# How to write a translator to Dedukti

## The case of Agda

**Jesper Cockx**
**Thiago Felicissimo**

25 June 2022

# How to write a translator to Dedukti

Previous talks. How to define theories and write proofs in Dedukti (eg. the theory $\mathcal{U}$).

This talk. How to write an automatic translator from a proof assistant to Dedukti:

- General principles on writing such a translator
- Specific case of the Agda2Dedukti translator

# From Agda to Dedukti

1. Principles on translating from a proof assistant to Dedukti
2. What is Agda?
3. Encoding Agda in Dedukti
4. Implementation of Agda2Dedukti
5. Inductive types and dependent pattern matching
6. Universe polymorphism
7. Eta equality & irrelevance
8. Conclusion

# How to translate from a proof assistant to Dedukti

Step 0. Find/define a system $\mathcal{O}$ corresponding to the proof assistant's logic (not easy!)

Step 1. Define a Dedukti theory $D[\mathcal{O}] = (\Sigma, \mathcal{R})$ representing the object logic in Dedukti.

Step 2. Define a translation $[\![-]\!] : \Lambda_{\mathcal{O}} \to \Lambda_{DK}$. The pair $(D[\mathcal{O}], [\![-]\!])$ is an encoding of $\mathcal{O}$.

Step 3. Implement the translating function, making use of the APIs and other tools offered by the proof assistant.

# Not all encodings are created equal

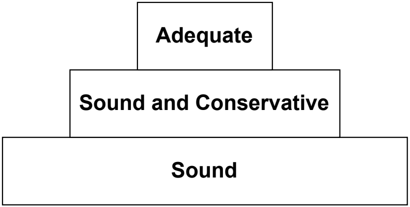
- An encoding is sound if:

$$\vdash_{\mathcal{O}} M : A \quad \text{implies} \quad \vdash_{D[\mathcal{O}]} [\![M]\!] : El\ [\![A]\!]$$

- An encoding is conservative if:

$$\vdash_{D[\mathcal{O}]} M : El\ [\![A]\!] \quad \text{implies} \quad \exists N,\ \vdash_{\mathcal{O}} N : A$$

- An encoding is adequate if for each type $A$:
  $[\![-]\!]$ is a *compositional bijection* between $A$ and $El\ [\![A]\!]$



# Nor are all proof assistants equal

The difficulty of encoding (the core language of) a proof assistant depends on its features:

Dependent types are in Coq, Agda, Lean, …

Inductive types are in most proof assistants.[1]

Universe polymorphism is in Coq, Agda, Lean, …

Impredicativity is in all proof assistants, except Agda and Epigram.

Eta-equality & irrelevance are present in different shapes in different proof assistants.

[1] Most type-theoretic proof assistants also support inductive families.

# Neither are their implementations

The difficulty of writing a translator also depends on the *implementation* of the proof assistant:

- In systems based on Curry-Howard (Coq/Agda/Matita), proof terms are already *in the internal syntax*, so are easier to translate.
- In LCF-like assistants (Isabelle/HOL), there are no proof terms, so we need to *reconstruct* them from proof derivations.
- In other systems (PVS), proofs derivations are not even internally available.[2]
- ...

[2] See Gabriel's talk for a solution.

# From Agda to Dedukti

# What is Agda?

Agda is a dependently typed programming language and proof assistant based on Martin-Löf type theory.

It has indexed datatypes, dependent pattern matching, and explicit universe polymorphism.

Its type checker identifies terms up to $\beta$-equality and $\eta$-equality for functions and records, and supports definitional proof irrelevance.

# Data types in Agda

$$
\begin{array}{l}
\textsf{data}\ \_\uplus\_\ (A\ B : \textsf{Set}) : \textsf{Set}\ \textsf{where} \\
\quad \textsf{left}\ \ : A \to A \uplus B \\
\quad \textsf{right} : B \to A \uplus B
\end{array}
$$

$$
\begin{array}{l}
\textsf{data}\ \_\leq\_ : \mathbb{N} \to \mathbb{N} \to \textsf{Set}\ \textsf{where} \\
\quad \leq\textsf{-zero} : \forall\ \{n\} \qquad\qquad\quad\ \ \to \textsf{zero}\ \ \ \ \leq n \\
\quad \leq\textsf{-suc}\ \ : \forall\ \{m\ n\} \to m \leq n \to \textsf{suc}\ m \leq \textsf{suc}\ n
\end{array}
$$

# Pattern matching in Agda

```
_<_ : ℕ → ℕ → Set
m < n = m ≤ suc n

compare : (m n : ℕ) → (m ≤ n) ⊎ (n < m)
compare zero    n       = left  ≤-zero
compare (suc m) zero    = right ≤-zero
compare (suc m) (suc n) with compare m n
... | left  m≤n         = left  (≤-suc m≤n)
... | right n<m         = right (≤-suc n<m)
```

# Agda as a PTS

At its core, Agda is a pure type system with sorts Set $\ell$ where $\ell$ is a universe level.

$$
\begin{aligned}
&\mathsf{U} : (\ell : \mathsf{Level}) \to \mathsf{Set}\ (\mathsf{lsuc}\ \ell) \\
&\mathsf{U}\ \ell = \mathsf{Set}\ \ell
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{prod} :\ & (\ell_1\ \ell_2 : \mathsf{Level}) \\
& (A : \mathsf{Set}\ \ell_1)\ (B : A \to \mathsf{Set}\ \ell_2) \\
& \to \mathsf{Set}\ (\ell_1 \sqcup \ell_2) \\
\mathsf{prod}\ \_\ \_\ & A\ B = (x : A) \to B\ x
\end{aligned}
$$

# From Agda to Dedukti

# Encoding Agda terms in Dedukti

| | | |
|---|---|---|
| Variable | $[\![x]\!]$ | $=$ |
| Def. symbol | $[\![\mathsf{f}]\!]$ | $=$ |
| Constructor | $[\![\mathsf{D.c}]\!]$ | $=$ |
| Lambda | $[\![\lambda x \to u]\!]$ | $=$ |
| Application | $[\![u\ v]\!]$ | $=$ |
| Pi type | $[\![(x : A) \to B]\!]$ | $=$ |
| Universe | $[\![\mathsf{Set}\ \ell]\!]$ | $=$ |

# Encoding Agda terms in Dedukti

| | | | |
|---|---|---|---|
| Variable | $[\![x]\!]$ | $=$ | $x$ |
| Def. symbol | $[\![\mathsf{f}]\!]$ | $=$ | `f` |
| Constructor | $[\![\mathsf{D.c}]\!]$ | $=$ | `D__c` |
| Lambda | $[\![\lambda x \to u]\!]$ | $=$ | |
| Application | $[\![u\ v]\!]$ | $=$ | |
| Pi type | $[\![(x : A) \to B]\!]$ | $=$ | |
| Universe | $[\![\mathsf{Set}\ \ell]\!]$ | $=$ | |

# Encoding Agda terms in Dedukti

| | | | |
|---|---|---|---|
| Variable | $[\![x]\!]$ | $=$ | $x$ |
| Def. symbol | $[\![\mathsf{f}]\!]$ | $=$ | `f` |
| Constructor | $[\![\mathsf{D.c}]\!]$ | $=$ | `D__c` |
| Lambda | $[\![\lambda x \to u]\!]$ | $=$ | $x \Rightarrow [\![u]\!]$ |
| Application | $[\![u\ v]\!]$ | $=$ | $[\![u]\!]\ [\![v]\!]$ |
| Pi type | $[\![(x : A) \to B]\!]$ | $=$ | |
| Universe | $[\![\mathsf{Set}\ \ell]\!]$ | $=$ | |

# Encoding Agda terms in Dedukti

| | | | |
|---|---|---|---|
| Variable | $[\![x]\!]$ | $=$ | $x$ |
| Def. symbol | $[\![\mathsf{f}]\!]$ | $=$ | `f` |
| Constructor | $[\![\mathsf{D.c}]\!]$ | $=$ | `D__c` |
| Lambda | $[\![\lambda x \to u]\!]$ | $=$ | $x \Rightarrow [\![u]\!]$ |
| Application | $[\![u\ v]\!]$ | $=$ | $[\![u]\!]\ [\![v]\!]$ |
| Pi type | $[\![(x : A) \to B]\!]$ | $=$ | ??? |
| Universe | $[\![\mathsf{Set}\ \ell]\!]$ | $=$ | ??? |

# Tarski- vs. Russell-style universes[3]

Agda uses Russell-style universes: Elements are *types* themselves.

$$\frac{A : \mathsf{Set}_l}{A \;\; \textsc{type}}$$

In Dedukti, if $A : \mathsf{Set}$, we cannot have $a : A$.
Thus, Dedukti uses a form of Tarski-style universes: Elements are *codes* that can be *interpreted* as types.

$$\frac{c : \mathtt{U}\ (\mathtt{set}\ l)}{\mathtt{El}\ (\mathtt{set}\ l)\ c \;\; \textsc{type}}$$

[3] https://www.cs.rhul.ac.uk/home/zhaohui/universes.pdf

# Encoding Agda's PTS in Dedukti

```
Sort : Type.
set  : Lvl -> Sort.

U : (s : Sort) -> Type.
def El : (s : Sort) -> (a : U s) -> Type.

def axiom : Sort -> Sort.
[i] axiom (set i) --> set (s i).

def rule : Sort -> Sort -> Sort.
[i, j] rule (set i) (set j) --> set (max i j).
```

(We postpone the definition of `Lvl` until later,
for now you can assume `lvl` $= \mathbb{N}$.)

# Encoding pi types

- Add a constant $\texttt{prod}$ for encoding the pi type:

$$\frac{A : \texttt{U}\ s_A \quad B : \texttt{El}\ s_A\ A \to \texttt{U}\ s_B}{\texttt{prod}\ s_A\ s_B\ A\ B : \texttt{U}\ (\texttt{rule}\ s_A\ s_B)}$$

# Encoding pi types

- Add a constant $\texttt{prod}$ for encoding the pi type:

$$\frac{A : \texttt{U}\ s_A \quad B : \texttt{El}\ s_A\ A \to \texttt{U}\ s_B}{\texttt{prod}\ s_A\ s_B\ A\ B : \texttt{U}\ (\texttt{rule}\ s_A\ s_B)}$$

- Identify elements of $\texttt{prod}$ with the *metatheoretic arrow type*:

$$\begin{aligned}&\texttt{El}\ \_\ (\texttt{prod}\ s_A\ s_B\ A\ B)\\ &\qquad = (x : \texttt{El}\ s_A\ A) \to El\ s_B\ (B\ x)\end{aligned}$$

# Encoding pi types in Dedukti

```
prod : (s_A : Sort) ->
       (s_B : Sort) ->
       (A : U s_A) ->
       (B : (El s_A A -> U s_B)) ->
       U (rule s_A s_B).

[s_A, s_B, A, B]
      El _ (prod s_A s_B A B)
  --> (x : El s_A A) -> El s_B (B x).
```

# Reconstructing sorts

For translating pi types, we need access to the sort of the domain and codomain.

Luckily, Agda's type checker already annotates each type $A$ with its sort $s(A)$.

**Examples.** $s(\mathbb{N}) = \mathsf{Set}$, $s(\mathsf{Set}) = \mathsf{Set}_1$, $s(\mathsf{Set}_1 \to \mathsf{Set}) = \mathsf{Set}_2$

# Encoding Agda terms in Dedukti

| | | | |
|---|---|---|---|
| Variable | $[\![x]\!]$ | $=$ | $x$ |
| Def. symbol | $[\![\mathsf{f}]\!]$ | $=$ | $\mathtt{f}$ |
| Constructor | $[\![\mathsf{D.c}]\!]$ | $=$ | $\mathtt{D\_\_c}$ |
| Lambda | $[\![\lambda x \to u]\!]$ | $=$ | $x \Rightarrow [\![u]\!]$ |
| Application | $[\![u\ v]\!]$ | $=$ | $[\![u]\!]\ [\![v]\!]$ |
| Pi type | $[\![(x : A) \to B]\!]$ | $=$ | ??? |
| Universe | $[\![\mathsf{Set}\ \ell]\!]$ | $=$ | ??? |

# Encoding Agda terms in Dedukti

| | | | |
|---|---|---|---|
| Variable | $[\![x]\!]$ | $=$ | $x$ |
| Def. symbol | $[\![\mathsf{f}]\!]$ | $=$ | $\texttt{f}$ |
| Constructor | $[\![\mathsf{D.c}]\!]$ | $=$ | $\texttt{D\_\_c}$ |
| Lambda | $[\![\lambda x \to u]\!]$ | $=$ | $x \Rightarrow [\![u]\!]$ |
| Application | $[\![u\ v]\!]$ | $=$ | $[\![u]\!]\ [\![v]\!]$ |
| Pi type | $[\![(x : A) \to B]\!]$ | $=$ | $\texttt{prod}\ \lvert s(A) \rvert\ \lvert s(B) \rvert\ [\![A]\!]\ (x \Rightarrow [\![B]\!])$ |
| | where $\lvert \mathsf{Set}\ \ell \rvert$ | $=$ | $\mathsf{set}\ [\![\ell]\!]$ |
| Universe | $[\![\mathsf{Set}\ \ell]\!]$ | $=$ | ??? |

(We will see how to translate levels later.)

# Encoding universe codes

- Add a constant $\mathtt{u}$ for encoding the Set type:

$$\frac{s : \mathtt{Sort}}{\mathtt{u}\ s : \mathtt{U}\ (\mathtt{axiom}\ s)}$$

# Encoding universe codes

- Add a constant $\mathtt{u}$ for encoding the Set type:

$$\frac{s : \mathtt{Sort}}{\mathtt{u}\ s : \mathtt{U}\ (\mathtt{axiom}\ s)}$$

- Identify elements of $\mathtt{u}\ s$ with the ones of $\mathtt{U}\ s$:

$$\mathtt{El}\ \_\ (\mathtt{u}\ s) = \mathtt{U}\ s$$

In Dedukti:

```
u : (s : Sort) -> U (axiom s).
[i] El _ (u s) --> U s.
```

# Encoding Agda terms in Dedukti

| | | | |
|---|---|---|---|
| Variable | $[\![x]\!]$ | $=$ | $x$ |
| Def. symbol | $[\![\mathsf{f}]\!]$ | $=$ | $\mathtt{f}$ |
| Constructor | $[\![\mathsf{D.c}]\!]$ | $=$ | $\mathtt{D\_\_c}$ |
| Lambda | $[\![\lambda x \to u]\!]$ | $=$ | $x \Rightarrow [\![u]\!]$ |
| Application | $[\![u\ v]\!]$ | $=$ | $[\![u]\!]\ [\![v]\!]$ |
| Pi type | $[\![(x : A) \to B]\!]$ | $=$ | $\mathtt{prod}\ \lvert s(A)\rvert\ \lvert s(B)\rvert$ $[\![A]\!]\ (x \Rightarrow [\![B]\!])$ |
| | where $\lvert \mathsf{Set}\ \ell \rvert$ | $=$ | $\mathsf{set}\ [\![\ell]\!]$ |
| Universe | $[\![\mathsf{Set}\ \ell]\!]$ | $=$ | ??? |

(We will see how to translate levels later.)

# Encoding Agda terms in Dedukti

$$
\begin{array}{lrcl}
\text{Variable} & [\![x]\!] & = & x \\
\text{Def. symbol} & [\![\mathsf{f}]\!] & = & \mathtt{f} \\
\text{Constructor} & [\![\mathsf{D.c}]\!] & = & \mathtt{D\_\_c} \\
\text{Lambda} & [\![\lambda x \to u]\!] & = & x \Rightarrow [\![u]\!] \\
\text{Application} & [\![u\ v]\!] & = & [\![u]\!]\ \ [\![v]\!] \\
\text{Pi type} & [\![(x : A) \to B]\!] & = & \mathtt{prod}\ |s(A)|\ |s(B)| \\
 & & & \quad [\![A]\!]\ \ (x \Rightarrow [\![B]\!]) \\
 & \text{where } |\mathsf{Set}\ \ell| & = & \mathsf{set}\ \ [\![\ell]\!] \\
\text{Universe} & [\![\mathsf{Set}\ \ell]\!] & = & \mathtt{u}\ (\mathsf{set}\ \ [\![\ell]\!])
\end{array}
$$

(We will see how to translate levels later.)

# Encoding Agda definitions in Dedukti

Data types (no parameters or indices)

$$\left[\!\!\left[ \begin{array}{l} \mathsf{data}\ \mathsf{D} : U\ \mathsf{where} \\ \quad \mathsf{c} : A \end{array} \right]\!\!\right] = \begin{array}{l} \mathtt{D} : \mathtt{El}\ |s(U)|\ \ [\![U]\!]\ . \\ \mathtt{D\_\_c} : \mathtt{El}\ |U|\ \ [\![A]\!]\ . \end{array}$$

Function definitions (no pattern matching)

$$\left[\!\!\left[ \begin{array}{l} \mathsf{f} : A \\ \mathsf{f}\ x = v \end{array} \right]\!\!\right] = \begin{array}{l} \mathtt{def\ \ f} : \mathtt{El}\ |s(A)|\ \ [\![A]\!]\ . \\ \mathtt{[x]\ \ f\ \ x\ \ -\!->}\ \ [\![v]\!]\ . \end{array}$$

# From Agda to Dedukti

1. Principles on translating from a proof assistant to Dedukti
2. What is Agda?
3. Encoding Agda in Dedukti
4. Implementation of Agda2Dedukti
5. Inductive types and dependent pattern matching
6. Universe polymorphism
7. Eta equality & irrelevance
8. Conclusion

# Implementation of Agda2Dedukti

Agda2Dedukti is implemented as an Agda backend.

This allows us to reuse parts of Agda's implementation:

- Internal syntax representation
- Type checking monad `TCM`

# Structure of the Agda typechecker

$$
\begin{array}{rcccc}
 & \boxed{\text{.agda file}} & & & \\
\textit{lexer \& parser} & \Downarrow & & & \\
 & \boxed{\text{Concrete syntax}} & & & \\
\textit{scope checker} & \Downarrow & & & \\
 & \boxed{\text{Abstract syntax}} & & & \\
\textit{type checker} & \Downarrow & & & \\
 & \boxed{\text{Internal syntax}} & & & \\
\textit{optimizer} & \Downarrow & & & \\
 & \boxed{\text{Treeless syntax}} & & & \\
\textit{MAlonzo} & \Downarrow & & & \\
 & \boxed{\text{.hs file}} & \xRightarrow{\textit{GHC}} & \boxed{\text{Binary}} &
\end{array}
$$

# Structure of the Agda typechecker

.agda file

*lexer & parser* ⇓

Concrete syntax

*scope checker* ⇓

Abstract syntax

*type checker* ⇓

Internal syntax $\xRightarrow{\textit{Agda2Dk}}$ .dk file

*optimizer* ⇓

Treeless syntax

*MAlonzo* ⇓

.hs file $\xRightarrow{\textit{GHC}}$ Binary

# Agda's internal syntax[4]

```
data Term
  = Var Int Elims                  -- x u v ..
  | Lam ArgInfo (Abs Term)         -- λ x → v
  | Lit Literal                    -- 42, 'a', ...
  | Def QName Elims                -- f u v ..
  | Con ConHead ConInfo Elims      -- c u v ..
  | Pi (Dom Type) (Abs Type)       -- (x : A) → B
  | Sort Sort                      -- Set, Set₁, Prop, ...
  | Level Level                    -- lzero, ...
  | MetaV MetaId Elims             -- _X_235
  | DontCare Term
  | Dummy String Elims
```

[4] Code from `Agda.Syntax.Internal`

# Agda's TCM monad

Agda's typechecker uses a type-checking monad **TCM**:

```
type TCM a
getConstInfo :: QName -> TCM Definition
getBuiltin :: String -> TCM Term
getContext :: TCM Context
addContext :: (Name, Dom Type) -> TCM a -> TCM a
checkInternal :: Term -> Type -> TCM ()
reconstructParameters :: Type -> Term -> TCM Term
...
```

# Putting it all together

$$\text{example} : (1 \leq 2) \uplus (2 < 1)$$
$$\text{example} = \text{left}\ (\leq\text{-suc}\ \leq\text{-zero})$$

```
{|!_⊎___left|}
  ({|!_≤_|}
    (Nat__suc Nat__zero)
    (Nat__suc (Nat__suc Nat__zero)))
  ({|!_<_|}
    (Nat__suc (Nat__suc Nat__zero))
    (Nat__suc Nat__zero))
  ({|!_≤___≤-suc|}
    Nat__zero
    (Nat__suc Nat__zero)
    ({|!_≤___≤-zero|} (Nat__suc Nat__zero)))
```

# From Agda to Dedukti

# Translating datatypes and constructors to constants

Data types and their constructors do not reduce, so we translate them to constants in Dedukti.

**Example.** $\_\leq\_$ is translated to:

```
{|!_≤_|} : El (set (s 0)) (prod (set 0) (set (s 0))
  Nat (_0 => (prod (set 0) (set (s 0))
    Nat (_0 => (u (set 0)))))).

{|!_≤___≤-zero|} : El (set 0) (prod (set 0) (set 0)
  Nat (n => ({|!_≤_|} Nat__zero n))).

{|!_≤___≤-suc|} : El (set 0) (prod (set 0) (set 0) Nat
  (m => (prod (set 0) (set 0)
    Nat (n => (prod (set 0) (set 0)
      ({|!_≤_|} m n)
      (_0 => ({|!_≤_|} (Nat__suc m) (Nat__suc n)))))))).
```

# Reconstruction of data parameters

Constructors in Agda do *not* store their parameters.

Reconstructing parameters requires a type-directed traversal of the syntax.

We can reuse Agda's `reconstructParameters`, which does exactly this!

# Filling implicit arguments & reconstructing parameters

$\mathsf{left}\ (\leq\text{-suc}\ \leq\text{-zero}) : (1 \leq 2) \uplus (2 < 1)$

# Filling implicit arguments & reconstructing parameters

Agda's type checker infers implicit arguments during type checking.

$$\mathsf{left}\ (\leq\text{-}\mathsf{suc}\ \leq\text{-}\mathsf{zero}) : (1 \leq 2) \uplus (2 < 1)$$
$$\Downarrow$$
$$\mathsf{left}\ (\leq\text{-}\mathsf{suc}\ \{\mathsf{m} = 0\}\ \{\mathsf{n} = 1\}\ (\leq\text{-}\mathsf{zero}\ \{\mathsf{n} = 1\}))$$

# Filling implicit arguments & reconstructing parameters

Agda's type checker infers implicit arguments during type checking.

Agda2Dk makes all implicit arguments explicit and reconstructs constructor parameters.

$$\text{left } (\leq\text{-suc } \leq\text{-zero}) : (1 \leq 2) \uplus (2 < 1)$$
$$\Downarrow$$
$$\text{left } (\leq\text{-suc } \{m = 0\} \; \{n = 1\} \; (\leq\text{-zero } \{n = 1\}))$$
$$\Downarrow$$
$$\text{left } (1 \leq 2) \; (2 < 1) \; (\leq\text{-suc } 0 \; 1 \; (\leq\text{-zero } 1))$$

# Translating clauses to rewrite rules

Functions in Agda are defined by a set of clauses, so we translate them to a constant + a set of rewrite rules.

**Example.** compare is translated to:

```
def compare : El (set 0) (prod (set 0) (set 0)
  Nat (m => (prod (set 0) (set 0)
    Nat (n => ({|!_⊎_|} ({|!_≤_|} m n) ({|!_<_|} n m)))))).
[n] compare Nat__zero n -->
  {|!_⊎___left|} ({|!_≤_|} Nat__zero n)
    ({|!_<_|} n Nat__zero) ({|!_≤___≤-zero|} n).
[m] compare (Nat__suc m) Nat__zero -->
  {|!_⊎___right|} ({|!_≤_|} (Nat__suc m) Nat__zero)
    ({|!_<_|} Nat__zero (Nat__suc m))
    ({|!_≤___≤-zero|} (Nat__suc (Nat__suc m))).
[m, n] compare (Nat__suc m) (Nat__suc n) -->
  {|!with-66|} m n (compare m n).
```

# Drawbacks of generating rewrite rules

Generating a new rewrite rule for each clause means that we are extending the theory with each definition.

Moreover, checking correctness (completeness & termination) of rewrite rules is very hard.

**Ongoing work:** Instead, we can translate definitions by pattern matching to eliminators.[5]

```
def compare := Nat__ind...
```

[5] Ask Thiago for details!

# From Agda to Dedukti

1. Principles on translating from a proof assistant to Dedukti
2. What is Agda?
3. Encoding Agda in Dedukti
4. Implementation of Agda2Dedukti
5. Inductive types and dependent pattern matching
6. Universe polymorphism
7. Eta equality & irrelevance
8. Conclusion

# Universe polymorphism

Sometimes one wishes to use a definition at multiple universes (e.g. List Nat but also List $\mathrm{Set}_0$).

.

# Universe polymorphism

Sometimes one wishes to use a definition at multiple universes (e.g. List Nat but also List $\text{Set}_0$).

**Bad solution.** Define a new $\text{List}_i$ for each level $i$.

# Universe polymorphism

Sometimes one wishes to use a definition at multiple universes (e.g. List Nat but also List $\mathsf{Set}_0$).

**Bad solution.** Define a new $\mathsf{List}_i$ for each level $i$.

**Universe polymorphism** allows definitions that can be used at multiple universe levels:

```
data List {i} (A : Set i) : Set i where
  [] : List A
  _::_ : A → List A → List A

map : {i j : Level} → {A : Set i} → {B : Set j}
  → (f : A → B) → List A → List B
map f [] = []
map f (x :: l) = f x :: map f l
```

# Other forms of universe polymorphism

Universe polymorphism in Agda is very different from universe polymorphism in Coq:

| | Coq | Agda |
|---|---|---|
| Typical ambiguity | | |
| Cumulativity ($Set_i \subseteq Set_{i+1}$) | | |
| Definitions carry constraints | | |

# Other forms of universe polymorphism

Universe polymorphism in Agda is very different from universe polymorphism in Coq:

| | Coq | Agda |
|---|---|---|
| Typical ambiguity | Yes | No |
| Cumulativity ($Set_i \subseteq Set_{i+1}$) | | |
| Definitions carry constraints | | |

# Other forms of universe polymorphism

Universe polymorphism in Agda is very different from universe polymorphism in Coq:

| | Coq | Agda |
|---|---|---|
| Typical ambiguity | Yes | No |
| Cumulativity ($Set_i \subseteq Set_{i+1}$) | | |
| Definitions carry constraints | | |

# Other forms of universe polymorphism

Universe polymorphism in Agda is very different from universe polymorphism in Coq:

| | Coq | Agda |
|---|---|---|
| Typical ambiguity | Yes | No |
| Cumulativity ($Set_i \subseteq Set_{i+1}$) | Yes | No |
| Definitions carry constraints | | |

# Other forms of universe polymorphism

Universe polymorphism in Agda is very different from universe polymorphism in Coq:

| | Coq | Agda |
|---|---|---|
| Typical ambiguity | Yes | No |
| Cumulativity ($Set_i \subseteq Set_{i+1}$) | Yes | No |
| Definitions carry constraints | Yes | No |

# Other forms of universe polymorphism

Universe polymorphism in Agda is very different from universe polymorphism in Coq:

| | Coq | Agda |
|---|---|---|
| Typical ambiguity | Yes | No |
| Cumulativity ($Set_i \subseteq Set_{i+1}$) | Yes | No |
| Definitions carry constraints | Yes | No |

In this talk we only see the encoding of Agda's universe polymorphism.

For Coq's version, see Gaspard Ferey's PhD thesis.

# Universe polymorphism in Dedukti

**Idea.** Generalize the encoding of the arrow type:

```
setOmega  : Sort.

forall : (l : (Lvl -> Sort)) ->
         ((i : Lvl) -> U (l i)) -> U setOmega.

[l, t] El _ (forall l t) -->
           (i : Lvl) -> El (l i) (t i).
```

# Universe polymorphism in Dedukti

**Idea.** Generalize the encoding of the arrow type:

```
setOmega  : Sort.

forall : (l : (Lvl -> Sort)) ->
         ((i : Lvl) -> U (l i)) -> U setOmega.

[l, t] El _ (forall l t) -->
          (i : Lvl) -> El (l i) (t i).
```

We extend the translation function with:

| | | | |
|---|---|---|---|
| Level quantification | $[\![(i : Level) \to A]\!]$ | $=$ | $\texttt{forall}\ (i \Rightarrow [\![s(A)]\!])\ (i \Rightarrow [\![A]\!])$ |
| Level application | $[\![M\ l]\!]$ | $=$ | $[\![M]\!]\ [\![l]\!]$ |
| Level abstraction | $[\![\lambda i.M]\!]$ | $=$ | $i \Rightarrow [\![M]\!]$ |

# Back to List

Now the constant `List` can be given the type:

```
El setOmega
   (forall (i => set (suc i))
           (i => prod (set (suc i))
                      (set (suc i))
                      (u (set i))
                      (_ => u (set i))))
```

Which, as expected, computes to:

```
(i : Lvl) -> U (set i) -> U (set i)
```

# Universe levels

Levels are given by the syntax:

$$l, l_1, l_2 ::= i \mid \mathsf{lzero} \mid \mathsf{lsuc}\ l \mid l_1 \sqcup l_2\,.$$

# Universe levels

Levels are given by the syntax:

$$l, l_1, l_2 ::= i \mid \textsf{lzero} \mid \textsf{lsuc}\ l \mid l_1 \sqcup l_2\,.$$

Levels are not freely generated, they satisfy:

# Universe levels

Levels are given by the syntax:

$$l, l_1, l_2 ::= i \mid \mathsf{lzero} \mid \mathsf{lsuc}\ l \mid l_1 \sqcup l_2\,.$$

Levels are not freely generated, they satisfy:

Idempotence: $a \sqcup a = a$

# Universe levels

Levels are given by the syntax:

$$l, l_1, l_2 ::= i \mid \mathsf{lzero} \mid \mathsf{lsuc}\ l \mid l_1 \sqcup l_2\,.$$

Levels are not freely generated, they satisfy:

Idempotence: $a \sqcup a = a$

Associativity: $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$

# Universe levels

Levels are given by the syntax:

$$l, l_1, l_2 ::= i \mid \text{lzero} \mid \text{lsuc}\ l \mid l_1 \sqcup l_2\,.$$

Levels are not freely generated, they satisfy:

Idempotence: $a \sqcup a = a$

Associativity: $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$

Commutativity: $a \sqcup b = b \sqcup a$

# Universe levels

Levels are given by the syntax:

$$l, l_1, l_2 ::= i \mid \mathsf{lzero} \mid \mathsf{lsuc}\ l \mid l_1 \sqcup l_2 \,.$$

Levels are not freely generated, they satisfy:

Idempotence: $a \sqcup a = a$

Associativity: $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$

Commutativity: $a \sqcup b = b \sqcup a$

Distributivity: $\mathsf{lsuc}\ (a \sqcup b) = \mathsf{lsuc}\ a \sqcup \mathsf{lsuc}\ b$

# Universe levels

Levels are given by the syntax:

$$l, l_1, l_2 ::= i \mid \mathsf{lzero} \mid \mathsf{lsuc}\ l \mid l_1 \sqcup l_2\,.$$

Levels are not freely generated, they satisfy:

Idempotence: $a \sqcup a = a$

Associativity: $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$

Commutativity: $a \sqcup b = b \sqcup a$

Distributivity: $\mathsf{lsuc}\ (a \sqcup b) = \mathsf{lsuc}\ a \sqcup \mathsf{lsuc}\ b$

Neutrality: $a \sqcup \mathsf{lzero} = a$

# Universe levels

Levels are given by the syntax:

$$l, l_1, l_2 ::= i \mid \mathsf{lzero} \mid \mathsf{lsuc}\ l \mid l_1 \sqcup l_2\,.$$

Levels are not freely generated, they satisfy:

Idempotence: $a \sqcup a = a$

Associativity: $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$

Commutativity: $a \sqcup b = b \sqcup a$

Distributivity: $\mathsf{lsuc}\ (a \sqcup b) = \mathsf{lsuc}\ a \sqcup \mathsf{lsuc}\ b$

Neutrality: $a \sqcup \mathsf{lzero} = a$

Subsumption: $a \sqcup \mathsf{lsuc}^n\ a = \mathsf{lsuc}^n\ a$

# The challenge of representing universe polymorphism

To establish the encoding's soundness,

$$l_1 \equiv l_2 \text{ should imply } [\![ l_1 ]\!] \equiv [\![ l_2 ]\!]$$

Possible solutions:

# The challenge of representing universe polymorphism

To establish the encoding's soundness,

$$l_1 \equiv l_2 \text{ should imply } [\![l_1]\!] \equiv [\![l_2]\!]$$

Possible solutions:

1. Representing levels as naturals?

# The challenge of representing universe polymorphism

To establish the encoding's soundness,

$$l_1 \equiv l_2 \text{ should imply } [\![ l_1 ]\!] \equiv [\![ l_2 ]\!]$$

Possible solutions:

1. Representing levels as naturals? *Open terms do not satisfy all equalities (e.g. $i \sqcup j \not\equiv j \sqcup i$).*

# The challenge of representing universe polymorphism

To establish the encoding's soundness,

$$l_1 \equiv l_2 \text{ should imply } [\![l_1]\!] \equiv [\![l_2]\!]$$

Possible solutions:

1. Representing levels as naturals? *Open terms do not satisfy all equalities (e.g. $i \sqcup j \not\equiv j \sqcup i$).*
2. Representing levels as a set of variables with natural increments? **(current solution)**

# The challenge of representing universe polymorphism

To establish the encoding's soundness,

$$l_1 \equiv l_2 \text{ should imply } [\![l_1]\!] \equiv [\![l_2]\!]$$

Possible solutions:

1. Representing levels as naturals? *Open terms do not satisfy all equalities (e.g. $i \sqcup j \not\equiv j \sqcup i$).*
2. Representing levels as a set of variables with natural increments? **(current solution)** *Works well, but there is a catch (next slide).*

# The challenge of representing universe polymorphism

To establish the encoding's soundness,

$$l_1 \equiv l_2 \text{ should imply } [\![l_1]\!] \equiv [\![l_2]\!]$$

Possible solutions:

1. Representing levels as naturals? *Open terms do not satisfy all equalities (e.g. $i \sqcup j \not\equiv j \sqcup i$).*
2. Representing levels as a set of variables with natural increments? **(current solution)** *Works well, but there is a catch (next slide).*
3. Decision procedure integrated in Dedukti?

# The challenge of representing universe polymorphism

To establish the encoding's soundness,

$$l_1 \equiv l_2 \text{ should imply } [\![l_1]\!] \equiv [\![l_2]\!]$$

Possible solutions:

1. Representing levels as naturals? *Open terms do not satisfy all equalities (e.g. $i \sqcup j \not\equiv j \sqcup i$).*
2. Representing levels as a set of variables with natural increments? **(current solution)** *Works well, but there is a catch (next slide).*
3. Decision procedure integrated in Dedukti? *We leave this to the future generations.*

# Current solution: levels as sets

**Idea.** Every level $l$ admits a unique canonical form

$$l = \max\{n, i_1 + m_1, ..., i_k + m_k\}$$

where $i_1, .., i_k \in FV(l)$, $n, m_1, .., m_k \in \mathbb{N}$ and $m_j \leq n$.

# Current solution: levels as sets

**Idea.** Every level $l$ admits a unique canonical form

$$l = \max\{n, i_1 + m_1, ..., i_k + m_k\}$$

where $i_1, .., i_k \in FV(l)$, $n, m_1, .., m_k \in \mathbb{N}$ and $m_j \leq n$.

A rewrite system can calculate such forms by using rewriting modulo associativity-commutativity.

# Current solution: levels as sets

**Idea.** Every level $l$ admits a unique canonical form

$$l = \max\{n, i_1 + m_1, ..., i_k + m_k\}$$

where $i_1, .., i_k \in FV(l)$, $n, m_1, .., m_k \in \mathbb{N}$ and $m_j \leq n$.

A rewrite system can calculate such forms by using rewriting modulo associativity-commutativity.

But idempotence and subsumption require a non-linear rule:

$$\max\{i + n, i + m\} = i + \max\{n, m\}$$

# Current solution: levels as sets

**Idea.** Every level $l$ admits a unique canonical form

$$l = \max\{n, i_1 + m_1, ..., i_k + m_k\}$$

where $i_1, .., i_k \in FV(l)$, $n, m_1, .., m_k \in \mathbb{N}$ and $m_j \leq n$.

A rewrite system can calculate such forms by using rewriting modulo associativity-commutativity.

But idempotence and subsumption require a non-linear rule:

$$\max\{i + n, i + m\} = i + \max\{n, m\}$$

This *breaks confluence of pre-terms*, and prevents proving conservativity.

# From Agda to Dedukti

# Eta equality in Agda

Agda supports two kinds of eta-equality:

1. Eta for functions:

$$\frac{f : (x : A) \to B}{f = (\lambda x \to f\ x) : (x : A) \to B}$$

2. Eta for records:[6]

$$\frac{u : \Sigma\ A\ B}{u = (\mathsf{proj}_1\ u, \mathsf{proj}_2\ u) : \Sigma\ A\ B}$$

[6] Also known as surjective pairing for $\Sigma$.

# Definitional singleton types

Agda supports eta for *all* record types, not just Σ!
In particular, it has eta for the unit type:

```
record ⊤ : Set where -- no fields
  constructor tt

eta-unit : (x y : ⊤) → x ≡ y
eta-unit x y = refl
```

*Two distinct variables might be equal*!

⇒ To check if two terms are convertible, it does not suffice to compare their normal forms.

# Encoding eta in Dedukti

1. Eta-expand everything when translating?

# Encoding eta in Dedukti

1. Eta-expand everything when translating?
   *This is not stable under substitution:*

   $$(\lambda a : A.a)\{\mathbb{N} \to \mathbb{N}/A\}$$

   *is not in eta-long form, but* $\lambda a : A.a$ *and* $\mathbb{N} \to \mathbb{N}$ *are.*

# Encoding eta in Dedukti

1. Eta-expand everything when translating?
   *This is not stable under substitution:*

   $$(\lambda a : A.a)\{\mathbb{N} \to \mathbb{N}/A\}$$

   *is not in eta-long form, but* $\lambda a : A.a$ *and* $\mathbb{N} \to \mathbb{N}$ *are.*
2. Eta-reduce everything when translating?

# Encoding eta in Dedukti

1. Eta-expand everything when translating?
   *This is not stable under substitution:*

   $$(\lambda a : A.a)\{\mathbb{N} \to \mathbb{N}/A\}$$

   *is not in eta-long form, but* $\lambda a : A.a$ *and* $\mathbb{N} \to \mathbb{N}$ *are.*
2. Eta-reduce everything when translating?
   *This is not stable under substitution and* $\beta$*:*

   $$(\lambda x.y\ x\ x)\{(\lambda\_.z)/y\} \longrightarrow_\beta \lambda x.z\ x \longrightarrow_\eta z$$

   *but* $\lambda x.y\ x\ x \not\longrightarrow_\eta$ *and* $\lambda\_.z \not\longrightarrow_\eta$.

# Encoding eta in Dedukti

3. Add eta-equality to the metatheory?

# Encoding eta in Dedukti

3. Add eta-equality to the metatheory?
   *This only handles eta for the arrow type.*

# Encoding eta in Dedukti

3. Add eta-equality to the metatheory?
   *This only handles eta for the arrow type.*

4. Use eta-reduction for record types?

# Encoding eta in Dedukti

3. Add eta-equality to the metatheory?
   *This only handles eta for the arrow type.*

4. Use eta-reduction for record types?
   *This does not work for unit type, and needs non-linearity for the others:*
   ```
   mk_pair (pi_1 p) (pi_2 p) --> p
   ```

# Encoding eta in Dedukti

3. Add eta-equality to the metatheory?
   *This only handles eta for the arrow type.*

4. Use eta-reduction for record types?
   *This does not work for unit type, and needs non-linearity for the others:*
   `mk_pair (pi_1 p) (pi_2 p) --> p`

5. Annotate terms with their types to be able to match them to eta expand? e.g.
   `eta (arrow nat nat) f --> x => f x`

# Encoding eta in Dedukti

3. Add eta-equality to the metatheory?
   *This only handles eta for the arrow type.*

4. Use eta-reduction for record types?
   *This does not work for unit type, and needs non-linearity for the others:*
   `mk_pair (pi_1 p) (pi_2 p) --> p`

5. Annotate terms with their types to be able to match them to eta expand? e.g.
   `eta (arrow nat nat) f --> x => f x`
   *We get bigger terms, and the other rules make the system non-confluent on pre-terms. Moreover, variables not translated as variables.*

# Encoding eta in Dedukti

**The next idea.** Extend Dedukti with typed-directed rewrite rules.

Take inspiration from already existing works:

- Agda's implementation of eta[7]
- Andromeda 2's extensionality rules[8]

Or maybe there are still other unexplored options?

[7] A. Bauer, A. Petković, An extensible equality checking algorithm for dependent type theories

[8] `https://agda.readthedocs.io/en/v2.6.2.2/language/record-types.html`

# Definitional irrelevance

Agda also supports definitional proof irrelevance[9] for irrelevant functions and elements of Prop:

```
postulate
  P : Prop
  f : P → ℕ

P-irrelevant : (x y : P) → f x ≡ f y
P-irrelevant x y = refl
```

This causes very similar problems to eta for ⊤, that also requires type-directed conversion to solve.

[9] In the encoding of PVS we have a simpler form of proof irrelevance, which can be encoded in Dedukti.

# From Agda to Dedukti

# Summary

Many features of a dependently typed language can be encoded in Dedukti directly:

- Defined symbols are mapped to constants.
- Clauses are mapped to rewrite rules.

Other features require some more work:

- Erased constructor parameters need to be reconstructed.
- Universe levels require an equational theory.

Finally, other features we don't yet know how to encode:

- Eta-equality for record types?
- Definitional proof irrelevance?

# Future work

Like most translators, Agda2Dedukti is still a work in progress.

In the future, we would like to have:

- Compilation of clauses to elimination principles,
- A conservative encoding of universe polymorphism,
- An adequate and computational encoding of Agda,[10]
- An encoding of eta-equality and irrelevance (probably requires extending Dedukti).

[10] For details, see Thiago's talk about Adequate and Computational Encodings in Dedukti, at FSCD 2022

# References


- G. Genestier. Encoding Agda Programs Using Rewriting. In Proceedings of the 5th International Conference on Formal Structures for Computation and Deduction, Leibniz International Proceedings in Informatics 167, 2020.[11]
- T. Felicissimo. Representing Agda and coinduction in the lambda-pi calculus modulo rewriting. Master thesis, 2021.[12]


[11] https://drops.dagstuhl.de/opus/volltexte/2020/12353/pdf/LIPIcs-FSCD-2020-31.pdf

[12] https://hal.inria.fr/hal-03343699