M1 Internship
Report

Translating proofs between Isabelle and Dedukti

Yann Leray
Département d'informatique
École Normale Supérieure

Supervised by
Frédéric Blanqui

Deducteam, Inria Saclay team in association with ENS Paris-Saclay
4, avenue des Sciences, 91190 Gif-sur-Yvette

March - July 2021

Contents

1 Introduction

Proof systems are computer programs used to write and prove mathematical statements. They are mostly used in two different domains: mathematics and program verification. In maths, they allow the formalisation of theories, laying down all definitions, axioms, theorems and (most importantly) verifying the proofs users write. Similarly, in program verification, users can provide specifications and write verifiable proofs that this program does follow those specifications.

Formal proofs need extreme detail so that they can be precisely checked against the axioms. To make the process more bearable, most proof systems are also assistants which can do reasoning steps automatically or allow the proof to have a different more natural organisation.

Multiple proof systems were developed independently, which lead to multiple incompatible logics being used. This also has the consequence that most logics have only one implementation of the proof checker, which means that one can only trust the results so long as this single implementation is correct, and this applies for every proof system.

A solution to this problem would be to use a universal logic, so weak it can support all others, and a set of translations (one to and one from) every proof system. This way, the needed trust is shared by all supported logics and proof systems, all while only needing a linear number of translations.

This solution is being actively developed by the Deducteam with whom I did my internship. I worked on the translations between the universal proof system the team develops, Dedukti, and one of the most widely used proof assistant, Isabelle.

1.1 Dedukti and Lambdapi

General presentation

Dedukti [1, 2] is the proof system developed by the Deducteam. It is based on the $\lambda\Pi$-calculus modulo theories (shortened to $\lambda\Pi/\equiv$) which I will describe below. The Deducteam also works on a new version of Dedukti with proof assistant features and a syntactic revision, which they named Lambdapi. Because their underlying logic is the same and Lambdapi's additional features weren't useful to my work, they can be understood as two equivalent syntaxes for $\lambda\Pi/\equiv$.

$\lambda\Pi$-calculus, as the name suggests, is an extension of λ-calculus which adds 'Π-constructions': dependent products. This allows for types depending on regular values, the prime example being the `Vector` type family, a family indexed on natural numbers which has elements having their size included in their types. Constructors for this type family would be :

$$\texttt{Nil} : \texttt{Vector 0} \qquad \texttt{Cons} : \Pi(n : \texttt{nat}) \cdot A \to \texttt{Vector}\ n \to \texttt{Vector}(S\ n)$$

with A being the type of elements of the vector (A is fixed here, we cannot quantify nor depend on types within $\lambda\Pi/\equiv$ but there exists a workaround) and S being the 'successor' constructor on natural numbers.

The second part of the name, 'modulo theories', refers to the addition of rewriting to $\lambda\Pi$-calculus. Rewriting refers to constructs of the form $c\ arg_1\ arg_2\ \cdots \longrightarrow t'$; they should be understood as (potentially partial) definitions of c. It can be understood as pattern matching, with some support for non-left-linear rules and higher-order unification, but these last concerns are out of the scope of this report. The identification of the terms on both sides is then taken into account for further matches or unifications, including type unifications : two types which are congruent modulo the reflexive transitive contextual closure of the rewriting rule are now equal, as if they were β-equivalent.

Theorems in $\lambda\Pi/\equiv$: the Curry-Howard equivalence

To construct propositions, one needs atoms which are here constants that were defined previously and the usual logical connectors like $=, \top, \bot, \neg, \implies, \wedge, \vee, \forall, \exists$. They can be typed ($= : \iota \to \iota \to prop$, $\top, \bot : prop$, $\neg : prop \to prop$, $\implies, \wedge, \vee : prop \to prop \to prop$, $\forall, \exists : (\iota \to prop) \to prop$, ι being the type of non-proposition values, a version with more types will be presented in section 2.2) but it would be nice to give them some inherent value to express some of their relations.

The Curry-Howard equivalence, also known as the propositions-as-types principle, states that a natural interpretation for propositions is to see them as types whose elements are then the proofs of the proposition. Proving a proposition is then equivalent to constructing an expression (then named proof-term) whose type will be that proposition.

Because of the weakness of the $\lambda\Pi/\equiv$ typing system, it is not possible to strictly adhere to the principle: it is impossible to have elements of a type be types themselves. Fortunately, an easy solution is to declare a constructor Prf relating propositions to the type which will contain their proofs; to prove a proposition p is now to construct an expression whose type will be $\mathrm{Prf}\, p$.

With Curry-Howard in mind, parallels between proposition connectors and type constructors can be made which illustrate the elegance of it all :

- $\mathrm{Prf}(A \implies B)$ can rewrite to $(\mathrm{Prf}\, A) \to (\mathrm{Prf}\, B)$ because proving $A \implies B$ is the same as being able to provide a proof-term proving B when one is given a proof-term proving A;
- $\mathrm{Prf}(\forall P)$, usually written $\mathrm{Prf}(\forall x : \iota \cdot P\ x)$ can rewrite to $\Pi(x : \iota) \cdot \mathrm{Prf}(P\ x)$ because proving $\forall x : \iota \cdot P\ x$ is the same as being able to provide a proof-term proving $P\ x$ for every x;
- $\mathrm{Prf}\, \top$ should be a type with an obvious element, it is usually defined as $\mathrm{Prf}(\forall z : prop \cdot z \Rightarrow z)$ (which is further rewritten using the previous rules) with the proof-term being the identity on proof-terms;
- $\mathrm{Prf}\, \bot$ should be a type with no element; to express the 'ex falso quodlibet' principle it is often defined as $\mathrm{Prf}(\forall z : prop \cdot z)$ so that a proof would be able to prove any proposition;
- $\neg p$ is synonymous with $p \implies \bot$, so $\mathrm{Prf}(\neg p)$ rewrites to $\mathrm{Prf}(p \implies \bot)$
- $\mathrm{Prf}(a \wedge b)$ would be the product type of proofs of a and proofs of b, but there's no builtin product in $\lambda\Pi/\equiv$, so we resort to currification, giving $\mathrm{Prf}(\forall z : prop \cdot (a \Rightarrow b \Rightarrow z) \Rightarrow z)$
- $\mathrm{Prf}(a \vee b)$ would be the sum type of proofs of a and proofs of b, but there's no builtin sum type constructor in $\lambda\Pi/\equiv$, so we do a similar transformation as $\wedge$, giving $\mathrm{Prf}(\forall z : prop \cdot (a \Rightarrow z) \Rightarrow (b \Rightarrow z) \Rightarrow z)$
- $\mathrm{Prf}(\exists x : \iota \cdot P\ x)$ is difficult to interpret naturally, but we can do a similar transformation as before, giving $\mathrm{Prf}(\forall z : prop \cdot (\forall x : \iota \cdot P\ x \Rightarrow z) \Rightarrow z)$

When using the Curry-Howard equivalence, checking a proof becomes the same as typing an expression, so any translation must heavily focus on the type system.

Typing rules of $\lambda\Pi/\equiv$

Figure 1 lists the typing rules of the $\lambda\Pi$-calculus modulo theories, which is the reference I will need for the formalisation of the translations and the proofs of correction. I also provided the typing rules for regular λ-calculus in fig. 5, for comparison. Let me now quickly explain the typing rules, so you can better understand how $\lambda\Pi/\equiv$ works:

In the following, Σ stands for signatures, $\mathcal{R}$ stands for a set of rewriting rules, 'full-sig' is a short-hand name for a pair (signature, rewriting rules set), s stands for either `Type` or `Kind` and $w.f.$ stands for 'well-formed'.

Rules about context and signatures

$$\frac{\Sigma, \mathcal{R}\ w.f.\ \text{full-sig}}{\vdash^{\Sigma}_{\mathcal{R}}\ []\ w.f.}\ \varnothing\text{-CON} \qquad \frac{[] \vdash^{\Sigma}_{\mathcal{R}} A : s \qquad c \notin \operatorname{dom}\Sigma}{(\Sigma, c : A), \mathcal{R}\ w.f.\ \text{full-sig}}\ \text{DECL} \qquad \frac{\Gamma \vdash^{\Sigma}_{\mathcal{R}} A : \texttt{Type}}{\vdash^{\Sigma}_{\mathcal{R}} \Gamma, x : A\ w.f.}\ \text{NEW-VAR}$$

$$\frac{\Gamma\ w.f. \qquad (x : A) \in \Sigma}{\Gamma \vdash^{\Sigma}_{\mathcal{R}} x : A}\ \text{CONST} \qquad \frac{\Gamma\ w.f.}{\Gamma \vdash^{\Sigma}_{\mathcal{R}} \texttt{Type} : \texttt{Kind}}\ \text{TYPE}$$

$$\frac{(\Gamma, x : A, \Gamma')\ w.f. \qquad x \notin \operatorname{dom}\Gamma'}{(\Gamma, x : A, \Gamma') \vdash^{\Sigma}_{\mathcal{R}} x : A}\ \text{VAR}$$

Rules specifically about types and type families

$$\frac{\Gamma \vdash^{\Sigma}_{\mathcal{R}} A : \texttt{Type} \qquad \Gamma, x : A \vdash^{\Sigma}_{\mathcal{R}} B : s}{\Gamma \vdash^{\Sigma}_{\mathcal{R}} (\Pi x : A \cdot B) : s}\ \Pi$$

$$\frac{\Gamma \vdash^{\Sigma}_{\mathcal{R}} e : A \qquad \Gamma \vdash^{\Sigma}_{\mathcal{R}} B : s \qquad A \equiv_{\beta} B}{\Gamma \vdash^{\Sigma}_{\mathcal{R}} e : B}\ \beta\text{-CONV}$$

Rules about expressions

$$\frac{\Gamma \vdash^{\Sigma}_{\mathcal{R}} A : \texttt{Type} \qquad \Gamma, x : A \vdash^{\Sigma}_{\mathcal{R}} B : s \qquad \Gamma, x : A \vdash^{\Sigma}_{\mathcal{R}} e : B}{\Gamma \vdash^{\Sigma}_{\mathcal{R}} (\lambda x : A \cdot e) : \Pi x : A \cdot B}\ \lambda$$

$$\frac{\Gamma \vdash^{\Sigma}_{\mathcal{R}} e_1 : \Pi x : A \cdot B \qquad \Gamma \vdash^{\Sigma}_{\mathcal{R}} e_2 : A}{\Gamma \vdash^{\Sigma}_{\mathcal{R}} (e_1\ e_2) : B[x := e_2]}\ \text{APP}$$

Additional rules for $\lambda\Pi/\equiv$ ($\beta\mathcal{R}$-CONV replaces β-CONV)

$$\frac{l \text{ is a pattern} \qquad \operatorname{dom}\Gamma = FV(l) \qquad \Gamma \vdash^{\Sigma}_{\mathcal{R}} l : T \qquad \Gamma \vdash^{\Sigma}_{\mathcal{R}} r : T}{\Sigma, (\mathcal{R}, l \longrightarrow^{\Gamma} r)\ w.f.\ \text{full-sig}}\ \text{REWRITE}$$

$$\frac{\Gamma \vdash^{\Sigma}_{\mathcal{R}} e : A \qquad \Gamma \vdash^{\Sigma}_{\mathcal{R}} B : s \qquad A \equiv_{\beta\mathcal{R}} B}{\Gamma \vdash^{\Sigma}_{\mathcal{R}} e : B}\ \beta\mathcal{R}\text{-CONV}$$

Figure 1: Typing rules for $\lambda\Pi/\equiv$-calculus [2]

- Rules $\varnothing$-CON and DECL illustrate the difference between typing contexts (Γ) and signatures (Σ). Until we reach the rules around rewriting, $\mathcal{R}$ can be ignored.

  Signatures contain information about previously declared values; they are mappings from constants' names to their type, with the restriction that the type must not have any free variable (i.e. the constant's type can be itself typed with an empty typing context). Typing contexts store the types of the variables, but only in the context of a single expression. It need only hold the same kind of variables as binders, and for this reason it cannot contain variables whose type is a `Kind`, that is type families or types.

  With that in mind, rule $\varnothing$-CON is the beginning of an expression's typing, the introduction of a new typing context, while rule DECL is its conclusion (for types), allowing declarations of constant of this type or of this kind.

  In typing, NEW-VAR means that we can add declarations of (local) variables, provided they are regular values and not type or type families.

  Rules CONST and VAR are used to type the constants and free variables used in the expression. Rule TYPE is for typing the constant `Type` with type `Kind`. Note that CONST and TYPE do not use the typing context at all, since constants truly are constants. The slightly roundabout notation for the context in rule VAR allows name shadowing: only the right-most type associated to the variable is taken into account. Name shadowing simplifies all rules adding variables to the context (rules NEW-VAR, Π and λ).

- Rule Π allows the creation of type families ($s = \texttt{Kind}$), that is a collection of types indexed by values of the argument type. It also enables the creation of product types ($s = \texttt{Type}$) which are an extension of the arrow $\to$ to allow the codomain type to depend on arguments (i.e. an element of a type family).

  Rule β-CONV means that all typing judgments are considered modulo β-equivalence on the right, just like β-equivalent expressions are treated equal, so we can convert from one type to the other

- Rule λ allows the creation of λ-abstractions, similarly to λ-calculus (and exactly identical when $s = \texttt{Type}$ and B doesn't depend on x). Now B can depend on x so that values can have a dependent type and we can have $s = \texttt{Kind}$ to represent type families as a functions.

  Rule APP is nearly exactly the same as in λ-calculus, but application entails substitution in the type as well as the expression, and it works as well for type families as it does for regular values.

- Rule REWRITE is a very condensed (and slightly erroneous) summary of the rules of rewriting, which are explained precisely in [2]. A good enough approximation is that the types of both sides must be the same for all valid types of the free variables. If the rewriting rule is valid, REWRITE adds its reflexive transitive contextual closure to $\mathcal{R}$.

  Rule $\beta\mathcal{R}$-CONV extends rule β-CONV to add the rewriting relations in $\mathcal{R}$ to the relations modulo which judgement (and in fact expressions) are treated equal.

1.2 Isabelle and HOL

General presentation

Isabelle [3] is a proof system and assistant. It is widely used, be it for mathematical and logical formalisation, programming languages specification or program soundness proofs. One of the biggest and most famous applications of Isabelle was the proof of an entire microkernel (seL4).

By design, Isabelle relies on a minimal kernel for its logic, named Isabelle/Pure. This kernel is a purposefully weak logic for reasons similar to Dedukti : Isabelle supports multiple 'object-logics' which sit on top of Isabelle/Pure like Isabelle/ZF, Isabelle/FOL and the most widely used Isabelle/HOL.

For reasons I will present in sections 2 and 3, I will only focus on the common logic Isabelle/Pure and the widely used Isabelle/HOL. Isabelle/Pure is based on Simple Type Theory, the first application of type theory to logics by Church. As such, its semantics are those of λ-calculus with the following extensions :

- Type classes : types are given a type class that can be recorded in the context. Classes are effectively the set of 'traits' a given type has, so a default type will have type class $\{\}$
- Prenex type polymorphism : type constructors which have type arguments can be declared, they work similarly to the arrow $\to$ in regular λ-calculus. The classes of their arguments can be restricted and the class of the resulting type can be described (a proof that it has the declared traits will obviously be needed)
- Support for free variables as parameters : variables (whose type is then inferred) can be left unbound; they will be treated as parameters and be generalised when the expression is fully typed

Typing also works just like in regular λ-calculus (see fig. 5), with the changes described in the extensions.

Theorems in Isabelle

Contrary to $\lambda\Pi/\equiv$, theorem proving works in a different and orthogonal way to typing. In Isabelle/Pure, the logical connectives are declared as constants : *prop* is again the type of propositions, $\Longrightarrow :: prop \Rightarrow prop \Rightarrow prop$ is implication ($\Rightarrow$ is the function type constructor in Isabelle), $\bigwedge :: (\alpha \Rightarrow prop) \Rightarrow prop$ is universal quantification, $\&\&\& :: prop \Rightarrow prop \Rightarrow prop$ is an internal conjunction and $\equiv :: \alpha \Rightarrow \alpha \Rightarrow prop$ is equality, Isabelle/Pure doesn't define the other connectives.

Isabelle/HOL defines new constructors for propositions (resp. *bool*, $\to$, $\forall$) and also provides the full list of connectives, defining them with Pure's connectives for the first two and using definitions similar to those I gave in section 1.1 for the other ones (note that it corresponds to the definitions that don't depend on $\lambda\Pi$'s type structure). The types *prop* and *bool* are linked by the constant $Trueprop :: bool \Rightarrow prop$ which states that its argument is true.

In Isabelle, propositions and (proved) theorems are fundamentally different; they do not have the same internal type. Elements of *thm*, the type of proved theorems, can only be constructed and modified using a fixed handful of primitives, each representing a deduction rule. This way, any *thm* is guaranteed to be derived from that set of deduction rules, thanks to the soundness of static typing.

This conception also allows defining functions on theorems by composing the primitives, to form higher-level rules. This has the downside that theorems do not store a proof, and it is impossible to reconstruct one without modifying Isabelle's kernel.

Deduction rules, built-in axioms

Deduction rules are described in fig. 2 with equality having the following axioms:

$$(\lambda x \cdot b(x))\ a \equiv b(a) \qquad \text{Beta-conversion}$$
$$x \equiv x \qquad \text{Reflexivity}$$
$$x \equiv y \Longrightarrow P\ x \Longrightarrow P\ y \qquad \text{'Substitution' (Leibniz law)}$$
$$\left(\bigwedge x \cdot f\ x \equiv g\ x\right) \Longrightarrow f \equiv g \qquad \text{Extensionality}$$
$$(A \Longrightarrow B) \Longrightarrow (B \Longrightarrow A) \Longrightarrow A \equiv B \qquad \text{Logical equivalence } (\equiv \text{ on props is } \Leftrightarrow)$$

$$\frac{A \text{ is an axiom}}{\vdash A}\ \textsc{Axiom} \qquad \frac{}{A \vdash A}\ \textsc{Hypothesis}$$

$$\frac{\Gamma \vdash B(x) \qquad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \bigwedge x \cdot B(x)}\ \textstyle\bigwedge\text{-}\textsc{Int} \qquad \frac{\Gamma \vdash \bigwedge x \cdot B(x)}{\Gamma \vdash B(a)}\ \textstyle\bigwedge\text{-}\textsc{Elim}$$

$$\frac{\Gamma \vdash B}{\Gamma \setminus A \vdash A \Longrightarrow B}\ \Longrightarrow\text{-}\textsc{Int} \qquad \frac{\Gamma_1 \vdash A \qquad \Gamma_2 \vdash A \Longrightarrow B}{\Gamma_1 \cup \Gamma_2 \vdash B}\ \Longrightarrow\text{-}\textsc{Elim}$$

Figure 2: Deriving rules for Isabelle/Pure [4]

In Isabelle/HOL, an additional axiom states that all *bool*s (all of its propositions) are either true or false, making it a classical logic.

2 From Isabelle/Pure to Dedukti[HOL]

The first part of my internship dealt with the translation from Isabelle/Pure to Dedukti, and more precisely to an extended Dedukti[HOL] (where Dedukti[HOL] is the name of the encoding of HOL into the framework of Dedukti). The code related to this is accessible on my branch of the GitHub repository of the translator, which should soon be merged into the master branch.

2.1 Isabelle's Curry-Howard proof-term module

As explained in sections 1.1 and 1.2, the Isabelle framework uses theorem-deriving primitives to prove its theorems while Dedukti uses Curry-Howard proof-terms. Considering Isabelle theorems do not memorise how they came to be, we need more than the underlying ML value of the theorem to be able to derive a proof-term of it.

To solve this issue and allow the export of proofs, Isabelle developer Stephan Berghöfer added a proof-term module which, when enabled, constructs and maintains a proof-term as theorems

are derived and proved. This effectively means that the Isabelle proofs that I had to translate were in a proof-term format.

However, the module is still experimental and doesn't scale well (consequences of this are addressed in section 2.3). Also, the proof-terms are handled at the lowest level, Isabelle/Pure, so the embedding of Isabelle/HOL into Isabelle/Pure weighs down on the module and on the size of proof-terms.

2.2 The Dedukti[HOL] embedding, the first version of the translator and its formalisation on paper

Before my internship, a first version of the Isabelle/Pure-to-Dedukti[HOL] translator had already been written by Isabelle lead developer Makarius Wenzel. It targeted slightly outdated versions of Dedukti and Lambdapi so, during the first few weeks, I updated the translator to the most recent versions of Dedukti and Lambdapi as I dug into its internals.

The encoding of the HOL logic system (sometimes referred to as Simple Type Theory) into Dedukti, written Dedukti[HOL], was first devised by Ali Assaf in [5], it was then put into context in [1] and inscribed in a unified framework for Dedukti called the theory $\mathcal{U}$ in [2].

In this encoding, proofs work in the same way as standard Curry-Howard proof-terms (see section 1.1). Encoded types are grouped in the Dedukti type *Set* and, because of the same limitation as proofs, these encoded types are associated to the real Dedukti types of their elements using the type family $\mathrm{El} : \mathrm{Set} \to \texttt{Type}$. Finally, just like we want proofs of implications to be functions on proofs, we want encoded function types to be first-hand functions on encoded types, so $\mathrm{El}(A \Rightarrow B)$ rewrites to $\mathrm{El}\, A \to \mathrm{El}\, B$.

This encoding was extended for the translation of Isabelle, to accommodate for the extensions to Simple Type Theory Isabelle makes:

- Type classes are translated by the proof-term module into predicates on types, so a construct for functions from types to $\mathrm{El}\, prop$ needs to be possible. The relations between type classes are translated to axioms, they don't need more expressing power.
- Type constructors need a construct for functions with an arbitrary number of arguments in *Set* and a return type of *Set*. Also, all terms may depend on type variables, so it must be possible to give a type argument to both terms and proofs.

Because the target language was Dedukti and all of these constructs are easily built in it, I imagine little thought was given on the exact description of the target logic, but it is in fact extremely close to STT$\forall$, a logic which I will talk about in section 3. The translation from Isabelle with proof-terms to Dedukti[HOL] is straightforward, it is illustrated by fig. 3 and its proof of correctness I wrote is in appendix B:

2.3 Distant machine; bugs and limits of the proof-term module

As explained in section 2.1, the module for proof-terms in Isabelle doesn't scale well. The memory it needs quickly reaches 12 GB (the available RAM+swap on my personal laptop), well before the HOL standard library is successfully compiled. To try and go further, the Inria gave me remote access to the Gulliver computer cluster.

This way, the build process could go further and not be blocked by RAM anymore. Instead, the next limitation would come from the proof-term module itself: it has an internal limit on

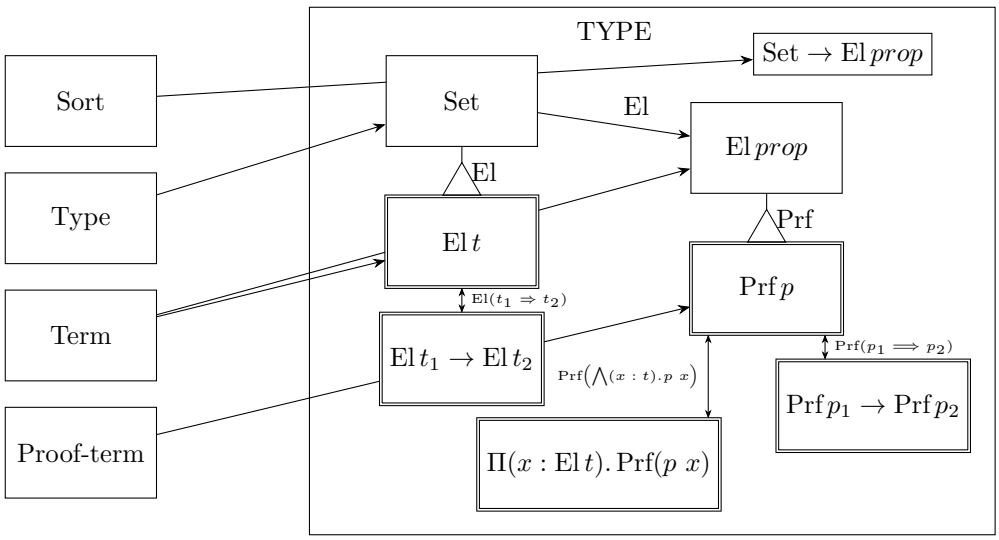

Figure 3: The modified Dedukti[HOL] embedding and the correspondence with Isabelle's elements

the size of the proof-terms it can generate and maintain. Unfortunately, in the theory Enum of the standard library which deals with small finite-size sets, some proofs on binary operators are solved by exhaustion, which for 5-element sets results in proof-terms larger than the module can support (the JVM heap space is out of memory).

Unfortunately for the internship, this issue is out of scope and depends on the developers of Isabelle. I managed to talk with Isabelle lead developer and the translator's first developer, Makarius Wenzel, and he told me the developers were aware of this limitation but their limited availability meant that the issue wouldn't be solved in the near future.

Still, he gave me a temporary way to side-step the issue: we could add an axiom proving any fact and replace all the problematic proofs with a call to said axiom. It did allow compiling theory Enum, but the very next theory, String, also blocked for the same reason. I repeated the process and managed to compile it but then another theory, Quickcheck_Random, blocked for an issue apparently related to type classes. Unfortunately, I haven't had enough time to try to solve it (each try taking an hour) and I had no way of avoiding it.

2.4 Infix notations, handling implicit arguments in Lambdapi

With proof assistants, just like in any programming language, it is very useful to be able to define and use prefix and infix operators, both for writing and reading code. Compare the following :

$$\text{Prf}\,\texttt{(imp (imp A B) ((imp B A) (equal A B)))}$$

$$\text{Prf}((A \Longrightarrow B) \Longrightarrow (B \Longrightarrow A) \Longrightarrow A == B)$$

Because Isabelle exports the notation declarations along with the constant declarations, I decided to add the translation of these notations to Lambdapi (Dedukti doesn't support them).

While dealing with associativity and when parentheses were needed gave me some trouble, the process is mostly uninteresting and won't be elaborated here.

Another more interesting difficulty is that of polymorphic operators : some operators which are declared infix may take as first argument(s) the type that they operate on, before the two arguments which are supposed to be around the symbol. Take polymorphic equality : its type is $'a \Rightarrow 'a \Rightarrow prop$ (in Isabelle). The type variable $'a$ is given completely implicitly in Isabelle, so it doesn't cause problems when declaring the symbol = infix. However, this type argument is a priori explicit in the translation to Lambdapi, so declaring = infix won't work.

To circumvent this issue (which is more general than this translation), Lambdapi supports the declaration of implicit arguments. They are denoted by curly braces and, when the function is called, their content is deduced by the constraints posed by the other arguments and the context of the call. If we come back to the example of equality, its type in Lambdapi would be $\Pi\{'a : \mathrm{Set}\}.\, \mathrm{El}('a \Rightarrow 'a \Rightarrow prop)$ and the call $A = B$ where $\Gamma \vdash A : 'a, \Gamma \vdash B : 'b$ would generate the constraints $?0 \equiv 'a, ?0 \equiv 'b$ (here, ?0 is a representation of the internal value of the implicit argument, a meta-variable).

In Isabelle, there is no notion of implicit arguments : type arguments are always handled implicitly and all arguments are given explicitly. It does support argument-value inference however, which means that if the value of an argument can be inferred by the context, one can decide to give _ as the argument, prompting Isabelle to infer the value itself.

Isabelle's exported terms and the proof-term module both contain the values of the type argument for all expressions, so everything is explicit at the source of the translation and the translator alone has to characterise which arguments are going to be implicit and which ones are going to be explicit.

My first idea was to make all type variables implicit, but there was a problem with constants which don't take arguments whose type depends on that type variable, the most problematic being $\_ : \Pi\{'a : \mathrm{Set}\}.\, \mathrm{El}('a)$

The second idea was to only make implicit the type variables which are mentioned in the types of the (other) arguments of the constant. The problem now was that when we referred to functions without those other arguments, the implicit argument was still impossible to guess. For this reason, I had to restrict it heavily and make it so that only infix operators with the correct number of normal arguments would have their implicit arguments omitted, all other implicit arguments being given explicitly.

However, explicit passing of implicit arguments was still an incomplete feature, because it didn't work with operators (symbols which had a notation). At the time, the parsing of infix notations relied on undocumented behaviour of the parser (which now doesn't work anymore) and managed to correctly interpret the expression '= 0' even when '=' was declared infix. This was incompatible with the way of passing explicitly implicit arguments, using a syntax that was only expected at regular applications.

Therefore, I patched Lambdapi's parser to recognise a new syntax for operators: being wrapped in parentheses. It has the same effect as in OCaml, that is, it removes its properties in the parser, and it's therefore the only way to have access to the operator before implicit arguments are filled in with metavariables. This modification was then quickly integrated to the master branch.

2.5 η-contraction, η-expansion and equivalence modulo η

Isabelle's equivalence is inherently modulo η, but for Dedukti and Lambdapi this is only a setting and for some proof systems it is not possible to work like that. Therefore, my supervisor prompted me to see if it was possible to remove the need of this setting in the translation.

The key insight is that η-equivalence is only useful between the η-contracted f and the once-expanded $\lambda x.f\ x$. If we have terms that are η-expanded twice or more, they will β-reduce to the once-expanded form :

$$\lambda x.(\lambda y.f\ y)\ x \hookrightarrow_\beta \lambda x.(f\ y)[y := x] = \lambda x.f\ x \quad \text{(reduction under the outer } \lambda)$$

First though, I wrote a function to η-contract all η-long forms. This direction is also useful to reduce some very long terms that were η-expanded up to three times (eg $\lambda x_2\ y_2 \cdot (\lambda x_1\ y_1 \cdot (\lambda\ x\ y \cdot \texttt{less}\ x\ y)\ x_1\ y_1)\ x_2\ y_2$) and can be used at every translation, not just when the user wishes to unset Dedukti's eta-equivalence setting.

This part was the easiest as it didn't involve name creation and renamings but only removal. A particular attention was needed to deal simultaneously with terms and their types, as well as to account for shared parameters (in '`symbol c ('a: Set) : Set := c2 'a;`', `'a` is shared between the term (as a λ-abstraction) and the type (as a Π-abstraction)). I also added a function to factor out such shared parameters, so that they are not duplicated in the term and the type.

Second, I wrote a function to η-expand all function-typed values, to the extent of their arity (i.e. $f : {'a} \Rightarrow {'b} \Rightarrow {'c}$ is expanded to $\lambda(x : {'a})(y : {'b}) \cdot f\ x\ y$). This time, I ran into the issue of name creation and mostly capture-free renamings. To expand all function-typed values, I had to know the type of all variables and constants at all times, even across multiple files. This only required me to keep a map of constants to types, these types being explicitly given in all declarations. To avoid captures in renamings, after I fell in the trap, I went for the easy but dirty trick that is tagging new variables with a character unused by Isabelle. It has the obvious downside of polluting simple names with '€' (the character I went for).

The last problem I encountered was that of functions returning something whose type is a given argument. Consider $id = \lambda\ \_\ x \cdot x : \Pi(a : \mathrm{Set}) \cdot \mathrm{El}\, a \to \mathrm{El}\, a$, the polymorphic identity function. When the argument type is that of a function, the whole expression becomes a function, which means I would have to look at every argument in every function application to see if it changes (through specialisation) the expected final type to that of a function. However, the true problem comes when we consider the following :

$$id\_predicate : \Pi(a : \mathrm{Set}) \cdot \Pi(x : \mathrm{El}\, a) \cdot \mathrm{Prf}(id\ a\ x = x)$$

Such a predicate would be easy to define (if we peered into the details of $=$, but we won't), yet it makes it impossible to η-expand the left-hand side when x is a function (x itself can be in η-long form, but '$\lambda z \cdot id\ (a \Rightarrow a)(\lambda x \cdot x)z = \lambda x \cdot x$' is what we would want and cannot reach because of the equal sign).

This makes it impossible not to rely on η-equivalence during the translation; the only solution is for Dedukti to track down by itself where η-equivalence is used and, if the target system can support it, implement an axiom so that η-short and η-long are made equal and use it explicitly when necessary.

3 From Dedukti[STT∀] to Isabelle/HOL

For the second part of my internship, I studied the translation from Dedukti[STT∀] and Logipedia's code base to Isabelle, where I decided to use the HOL object-logic. The code related to this is accessible on my branch of the Github repository of Logipedia.

3.1 Logipedia and STT∀

Logipedia [6] is a library of definitions, theorems and proofs, all available in multiple proof systems. As such, it makes for an ideal code-base when a new translator from Dedukti is developed.

Contrary to what one might believe, the system in which the proofs are stored is not pure Dedukti, but another system called STT∀ [7]. It is based on Simple Type Theory, with one major extension: prenex polymorphism. This makes it a system even weaker than $\lambda\Pi/\equiv$, ideal for outwards translation.

STT∀ is very close to Isabelle/Pure in terms of expressive power: starting from STT which is a subtheory, type classes look like they can be axiomatised with type predicates being fixed constants of type $\forall\_, prop$ (a function type from Set to *prop*, with the caveat that Set cannot be expressed explicitly as it truly is the type of all types within STT∀), and free variables are only variables whose binding is implicit, not removed entirely. Based on the description of Isabelle/Pure I gave in section 1.2, it appears that they could in fact be equivalent, some more exploration would be needed to confirm.

Types and terms behave the same way in STT∀ and Isabelle (besides the fact that we have to bind all variables), the way proofs are handled and stored is quite different however. In STT∀, proofs are stored as full derivation trees. This makes it a hybrid between proof-terms and primitives, compatible with both approaches:

- because of Curry-Howard equivalence, each rule can be interpreted as a typing rule, and it is trivial to construct a proof-term from a derivation tree;
- it is also trivial to construct a theorem using the primitives in the same order as they are presented in the tree.

Figure 4 lists the typing rules of STT∀, which are basically λ-calculus, some rules to formalise prenex polymorphism properly and Isabelle's deduction rules.

3.2 Translating terms

The first few steps for writing the compiler were as follows : first, understanding how Logipedia's code base works in terms of adding support for a new language; second, finding out the necessary boilerplate code to construct an Isabelle theory file; third comes the translation proper.

The first two steps were dealt with quickly, mostly by looking at other files for comparison (I started from Logipedia's files for the translation to HOL-Light, a close relative of Isabelle's, and I extracted the boilerplate from the first few theories in Isabelle's standard library).

Because Isabelle's way of writing proofs is very different to the one used in other systems (including Logipedia), I first focused on translating definitions, axioms, propositions and parameters. With the similarities between the systems this was done easily, but a question arose: should I translate to Isabelle/Pure or Isabelle/HOL ? At this level the question was mostly syntactic, I needed to know which system of symbols I was going to use, switching between them wouldn't be difficult. As I was more familiar with Isabelle/Pure, I chose it at first.

Rules about context

$$\frac{}{[]\ w.f.}\ \varnothing\text{-CON}$$

$$\frac{\Gamma\ w.f. \qquad p \notin \operatorname{dom}\Gamma}{\Gamma, (p,n)\ \text{type op}\ w.f.}\ \text{NEW-OP} \qquad \frac{\Gamma\ w.f.}{\Gamma, X\ \text{tvar}\ w.f.}\ \text{NEW-TVAR} \qquad \frac{\Gamma \vdash A : \texttt{Type}}{\Gamma, x : A\ w.f.}\ \text{NEW-VAR}$$

$$\frac{\Gamma \vdash A_i : \texttt{Type} \quad (i \in [\![1,n]\!]) \qquad (p,n)\ \text{type op} \in \Gamma}{\Gamma \vdash p\ A_1\ \ldots\ A_n : \texttt{Type}}\ \text{OP}$$

$$\frac{\Gamma\ w.f. \qquad X\ \text{tvar} \in \Gamma}{\Gamma \vdash X : \texttt{Type}}\ \text{TVAR} \qquad \frac{(\Gamma, x : A, \Gamma')\ w.f. \qquad x \notin \operatorname{dom}(\Gamma')}{(\Gamma, x : A, \Gamma') \vdash x : A}\ \text{VAR}$$

---

prop and its operators

$$\frac{\Gamma\ w.f.}{\Gamma \vdash \mathit{prop} : \texttt{Type}}\ \text{PROP} \qquad \frac{\Gamma\ w.f.}{\Gamma \vdash (\Longrightarrow) : \mathit{prop} \to \mathit{prop} \to \mathit{prop}}\ \Longrightarrow$$

$$\frac{\Gamma\ w.f.}{\Gamma \vdash (\forall) : \forall A \cdot (A \to \mathit{prop}) \to \mathit{prop}}\ \forall \qquad \frac{\Gamma, X\ \text{tvar} \vdash p : \mathit{prop}}{\Gamma \vdash \forall_p X \cdot p : \mathit{prop}}\ \forall\text{-PROP}$$

Rules about types and polymorphism

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma \vdash B : \texttt{Type}}{\Gamma \vdash A \to B : \texttt{Type}}\ \to \qquad \frac{\Gamma, X\ \text{tvar} \vdash T : \texttt{Type}}{\Gamma \vdash (\forall X \cdot T) : \texttt{Type}}\ \forall\text{-TYPE}$$

Rules about terms (what we intend by λ-calculus) and about terms depending on type variables

$$\frac{\Gamma \vdash A, B : \texttt{Type} \qquad \Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x : A \cdot e) : A \to B}\ \lambda \qquad \frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash (e_1\ e_2) : B}\ \text{APP}$$

$$\frac{\Gamma, X\ \text{tvar} \vdash t : T : \texttt{Type}}{\Gamma \vdash (\lambda X \cdot t) : \forall X \cdot T}\ \lambda\text{-TYPE} \qquad \frac{\Gamma \vdash t : \forall X \cdot T \qquad \Gamma \vdash A : \texttt{Type}}{\Gamma \vdash (t\ A) : T[X := A]}\ \text{APP-TYPE}$$

Deduction rules in STT$\forall$

$$\frac{\Gamma \vdash p : \mathit{prop}}{\Gamma, p\ \text{true}\ w.f.}\ \text{NEW-HYP} \qquad \frac{(\Gamma, p\ \text{true}, \Gamma')\ w.f. \qquad FV(p) \cap \operatorname{dom}\Gamma' = \varnothing}{(\Gamma, p\ \text{true}, \Gamma') \Vdash p}\ \text{HYP}$$

$$\frac{\Gamma \Vdash t \qquad t \equiv_\beta u}{\Gamma \Vdash u}\ \beta\text{-CONV} \qquad \frac{\Gamma, p\ \text{true} \Vdash q}{\Gamma \Vdash p \Longrightarrow q}\ \Longrightarrow\text{-I} \qquad \frac{\Gamma \Vdash p \Longrightarrow q \qquad \Gamma \Vdash p}{\Gamma \Vdash q}\ \Longrightarrow\text{-E}$$

$$\frac{\Gamma \vdash t : (A : \texttt{Type}) \to \mathit{prop} \qquad \Gamma, x : A \Vdash t\ x}{\Gamma \Vdash \forall (x : A) \cdot t\ x}\ \forall\text{-I} \qquad \frac{\Gamma \Vdash \forall (x : A) \cdot t\ x \qquad \Gamma \vdash u : A}{\Gamma \Vdash t\ u}\ \forall\text{-E}$$

$$\frac{\Gamma, A\ \text{tvar} \vdash p : \mathit{prop} \qquad \Gamma, A\ \text{tvar} \Vdash p(A)}{\Gamma \Vdash \forall_p A \cdot p(A)}\ \forall_p\text{-I} \qquad \frac{\Gamma \Vdash \forall_p A \cdot p(A) \qquad \Gamma \vdash X\ \text{tvar}}{\Gamma \Vdash p[A := X]}\ \forall_p\text{-E}$$

Figure 4: Typing and deduction rules for STT$\forall$ [7]

However, when I had Isabelle first parse the output of the translator, it made me understand (through quite obscure error messages) that it cannot run at the user-level without using an object-logic. This meant that I would need to target a logic encoded in Pure and not directly Pure. Therefore, I changed the target logic to Isabelle/HOL; this is the most widely used object-logic and as such had the most tools to help me on my translations, this also means that the translation would be interpreted in classical logic contrary to STT$\forall$ or Isabelle/Pure.

3.3 Translating proofs

The first question surrounding the translation of proofs was how to have them interpreted by Isabelle. Its native proof system emulates mathematical proofs, where you list the propositions in order of their deduction, potentially adding a sub-proof to fork the reasoning. This organisation is very unnatural when the input proofs are in a λ-term-like form (it is primitives with arguments and not the proved propositions at each 'step', there's not even a natural notion of 'step' with this form).

My supervisor first inquired whether we could use the proof-term module the other way around, having it read a proof-term and reconstruct the theorem it proves. Such a function did exist in the module, however I needed to add a way to access it to the user-level. Thanks to the help of the Isabelle forum and Burkhart Wolff, an expert on Isabelle proofs with whom I had the opportunity to talk, I managed to write such an entry point to the proof-term interpreter.

Then came the moment to write down the proofs. Because I had to target Isabelle/HOL, the fact that it is a logic encoded in another appeared there, as each proof-term construct (think abstraction or application) needed to be translated to a call to the corresponding HOL axiom, which also asks for some type information on its arguments and especially proofs that all types are HOL types (a trivial proof, but which can take up a lot of space). This made the proofs very large when compared to the theorem they proved.

This also vastly increased the room for error and brought up an issue of using the proof-term module: because the module (and especially this interpretation function) are very experimental, they do not properly handle error reporting, which made debugging close to impossible. Any error would stop everything while leaving no information, and the size of terms also made it impossible to minimise the problematic translations. Adding to this the fact that abstractions in proofs were not properly parsed, these issues prevented me from completing the translation.

4 Current state, future developments

4.1 Summary of the current state

To sum-up, at the moment:

- The translator from Isabelle/Pure to Dedukti[HOL, augmented for Isabelle] works for current versions, was improved on some points, and was formalised and proved sound;
  - it can now support infix and prefix notations while making some arguments implicit and lightening the resulting translation, at the cost of a much higher memory demand;
  - a module to skip η-equivalence was constructed, but it doesn't work because it theoretically cannot; η-shortening however is implemented and does work;
- The translator from Dedukti[STT$\forall$] to Isabelle/HOL partially works
  - Everything not requiring proofs is translated well

– Everything requiring proofs tries to use Isabelle's proof-term module, but the translation is not finished because of my issues with Isabelle's proof-term module

4.2 Future developments, unused ideas

During the internship, a few ideas for further developments came to my mind:

1. Although the choice of using Isabelle's proof-term module for the translation of Dedukti[STT∀] seems more natural, the native proof method may be easier to set-up and we may reach a working version sooner with it.

2. Because Isabelle/HOL is by far the most widely used of Isabelle's object-logics, it may be worth it to write a translation from it directly, to get away with the encoding in Isabelle/Pure and simplify the translated terms. A difficulty would be that Isabelle's low-level only uses Pure and it is responsible for exports and some of the automation in proofs, making it difficult to abstract away.

3. While Isabelle supports definitions at the user-level, there is no such construct at the low-level: they are reduced to a constant declaration and an axiom of equality between the constant and its definition. Because Dedukti supports definitions as a more native construct than application of equality (which uses 2 axioms: the definition axiom and the Leibniz equality axiom), it may be a good idea to fuse back the constant declaration and axiom into a single definition. A difficulty would be that detecting which axiom is a definition and pairing the constructs would need to be done on a name basis, with term analysis; this may need a quadratic exploration of all pairs (constant, axiom) to successfully find the pairs, or get some help from a modification in Isabelle indicating what those pairs are.

4. This idea was suggested to me by Makarius Wenzel and falls a bit out of the scope of the internship. The current way type classes algebra is implemented in the proof-term module and in Dedukti is quite primitive, having all 'subclassing' be explicit and defining a potentially quadratic number of such 'subclassing' axioms. It may be wise to look into Dedukti's rewriting abilities to better represent this algebra, possibly limiting the number of 'subclassing' indicators to a minimal number. While knowledge of Dedukti's rewriting system is needed, this change would mostly affect the proof-term module as it is responsible of reducing type classes to something Dedukti supports.

5 Conclusion

The internship allowed me to study in-depth the logics of different proof systems which I didn't know (except by name for Isabelle) which complemented well my first internship about proving distributed algorithms in Coq. My curiosity about proof assistants and proof systems was well satisfied with this first but deep insight into their foundations.

On another level, this internship was mixed between bibliographic research to find the written bases of the proof systems I studied and programming between the translators, the exploration I did in Isabelle's proof-term module and the modifications I did in Lambdapi. While I already knew I would enjoy the latter more, I imagine this balance will stay the same during my future internships and my career, so being confronted to it soon is most desirable.

Finally, on a personal level, this internship is the first one in which I had the opportunity to go to a lab and work around other people. Following 7 months of lockdown, it was a very nice opportunity to be able to experience research as it is usually done.

References

[1] Ali Assaf, G. Burel, Raphaël Cauderlier, D. Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, O. Hermant and Ronan Saillard. 'Dedukti : A Logical Framework Based on the Π-Calculus Modulo Theory'. In: 2016. URL: https://www.semanticscholar.org/paper/Dedukti-%3A-a-Logical-Framework-based-on-the-%CE%BB-Modulo-Assaf-Burel/b83480c7d1578d8e6eb57fa1fda46d051a715ace.

[2] Frédéric Blanqui, Gilles Dowek, Émilie Grienenberger, Gabriel Hondet and François Thiré. 'Some Axioms for Mathematics'. In: *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021)*. Ed. by Naoki Kobayashi. Vol. 195. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 20:1–20:19. ISBN: 978-3-95977-191-7. DOI: 10.4230/LIPIcs.FSCD.2021.20. URL: https://drops.dagstuhl.de/opus/volltexte/2021/14258.

[3] *Isabelle*. URL: https://isabelle.in.tum.de/ (visited on 28/08/2021).

[4] *implementation.pdf*. URL: https://isabelle.in.tum.de/dist/Isabelle2021/doc/implementation.pdf (visited on 28/08/2021).

[5] Ali Assaf and Guillaume Burel. 'Translating HOL to Dedukti'. In: *Electronic Proceedings in Theoretical Computer Science* 186 (30th July 2015), pp. 74–88. ISSN: 2075-2180. DOI: 10.4204/EPTCS.186.8. URL: http://arxiv.org/abs/1507.08720 (visited on 15/07/2021).

[6] *Logipedia*. URL: https://logipedia.inria.fr (visited on 28/08/2021).

[7] François Thire. 'Interoperability between Proof Systems Using the Logical Framework Dedukti'. Theses. Université Paris-Saclay, Dec. 2020. URL: https://hal.archives-ouvertes.fr/tel-03224039 (visited on 28/08/2021).

A Typing rules for λ

Rules about context

$$\frac{}{[]\ w.f.}\ \varnothing\text{-CON} \qquad \frac{\Gamma\ w.f.}{\Gamma, x : \texttt{Type}\ w.f.}\ \text{TYPE-DECL} \qquad \frac{\Gamma \vdash A : \texttt{Type}}{\Gamma, x : A\ w.f.}\ \text{VALUE-DECL}$$

$$\frac{(\Gamma, x : A, \Gamma')\ w.f. \qquad x \notin \mathrm{dom}(\Gamma')}{(\Gamma, x : A, \Gamma') \vdash x : A}\ \text{VAR}$$

Rules about types

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma \vdash B : \texttt{Type}}{\Gamma \vdash A \to B : \texttt{Type}}\ \to$$

Rules about expressions

$$\frac{\Gamma \vdash A : \texttt{Type} \qquad \Gamma \vdash B : \texttt{Type} \qquad \Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x : A \cdot e) : A \to B}\ \lambda$$

$$\frac{\Gamma \vdash e_1 : A \to B \qquad \Gamma \vdash e_2 : A}{\Gamma \vdash (e_1\ e_2) : B}\ \text{APP}$$

Figure 5: Typing rules for λ-calculus

B Proof of the translation from Isabelle/Pure to Dedukti[HOL] modified

Before starting, note that initially:

$$\Sigma = \Big[\mathrm{Set} : \mathtt{Type}, \mathrm{El} : \mathrm{Set} \to \mathtt{Type}, \mathit{prop} : \mathrm{Set}, \mathrm{Prf} : \mathrm{El}\,\mathit{prop} \to \mathtt{Type},$$
$$(\Rightarrow) : \mathrm{Set} \to \mathrm{Set} \to \mathrm{Set}, (\Longrightarrow) : \mathrm{El}(\mathit{prop} \Rightarrow \mathit{prop} \Rightarrow \mathit{prop}),$$
$$\Big(\bigwedge\Big) : \Pi(\alpha : \mathrm{Set}) \cdot \mathrm{El}((\alpha \Rightarrow \mathit{prop}) \Rightarrow \mathit{prop}) \Big]$$

$$\mathcal{R} = \Big\{ [A, B : \mathtt{Type}]\ \ \mathrm{El}(A \Rightarrow B) \hookrightarrow \mathrm{El}\,A \to \mathrm{El}\,B,$$
$$[P, Q : \mathrm{El}\,\mathit{prop}]\ \ \mathrm{Prf}(P \Longrightarrow Q) \hookrightarrow \mathrm{Prf}\,P \to \mathrm{Prf}\,Q,$$
$$[\alpha : \mathrm{Set}, P : \alpha \to \mathrm{El}\,\mathit{prop}]\ \ \mathrm{Prf}\Big(\bigwedge\ \alpha\ P\Big) \hookrightarrow \Pi(x : \mathrm{El}\,\alpha) \cdot \mathrm{Prf}(P\ x) \Big\}$$

First, types and type constructors aren't modified when translated, so a basic induction shows that every Isabelle type gets translated to an element of *Set* (Sound-Types).

Second, we need to prove that every Isabelle term of type τ gets translated to an expression of type $\mathrm{El}\,\tau$ (Sound-Terms):

- In contexts, everything works easily :

$$\frac{\Gamma \vdash \tau : \mathtt{Type}}{\Gamma, x : \tau\ w.f.}\ \textsc{value-decl} \qquad \text{gives}$$

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash^{\Sigma}_{\mathcal{R}} [\![\Gamma]\!]\ w.f.}\ \textsc{induction} \qquad (\mathrm{El} : \mathrm{Set} \to \mathtt{Type}) \in \Sigma}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} \mathrm{El} : \mathrm{Set} \to \mathtt{Type}}\ \textsc{const} \qquad \dfrac{\vdots}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} \tau : \mathrm{Set}}\ \textsc{Sound-Types}}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} \mathrm{El}\,\tau : \mathtt{Type}}\ \textsc{app}}{\vdash^{\Sigma}_{\mathcal{R}} [\![\Gamma]\!], x : \mathrm{El}\,\tau\ w.f.}\ \textsc{decl}$$

  From now on, Ty will represent the tree rooted at app, so other trees are not too charged.

- The var rule is identical, and $[\![(\Gamma, x : \tau, \Gamma')]\!] = ([\![\Gamma]\!], x : \mathrm{El}\,\tau, [\![\Gamma']\!])$

$$\frac{(\Gamma, x : \tau, \Gamma')\ w.f. \qquad x \notin \mathrm{dom}(\Gamma')}{(\Gamma, x : \tau, \Gamma') \vdash x : A}\ \textsc{var} \quad \text{gives} \quad \frac{\dfrac{\vdots}{[\![(\Gamma, x : \tau, \Gamma')]\!]\ w.f. \qquad x \notin \mathrm{dom}([\![\Gamma']\!])}\ \textsc{induction}}{[\![(\Gamma, x : A, \Gamma')]\!] \vdash^{\Sigma}_{\mathcal{R}} x : \mathrm{El}\,\tau}\ \textsc{var}$$

- Abstractions are a simple induction step, with $[\![\lambda x :: \tau \cdot e]\!] = \lambda x : \mathrm{El}\,\tau \cdot [\![e]\!]$ and $[\![\tau \Rightarrow \tau']\!] = \mathrm{El}\,\tau \to \mathrm{El}\,\tau'$

$$\frac{\Gamma \vdash \tau : \mathtt{Type} \qquad \Gamma \vdash \tau' : \mathtt{Type} \qquad \Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x :: \tau \cdot e) : \tau \to \tau'}\ \lambda \qquad \text{gives}$$

$$\frac{\dfrac{\vdots}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} \mathrm{El}\,\tau : \mathtt{Type}}\ \textsc{Ty} \qquad \dfrac{\vdots}{[\![\Gamma]\!], x : \mathrm{El}\,\tau \vdash^{\Sigma}_{\mathcal{R}} \mathrm{El}\,\tau' : \mathtt{Type}}\ \textsc{Ty} \qquad \dfrac{\vdots}{[\![\Gamma]\!], x : \tau \vdash^{\Sigma}_{\mathcal{R}} [\![e]\!] : \tau'}\ \textsc{induction}}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} (\lambda x : \mathrm{El}\,\tau \cdot e) : \mathrm{El}\,\tau \to \mathrm{El}\,\tau'}\ \lambda$$

- Applications are also a simple induction step, with $[\![e_1\ e_2]\!] = [\![e_1]\!]\ [\![e_2]\!]$

$$\frac{\Gamma \vdash e_1 : \tau \Rightarrow \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \tau'}\ \textsc{app} \quad \text{gives}$$

$$\frac{\dfrac{\vdots}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} [\![e_1]\!] : \mathrm{El}\,\tau \to \mathrm{El}\,\tau'}\ \textsc{induction} \qquad \dfrac{\vdots}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} [\![e_2]\!] : \mathrm{El}\,\tau}\ \textsc{induction}}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} ([\![e_1]\!]\ [\![e_2]\!]) : \mathrm{El}\,\tau'}\ \textsc{app}$$

Finally, we can prove that the deduction rules are preserved

- Axioms are present in the signature, they were well translated thanks to Sound-Terms.

$$\frac{A \text{ is an axiom}}{\vdash A}\ \textsc{Axiom} \qquad \text{gives} \qquad \frac{(\mathrm{Ax}_A : \mathrm{Prf}[\![A]\!]) \in \Sigma \qquad \dfrac{\vdots}{\vdash^{\Sigma}_{\mathcal{R}} [\![\Gamma]\!]\ w.f.}\ \textsc{induction}}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} \mathrm{Ax}_A : \mathrm{Prf}[\![A]\!]}\ \textsc{const}$$

- Hypotheses are present in the context as previously bound variables

$$\frac{}{A \vdash A}\ \textsc{Hypothesis} \qquad \text{becomes} \qquad \frac{H_A : \mathrm{Prf}[\![A]\!] \in \Gamma \qquad \dfrac{\vdots}{\vdash^{\Sigma}_{\mathcal{R}} [\![\Gamma]\!]\ w.f.}\ \textsc{induction}}{\Gamma \vdash^{\Sigma}_{\mathcal{R}} H_A : \mathrm{Prf}[\![A]\!]}\ \textsc{var}$$

- Variables which appear free in Isabelle are bound by the proof-term module, so they will be in context with their types. Note that $\mathrm{Prf}\left[\!\!\left[\bigwedge (x :: \tau) \cdot B(x)\right]\!\!\right] = \Pi(x : \mathrm{El}\,\tau) \cdot \mathrm{Prf}[\![B(x)]\!]$. For a proposition p, we'll name $\textsc{pr}$ the derivation tree

$$\frac{\dfrac{(\mathrm{Prf} : \mathrm{El}\,prop \to \texttt{Type}) \in \Sigma \qquad \dfrac{\vdots}{\vdash^{\Sigma}_{\mathcal{R}} [\![\Gamma]\!]\ w.f.}\ \textsc{induction}}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} \mathrm{Prf} : \mathrm{El}\,prop \to \texttt{Type}}\ \textsc{const} \qquad \dfrac{\vdots}{[\![\Gamma]\!] \vdash [\![p]\!] : \mathrm{El}\,prop}\ \textsc{Sound-Terms}}{[\![\Gamma]\!] \vdash \mathrm{Prf}[\![p]\!] : \texttt{Type}}\ \textsc{app}$$

  Then, with $[\![\Gamma']\!] = [\![\Gamma, x : \tau]\!] = [\![\Gamma]\!], x : \mathrm{El}\,\tau$,

$$\frac{\Gamma \vdash B(x) \qquad x \notin \Gamma}{\Gamma \vdash \bigwedge x \cdot B(x)}\ \textstyle\bigwedge\textsc{-Int} \qquad \text{gives}$$

$$\frac{\dfrac{\vdots}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} \mathrm{El}\,\tau : \texttt{Type}}\ \textsc{Ty} \quad \dfrac{\vdots}{[\![\Gamma']\!] \vdash^{\Sigma}_{\mathcal{R}} \mathrm{Prf}[\![B(x)]\!] : \texttt{Type}}\ \textsc{pr} \quad \dfrac{\vdots}{[\![\Gamma']\!] \vdash^{\Sigma}_{\mathcal{R}} H_B(x) : \mathrm{Prf}[\![B(x)]\!]}\ \textsc{induction}}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} (\lambda x : \mathrm{El}\,\tau \cdot H_B(x)) : \mathrm{Prf}\left[\!\!\left[\bigwedge x :: \tau \cdot B(x)\right]\!\!\right]}\ \lambda$$

- $\bigwedge$-Elim translates easily to $\textsc{app}$ ($\mathrm{Prf}\left[\!\!\left[\bigwedge x :: \tau \cdot B(x)\right]\!\!\right] = \Pi(x : \mathrm{El}\,\tau) \cdot \mathrm{Prf}[\![B(x)]\!]$):

$$\frac{\Gamma \vdash \bigwedge x \cdot B(x)}{\Gamma \vdash B(a)} \bigwedge\text{-Elim} \qquad \text{gives}$$

$$\frac{\dfrac{\vdots}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} H : \mathrm{Prf}\left[\!\!\left[\bigwedge x :: \tau \cdot B(x)\right]\!\!\right]}\ \textsc{induction} \qquad \dfrac{\vdots}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} [\![a]\!] : \mathrm{El}\,\tau}\ \textsc{Sound-Terms}}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} H\ [\![a]\!] : \mathrm{Prf}[\![B(a)]\!]}\ \textsc{app}$$

- $\Longrightarrow$-Int translates easily to λ ($\mathrm{Prf}[\![A \Longrightarrow B]\!] = (\mathrm{Prf}[\![A]\!]) \to (\mathrm{Prf}[\![B]\!])$), with $[\![\Gamma']\!] = [\![\Gamma, A]\!] = [\![\Gamma]\!], H_A : \mathrm{Prf}[\![A]\!]$:

$$\frac{\Gamma \vdash B}{\Gamma \setminus A \vdash A \Longrightarrow B} \Longrightarrow\text{-Int} \qquad \text{gives}$$

$$\frac{\dfrac{\vdots}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} \mathrm{Prf}[\![A]\!] : \mathtt{Type}}\ \textsc{pr} \quad \dfrac{\vdots}{[\![\Gamma']\!] \vdash^{\Sigma}_{\mathcal{R}} \mathrm{Prf}[\![B]\!] : \mathtt{Type}}\ \textsc{pr} \quad \dfrac{\vdots}{[\![\Gamma']\!] \vdash^{\Sigma}_{\mathcal{R}} H_B(H_A) : \mathrm{Prf}[\![B]\!]}\ \textsc{induction}}{[\![\Gamma]\!] \vdash^{\Sigma}_{\mathcal{R}} (\lambda H_A : \mathrm{Prf}\,A \cdot H_B(H_A)) : \mathrm{Prf}[\![A \Longrightarrow B]\!]}\ \lambda$$

- $\Longrightarrow$-Elim translates easily to app, using $\Gamma = \Gamma_1 \cup \Gamma_2$ ($\mathrm{Prf}[\![A \Longrightarrow B]\!] = (\mathrm{Prf}[\![A]\!]) \to (\mathrm{Prf}[\![B]\!])$):

$$\frac{\Gamma_1 \vdash A \qquad \Gamma_2 \vdash A \Longrightarrow B}{\Gamma_1 \cup \Gamma_2 \vdash B} \Longrightarrow\text{-Elim} \qquad \text{gives}$$

$$\frac{\dfrac{\vdots}{[\![\Gamma]\!] \vdash H_A : \mathrm{Prf}[\![A]\!]}\ \textsc{induction} \qquad \dfrac{\vdots}{[\![\Gamma]\!] \vdash H_{AB} : \mathrm{Prf}[\![A \Longrightarrow B]\!]}\ \textsc{induction}}{[\![\Gamma]\!] \vdash H_{AB}\ H_A : \mathrm{Prf}[\![B]\!]}\ \textsc{app}$$