Connecting Agda to other theorem provers via EuroProofNet

or, how to implement an Agda backend

Jesper Cockx

11 November 2022

From Agda to Dedukti

1. EuroProofNet and Dedukti
2. Encoding Agda in Dedukti
3. Translating Agda to Dedukti
4. Encoding universe polymorphism and eta equality
5. Conclusion

What is EuroProofNet?[1]

- A European network of researchers working on improving interoperability between proof systems.
- 300 researchers from 40 different countries (you're welcome to join too!).
- Centered around Dedukti as an intermediate language for translation.

[1] `https://europroofnet.github.io/`

What is Dedukti?[2]

- A logical framework based on the $\lambda\Pi$-calculus with higher-order rewrite rules.
- Used for checking and transforming proofs, as well as translating between proof systems.
- Has a small core, so it should be easy to implement many different independent checkers (currently 3).
- Has encodings of many proof assistants: Coq, Isabelle/HOL, Matita, ...
- Also has encodings of other proof systems: Zenon Modulo, iProver, veriT, ...

[2] `https://deducteam.github.io/`

One-slide Dedukti tutorial

```
ty  : Type.
arr : ty -> ty -> ty.
```

One-slide Dedukti tutorial

```
ty  : Type.
arr : ty -> ty -> ty.

tm  : ty -> Type.
lam : a:ty -> b:ty
   -> (tm a -> tm b) -> tm (arr a b).
def id : a:ty -> tm (arr a a)
  := a => lam a a (x => x).
```

One-slide Dedukti tutorial

```
ty  : Type.
arr : ty -> ty -> ty.

tm  : ty -> Type.
lam : a:ty -> b:ty
   -> (tm a -> tm b) -> tm (arr a b).
def id : a:ty -> tm (arr a a)
  := a => lam a a (x => x).

def app : a:ty -> b:ty
        -> tm (arr a b) -> tm a -> tm b.
[u,v] app _ _ (lam _ _ u) v --> u v.
def twice : a:ty -> tm (arr (arr a a) (arr a a))
  := a => lam (arr a a) (arr a a) (f =>
          lam a a (x => app a a f (app a a f x))).
```

A success story

Recently, Thiago Felicissimo and Frédéric Blanqui (CSL '23)[3] translated a big mathematical library from Matita to Agda via Dedukti.

This includes proofs of **Bertrand's Postulate** and **Fermat's Little Theorem**, which were so far not done in Agda.

Hence translating between proof assistants can *already save us work today*!

[3] `http://files.inria.fr/blanqui/pred.pdf`

Agda2Dedukti

To connect Agda to EuroProofNet, we need to:

- Encode basic logic of Agda in Dedukti
- Translate Agda files to this encoding in Dedukti

This is the goal of **Agda2Dedukti**.

From Agda to Dedukti

1. EuroProofNet and Dedukti
2. Encoding Agda in Dedukti
3. Translating Agda to Dedukti
4. Encoding universe polymorphism and eta equality
5. Conclusion

Encoding Agda's universe levels

```
Lvl : Type.
0   : Lvl.
S   : Lvl -> Lvl.

def max : Lvl -> Lvl -> Lvl.
[n]   max 0     n     --> n.
[m]   max m     0     --> m.
[m,n] max (S m) (S n) --> S (max m n).
```

(Universe polymorphism comes later.)

Tarski- vs. Russell-style universes[4]

Agda uses Russell-style universes:
Elements of a universe are types themselves.

$$\frac{A : \mathsf{U}_l}{A \ \ \text{TYPE}}$$

Dedukti uses a form of Tarski-style universes:
Elements are codes that can be *interpreted* as types.

$$\frac{A : \mathsf{U}_l}{\texttt{El}\ A \ \ \text{TYPE}}$$

$\Rightarrow$ Encoding Agda in Dedukti takes some overhead.

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

Encoding Agda's sorts

We define Agda's sort system[5] in three easy steps:

1. Define a Dedukti type of Agda sorts, and add a concrete sort `set` *l* for each level *l*:

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

[5] or at least, a simplified version of it

Encoding Agda's sorts

We define Agda's sort system[5] in three easy steps:

1. Define a Dedukti type of Agda sorts, and add a concrete sort `set` *l* for each level *l*:

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

2. Interpret each sort *s* as a Dedukti type `U` *s*:

```
U : s : Sort -> Type.
```

[5] or at least, a simplified version of it

Encoding Agda's sorts

We define Agda's sort system[5] in three easy steps:

1. Define a Dedukti type of Agda sorts, and add a concrete sort `set` *l* for each level *l*:

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

2. Interpret each sort *s* as a Dedukti type `U` *s*:

   ```
   U : s : Sort -> Type.
   ```

3. Interpret each type *a* : `U` *s* as a Dedukti type `El` *s* *a*:

   ```
   def El : s : Sort -> a : U s -> Type.
   ```

[5] or at least, a simplified version of it

Encoding Agda's universes

Problem: Now Set_l is a *sort*, but not yet a *term*!

Encoding Agda's universes

Problem: Now Set_l is a *sort*, but not yet a *term*!

Solution:

1. For each sort s, define its successor `axiom` s:

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

Encoding Agda's universes

Problem: Now Set_l is a *sort*, but not yet a *term*!

Solution:

1. For each sort *s*, define its successor `axiom` *s*:

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

2. Add a constant `u` internalizing any sort *s* as a term `u` *s* of sort `axiom` *s*:

```
u : s : Sort -> U (axiom s).
```

Encoding Agda's universes

Problem: Now Set_l is a *sort*, but not yet a *term*!

Solution:

1. For each sort s, define its successor `axiom` s:

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

2. Add a constant `u` internalizing any sort s as a term `u` s of sort `axiom` s:

   ```
   u : s : Sort -> U (axiom s).
   ```

3. Identify elements of `u` s with the ones of `U` s:

   ```
   [s] El _ (u s) --> U s.
   ```

Encoding Agda's function types

Function types follow the same pattern:

1. For two types $A : s_1$ and $x : A \vdash B : s_2$, define the sort `rule` s_1 s_2 of the pi type $(x : A) \to B$:

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

Encoding Agda's function types

Function types follow the same pattern:

1. For two types $A : s_1$ and $x : A \vdash B : s_2$, define the sort `rule` s_1 s_2 of the pi type $(x : A) \to B$:

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

2. Add a constant `prod` for encoding the pi type:

```
prod : s1 : Sort -> s2 : Sort ->
  A : U s1 -> B : (El s1 A -> U s2) ->
  U (rule s1 s2).
```

Encoding Agda's function types

Function types follow the same pattern:

1. For two types $A : s_1$ and $x : A \vdash B : s_2$, define the sort `rule` s_1 s_2 of the pi type $(x : A) \to B$:

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

2. Add a constant `prod` for encoding the pi type:

```
prod : s1 : Sort -> s2 : Sort ->
  A : U s1 -> B : (El s1 A -> U s2) ->
  U (rule s1 s2).
```

3. Identify elements of `prod` with Dedukti's arrow type:

```
[s1, s2, A, B] El _ (prod s1 s2 A B)
            --> x : El s1 A -> El s2 (B x).
```

From Agda to Dedukti

1. EuroProofNet and Dedukti
2. Encoding Agda in Dedukti
3. Translating Agda to Dedukti
4. Encoding universe polymorphism and eta equality
5. Conclusion

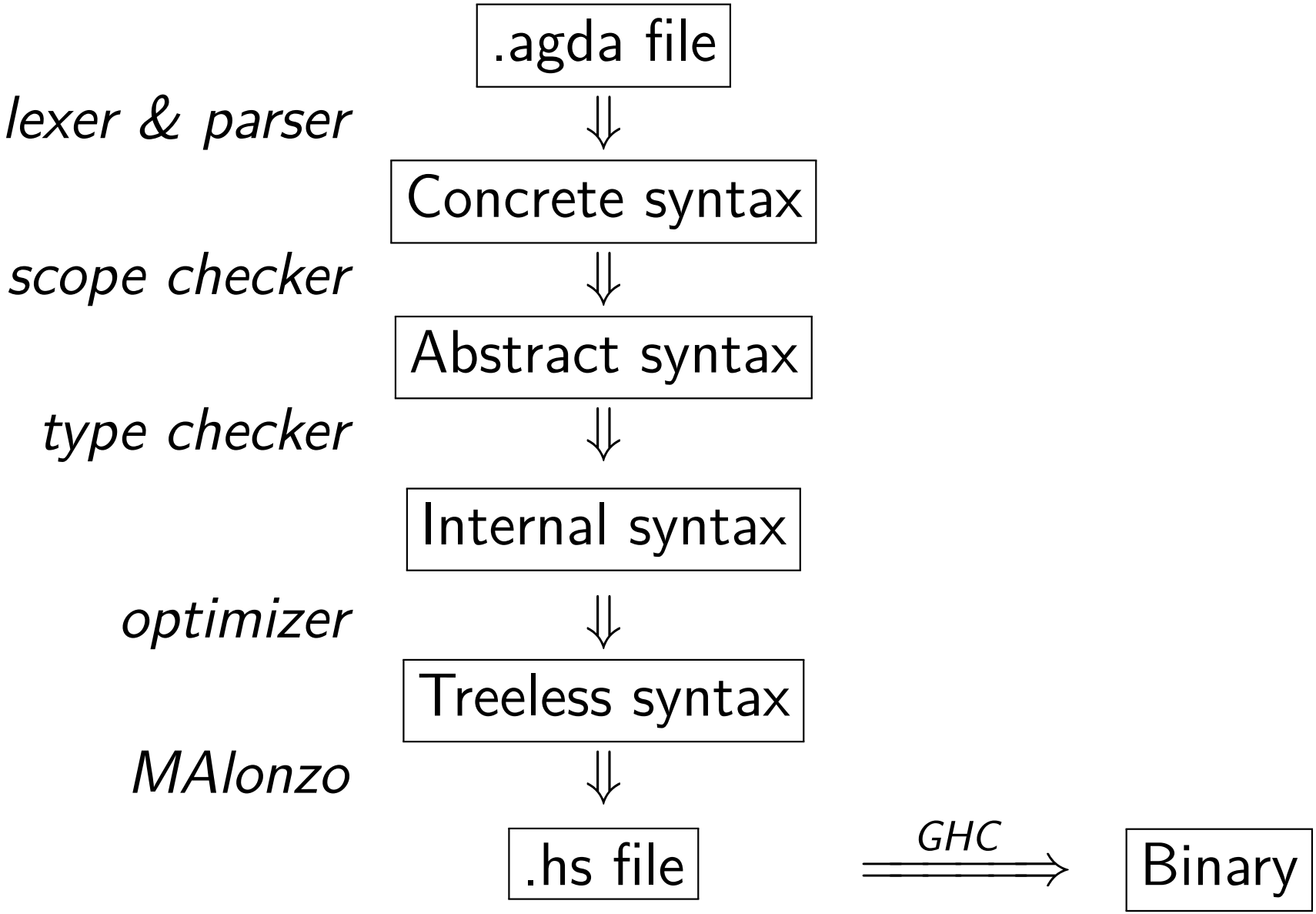
Implementation of Agda2Dedukti

Agda2Dedukti is implemented as an Agda backend.

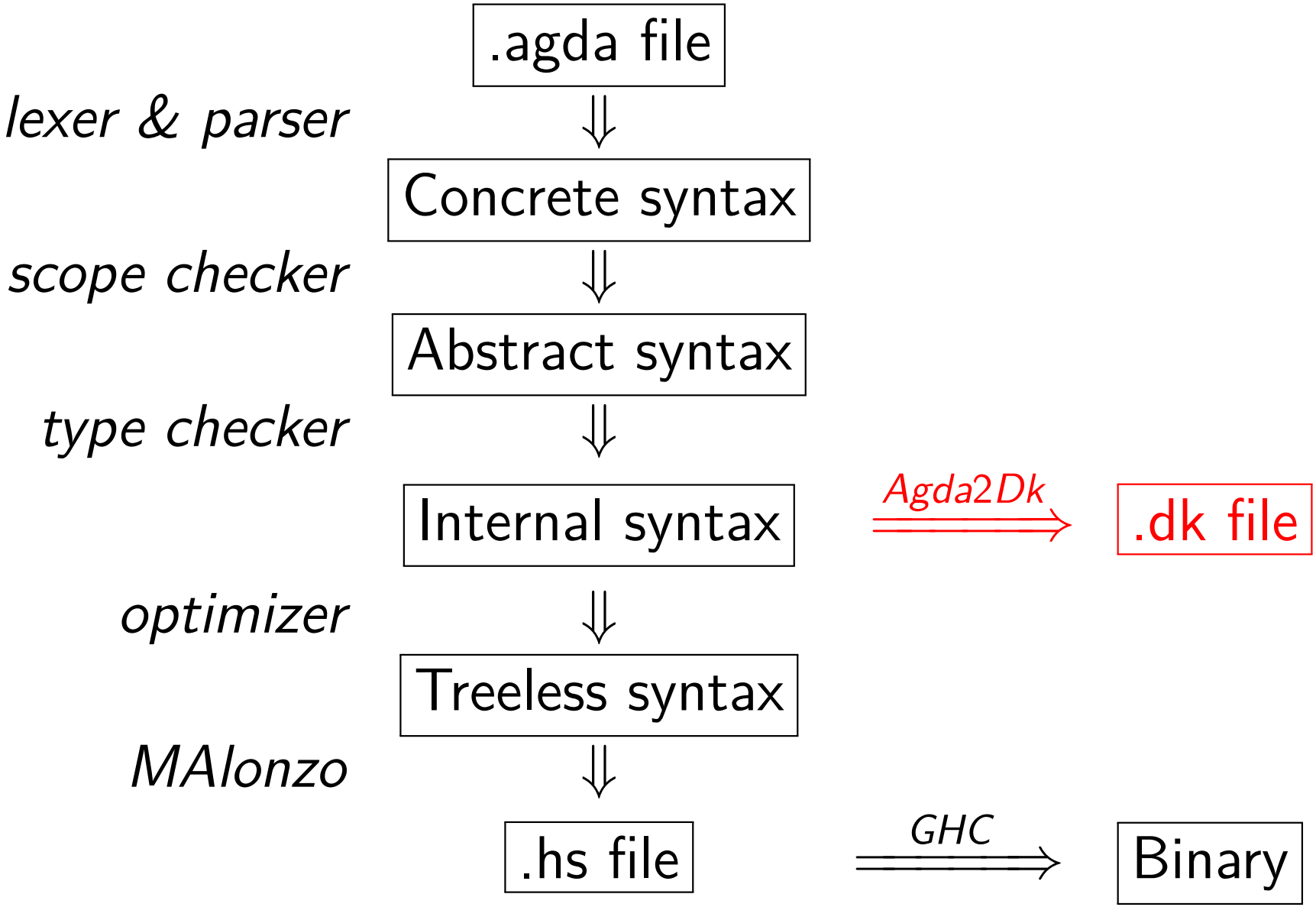
This allows us to reuse parts of Agda's implementation:

- Internal syntax representation
- Type checking monad `TCM`

Structure of the Agda typechecker



Structure of the Agda typechecker



Agda's internal syntax[6]

```
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
```

[6] 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
...
```

Translating terms

Variable	$[\![x]\!]$	$=$	x
Def. symbol	$[\![\mathsf{f}]\!]$	$=$	$\mathtt{f}$
Constructor	$[\![\mathsf{c}]\!]$	$=$	$\mathtt{c}$
Lambda	$[\![\lambda x \to u]\!]$	$=$	$x \Rightarrow [\![u]\!]$
Application	$[\![u\ v]\!]$	$=$	$[\![u]\!]\ [\![v]\!]$
Pi type	$[\![(x : A) \to B]\!]$	$=$	$\mathtt{prod}\ [\![\mathtt{sortOf}(A)]\!]_{\mathtt{s}}$ $[\![\mathtt{sortOf}(B)]\!]_{\mathtt{s}}$ $[\![A]\!]\ (x \Rightarrow [\![B]\!])$
Universe	$[\![s]\!]$	$=$	$\mathtt{u}\ [\![s]\!]_{\mathtt{s}}$
Sort	$[\![\mathsf{Set}\ \ell]\!]_{\mathtt{s}}$	$=$	$\mathtt{set}\ [\![\ell]\!]$

Translating datatypes and constructors

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

Example. Nat is translated to:

```
Nat  : U (set 0).
zero : El (set 0) Nat.
suc  : El (set 0)
  (prod (set 0) (set 0) Nat (_ => Nat)).
```

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.** pred is translated to:

```
def pred : El (set O)
  (prod (set O) (set O) Nat (_ => Nat)).
[]  pred zero     --> zero.
[m] pred (suc m) --> m.
```

$\Rightarrow$ We extend the theory with each definition.

[7] Contact Thiago for details!

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.** pred is translated to:

```
def pred : El (set O)
  (prod (set O) (set O) Nat (_ => Nat)).
[]  pred zero     --> zero.
[m] pred (suc m) --> m.
```

$\Rightarrow$ We extend the theory with each definition.

Ongoing work: Instead, we can translate definitions by pattern matching to eliminators.[7]

[7] Contact Thiago for details!

From Agda to Dedukti

1. EuroProofNet and Dedukti
2. Encoding Agda in Dedukti
3. Translating Agda to Dedukti
4. Encoding universe polymorphism and eta equality
5. Conclusion

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:

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

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

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.

Eta equality for records

1. Eta for Σ:

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

Eta equality for records

1. Eta for Σ:

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

2. Eta for $\top$:

$$\frac{u : \top}{u = \mathsf{tt} : \top}$$

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

Encoding eta

1. Eta-expand everything when translating?

Encoding eta

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

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

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

Encoding eta

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

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

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

Encoding eta

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

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

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

3. Annotate terms with their types to be able to match them to eta expand? e.g.

```
eta (arrow nat nat) f --> x => f x
```

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

Encoding eta

3. 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.

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

Encoding eta

3. 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.

4. **The next idea.** Extend Dedukti with typed-directed rewrite rules, taking inspiration from Andromeda 2's extensionality rules.[8]

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

From Agda to Dedukti

1. EuroProofNet and Dedukti
2. Encoding Agda in Dedukti
3. Translating Agda to Dedukti
4. Encoding universe polymorphism and eta equality
5. Conclusion

Summary

Many parts of Agda can be encoded directly:

- Defined symbols $\mapsto$ constants
- Clauses $\mapsto$ rewrite rules

Other features require some more work:

- Reconstruction of constructor parameters
- Equational theory for universe levels

Yet other features we don't yet know how to encode (e.g. eta-equality for records).

Future work

- Compilation of clauses to elimination principles
- A conservative encoding of universe polymorphism
- An adequate and computational encoding of Agda[9]
- An encoding of eta-equality and irrelevance[10]

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

[10] This probably requires extending Dedukti with type-aware conversion.

References

- G. Genestier. Encoding Agda Programs Using Rewriting. FSCD 2020.[11]
- T. Felicissimo. Representing Agda and coinduction in the lambda-pi calculus modulo rewriting. Master thesis, 2021.[12]
- T. Felicissimo, F. Blanqui, A.K. Barnawal. Translating proofs from an impredicative type system to a predicative one. CSL 2023.[13]

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

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

[13] http://files.inria.fr/blanqui/pred.pdf