lean extended with some kind of limited equality reflection rule:

$$\frac{\Delta \vdash^-_{e*} A : \texttt{Sort}\ \ell \quad \Delta \vdash^-_{e*} t, u : A \quad \text{compeq}(\Delta, A, t, u)}{\Delta \vdash^-_{e*} t \equiv u}\ [\text{RFL*}]$$

where the compeq(Γ, A, t, u) criteria states that, in context Γ, t `==` u is provable automatically in Lean, due to it having been registered directly by a user, or being derivable from other registered equalities. The Lean kernel itself could then be extended to accept user-defined extensional equalities, with the assurance that it will be possible to translate it back to Lean's theory via the modified

[8] Alternatively, if this proof is first expressed in Lean$^+$ itself, it can be translated to Lean$^-$ via a composite translation first from Lean$^+$ to Lean, then from Lean to Lean$^-$.

Lean4Less translation. On the other hand, if we wish to continue using the current Lean kernel, another option is to integrate Lean4Less with existing elaboration routines to allow for a real-time translation that would simulate native kernel support for extensional reasoning.

Regarding the user input of extensional equalities, it will be important to distinguish between "directed" and "undirected" equalities. Undirected equalities are analogous to proof irrelevance, unit-η and function-η in Lean, (and are implemented in the Lean4Lean typechecker kernel's `isDefEqCore` function). Suppose we have a hypothetical constant annotation `@[deq]` thats marks an equality theorem as an extensional definitional equality that is "known" to the kernel. This would allow us to prove the following theorem by reflection:

```
@[deq]
theorem addComm (x y : Nat) : x + y = y + x := ...
example (x y z : Nat) : x + (y + z) = x + (z + y) := rfl
```

Here, Lean checks the definitional equality of the arguments in turn, invoking [RFL*] via `addComm` on the second argument of the outermost addition. However, an undirected definitional equality would *not* allow us to prove:

```
-- (Lean's addition function matches on the second argument,
-- so this does not hold definitionally)
@[deq]
theorem incEq (x : Nat) : 1 + x = Nat.succ x := ...
-- cannot be proven with `rfl`
example (x y : Nat) : y + (1 + a) = Nat.succ (y + a) := sorry
```

The problem is the following: for the outermost application to reduce, the second argument's weak-head normal form must be an application of `Nat.succ`, which is not the case for `1 + a`. While `1 + a` is definitionally equal to `Nat.succ a` by `incEq`, this equality does not apply when computing its weak-head normal form.

For this, we instead require a "directed equality" (a.k.a. "rewrite rule") that can be applied during reduction, allowing us to "rewrite" the addition to a constructor application. Let us use the hypothetical annotation `@[drw]` to register a directed extensional equality theorem, enabling here a proof by `rfl`:

```
@[drw]
theorem incEq (x : Nat) : 1 + x = Nat.succ x := ...
example (x y : Nat) : y + (1 + a) = Nat.succ (y + a) := rfl
```

Directed equalities may seem to be strictly more powerful than undirected ones, but they are only practically applicable as long as they satisfy the properties of termination and confluence, which are well-studied in other systems such as Dedukti [10] where rewrite rules are built-in. Without a terminating set of rewrite rules, typechecking/elaboration will also not terminate (for instance, it would not be acceptable to register the commutativity of addition as a directed equality). Confluence is an important property in ensuring that the user-provided reduction rules are unambiguous – in particular, it ensures that definitional equality checking via comparison of normal forms effectively decides the equational theory that they define[9]. Termination and confluence must also be considered in light of the reduction rules that Lean natively implements, namely those of recursor, K-like, struct-like and quotient reduction.

[9]The equational theory defined by a rewrite system is the reflexive, transitive, symmetric closure of the relation between terms defined by the individual rewrite rules.

Regarding Lean's `cc` / `grind` Tactic

Some of the functionality suggested above for allowing Lean to decide a larger class of definitional equalities may be reminiscent of automation already present in Lean for congruence closure [34], which was first introduced in Lean 3's `cc` tactic, and more recently superseded by Lean 4's `grind` tactic. Lean's congruence closure procedure uses a powerful algorithm widely used by SMT solvers that attempts to find an equality proof between two specified terms, taking local equality assumptions into consideration. The fact that automation already exists for this purpose may bring up some questions regarding what potential "benefits" an approach for equality proof reconstruction based on an extensional-to-intensional translation may have over existing, more well-established approaches such as congruence closure.

For instance, one could imagine translating from an "extensional" version of Lean in which the elaborator automatically calls a congruence closure algorithm whenever a typing discrepancy is encountered, and, if the algorithm returns a proof, uses the returned proof to "patch up" the discrepancy via a type cast (as is already implemented in Lean4Less) to help build a finally elaborated term. Such an approach could work in principle, however from a practical perspective it is hardly reasonable. An implicit trade-off that many proof assistant kernels have to make is between providing convenient automation that allows the kernel to identify as many equal terms as possible (avoiding the need for users to manually provide equality proofs), and providing timely negative feedback in the event of a typing error. From a user perspective, it would be unacceptable to call the equivalent of Lean's `grind` tactic to try to resolve every single instance of a typing discrepancy that is encountered. These tactics are much better suited for when the user already heavily suspects that equality can be proven beforehand.

The approach we suggest is rather to extend the existing kernel `isDefEq` routine in simple, limited, and efficient ways, allowing it to identify a larger class of provably equal terms while minimally sacrificing the responsiveness of the system in the event of ill-typedness. The proof reconstruction algorithm we could implement for translating from this extensional version of Lean could then simply extend on the implementation we already have for Lean4Less's `isDefEq` function. While this may not cover as much as an approach based on a full-blown congruence closure algorithm in terms of enabling more definitional equalities, it could be a very reasonable compromise allowing for some level of user-specified definitional equalities while still providing timely negative feedback to the user.

Conclusion

In this thesis, we have described several aspects of the theory, design, and implementation behind a tool for translating proofs from the Lean proof assistant to the Dedukti logical framework, for the ultimate purpose of exporting proofs from Lean to other proof assistants. Guided by certain theoretical correctness requirements, we have derived a base encoding of Lean in the $\lambda\Pi/R$ type theory of Dedukti, along with a two-step translation consisting of an initial translation from Lean to a smaller theory Lean$^-$, followed by a translation from Lean$^-$ to Dedukti.

Designing this translation required special attention to the specifics of Lean's type theory, in particular its use of specialized definitional equalities such as proof irrelevance and K-like reduction. Lean's use of universe polymorphism and impredicativity also required us to derive an encoding of universe level terms in Dedukti that allowed for Dedukti to decide equivalence of the translated universe level terms through the computation of normal forms. The translation from Lean$^-$ to Dedukti is based directly on previous work defining a generic translation from a class of "pure type systems" to Dedukti, with some extensions to the translation to account for Lean's particular definitional equality rules. The preliminary translation step from Lean to Lean$^-$ was adapted from previous work on the translation from extensional and intensional type theory, and implemented in the tool "Lean4Less", which itself is based on a typechecker kernel implementation for Lean. We

also describe a number of output optimizations that we have implemented for Lean4Less to help make the translation somewhat practical.

While a great deal of work remains to be done to make the translation as a whole capable of scaling to large input libraries, we have already had some notable success in translating moderately sized Lean libraries to Lean$^-$ and Dedukti. We have also described some possible avenues to consider to help make the translation more practical, as well as some interesting possible implications our translation could have for certain meta-theoretical analyses. Overall, we believe that the work described in this thesis lays the foundation for the export of Lean to other proof assistants through the intermediate Dedukti logical framework, eventually enabling improved interoperability and making Lean formalizations accessible to a wider array of formal systems.

Bibliography

[1] Andreas Abel and Thierry Coquand. "Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality". In: *Logical Methods in Computer Science* Volume 16, Issue 2, 14 (June 2020). ISSN: 1860-5974. DOI: 10.23638/LMCS-16(2:14)2020.

[2] Stuart Allen et al. "The Nuprl Open Logical Environment". In: Dec. 2006, pp. 170–176. ISBN: 978-3-540-67664-5. DOI: 10.1007/10721959_12.

[3] Andrea Asperti et al. "The Matita interactive theorem prover". In: *Proceedings of the 23rd International Conference on Automated Deduction*. CADE'11. Wroc a w, Poland: Springer-Verlag, 2011, pp. 64–69. ISBN: 9783642224379.

[4] Ali Assaf. "A calculus of constructions with explicit subtyping". In: *LIPICS*. Ed. by Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau. Vol. 39. LIPICS. Institut Henri Poincaré, Paris, France, May 2014. URL: https://hal.science/hal-01097401.

[5] Ali Assaf. "A framework for defining computational higher-order logics". Theses. École polytechnique, Sept. 2015. URL: https://pastel.hal.science/tel-01235303.

[6] Henk Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013.

[7] Andrej Bauer et al. "Design and Implementation of the Andromeda Proof Assistant". In: *22nd International Conference on Types for Proofs and Programs (TYPES 2016)*. Ed. by Silvia Ghilezan, Herman Geuvers, and Jelena Ivetic. Vol. 97. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018, 5:1–5:31. ISBN: 978-3-95977-065-1. DOI: 10.4230/LIPIcs.TYPES.2016.5.

[8] Frédéric Blanqui. "Encoding Type Universes Without Using Matching Modulo Associativity and Commutativity". In: *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*. Vol. 228. 7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022). Haifa, Israel, Aug. 2022, p. 14. DOI: 10.4230/LIPIcs.FSCD.2022.24. URL: https://inria.hal.science/hal-03708036.

[9] Frédéric Blanqui. *Type theory and rewriting*. 2001. URL: https://inria.hal.science/inria-00105525.

[10] Frédéric Blanqui et al. "A modular construction of type theories". In: *Logical Methods in Computer Science* Volume 19, Issue 1, 12 (Feb. 2023). ISSN: 1860-5974. DOI: 10.46298/lmcs-19(1:12)2023.

[11] Ana Bove, Peter Dybjer, and Ulf Norell. "A Brief Overview of Agda — A Functional Language with Dependent Types". In: *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*. TPHOLs '09. Munich, Germany: Springer-Verlag, 2009, pp. 73–78. ISBN: 9783642033582. DOI: 10.1007/978-3-642-03359-9_6. URL: https://doi.org/10.1007/978-3-642-03359-9_6.

[12] Mario Carneiro. *Lean4Lean: Towards a formalized metatheory for the Lean theorem prover*. 2024. arXiv: 2403.14064 [cs.PL]. URL: https://arxiv.org/abs/2403.14064. Github repository: https://github.com/digama0/lean4lean.

[13] Mario Carneiro. "The Type Theory of Lean". MA thesis. 2019. URL: https://github.com/digama0/lean-type-theory/releases/tag/v1.0.

[14] Thierry Coquand and Gérard Huet. "The calculus of constructions". In: *Information and Computation* 76.2 (1988), pp. 95–120. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(88)90005-3. URL: https://www.sciencedirect.com/science/article/pii/0890540188900053.

[15] Thiago Felicissimo. "Adequate and Computational Encodings in the Logical Framework Dedukti". In: *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*. Ed. by Amy P. Felty. Vol. 228. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 25:1–25:18. ISBN: 978-3-95977-233-4. DOI: 10.4230/LIPIcs.FSCD.2022.25. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2022.25.

[16] Thiago Felicissimo, Frédéric Blanqui, and Ashish Kumar Barnawal. "Translating proofs from an impredicative type system to a predicative one". In: *31st EACSL Annual Conference on Computer Science Logic (CSL 2023)*. Warsaw, Poland, 2023. DOI: 10.4230/LIPIcs.CSL.2023.19. URL: https://inria.hal.science/hal-03848584.

[17] Gaspard Ferey. "Higher-Order Confluence and Universe Embedding in the Logical Framework". Theses. Université Paris-Saclay, June 2021. URL: https://theses.hal.science/tel-03418761.

[18] Guillaume Genestier. "Encoding Agda Programs Using Rewriting". In: *FSCD - 5th International Conference on Formal Structures for Computation and Deduction*. Paris, France, June 2020. DOI: 10.4230/LIPIcs.FSCD.2020.31. URL: https://inria.hal.science/hal-03838613.

[19] Yoan Géran. "Encoding impredicative hierarchy of type universes with variables". working paper or preprint. Nov. 2023. URL: https://hal.science/hal-04311936.

[20] J.Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Éditeur inconnu, 1972. URL: https://books.google.fr/books?id=IRcVHAAACAAJ.

[21] Kurt Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I". German. In: *Monatshefte für Mathematik und Physik* 38 (1931), pp. 173–198.

[22] Martin Hofmann. "Conservativity of Equality Reflection over Intensional Type Theory". In: *Selected Papers from the International Workshop on Types for Proofs and Programs*. TYPES '95. Berlin, Heidelberg: Springer-Verlag, 1995, pp. 153–164. ISBN: 3540617809.

[23] Martin Hofmann and C. J. Rijsbergen. *Extensional Constructs in Intensional Type Theory*. Berlin, Heidelberg: Springer-Verlag, 1997. ISBN: 3540761217.

[24] William A. Howard. "The formulae-as-types notion of construction". In: 1969. URL: https://api.semanticscholar.org/CorpusID:118720122.

[25] Joe Hurd. "The OpenTheory Standard Theory Library". In: *NASA Formal Methods*. Ed. by Mihaela Bobaru et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 177–191. ISBN: 978-3-642-20398-5.

[26] Xavier Leroy. "Well-founded recursion done right". In: *CoqPL 2024: The Tenth International Workshop on Coq for Programming Languages*. ACM. London, United Kingdom, Jan. 2024. URL: https://inria.hal.science/hal-04356563.

[27] Leonardo de Moura and Sebastian Ullrich. "The Lean 4 Theorem Prover and Programming Language". In: *Automated Deduction – CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 625–635. ISBN: 978-3-030-79875-8. DOI: 10.1007/978-3-030-79876-5_37.

[28] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Jan. 2002. ISBN: 9783540433767. DOI: 10.1007/3-540-45949-9.

[29] Nicolas Oury. "Extensionality in the calculus of constructions". In: *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*. TPHOLs'05. Oxford, UK: Springer-Verlag, 2005, pp. 278–293. ISBN: 3540283722. DOI: 10.1007/11541868_18.

[30] Christine Paulin-Mohring. "Introduction to the Calculus of Inductive Constructions". In: *All about Proofs, Proofs for All*. Ed. by Bruno Woltzenlogel Paleo and David Delahaye. Vol. 55. Studies in Logic (Mathematical logic and foundations). College Publications, Jan. 2015. URL: https://inria.hal.science/hal-01094195.

[31] Rocq Community. *The Rocq Theorem Prover*. URL: https://rocq-prover.org/.

[32] B. Russell. "Les Paradoxes de la Logique". In: *Revue de Métaphysique et de Morale* 14.5 (1906), pp. 627–650.

[33] Ronan Saillard. "Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice". Theses. Ecole Nationale Supérieure des Mines de Paris, Sept. 2015. URL: https://pastel.hal.science/tel-01299180.

[34] Daniel Selsam and Leonardo de Moura. "Congruence Closure in Intensional Type Theory". In: *CoRR* abs/1701.04391 (2017). arXiv: 1701.04391.

[35] Matthieu Sozeau et al. "Coq Coq correct! verification of type checking and erasure for Coq, in Coq". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371076.

[36] SRI International Computer Science Laboratory. *Prototype Verification System (PVS)*. https://pvs.csl.sri.com/. Accessed: 2026-01-12. 2026.

[37] Nikhil Swamy et al. "Dependent types and multi-monadic effects in F*". In: *SIGPLAN Not.* 51.1 (Jan. 2016), pp. 256–270. ISSN: 0362-1340. DOI: 10.1145/2914770.2837655.

[38] The mathlib community. "The lean mathematical library". In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020. New Orleans, LA, USA: Association for Computing Machinery, 2020, pp. 367–381. ISBN: 9781450370974. DOI: 10.1145/3372885.3373824. URL: https://doi.org/10.1145/3372885.3373824.

[39] François Thiré. "Interoperability between proof systems using the logical framework Dedukti". Theses. Université Paris-Saclay, Dec. 2020. URL: https://hal.science/tel-03224039.

[40] François Thiré. "Sharing a Library between Proof Assistants: Reaching out to the HOL Family". In: *Electronic Proceedings in Theoretical Computer Science* 274 (July 2018), pp. 57–71. ISSN: 2075-2180. DOI: 10.4204/eptcs.274.5. URL: http://dx.doi.org/10.4204/EPTCS.274.5.

[41] Thomas Traversié. "Proofs for Free in the λΠ-Calculus Modulo Theory". In: *Electronic Proceedings in Theoretical Computer Science* 404 (July 2024), pp. 49–63. ISSN: 2075-2180. DOI: 10.4204/eptcs.404.4. URL: http://dx.doi.org/10.4204/EPTCS.404.4.

[42] Andrzej Trybulec and Howard Blair. "Computer assisted reasoning with MIZAR". In: *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*. IJCAI'85. Los Angeles, California: Morgan Kaufmann Publishers Inc., 1985, pp. 26–28. ISBN: 0934613028.

[43] Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. "Eliminating reflection from type theory". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs* (2019). URL: https://api.semanticscholar.org/CorpusID:57379755.

[44] Théo Winterhalter and Nicolas Tabareau. *ett-to-itt (Github)*. URL: %7Bhttps://github.com/TheoWinterhalter/ett-to-itt%7D.

Titre: Traduction de Preuves de Lean vers Dedukti

Mots clés: lambda-pi calcul modulo réécriture, interopérabilité des systèmes de preuve, la théorie des types

Résumé: Les assistants de preuve sont des logiciels qui permettent l'expression précise d'objets, de propriétés et de preuves mathématiques, et vérifient la validité des preuves.

Au cours des dernières décennies, plusieurs assistants de preuve ont été développés, chacun disposant de sa propre communauté et de ses bibliothèques. Malheureusement, le partage des résultats entre différents assistants s'avère souvent complexe car leur syntaxe et leurs théories sous-jacentes diffèrent de manière significative. Ceci peut entraîner une duplication du travail, les mêmes théories mathématiques étant formalisées indépendamment dans différents systèmes. Idéalement, nous souhaiterions disposer d'outils permettant la traduction des preuves entre différents systèmes.

Cette thèse décrit la traduction des preuves de l'assistant de preuve Lean vers le cadre logique Dedukti. Lean est un assistant de preuve qui a gagné en popularité auprès des mathématiciens ces dernières années grâce à sa bibliothèque «mathlib» contenant un vaste corpus de formalisations mathématiques en constante expansion. La théorie des types de Lean repose sur le calcul des constructions avec types inductifs, s'inspirant fortement de l'assistant de preuve Rocq. Dedukti est un cadre logique conçu pour faciliter l'exportation de preuves entre différents systèmes, avec une théorie des types basée sur le lambda-pi-calcul modulo réécriture.

Notre traduction s'appuie sur un encodage générique des «systèmes de types purs» en Dedukti. À partir de cet encodage de base, nous définissons une traduction des termes de Lean vers les termes de Dedukti, en tenant compte également de certaines spécificités du typage de Lean, notamment l'utilisation de certaines «égalités de définition» pour accroître sa puissance expressive en élargissant la classe des types que le noyau peut considérer comme équivalents. La gestion de ces égalités définitionnelles consiste généralement à ajouter des règles de réécriture. Nous ajoutons également des composants à notre encodage permettant à Dedukti de déterminer efficacement l'équivalence des expressions d'univers, compte tenu de la prise en charge par Lean du polymorphisme d'univers et de son univers propositionnel imprédicatif.

La traduction directe que nous définissons de Lean vers Dedukti ne prend cependant en compte qu'un sous-ensemble particulier des égalités définitionnelles de Lean. Certaines égalités spécifiques à Lean, notamment celles d'irrelevance des preuve, d'êta pour les types unités et de «réduction de type K», ne se prêtent malheureusement pas à un encodage direct via des règles de réécriture. Pour les gérer, nous intégrons une «étape de prétraduction» à notre pipeline, traduisant d'abord les termes Lean vers une théorie plus petite, «Lean$^-$», qui ne présente pas ces identités problématiques. Notre approche s'inspire d'une traduction générique de la théorie des types extensionnelle vers la théorie des types intensionnelle, que nous adaptons pour traduire de Lean vers Lean$^-$ en modifiant un noyau Lean afin d'effectuer la traduction en parallèle de la vérification de types classique.

La traduction que nous avons implémentée a donné des résultats préliminaires prometteurs, malgré la persistance de certains défis, notamment concernant son passage à l'échelle pour fonctionner avec des bibliothèques plus volumineuses. Les techniques de traduction décrites dans cette thèse suggèrent également des pistes de recherche intéressantes, avec des implications potentielles pour les développements futurs de Lean, notamment en ce qui concerne les extensions de noyau possibles et la vérification de certains résultats métathéoriques importants relatifs à la théorie des types de Lean.

Title: Translating Proofs from Lean to Dedukti
Keywords: lambda-pi calculus modulo rewriting, proof system interoperability, type theory

Abstract:

Proof assistants are software tools that enable the precise expression of mathematical objects, properties, and proofs, and include automation that checks the validity of proofs with respect to specific theorem statements. Proof assistants often feature a high degree of expressivity, allowing users formally state and verify many interesting results, spanning from foundational to research-level mathematics.

Over the past few decades, a number of proof assistants have been developed, each with their own communities and formal mathematical libraries. Unfortunately, however, it is often difficult to share results between different proof assistants, as they usually differ significantly in their syntax and underlying theories. This can lead to the duplication of work, with the same mathematical theories being independently formalized in different systems. Ideally, we would like to have tools that translate proofs between different systems, allowing for the automated export of mathematical proofs in such a way that they can be independently verified by other systems.

This thesis describes the topic of translating proofs from the Lean proof assistant to the logical framework known as Dedukti. Lean is a proof assistant developed by the Lean FRO that has become quite popular with mathematicians in recent years, with its "mathlib" library containing a large and growing body of formalized mathematics. Lean's type theory is based on the Calculus of Constructions with inductive types, taking closely after the proof assistant Rocq. Dedukti is a logical framework designed to facilitate the export of proofs between different systems, with a type theory based on the lambda-pi calculus with rewrite rules.

The basis of our translation is derived from a generic encoding of so-called "pure type systems" into Dedukti, with Lean being interpretable as a particular kind of pure type system. Starting from this base encoding, we define a translation from Lean terms to Dedukti terms, also taking into account some additional features of Lean's typing, notably around Lean's use of certain "definitional equalities" to augment its expressive power by expanding the class of types that the kernel is able to consider equivalent. Handling these definitional equalities usually consists in adding corresponding rewrite rules to our encoding, enabling the Dedukti kernel to identify the translation of terms that were originally identified by the Lean kernel. We also add components to our encoding enabling it to effectively decide equivalence between universe level expressions in light of Lean's support for prenex universe level polymorphism and its impredicative propositional universe.

The direct translation we define from Lean to Dedukti, however, only accounts for a particular subset of Lean's definitional equalities. Some specific definitional equalities in Lean, namely those of proof irrelevance, unit-η, and "K-like reduction", unfortunately do not give way to a straightforward encoding via Dedukti rewrite rules. To handle these, we incorporate a "pre-translation step" into our pipeline, initially translating Lean terms to a smaller theory "Lean$^-$" that does not feature these problematic identities. Our approach takes inspiration from a generic translation from extensional to intensional type theory, which we adapt for the purpose of translating from Lean to Lean$^-$ by modifying a Lean kernel to effect translation in parallel to normal typechecking.

The translation we have implemented has yielded some promising preliminary results, though certain challenges remain, particularly around scaling the translation to work with larger libraries. The translation techniques we describe in this thesis also suggest some interesting future directions for this work, with possible implications for future developments in Lean around possible kernel extensions as well as the verification of certain important meta-theoretical results relating to Lean's type theory.

Maison du doctorat de Université Paris-Saclay
2e étage, aile ouest, École normale supérieure Paris-Saclay
4 avenue des Sciencs
91190 Gif-sur-Yvette, France