Implementation of Two Layers Type Theory in Dedukti and Application to Cubical Type Theory

Bruno Barras, Valentin Maestracci

Implementation of Two Layers Type Theory in Dedukti and Application to Cubical Type Theory

Bruno Barras
Inria, Université Paris-Saclay,
ENS Paris-Saclay, CNRS, LSV,
91190, Gif-sur-Yvette, France.

Valentin Maestracci
Université Paris-Saclay,
ENS Paris-Saclay, CNRS, LSV,
91190, Gif-sur-Yvette, France.

In this paper, we make a substantial step towards an encoding of Cubical Type Theory (CTT) in the Dedukti logical framework. Type-checking CTT expressions features a decision procedure in a de Morgan algebra that so far could not be expressed by the rewrite rules of Dedukti. As an alternative, 2 Layer Type Theories are variants of Martin-Löf Type Theory where all or part of the definitional equality can be represented in terms of a so-called external equality. We propose to split the encoding by giving an encoding of 2 Layer Type Theories (2LTT) in Dedukti, and a partial encoding of CTT in 2LTT.

1 Introduction

The goal of this paper is to explore the possibility to express Homotopy Type Theory (HoTT, [9]) in the Dedukti [5] logical framework.

Dedukti is a logical framework which main distinctive feature is the possibility to extend the definitional equality (aka conversion) with a class of rewrite rules. It is intended to be used as a "hub" for proof systems. Many of the logics implemented by those systems can be encoded as *Dedukti theories*, and proofs in those systems can be expressed as Dedukti terms in the corresponding theory. The point is not just to collect the proofs of many logics, but rather to make it easier to translate proofs from one system to the other.

Expressing HoTT as a Dedukti theory is at the same time a problem that challenges the expressiveness of the rewrite rules accepted by Dedukti, but HoTT is an interesting formalism on its own. Moreover, in the long term, it would be interesting to have tools to investigate which proofs can be adapted from more conventional logics (HOL, set theory) to HoTT and conversely.

The paper is organized as follows. We first give an introduction on HoTT, Dedukti and encodings of Type Theory in Dedukti. This will motivate the introduction of Two Level Type Theories (2LTT) as a more flexible way to deal with type theories having a definitional equality two complex to be encoded as rewrite rules. In the second section, we will introduce 2LTTs and their encoding as a Dedukti theory. In the third section, we will give a partial encoding of Cubical as a 2LTT, and the corresponding piece of Dedukti theory.

Both encodings have been implemented and can be found here:

`https://github.com/valent20000/CTTDedukti`.

1.1 Homotopy Type Theories

In the broad sense, HoTT is based on an interpretation of Type Theory (ML, Coq, Agda, NuPRL) where types are topological spaces and proofs of an equality $x = y$ in type A are continuous paths between points x and y of space A.

Submitted to:
LFMTP 2020

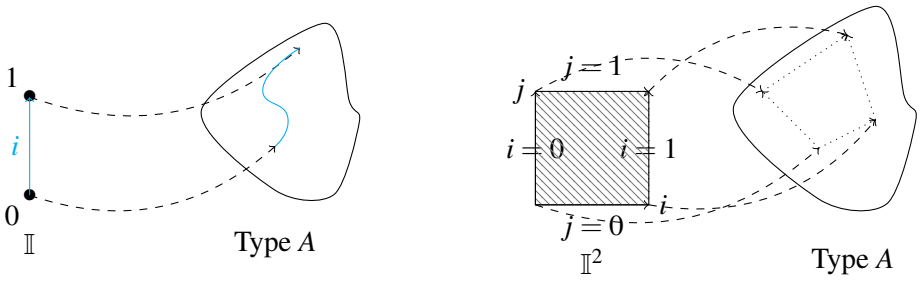

Figure 1: Meaning of judgments $i : \mathbb{I} \vdash t : A$ and $i, j : \mathbb{I} \vdash t : A$

This specific interpretation implies the possibility to extend the theory with new principles. The most famous one is *univalence* expressing that paths between types are exactly the weak equivalences (which is a particular case of isomorphism that deals with higher dimensions). Other extensions include higher inductive types, which are a generalization of inductive definitions. They are used to define spaces such as the circle, the torus, suspensions and many others.

Some members of the HoTT family are just regular Type Theory extended with axioms. The drawback of this approach is that axioms breaks metatheoretical properties such as canonicity. Worse, some useful notions such as simplicial sets seems impossible to express by mere axioms: face map equations require a coherence condition at dimension 2, which in turn requires another coherence condition at higher dimension, etc. This situation is often called "coherence hell".

In contrast, several members of the HoTT family, called *cubical* ([7],[2]), give a computational interpretation to univalence. In this paper we will focus on the Cubical Type Theory in [7] (called Cubical from now on) that we will briefly introduce in the next section. We believe that adapting this work to the others cubical theories should be easy.

1.2 Cubical Type Theory

The intuition behind Cubical is to follow the definition of paths as continuous functions from interval $[0;1]$ to the points of the topological space.

Cubical is a type theory introducing an interval pretype $\mathbb{I}$.[1]

Having an interval variable i in the context, a judgement $i : \mathbb{I} \vdash t : A$ represents a point t parameterized by the interval, hence a path in A (see Fig. 1, left). From that, one can define a type Path and a path constructor in a way similar to λ-abstraction. It is also possible to apply an expression to path to get back a point of A.

$$\frac{\Gamma \vdash A \text{ type}}{\Gamma \vdash \langle i \rangle t : \mathsf{Path}\ A\ t(i0)\ t(i1)} \qquad \frac{\Gamma \vdash p : \mathsf{Path}\ A\ x\ y \quad \Gamma \vdash e : \mathbb{I}}{\Gamma \vdash p\, e : A}$$

Formally, the interval $\mathbb{I}$ is defined in a synthetic way: as the free De Morgan algebra on $i, j, k \cdots$. Expanding the definition, this means its terms are elements of the following form : $0 \mid 1 \mid i \mid 1 - r \mid r \wedge s \mid r \vee s$, with $\wedge$ representing the inf and $\vee$ representing the sup of the elements.

Now if we have two interval variables, a judgment $i, j : \mathbb{I} \vdash t : A$ means t is a square in A, as illustrated by Fig. 1. Having n interval variable leads to a n-dimensional cube in A, hence the name of Cubical Type Theory.

[1] $\mathbb{I}$ is only a *pretype*, as it does not enjoy all properties of types: we should not identify 0 and 1 although they are connected by a path.

Some primitives of Cubical refer to expressions that may be defined only on a sub-polyhedra. To do that, we first describe the cube in a synthetic way like we did with the interval. A pretype $\mathbb{F}$ for the faces of the cube are defined with the following grammar:

- 1, the entire cube.
- 0, the empty face.
- $i = 0/i = 1$ the face where $i = 0/i = 1$
- $f_1 \wedge f_2$, the intersection of the faces f_1, f_2
- $f_1 \vee f_2$, the union of the faces f_1, f_2

Contexts may also contain a face to restrict judgments to a sub-polyhedra. For instance, the judgment $i, j : \mathbb{I}; i = 0 \vee j = 1 \vdash t : A$ represents the left and top edges of the square in Fig. 1.

In the general case, type-checking in Cubical features a decision procedure for the inclusion of faces. This is the main challenge in encoding Cubical in Dedukti.

1.3 Encoding Type Theories in Dedukti

Encoding a logic L in Dedukti usually consists in introducing a Dedukti theory (i.e. a set of constants and rewrite rules) $D(L)$, and a mapping $[\![]\!]_L$ from L-formulae to Dedukti types and from L-proofs to terms of type corresponding to the formula they prove.

In the case of Type Theory, one introduces a Dedukti type for "codes of types", and a decoding function that assigns a Dedukti type to each of these codes. Then, one introduces one constant for each type constructor, and constants for introduction and elimination rules.

The basic property of this encoding is that it must be well-typed, in the sense that a well-formed type must be translated to a well-typed Dedukti term. More specifically this requires that definitionally equal terms must be translated to convertible Dedukti terms. In other terms, we expect that the definitional equality can be expressed as rewrite rules.

As we have explained above, Cubical is a type theory which definitional equality includes the equational theory of a de Morgan algebra. It is far from obvious that it can actually be encoded by the Dedukti rewrite rules.

We prefer to investigate another approach, where part of the conversion is mapped to a kind of propositional equality. Unfortunately, we cannot express conversion as a propositional equality of Cubical (for the same reason that some notions in HoTT cannot be expressed by mere axioms).

Those remarks have led to the introduction of Two Level Type Theories [3].

1.4 Two Level Type Theories

We recall that in Type Theory there are two notions of equality:

- the propositional equality, that represents the intended equality of the logic
- the definitional equality, which in fact is a judgment, which represents objects that should be identified to ensure important properties of the judgments

Two-Layer Type Theories are a class of type theories. Their common point is that they are in fact made of two types theories (both copies of MLTT, with different additional axioms)

- **The internal**: It represents the theory we want to study. It is often equipped with Univalence Axiom, which makes its equality different than the usual equality, and incompatible with axioms like Uniqueness of Identity Proofs (UIP, or K) and functional extensionality (FunExt).
- **The external**: This theory will act as a sort of 'meta-theory' of the internal. We will use its propositional equality as an intermediate equality between the definitional ones (there are two definitional equalities here, the internal and the external), and the propositional equality of the internal theory. It has additional axioms (UIP & FunExt) to make its propositional equality not slightly weaker but as powerful as the usual one.

2 Two Layers Type Theories in Dedukti

2.1 Two Layer Type Theory as a Dedukti theory

In this section, we define Two Layers Type Theories by giving their encoding in Dedukti.

For a more comprehensive definition of 2LTT, we refer to [3], although we had to adapt the definition as they were defined semantically on some specific points.

2LTTs are basically two copies of Martin Löf's Type Theory: one internal layer and an external one. In order to avoid the complexity of having the notion of type and later introduce each universe as a subclass (as is usual in MLTT), we parameterize the notion of type by *levels*, that identify each universe, following Assaf (section 8.3 of [5]). We made the minimal assumptions of those levels, by just assuming a function `lsuc` such that universe l belongs to level `lsuc` l. The Dedukti declarations for that are:

```
Lev : Type.
lsuc : Lev -> Lev.
```

We can now introduce two codes of types: one for the internal layer (`T`) and one for the external one (`xT`), and their corresponding decoding functions `eps` and `xeps`. Let us point out that we have chosen a shallow embedding where 2LTT contexts are identified with Dedukti contexts. So codes of types are not explicitely parameterized by a notion of context.

```
T : Lev -> Type.
def eps : i : Lev -> T i -> Type.
```

```
xT : Lev -> Type.
def xeps : i:Lev -> T i -> Type.
```

In this encoding, an internal type A at level l is a term `A : T l`, an element t of that type A is a term `t : eps l A`. `T`, and likewise for external types.

The inclusion of a universe l in the bigger universe is taken care by first introducing a code in `t l : T (lsuc l)`, and a rewrite rule to assert that `t l` decodes to `T l`:

```
t : i : Lev -> T (lsuc i).
[i] eps (lsuc i) (t i) --> T i.
```

```
xt : i : Lev -> xT (lsuc i).
[i] xeps (lsuc i) (xt i) --> xT i.
```

It remains to lift each type of level l as a type of level `lsuc` l. We omit the external lift which is defined similarly.

```
lUp :  i : Lev -> a : T i -> T (lsuc i).
[i, a] eps (lsuc i) (lUp i a) --> eps i a.
```

The above rewrite rule relate codetypes at level l and their counterpart at level `lsuc` l in a very strong way: they decode to the same type, which means they have the *same* inhabitants.

We then implemented the usual primitive types of MLTT to populate the universes. As an example, internal dependent pairs are declared by introducing a constant `Sig` for the typecode, `pair` is the introduction rules, and `p1` and `p2` are the projections:

```
Sig : i : Lev -> A : T i -> (eps i A -> T i) -> T i.
def tSig := (i : Lev => A : T i => B : (eps i A -> T i)
  => eps i (Sig i A B)).

def pair : i : Lev -> A : T i -> B : (eps i A -> T i) ->
    a : eps i A -> b : eps i (B a) -> tSig i A B.
def p1 : i : Lev -> A : T i -> B : (eps i A -> T i) ->
    p : tSig i A B -> eps i A.
def p2 : i : Lev -> A : T i -> B : (eps i A -> T i) ->
    p : tSig i A B -> eps i (B (p1 i A B p)).

[i,A,B,a,b] p1 i A B (pair i A B a b) --> a.
[i,A,B,a,b] p2 i A B (pair i A B a b) --> b.
```

Actually, we define the following types, both at the internal and external layer, and at each level:

Internal	External
`False i : T i`	`xFalse i : xT i`
`True i : T i`	`xTrue i : xT i`
`Nat i : T i`	`xNat i : xT i`
`Pi i (A:T i)(B:x:eps i A->T i) : T i`	`xPi i (A:xT i)(B:x:xeps i A->xT i) : xT i`
`Sig i (A:T i)(B:x:eps i A->T i):T i`	`xSig i (A:xT i)(B:x:xeps i A->xT i):xT i`
`Sum i (A:T i) (B:T i) : T i`	`xSum i (A:xT i) (B:xT i) : xT i`
`Eq i (A:T i)(x:eps i A)(y:eps i A):T i`	`xEq i (A:xT i)(x:xeps i A)(y:xeps i A):xT i`

So far, 2LTTs feature two copies of MLTT, each one totally independent from the other. In order to include the internal layer into the external one, 2LTTs feature a coercion function `c` that assigns an external to each internal type.

```
def c  :  i : Lev -> T i -> xT i.
```

In [3], the coercion was defined in semantical terms. Here, coercion is such each internal type A is *isomorphic* to $\texttt{c}(A)$. By lifting internal types into the external world, the coercion allows us to see the internal world as a sort of sub-world of the external world. This allows to express properties of the internal world using the external equality.

This isomorphism can be encoded in different ways, from the most general to the most specific:

- Assuming the existence of functions between $\texttt{eps}(A)$ and $\texttt{xeps}(\texttt{c}(A))$, which are inverse one of each other, propositionally.
- Assuming the existence of functions between $\texttt{eps}(A)$ and $\texttt{xeps}(\texttt{c}(A))$, which are inverse one of each other, definitionally.
- Assuming that both (code)types decode to the same type.

The third option is similar to the one chosen for the universe inclusion, but the goal of 2LTTs is to be as general as possible, so we would better avoid it. However, the first one would probably need a coherence condition, and would make the system harder to use. For these reasons we have chosen the second option:

```
def isoUp :   i : Lev -> A : T i -> eps i A -> tc i A.
def isoDown : i : Lev -> A : T i -> tc i A  -> eps i A.
[i, A, a] isoDown i A (isoUp i A a) --> a.
[i, A, a] isoUp i A (isoDown i A a) --> a.
```

As stated before, 2LTTs are a class of type theories. They are a sort of 'à la carte' type theory where one can add additional axioms concerning coercion to tune it the way one wants it to be.

Like the definition of coercion, most of these axioms had a semantic definition. Here is a list of the ones (as introduced in [3]) we were able to express in a syntactic manner:

- Coercion can be required to be injective:

```
(; < T1 > ;)
def T1 : l : Lev -> A : T l -> B : T l ->
         p : xtTEq l (c l A) (c l B) -> tTEq l A B.
```

  where `xtTEq` and `tTEq` are shorthands for the types of elements of respectively external and internal equality

- The repletion axiom: an external type isomorphic to a $c(A)$ also has an antecedent by c.

```
(; < T3 > ;)
repletion : l : Lev -> A : xT l -> B : T l ->
            p : xtTEq l A (c l B) -> T l.
[l, A, B, e] c l (repletion l A B e) --> A.
```

- Integers are a fairly simple type with simple rules. One would expect $c(\mathbb{N})$ and $\mathrm{x}\mathbb{N}$ to be isomorphic, but it is actually not the case for technical reasons. There is only a morphism $\mathrm{x}\mathbb{N} \to c(\mathbb{N})$.

  A possible additional axiom is to make this morphism an isomorphism definitionally.

  The union, eq and false types can have similar additional axioms, while in the case of the product, sum and 1 types, the isomorphism is already there.

- One can also make these types isomorphic through that isomorphism not definitionally (ie by rewriting), but only up to external equality.

- One can make all these isomorphisms (including the ones for pi, sig, and 1, with example code below) equalities instead of just isomorphisms (cf option 3 for the coercion).

```
(; < T2 >  Primitive Isomorphisms c A ~ xA become equality ;)
[l] c l True --> xTrue l.
[l, A, B] c l (Pi l A B) --> xPi l (c l A) (clift l A B).
[l, A, B] c l (Sig l A B) --> xSig l (c l A) (clift l A B).
```

We implemented all the above axioms in Dedukti.

Interestingly, there was in the list of axioms that couldn't be implemented an axiom called $(A5)$, which requires that external equality validates the reflection rule (that is, externally equal types are considered definitionally equal). This could not be implemented in Dedukti. In it was possible, then we would be able to encode Cubical in Dedukti without resorting to 2LTTs.

We also tested the usability of this encoding by formulating the axiom of univalence. Here, for the sake of brevity, we only give the weaker form which states that the type $A = B$ (where A and B are types of level l) is weakly equivalent to $A \approx B$, the type of weak equivalence between A and B:

```
WeakUnivalence : l : Lev -> A : T l -> B : T l ->
  eps (lsuc l) (Equiv (lsuc l) (TEq l A B) (lUp l (Equiv l A B))).
```

Note that $A = B$ actually belongs to level `lsuc` l, hence the need to lift $A \approx B$ one universe up. Also, the notion of weak equivalence occurs twice but at different levels. This remark is the reason that made us opt for a presentation of type theories with a hierarchy of universes

2.2 Translating Two Layer Type Theories

We define a straightforward translation, with every type/term associated to the one with the same name in Dedukti, and the convention that variables share the same name in 2LTT and Dedukti:

- $[\![x]\!]_l = \texttt{x}$
- $[\![\underset{x:A}{\Sigma}(B)]\!]_l = \texttt{Sig l } [\![A]\!]_l \texttt{ (x: eps l } [\![A]\!]_l \texttt{ => } [\![B(x)]\!]_l\texttt{)}$
- $[\![\text{pair}_{[x:A]B}(a,b)]\!]_l = \texttt{pair l } [\![A]\!]_l \texttt{ (x: eps l } [\![A]\!]_l \texttt{ => } [\![B(x)]\!]_l\texttt{) } [\![a]\!]_l\ [\![b]\!]_l$
- $\cdots$

We also define the translation of context :

$$[\![x_1 :_{l_1} A_1, \cdots, x_n :_{l_n} A_n]\!] = \texttt{x1: eps } [\![A_1]\!]_{l_1}, \ldots, \texttt{xn: eps } [\![A_n]\!]_{l_n}.$$

As stated before when we first talked about how 2LTTs were implemented, encoding has the particularity that it encodes types, not into Dedukti types, but into type codes, that is elements of type $\texttt{T}\,l$ for internal types, and $\texttt{xT}\,l$ for external types. It encodes terms into terms of type the 'realization' of $[\![A]\!]_l$, that is $\texttt{eps}\ l\ [\![A]\!]_l$, and similarly for the external layer.

2.3 Soundness of the encoding

There are two important properties that the translation is expected to have:

Soundness: this means that the translation defined above preserves typability, and hence provability too. It also means that our encoding is powerful enough to prove everything that could be proven in the theory of 2LTTs.

Conservativity: this would mean that any property provable in the encoding can be proved in the thoery of 2LTTs, in other words that our encoding is not too powerful. From this we would be able to use Dedukti as a 2LTT type-checker.

While soundness is quite straightforward to prove (since all of the definitional equality of 2LTTs could be encoded by rewrite rules), conservativity is quite hard to prove.

The soundness property consists of 9 statements, one for each judgment kind:

THEOREM —

- If Γ 2LTT context, then $[\![\Gamma]\!]_l$ Dedukti context.
- If $\Gamma \vdash_{2L} A$ type l, then $[\![\Gamma]\!]_l \vdash_{Dk} [\![A]\!]_l : \texttt{T l}$
- If $\Gamma \vdash_{2L} A = A'$ type l, then $[\![\Gamma]\!]_l \vdash_{Dk} [\![A]\!]_l \leftrightarrow^* [\![A']\!]_l : \texttt{T l}$
- If $\Gamma \vdash_{2L} A$ type l, $\Gamma \vdash_{2L} t : A$, then $[\![\Gamma]\!]_l \vdash_{Dk} [\![t]\!]_l : \texttt{eps l } [\![A]\!]_l$
- If $\Gamma \vdash_{2L} A$ type l, $\Gamma \vdash_{2L} t = t' : A$ then $[\![\Gamma]\!]_l \vdash_{Dk} [\![t]\!]_l \leftrightarrow^* [\![t']\!]_l : \texttt{eps l } [\![A]\!]_l$
- If $\Gamma \vdash_{2L} A$ xtype l, then $[\![\Gamma]\!]_l \vdash_{Dk} [\![A]\!]_l : \texttt{xT l}$
- If $\Gamma \vdash_{2L} A = A'$ xtype l, then $[\![\Gamma]\!]_l \vdash_{Dk} [\![A]\!]_l \leftrightarrow^* [\![A']\!]_l : \texttt{xT l}$
- If $\Gamma \vdash_{2L} A$ xtype l, $\Gamma \vdash_{2L} t : A$, then $[\![\Gamma]\!]_l \vdash_{Dk} [\![t]\!]_l$: $\texttt{xeps l } [\![A]\!]_l$
- If $\Gamma \vdash_{2L} A$ xtype l,$\Gamma \vdash_{2L} t = t' : A$ then $[\![\Gamma]\!]_l \vdash_{Dk} [\![t]\!]_l \leftrightarrow^* [\![t']\!]_l$: $\texttt{xeps l } [\![A]\!]_l$

The proof is by mutual induction on the judgments.

Conservativity is a much harder problem, and we have not proven it yet. However, we make the following conjecture a conservativity result. Every Dedukti proof which context and type are in the image of the translation correspond to a 2LTT derivation:

CONJECTURE —
Given a Dedukti judgment

$$x_1 : T_1, \ldots, x_n : T_n \vdash_{Dk} t : A$$

where all T_is are of the form $\texttt{eps}\, l_i\, [\![U_i]\!]_{l_i}$ or $\texttt{xeps}\, l_i\, [\![U_i]\!]_{l_i}$ and A is convertible to a term of the form $\texttt{eps}\, l\, [\![B]\!]_l$ or $\texttt{xeps}\, l\, [\![B]\!]_l$, then there exists a 2LTT terms u such that

$$x_1 : U_1, \ldots, x_n : U_n \vdash_{2L} u : B$$

This obviously cannot hold if the logic of Dedukti is inconsistent (unless 2LTTs are themselves inconsistent). The idea of the proof is to translate only proofs in normal form, and assume strong normalization of the reduction rules of Dedukti. Another source of inspiration is [4], where conservativity is proven for an encoding of Pure Type Systems.

3 Cubical Type Theory in Dedukti

This section introduces a partial encoding of Cubical as an extension of the 2LTT Dedukti theory. We focused of the main primitive of Cubical: composition. As we have already mentionned, the typing rule of composition is probably beyond the capabilities of Dedukti's rewrite rules, and we expect 2LTTs to be a trade-off where expressivity is recovered at the cost of building parts of definitional equality derivations by hand. We try to express the largest fragment by rewrite rules, and the rest will be encoded in the external layer.

We do not consider the primitives related to glueing, which is the main feature that makes univalence provable in Cubical.

When viewing Cubical as an instance of a 2LTT, the leading idea is that the internal layer contains the object theory (Cubical) while the external layer is that of the meta-theory. More concretely, the internal layer contains the types of Cubical, while the external layer contains the pretypes ($\mathbb{I}$, and $\mathbb{F}$) and the judgments of Cubical.

We first introduce a level `cL` for the primitive pretypes of Cubical: $\mathbb{I}$ and $\mathbb{F}$ which are declared as external types.

```
cL : Lev.                def T := xT cL.          def ceps := xeps cL.
def cEq := xEq cL. (; and all types at level cL with prefix c ;)
```

Symbol `cEq` is used to express convertibility of preobjects. For the sake of conciseness we also define a symbol to represent convertibility of objects, using the coercion to lift internal objects to the external layer:

```
def CubicalEq (l : Lev) (A : T l) (a : eps l A) (b : eps l A) :=
      xEq l (c l A) (isoUp l A a) (isoUp l A b).
```

In order to interpret the conversion rule, we need more interaction between both layers, by allowing elimination of an external equality at level `cL` towards an internal type:

```
def CubicalJ :
  l:Lev -> A:cT -> x:ceps A -> P: (y:ceps A->cEq A x y->T l) ->
  eps l (P x (crefl A x)) ->
  y:ceps A -> e:cEq A x y -> eps l (P y e).
```

3.1 The interval pretype $\mathbb{I}$

Implementing the grammar of intervals and faces needs care, because the type-checking of Dedukti requires confluence of the set of rewrite rules. An algebra $(A, \vee, \wedge, 0, 1, \neg)$ is a De Morgan algebra if $\vee$ and $\wedge$ are associative and commutative and

$$x \wedge x = x \qquad x \wedge 0 = 0 \qquad x \wedge 1 = x$$
$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \qquad \neg(x \wedge y) = \neg x \vee \neg y \qquad \neg\neg x = x$$

The dual laws are derivable from these. In the currently distributed version of Dedukti, commutativity cannot be added as a rewrite rule, and idempotence being non-linear may break confluence. Zero, neutral, involution and De Morgan laws are straightforward. Associativity can be oriented in an arbitrary direction. Regarding distributivity, we cannot have a law and the dual one (neither the left nor the right one) or normalization is lost. Having the left and right laws at the same time creates a critical pair that cannot be closed without commutativity. So, we can have at most one of the four. Considering that distributivity is probably used only in very few cases, we decided to implement none as a rewrite rule. The rules that cannot be expressed as rewrite rules are thus stated as external equations.

```
I : cT.          0 : ceps I.        1 : ceps I.
def Imin : ceps I -> ceps I -> ceps I.
def Imax : ceps I -> ceps I -> ceps I.
def sym : ceps I -> ceps I.
(; rewrite rules, completed by symmetry ;)
[i] Imin 0 i --> 0  [i] Imin i 0 --> 0.
[i] Imin 1 i --> i [i] Imin i 1 --> i.
[i] Imax 0 i --> i  [i] Imax i 1 --> 0.
[i] Imax 1 i --> 1 [i] Imax i 1 --> 1.
[] sym 0 --> 1  [] sym 1 --> 0.
[i,j] sym (Imin i j) --> Imax (sym i) (sym j).
[i,j] sym (Imax i j) --> Imin (sym i) (sym j).
[i] sym (sym i) --> i.
[i,j,k] Imin (Imin i j) k --> Imin i (Imin j k).
[i,j,k] Imax (Imax i j) k --> Imax i (Imax j k).
(; properties expressed as external equations;
   Imin laws derived by duality ;)
Imax_idem : i:ceps I -> cEq (Imax i i) i.
Imax_comm : i:ceps I -> j:ceps I -> cEq I (Imax i j) (Imax j i).
Imax_dist : i : ceps I -> j : ceps I -> k : ceps I ->
   cEq (Imax (Imin i j) k) (Imin (Imax i k) (Imax j k).
```

3.2 Paths

We then define the type of paths together with its introduction and elimination rules. Since paths are types in Cubical, they are encoded in the internal layer.

```
def Path :   A : cT -> u : ceps A -> v : ceps A -> cT.
def lam : A : cT -> p : (ceps I -> ceps A) ->
     ceps (Path A (p 0) (p 1)).
def app : A : cT -> u : ceps A -> v : ceps A ->
     ceps (Path A u v) -> ceps I -> ceps A.
(; Computational rules ;)
[A,u,v,p] app A u v (lam A f) e --> f e.   (; beta ;)
[A,u,v,p] app A u v p 0 --> u  [A,u,v,p] app A u v p 1.
```

In the definition of Cubical, the last two definitional equalities above implement the rules

$$\frac{\Gamma \vdash p : \mathsf{Path}\ A\ u\ v}{\Gamma \vdash p\ 0 = u : A \qquad \Gamma \vdash p\ 1 = v : A}$$

which requires typing information about path p. This could be a problem since the conversion (and rewrite rules) of Dedukti is applied on terms without any typing information. Fortunately, in our encoding application is annotated with all of the information needed.

We also note that at this point paths are not related to the internal equality `Eq`.

It remains to express the key primitive of Cubical: composition. The typing rule involves the notions of interval variable (which are just external variables of type $\mathbb{I}$) and face ($\mathbb{F}$).

3.3 Faces

Let us first define faces, following the explanations in the introduction.

```
F : cT. (; Type of Faces;)
0f : ceps F. (; Empty face ;)
1f : ceps F. (; Whole cube ;)
def eq0 : ceps I -> ceps F. (; eq0 i is the face i = 0 ;)
def eq1 : ceps I -> ceps F. (; eq1 i is the face i = 1 ;)
def Fmin: ceps F -> ceps F -> ceps F. (; Intersection of faces ;)
def Fmax: ceps F -> ceps F -> ceps F. (; Union of faces ;)
(; Rewrite rules and equations (problem similar to the interval);)
[f] Fmin 0f f --> 0f.
...
Fdiscr : i : ceps I -> cEq F (Fmin (eq0 i) (eq1 i)) 0f.
```

We do not give details of how the algebraic properties of faces are turned into either rewrite rules or an equation. We only give the crucial property `Fdiscr` that there is no intersection between the opposite faces of a cube.

An important remark is that this does not exactly follow the syntax of the faces of Cubical, since the face $(i = 1)$ requires i to be a *variable*, which cannot be enforced in our shallow embedding. In Cubical when an interval variable i is substituted by an interval expression e, in a face $(i = 1)$, it is replaced following the rules (and similarly for $i = 0$):

$$(i=1)[i/0] = 0 \qquad (i=1)[i/1] = 1 \qquad (i=1)[i/1-e] = (i=0)[i/e]$$
$$(i=1)[i/e_1 \vee e_2] = (i=1)[i/e_1] \vee (i=1)[i/e_2]$$
$$(i=1)[i/e_1 \wedge e_2] = (i=1)[i/e_1] \wedge (i=1)[i/e_2]$$

This is quite naturally expressed by rewrite rules (straightfowardly adapted to `eq0`):

```
[] eq1 0 --> 0f       [] eq1 1 --> 1f    [e] eq1 (sym e) --> eq0 e.
[i,j] eq1 (Imax i j) --> Fmax (eq1 i) (eq1 j).
[i,j] eq1 (Imin i j) --> Fmin (eq1 i) (eq1 j).
```

Given a context Γ and a face ϕ of Γ, the context Γ, ϕ is a restriction of context Γ where the interval variables must belong to ϕ. Since we use a shallow embedding of context, we need to represent a face as an external type (actually a proposition) of witnesses that the interval variable belong to ϕ. So, the Dedukti type `F` above is a code of types with a decoding function `faceType`.

The definition one would like to make would thus be something along the lines of:

```
(; First attempt ;)
def faceType : ceps F -> cT.
[]     faceType 0f          --> cFalse.
[]     faceType 1f          --> cTrue.
[a]    faceType (eq1 a)     --> cEq I 1 a.
[a]    faceType (eq0 a)     --> cEq I 0 a.
[a, b] faceType (Fmax a b) --> cSum (faceType a) (faceType b).
[a, b] faceType (Fmin a b) --> cSig (faceType a) (_=>faceType b).
```

where we see that intersection (resp. union) is represented by the cartesian product (resp. sum), and the base constraint $(i = 0)$ by an external equality.

But this definition breaks confluence. Here is an example of critical pair:

```
faceType (Fmin 1f 1f) --> cSig cTrue (_=> cTrue)
faceType (Fmin 1f 1f) --> faceType 1f --> cTrue
```

If we try to recover confluence by closing this critical pair, the types `cTrue` and `cSig cTrue (_=> cTrue)` become convertible and hence allow to apply a projection to an inhabitant of `cTrue`. This destroys many good properties (e.g. canonicity) of the external layer, and this may lead to have erroneous Cubical proofs accepted by the encoding.

As a workaround, we decided to not have rewrite rules associated to `faceType`, but rather have an *isomorphism* between `faceType` ϕ and its intended type. We will also need that those types are actually propositions (i.e. all inhabitants must be equal). This is the case for faces 0, 1 and $\phi_1 \wedge \phi_2$. This holds also for $(i = 1)$ and $(i = 1)$ if the external layer enjoys Uniqueness of Identity Proofs (or axiom K). But having $\phi_1 \vee \phi_2$ isomorphic to a disjoint sum is a problem because the latter type may not be a proposition. We need a new external type, for instance a truncated sum. This type as the same introduction rules similar to disjoint sum, but the elimination rule requires a coherence condition (that will be given in the definition `TermSys` below).

3.4 Systems

We now focus on the Cubical notion of *system*, which allows to define functions whose value depends on where we are on the cube. A system is an expression of the form $[\phi_1 \to a \mid \phi_2 \to b]$, which evaluates to a on ϕ_1, and b on ϕ_2. This expression is defined on $\phi_1 \vee \phi_2$ which is called the extent of that system. Obviously, this makes sense only when a and b coincide on $\phi_1 \wedge \phi_2$ w.r.t. definitional equality. In Cubical, systems are valid terms only if their extent is equal to $1f$.

In our encoding, a system of type A and extent ϕ, is a term of type `ceps(faceType phi) -> eps 1 A`. The embedding into terms of type A is done by applying a proof that the current context is on face ϕ. So, a (binary) partial system is build with the following symbol:

```
def TermSys : l : Lev -> f1 : ceps F -> f2 : ceps F ->
```

```
  tau : T l ->
  A1 : (ceps(faceType f1) -> eps l tau) ->
  A2 : (ceps(faceType f2) -> eps l tau) ->
  coh : (e : ceps (faceType (Fmin f1 f2)) ->
         tCubicalEq l tau (A1 (fp1 f1 f2 e)) (A2 (fp2 f1 f2 e))) ->
  ceps (faceType (Fmax f1 f2)) ->  eps l tau.
```

which implements the rule

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, \phi_1 \vdash a_1 : A \qquad \Gamma, \phi_2 \vdash a_2 : A \qquad \Gamma, \phi_1 \wedge \phi_2 \vdash a_1 = a_2 : A}{\Gamma, \phi_1 \vee \phi_2 \vdash [\phi_1 \to a_1 \mid \phi_2 \to a_2] : A}$$

Actually, we implemented a more general rule where the first premisse is $\Gamma, \phi_1 \vee \phi_2 \vdash A$ type, but this adds a lot of technicalities on the handling of face witnesses. We shall not give all the details here, but making those witnesses irrelevent (either definitionaly or propositionaly) is necessary.

This condition and the fact that the theory uses dependent types makes the use of systems in practice really complex, especially when more than two branches are involved since one has to check multiple side conditions.

3.5 Composition and the example of filling

The formal definition of composition is

$$\frac{\Gamma \vdash \phi : \mathbb{F} \qquad \Gamma, i : \mathbb{I} \vdash A \text{ type} \qquad \Gamma, \phi, i : \mathbb{I} \vdash u : A \qquad \Gamma \vdash a_0 : A(i0)[\phi \to u(i0)]}{\Gamma \vdash \text{comp}^i\ A\ [\phi \to u]\ a_0\ :\ A(i1)[\phi \to u(i1)]}$$

where the notation $\Gamma \vdash t : A[\phi \to u]$ is a shorthand for $\Gamma \vdash t : A$ and $\Gamma, \phi \vdash u = u : A$, in other words t as type A and is definitionally equal to u on face ϕ. This cannot be expressed via rewriting. So we used external equality both for the premisse and the conclusion, which is split into two symbols:

```
def primCompTerm : l : Lev -> phi : ceps F -> A : (ceps I -> T l)
    -> u : (ceps (faceType phi) -> i : ceps I -> eps l (A i))
    -> a0 : eps l (A 0)
    -> (e : ceps (faceType phi)-> tCubicalEq l (A 0) a0 (u e 0))
    -> eps l (A 1).

def primCompEq : l : Lev -> phi : ceps F -> A : (ceps I -> T l)
     -> u : (ceps (faceDataType phi) -> i : ceps I -> eps l (A i))
    -> a0 : eps l (A 0)
    -> coh :(e:ceps(faceType phi -> tCubicalEq l (A 0) a0 (u e 0))
    -> e : ceps (faceType phi)
    -> tCubicalEq l (A 1) (primCompTerm l phi A u a0 coh) (u e 1).
```

Combining composition with systems, one can prove many important theorems such as transitivity of paths. To make an example use of our encoding as well as to test how usable it was in practice, we implemented the example of filling, which draws the line between a point a_0 and the composition $\text{comp}^i\ A\ [\phi \to u]\ a_0$.:

$$\text{fill}^i\ A\ [\phi \to u]\ a_0\ j\ :\ A(j)$$

Because of the verbosity of our encoding, making the proof by hand (*i.e.* creating the fill term) was quite complicated in Dedukti. The situation would be better with the development version of Dedukti which features an interactive proof construction engine.

3.6 Translation of Cubical expressions

The soundness and conservativity of the encoding (using an translation function) is clearly a quite hard problem. Here we will only sketch how those results should be obtained.

The first step is to translate Cubical terms into a Dedukti well-typed terms in the theory of 2LTTs extended with the Cubical-specific piece of theory described above. This is quite difficult since parts of the definitional equality of Cubical is translated to external equalities of 2LTTs. Given two Cubical-convertible types A and B, a term M of types A is also of type B, while in the 2LTT encoding, $[\![M]\!]$ has type $[\![A]\!]$, but getting a term of type $[\![B]\!]$ requires the transport principle `CubicalJ`. Formally, one would say that the translation domain is Cubical derivations rather than mere terms.

Proving the soundness of this translation raises the difficulty that the same term may be typed by different derivations (using conversion at different places), resulting in translated terms that may not be convertible. One important lemma is to show they are actually equal w.r.t. external equality. This is where it is important that the external equality satisfies axiom UIP and functional extensionality. We note that is problem is equivalent to the one of encoding extensional type theory into intensional type theory extended with the above two axioms, see [10]

The conservativity proof follows the same idea as the one of 2LTTs.

4 Conclusion

We gave an encoding of 2LTTs as a Dedukti theory, and specialized it to an encoding of a subsystem of Cubical Type Theory (excluding glueing).

The 2LTT encoding was rather straightforward, once the typing rules are expressed syntactically. Definitional equality could be encoded as rewrite rules, which made the soundness proof rather easy. However, we consider 2LTTs as an important logical framework to express theories which definitional equality is very rich. Other theories could be expressed in our encoding. Beyond the HoTT family of formalisms, we may cite CoqMTU [6], an extension of type theory with a decidable first-order theory.

The Cubical encoding required much more care. There are two main reasons for this. Firstly, Cubical is based on an algebraic structure (De Morgan algebra) which properties cannot be encoded easily as rewrite rules (commutativity, distributivity, idempotence). The answer to this problem may be to extend the expressivity of the rewrite rules of Dedukti: rewriting modulo AC is being considered, but it is not clear if this work is useful for theories with more properties, like having a unit element. The second reason is that the definitional equality of Cubical includes a decision procedure. Rewrite rules cannot inspect the context, so it does seem possible to encode it without giving up the shallow embedding for faces.

Future work

There are several directions that this work may take. Of course, one is to establish the soundness and conservativity results that we expect hold for our encodings.

We also plan to study how glueing can be added to the encoding. At first glance, the reduction rule associated to glue types cannot be expressed as rewrite rules, so there are probably several interesting problems to study there.

A more concrete concern would be to define the translation function from Cubical to our encoding. In this paper, we have done it by hand on several examples (like filling). It appeared to become very

complex when systems are involved. Writing an elaboration function that fills the gaps would be the next step before instrumenting Cubical proof systems to produce Dedukti terms that could be rechecked.

References

[1] Thorsten Altenkirch, Paolo Capriotti & Nicolai Kraus (2016): *Extending Homotopy Type Theory with Strict Equality*. doi:10.4230/LIPIcs.CSL.2016.21.

[2] Carlo Angiuli, Kuen-Bang Hou (Favonia) & Robert Harper (2018): *Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities*. In Dan R. Ghica & Achim Jung, editors: *27th EACSL Annual Conference on Computer Science Logic, CSL 2018, September 4-7, 2018, Birmingham, UK*, *LIPIcs* 119, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 6:1–6:17, doi:10.4230/LIPIcs.CSL.2018.6.

[3] Danil Annenkov, Paolo Capriotti, Nicolai Kraus & Christian Sattler (2017): *Two-Level Type Theory and Applications*.

[4] Ali Assaf (2015): *Conservativity of Embeddings in the lambda Pi Calculus Modulo Rewriting*. In Thorsten Altenkirch, editor: *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, July 1-3, 2015, Warsaw, Poland*, *LIPIcs* 38, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 31–44, doi:10.4230/LIPIcs.TLCA.2015.31.

[5] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant & Ronan Saillard (2016): *Dedukti: a logical framework based on the $\lambda\Pi$-calculus modulo theory*. *Unpublished manuscript.* Available at `http://www.lsv.fr/~dowek/Publi/expressing.pdf`.

[6] Bruno Barras, Jean-Pierre Jouannaud, Pierre-Yves Strub & Qian Wang (2011): *CoQMTU: A Higher-Order Type Theory with a Predicative Hierarchy of Universes Parametrized by a Decidable First-Order Theory*. In: *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*, IEEE Computer Society, pp. 143–151, doi:10.1109/LICS.2011.37.

[7] Cyril Cohen, Thierry Coquand, Simon Huber & Anders Mörtberg (2016): *Cubical Type Theory: a constructive interpretation of the univalence axiom*.

[8] Martin Hofmann (1997): *Syntax and semantics of dependent types*.

[9] The Univalent Foundations Program (2013): *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study.

[10] Théo Winterhalter, Matthieu Sozeau & Nicolas Tabareau (2019): *Eliminating reflection from type theory*. In Assia Mahboubi & Magnus O. Myreen, editors: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, ACM, pp. 91–103, doi:10.1145/3293880.3294095.