Abella2Coq: Translating Abella Specifications into Coq

Informatik Masterprojekt in der Theoretischen Informatik,
Friedrich-Alexander-Universität Erlangen-Nürnberg

Florian Guthmann <florian.guthmann@fau.de>
Philip Kaludercic <philip.kaludercic@fau.de>

With thanks to Johannes Lindner for his advice and input

2023-11-14

The proof assistant Abella takes a "two-level approach" to dealing with PL meta-theory and binders, distinguishing between a specification and reasoning logic.

The proof assistant Abella takes a "two-level approach" to dealing with PL meta-theory and binders, distinguishing between a specification and reasoning logic. This project starts the effort to translate these proofs into Coq.

The proof assistant Abella takes a “two-level approach” to dealing with PL meta-theory and binders, distinguishing between a specification and reasoning logic. This project starts the effort to translate these proofs into Coq.

What is to be done?

- Understand Abella and its specification logic (λProlog)

The proof assistant Abella takes a "two-level approach" to dealing with PL meta-theory and binders, distinguishing between a specification and reasoning logic. This project starts the effort to translate these proofs into Coq.

What is to be done?

- Understand Abella and its specification logic (λProlog)
- Introduce Coq-ELPI and Hybrid

The proof assistant Abella takes a "two-level approach" to dealing with PL meta-theory and binders, distinguishing between a specification and reasoning logic. This project starts the effort to translate these proofs into Coq.

What is to be done?

- Understand Abella and its specification logic (λProlog)
- Introduce Coq-ELPI and Hybrid
- Give an overview of software engineering process

Section 1

Necessary Background

What is Abella good for?

Baelde et al. 2014; Gacek, Miller, and Nadathur 2011; Tiu 2007

- A proof assistant based on a two-level approach
  1. Specification Logic Based on λProlog/hereditary Harrop formulas, with an *open world* assumption
  2. Reasoning Logic Based on "logic $\mathcal{G}$", with a *closed world* assumption, used to reason about specifications

What is Abella good for?

Baelde et al. 2014; Gacek, Miller, and Nadathur 2011; Tiu 2007

- A proof assistant based on a two-level approach
  1. Specification Logic Based on λProlog/hereditary Harrop formulas, with an *open world* assumption
  2. Reasoning Logic Based on "logic $\mathcal{G}$", with a *closed world* assumption, used to reason about specifications
- Reasoning over nominal constants using ∇ quantifiers (Baelde et al. 2014, p. 32f.).

$$\nabla x \nabla y.\, x \neq y$$

$$\nabla x.\, P = P \iff x \notin \mathrm{FV}(P)$$

$$\nabla x \nabla y.\, P = \nabla y \nabla x.\, P$$

What is Abella good for?

Baelde et al. 2014; Gacek, Miller, and Nadathur 2011; Tiu 2007

- A proof assistant based on a two-level approach
  1. Specification Logic Based on λProlog/hereditary Harrop formulas, with an *open world* assumption
  2. Reasoning Logic Based on "logic $\mathcal{G}$", with a *closed world* assumption, used to reason about specifications
- Reasoning over nominal constants using ∇ quantifiers (Baelde et al. 2014, p. 32f.).
- An interactive tactic-based system, with proof search.

```
Theorem add_step :
  forall A B C,
    add A B C ->
      add A (s B) (s C).
induction on 1.
intros. case H1.
  search.
  apply IH to H2. search.
```

What is Abella good for?

Baelde et al. 2014; Gacek, Miller, and Nadathur 2011; Tiu 2007

- A proof assistant based on a two-level approach
  1. Specification Logic Based on λProlog/hereditary Harrop formulas, with an *open world* assumption
  2. Reasoning Logic Based on "logic $\mathcal{G}$", with a *closed world* assumption, used to reason about specifications
- Reasoning over nominal constants using ∇ quantifiers (Baelde et al. 2014, p. 32f.).
- An interactive tactic-based system, with proof search.
- Brittle user experience.

```
...
Error: Sys_error("/home/philip
Error: Sys_error("/home/philip
Error: Sys_error("/home/philip
Error: Sys_error("/home/philip
Error: Sys_error("/home/philip
Error: Sys_error("/home/philip
...
```

Reasoning Logic Example Involving ∇

```
Theorem member_nominal_absurd :
  forall L T, nabla x,
    member (of x T) L -> false.
```

```
Theorem member_nominal_absurd :
  forall L T, nabla x,
    member (of x T) L -> false.
induction on 1.
```

Reasoning Logic Example Involving ∇

```
Theorem member_nominal_absurd :
  forall L T, nabla x,
    member (of x T) L -> false.
induction on 1.
intros.
```

```
Theorem member_nominal_absurd :
  forall L T, nabla x,
    member (of x T) L -> false.
induction on 1.
intros.
case H1.
```

Reasoning Logic Example Involving ∇

```
Theorem member_nominal_absurd :
  forall L T, nabla x,
    member (of x T) L -> false.
induction on 1.
intros.
case H1.
  apply IH to H2.
```

type-uniq.sig

```
sig type-uniq.

kind tm, ty type.
type of     tm -> ty -> o.
```

type-uniq.mod

```
module type-uniq.

of (abs R) (arrow T U) :-
  pi x\ of x T => of (R x) U.
of (app M N) T :- of M (arrow U T), of N U.
```

The *Specification Logic* as a *Relational Specifications*

Miller 1988; Miller and Nadathur 2012; Miller 2000; Belleannée, Brisset, and Ridoux 1999

λProlog "extends" Prolog with

- Generalising Horn-Clauses to intuitionistic Higher Order Hereditary Harrop Formulae.

$$
\begin{aligned}
H &::= x\ t_1 \ldots t_n \mid H \wedge H \mid \forall X.\, H \mid \exists x.\, H \mid G \implies H & \text{(Program)} \\
G &::= x\ t_1 \ldots t_n \mid X\ t_1 \ldots t_n \mid G \wedge G \mid G \vee G \mid & \text{(Queries)} \\
&\quad\ \exists X.\, G \mid \forall x.\, G \mid H \implies G &
\end{aligned}
$$

The *Specification Logic* as a *Relational Specifications*

Miller 1988; Miller and Nadathur 2012; Miller 2000; Belleannée, Brisset, and Ridoux 1999

λProlog "extends" Prolog with

- Generalising Horn-Clauses to intuitionistic Higher Order Hereditary Harrop Formulae.
- Static typing for predicates, functors and individuals.

```
kind list type.
type nil list.
type :: int -> list -> list.
type append list -> list -> list -> o.
append (X :: L) K (X :: M) :- append L K M.
append nil K K.
```

The *Specification Logic* as a *Relational Specifications*

Miller 1988; Miller and Nadathur 2012; Miller 2000; Belleannée, Brisset, and Ridoux 1999

λProlog "extends" Prolog with

- Generalising Horn-Clauses to intuitionistic Higher Order Hereditary Harrop Formulae.
- Static typing for predicates, functors and individuals.
- **H**igher **O**rder **A**bstract **S**yntax (HOAS) using typed λ-tree syntax

```
type app tm -> tm -> tm.
type abs (tm -> tm) -> tm.

% Y-Combinator: λf. λx. f(x x)λx. f(x x)
(abs f\(app (x\(app f (app x x))) (x\(app f (app x x)))))
```

The *Specification Logic* as a *Relational Specifications*

Miller 1988; Miller and Nadathur 2012; Miller 2000; Belleannée, Brisset, and Ridoux 1999

λProlog “extends” Prolog with

- Generalising Horn-Clauses to intuitionistic Higher Order Hereditary Harrop Formulae.
- Static typing for predicates, functors and individuals.
- **H**igher **O**rder **A**bstract **S**yntax (HOAS) using typed λ-tree syntax
- Higher-Order Unification, i.e. λ-term unification modulo $\alpha\beta\eta$-conversion

```
kind person type.
type bob sue ned person.

type age person -> int -> o.
age bob 23. age sue 24. age ned 23.
```

The *Specification Logic* as a *Relational Specifications*

Miller 1988; Miller and Nadathur 2012; Miller 2000; Belleannée, Brisset, and Ridoux 1999

λProlog "extends" Prolog with

- Generalising Horn-Clauses to intuitionistic Higher Order Hereditary Harrop Formulae.
- Static typing for predicates, functors and individuals.
- **H**igher **O**rder **A**bstract **S**yntax (HOAS) using typed λ-tree syntax
- Higher-Order Unification, i.e. λ-term unification modulo $\alpha\beta\eta$-conversion

```
kind person type.
type bob sue ned person.

type age person -> int -> o.
age bob 23. age sue 24. age ned 23.

?- forevery (x\ age x A) (ned :: bob :: nil).
A == 23.
```

The *Specification Logic* as a *Relational Specifications*

Miller 1988; Miller and Nadathur 2012; Miller 2000; Belleannée, Brisset, and Ridoux 1999

λProlog "extends" Prolog with

- Generalising Horn-Clauses to intuitionistic Higher Order Hereditary Harrop Formulae.
- Static typing for predicates, functors and individuals.
- **H**igher **O**rder **A**bstract **S**yntax (HOAS) using typed λ-tree syntax
- Higher-Order Unification, i.e. λ-term unification modulo $\alpha\beta\eta$-conversion

```
kind person type.
type bob sue ned person.

type age person -> int -> o.
age bob 23. age sue 24. age ned 23.

?- forevery (x\ age x A) (ned :: sue :: nil).
no.
```

The *Specification Logic* as a *Relational Specifications*

Miller 1988; Miller and Nadathur 2012; Miller 2000; Belleannée, Brisset, and Ridoux 1999

λProlog "extends" Prolog with

- Generalising Horn-Clauses to intuitionistic Higher Order Hereditary Harrop Formulae.
- Static typing for predicates, functors and individuals.
- **H**igher **O**rder **A**bstract **S**yntax (HOAS) using typed λ-tree syntax
- Higher-Order Unification, i.e. λ-term unification modulo $\alpha\beta\eta$-conversion
- Support for hypothetical reasoning

type-uniq.sig

```
sig type-uniq.

kind tm, ty type.
type of      tm -> ty -> o.
```

type-uniq.mod

```
module type-uniq.

of (abs R) (arrow T U) :-
  pi x\ of x T => of (R x) U.
of (app M N) T :- of M (arrow U T), of N U.
```

type-uniq.sig

```
sig type-uniq.

kind tm, ty type.
type of     tm -> ty -> o.
```

type-uniq.mod

```
module type-uniq.

of (abs R) (arrow T U) :-
  pi x\ of x T => of (R x) U.
of (app M N) T :- of M (arrow U T), of N U.
```

type-uniq.sig

```
sig type-uniq.

kind tm, ty type.
type of     tm -> ty -> o.
```

type-uniq.mod

```
module type-uniq.

of (abs R) (arrow T U) :-
  pi x\ of x T => of (R x) U.
of (app M N) T :- of M (arrow U T), of N U.
```

type-uniq.sig

```
sig type-uniq.

kind tm, ty type.
type of     tm -> ty -> o.
```

type-uniq.mod

```
module type-uniq.

of (abs R) (arrow T U) :-
  pi x\ of x T => of (R x) U.
of (app M N) T :- of M (arrow U T), of N U.
```

A Naïve Translation Attempt

Example

```
type app tm -> tm -> tm.
type abs (tm -> tm) -> tm.
```

Interpretation as an inductive datatype

A Naïve Translation Attempt

Example

```
type app tm -> tm -> tm.
type abs (tm -> tm) -> tm.
```

Interpretation as an inductive datatype

```
Inductive tm : Set :=
  app : tm -> tm -> tm
| abs : (tm -> tm) -> tm.
```

A Naïve Translation Attempt

Example

```
type app tm -> tm -> tm.
type abs (tm -> tm) -> tm.
```

Interpretation as an inductive datatype

```
Inductive tm : Set :=
  app : tm -> tm -> tm
| abs : (tm -> tm) -> tm.
(* Error: Non strictly positive occurrence of "tm" in
   "(tm -> tm) -> tm". *)
```

Example

```
type app tm -> tm -> tm.
type abs (tm -> tm) -> tm.
```

Interpretation as an inductive datatype

```
#[bypass_check(positivity)]
Inductive tm : Set :=
  app : tm -> tm -> tm
| abs : (tm -> tm) -> tm.
```

Example

```
type app tm -> tm -> tm.
type abs (tm -> tm) -> tm.
```

Interpretation as an inductive datatype (Chlipala 2022)

```
#[bypass_check(positivity)]
Inductive tm : Set :=
  app : tm -> tm -> tm
| abs : (tm -> tm) -> tm.

Definition uhoh (t : tm) : tm :=
  match t with abs f => f t | _ => t end.
```

The ELPI language and the Coq-ELPI Plugin

Tassi 2018; Dunchev, Sacerdoti Coen, and Tassi 2016

ELPI is an extended variant of λProlog with constraint handling rules. . .

The ELPI language and the Coq-ELPI Plugin

Tassi 2018; Dunchev, Sacerdoti Coen, and Tassi 2016

ELPI is an extended variant of λProlog with constraint handling rules...

- Can be embedded into applications, such as Coq-ELPI.

```
Elpi Query lp:{{
of (abs x\abs y\app y x) X.
}}.
```

The ELPI language and the Coq-ELPI Plugin

Tassi 2018; Dunchev, Sacerdoti Coen, and Tassi 2016

ELPI is an extended variant of λProlog with constraint handling rules...

- Can be embedded into applications, such as Coq-ELPI.
- Coq-ELPI can introspect the Coq environment

```
Elpi Query lp:{{
  coq.locate "nat" Nat,
  (global Nat) = {{ nat }}.
}}.
```

The ELPI language and the Coq-ELPI Plugin

Tassi 2018; Dunchev, Sacerdoti Coen, and Tassi 2016

ELPI is an extended variant of λProlog with constraint handling rules...

- Can be embedded into applications, such as Coq-ELPI.
- Coq-ELPI can introspect, and manipulate the Coq environment

```
Elpi Query lp:{{
  ...,
  coq.env.add-indt Decl _.
}}.
```

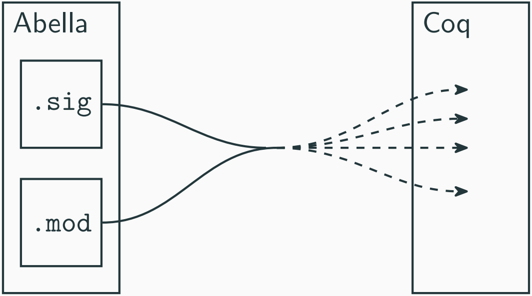
The ELPI language and the Coq-ELPI Plugin

Tassi 2018; Dunchev, Sacerdoti Coen, and Tassi 2016

ELPI is an extended variant of λProlog with constraint handling rules...

- Can be embedded into applications, such as Coq-ELPI.
- Coq-ELPI can introspect, and manipulate the Coq environment
- Tactics and commands can be implemented in Coq-ELPI.

```
Elpi Command Say.
Elpi Accumulate lp:{{
  main [str S] :- coq.say S.
}}.
Elpi Typecheck.
Elpi Export Say.
```


Abella
.sig
.mod
Coq

Abella

```
kind nat type.
type z nat.
type s nat -> nat.
```

Coq

```
Inductive nat : Set :=
| z : nat
| s : nat -> nat.
```

Abella

```
kind nat type.
type z nat.
type s nat -> nat.
```

Coq

```
Inductive nat : Set :=
| z : nat
| s : nat -> nat.
```

but that's hardly a typical Abella specification. . .

Abella is well-suited for reasoning about languages with binders by using HOAS...

What about Coq?

```
Inductive uexp : Set :=
| uapp : uexp -> uexp -> uexp
| uabs : (uexp -> uexp) -> uexp.
```

```
Inductive uexp : Set :=
| uapp : uexp -> uexp -> uexp
| uabs : (uexp -> uexp) -> uexp.

Error: Non strictly positive occurence of "uexp" in
  "(uexp -> uexp) -> uexp".
```

So what are the alternative approaches to representing binders in Coq?

```
Definition var : Set := string.

Inductive term : Set :=
| Var : var -> term
| App : term -> term -> term
| Abs : var -> term -> term.
```

de Bruijn indices

de Bruijn 1972

```
ABS (ABS (APP (BND 0) (BND 1)))
```

```
Inductive term : Set :=
| BND : nat -> term
| APP : term -> term -> term
| ABS : term -> term.
```

```
Inductive uexp : Set :=
| uapp : uexp -> uexp -> uexp
| uabs : (uexp -> uexp) -> uexp.
```

```
Parameter var : Set.
Inductive uexp : Set :=
| uapp : uexp -> uexp -> uexp
| uabs : (var -> uexp) -> uexp.
```

```
Parameter var : Set.
Inductive uexp : Set :=
| uapp : uexp -> uexp -> uexp
| uabs : (var -> uexp) -> uexp
| uvar : var -> uexp.
```

Autosubst{,2}

Schäfer, Tebbi, and Smolka 2015, Stark, Schäfer, and Kaiser 2019

Autosubst 1

- takes in a de Bruijn specification (in Coq) with Binding annotations

Autosubst 1

- takes in a de Bruijn specification (in Coq) with Binding annotations
- generates substitution algorithm and substitution lemmata

Autosubst{,2}

Schäfer, Tebbi, and Smolka 2015, Stark, Schäfer, and Kaiser 2019

Autosubst 1

- takes in a de Bruijn specification (in Coq) with Binding annotations
- generates substitution algorithm and substitution lemmata

Autosubst 2

Autosubst{,2}

Schäfer, Tebbi, and Smolka 2015, Stark, Schäfer, and Kaiser 2019

Autosubst 1

- takes in a de Bruijn specification (in Coq) with Binding annotations
- generates substitution algorithm and substitution lemmata

Autosubst 2

- takes a HOAS specification

Autosubst{,2}

Schäfer, Tebbi, and Smolka 2015, Stark, Schäfer, and Kaiser 2019

Autosubst 1

- takes in a de Bruijn specification (in Coq) with Binding annotations
- generates substitution algorithm and substitution lemmata

Autosubst 2

- takes a HOAS specification
- generates substitution algorithm and substitution lemmata

Two level approach

Object logic HOAS encoding of a formal system

Base level expands terms to de Bruijn representation

Two level approach

Object logic HOAS encoding of a formal system

Base level expands terms to de Bruijn representation

- Details of de Bruijn representation are hidden

Hybrid Framework

Momigliano, Martin, and A. P. Felty 2008; A. Felty and Momigliano 2010

Two level approach

Object logic HOAS encoding of a formal system

Base level expands terms to de Bruijn representation

- Details of de Bruijn representation are hidden
- Specification logic is flexible

Approach	HOAS-like	Reasoning Logic
Concrete Syntax		
de Bruijn		
PHOAS	✓	
Autosubst 1		
Autosubst 2	✓	
Hybrid	✓	✓

Isabelle	Coq

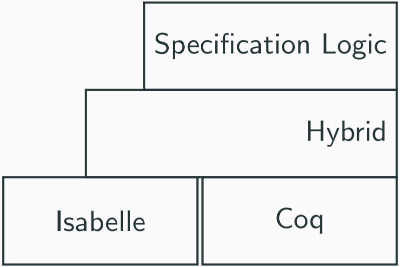
Simplification and Induction

Structure

Meta-Language for λ-Calculus

Simplification and Induction

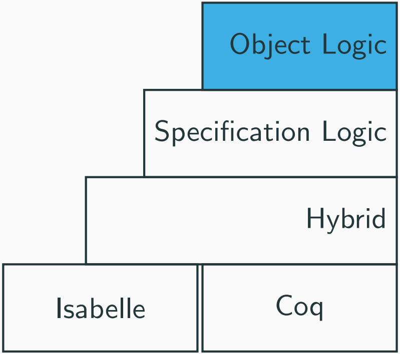
Structure



Sequent Calculus

Meta-Language for λ-Calculus

Simplification and Induction

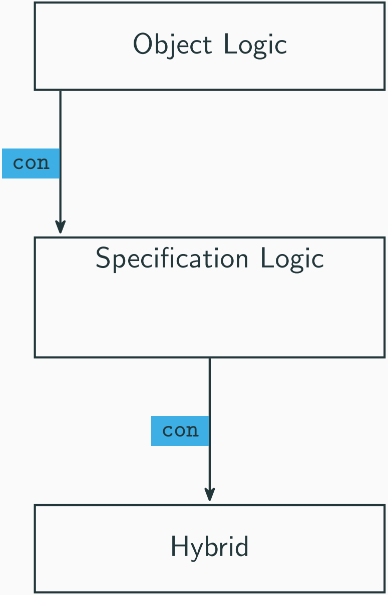


Syntax and Semantics

Sequent Calculus

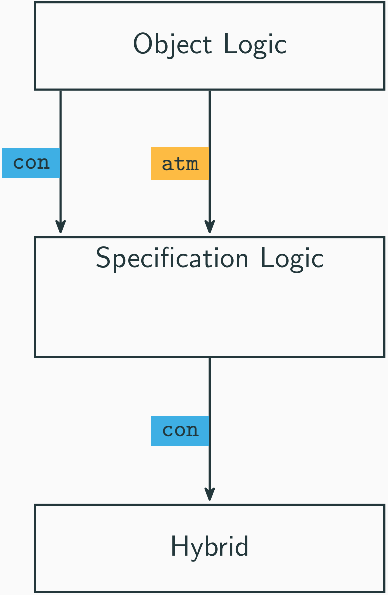
Meta-Language for λ-Calculus

Simplification and Induction



Types

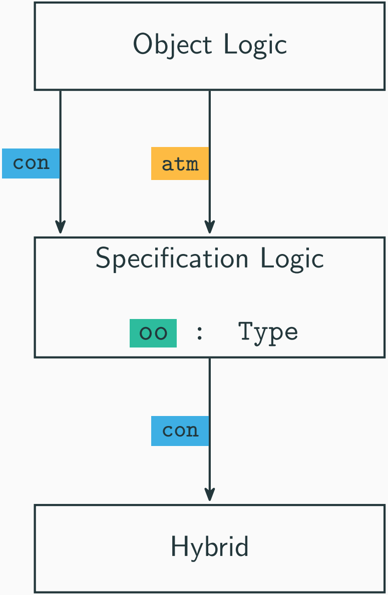
`con` : Set



Types

con : Set

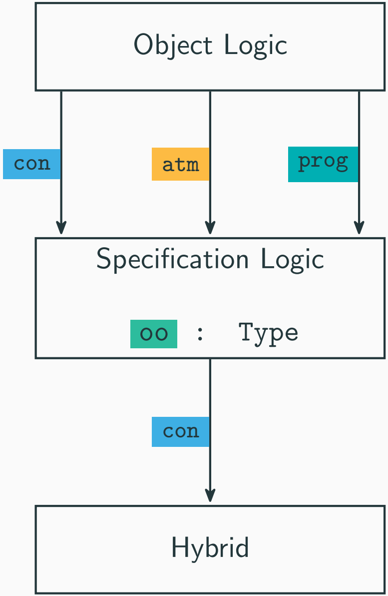
atm : Set



Types

`con` : Set

`atm` : Set



Types

```
con : Set
atm : Set
prog : atm -> oo -> Prop
```

- Second-Order Minimal logic (Momigliano, Martin, and A. P. Felty 2008)

- Second-Order Minimal logic (Momigliano, Martin, and A. P. Felty 2008)
- Hereditary Harrop Formulas (Battell and A. Felty 2016)

- Second-Order Minimal logic (Momigliano, Martin, and A. P. Felty 2008)
- Hereditary Harrop Formulas (Battell and A. Felty 2016)

In both cases, a sequent calculus is provided as an inductive type.

```
Inductive oo : Type :=
  | atom : atm -> oo
```

```
Notation "<< a >>" := (atom a).
```

```
Inductive oo : Type :=
  | atom : atm -> oo
  | T : oo
```

```
Notation "<< a >>" := (atom a).
```

```
Inductive oo : Type :=
  | atom : atm -> oo
  | T : oo
  | Conj : oo -> oo -> oo
```

```
Notation "<< a >>" := (atom a).
Notation "a & b" := (Conj a b).
```

```
Inductive oo : Type :=
  | atom : atm -> oo
  | T : oo
  | Conj : oo -> oo -> oo
  | Imp : oo -> oo -> oo
```

```
Notation "<< a >>" := (atom a).
Notation "a & b" := (Conj a b).
Notation "a ---> b" := (Imp a b).
```

HHF as a specification logic

```
Inductive oo : Type :=
  | atom : atm -> oo
  | T : oo
  | Conj : oo -> oo -> oo
  | Imp : oo -> oo -> oo
  | All : (expr con -> oo) -> oo
  | Some : (expr con -> oo) -> oo.

Notation "<< a >>" := (atom a).
Notation "a & b" := (Conj a b).
Notation "a ---> b" := (Imp a b).
```

```
Variable con : Set.
Variable atm : Set.
Variable prog : atm -> oo -> Prop.

Notation "A :- b" := (prog A b).
```

HOAS encoding of some formal system where

`con` Set of constants

HOAS encoding of some formal system where

- `con` Set of constants
- `atm` Set of relations

HOAS encoding of some formal system where

- `con` Set of constants
- `atm` Set of relations
- `prog` Inductive `Property` specifying when the relations hold

Peano arithmetic as an object logic

Abella

```
kind nat type.

type zero nat.
type succ nat -> nat.
```

Coq

```
Inductive con : Set :=
| czero : con
| csucc : con.
```

Abella

```
kind nat type.

type zero nat.
type succ nat -> nat.
```

Coq

```
Inductive con : Set :=
| czero : con
| csucc : con.


Definition zero : uexp :=
  CON czero.

Definition succ : uexp -> uexp :=
  fun e =>
    (APP (CON csucc) e).
```

Abella

```
type plus nat -> nat -> nat -> o.

type leq nat -> nat -> o.
```

Coq

```
Inductive atm : Set :=
| plus : uexp -> uexp -> uexp -> atm
| leq  : uexp -> uexp -> atm.
```

Abella

```
type plus nat -> nat -> nat -> o.
```

Coq

```
Inductive prog : atm -> oo -> Prop :=
```

Abella

```
type plus nat -> nat -> nat -> o.

plus z N N.
```

Coq

```
Inductive prog : atm -> oo -> Prop :=
| plus0 : forall N,
    plus z N N :- T
```

Abella

```
type plus nat -> nat -> nat -> o.

plus z N N.
plus (s M) N (s P) :- plus M N P.
```

Coq

```
Inductive prog : atm -> oo -> Prop :=
| plus0 : forall N,
    plus z N N :- T
| plus1 : forall M N P,
    plus (s M) N (s P) :-
      << plus M N P >>
```

STLC as an object logic

```
kind    tm, ty        type.
type    app           tm -> tm -> tm.
type    abs           (tm -> tm) -> tm.
type    arrow         ty -> ty -> ty.
type    of            tm -> ty -> o.
```

Abella

Coq

```
Inductive prog : atm -> oo -> Prop :=
...
```

Abella

```
of (app M N) T :- of M (arrow U T), of N U.
```

Coq

```
Inductive prog : atm -> oo -> Prop :=
| of0 : forall (M N T U : uexp),
      of (app M N) T :-
        << of M (arrow U T) >> &
        << of N U >>
```

Abella

```
of (app M N) T :- of M (arrow U T), of N U.
of (abs R) (arrow T U) :- pi x\ (of x T => of (R x) U).
```

Coq

```
Inductive prog : atm -> oo -> Prop :=
...
| of1 : forall (R : uexp -> uexp) (T U : uexp),
        abstr R ->
        of (abs R) (arrow T U) :-
          All (fun x =>
                  << of x T >> --->
                    << of (R x) U >>)
```

abstr

```
abstr : (uexp -> uexp) -> Prop
```

```
abstr : (uexp -> uexp) -> Prop
```

`abstr R` asserts that:

`abstr : (uexp -> uexp) -> Prop`

`abstr R` asserts that:

- `R` contains no dangling indices

```
abstr : (uexp -> uexp) -> Prop
```

`abstr R` asserts that:

- `R` contains no dangling indices
- `R` is not an *exotic* term

Dangling Indices

`ABS (ABS (APP (BND 0) (BND 7)))`

Exotic Terms

```
Parameter P : uexp -> bool.
Definition foo : uexp := CON cfoo.
Definition bar : uexp := CON cbar.

Definition exotic : uexp :=
  (lambda (fun e => if (P e) then foo else bar)).
```

Section 2

Software Engineering

How to situate this in a Coq environment?

How to situate this in a Coq environment?

1. How to parse `.sig` and `.mod` files?
   - Re-use the Abella parser
   - Implement a parser in ELPI

How to situate this in a Coq environment?

1. How to parse `.sig` and `.mod` files?
   - Re-use the Abella parser
   - Implement a parser in ELPI
2. How facilitate the translation?
   - Write a Coq Plugin that translates
   - Use Coq-ELPI invoke a translation predicate

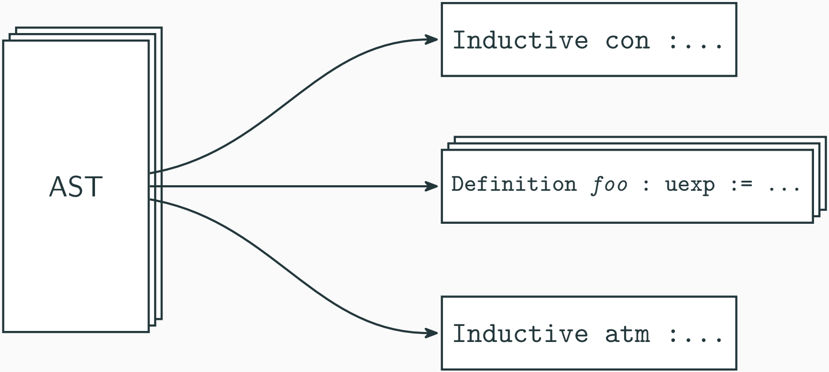
How to situate this in a Coq environment?

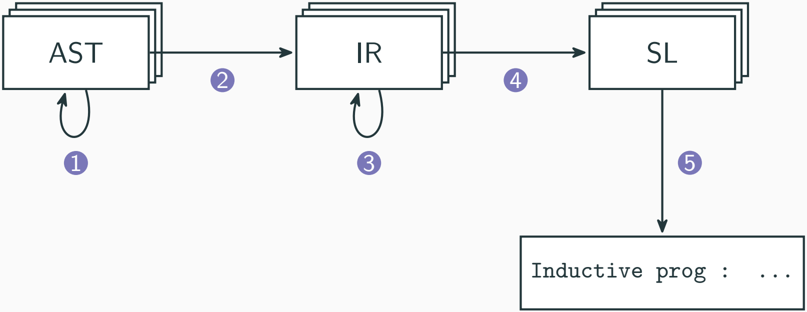
1. How to parse `.sig` and `.mod` files?
   - Re-use the Abella parser
   - Implement a parser in ELPI
2. How facilitate the translation?
   - Write a Coq Plugin that translates
   - Use Coq-ELPI invoke a translation predicate
3. How to invoke the translation from Coq?
   - Manually write a vernacular command
   - Use Coq-ELPI to define a vernacular command

How to situate this in a Coq environment?

1. How to parse `.sig` and `.mod` files?
   - Re-use the Abella parser
   - Implement a parser in ELPI
2. How facilitate the translation?
   - Write a Coq Plugin that translates
   - Use Coq-ELPI invoke a translation predicate
3. How to invoke the translation from Coq?
   - Manually write a vernacular command
   - Use Coq-ELPI to define a vernacular command

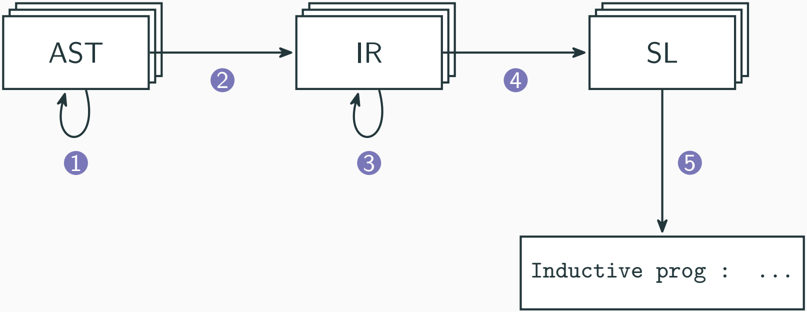
Translation Pipeline [.sig]





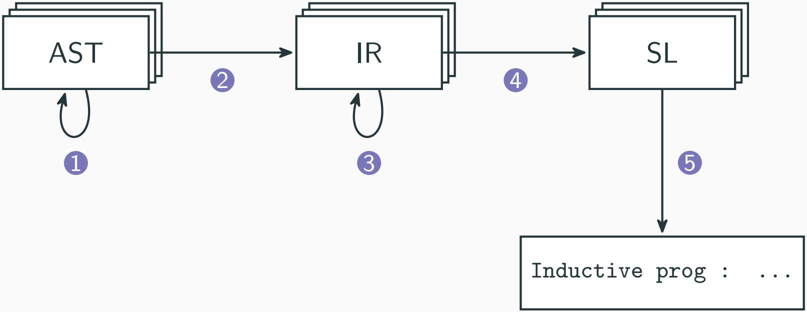
1. Preprocessing of AST

Translation Pipeline [.mod]



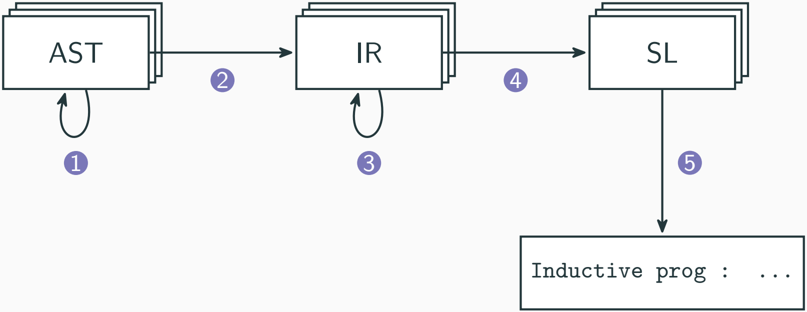
1. Preprocessing of AST
2. Translating from AST to IR

Translation Pipeline [.mod]



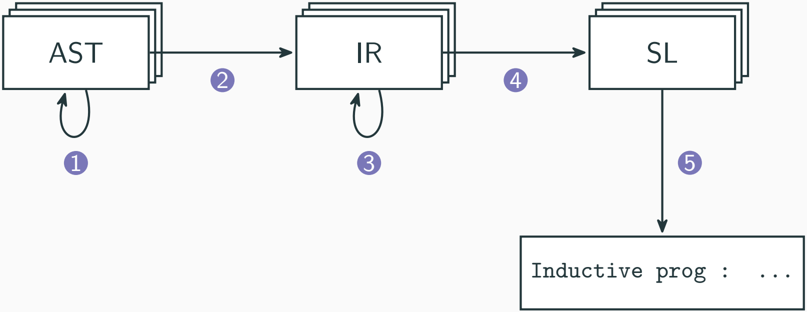
1. Preprocessing of AST
2. Translating from AST to IR
3. Identify head and body of a clause

Translation Pipeline [.mod]



1. Preprocessing of AST
2. Translating from AST to IR
3. Identify head and body of a clause
4. Translate into representations of head and body in the specification logic

Translation Pipeline [.mod]



1. Preprocessing of AST
2. Translating from AST to IR
3. Identify head and body of a clause
4. Translate into representations of head and body in the specification logic
5. Generate inductive Property

AST → IR

- the AST representation uses de Bruijn indices for variables

- the AST representation uses de Bruijn indices for variables
- these are resolved to string references for a consistent handling of variables and constants

- the AST representation uses de Bruijn indices for variables
- these are resolved to string references for a consistent handling of variables and constants
- some idiosyncracies of the AST are resolved

Assumption

Each term is a clause, i.e. it has the form

$$\forall X_0 . \forall X_1 \ldots \forall X_n . \mathsf{Head} \mathrel{:\!-} B_0, \ldots, B_m$$

where B_i are conjuncts in the clause body

```
conjuncts [] {{T}}.
conjuncts [B | Bs] (Conj B1 Bs1) :-
  translate B B1,
  conjuncts Bs Bs1
```

```
Inductive prog : atm -> oo -> Prop :=
| clause0 : ...
| clause1 : ...
⋮
```

```
Inductive prog : atm -> oo -> Prop :=
| clause0 : ...
| clause1 : ...
⋮
```

We need a representation of Coq terms in ELPI

Elpi

```
type app list term
      -> term.
```

Coq

```
Definition n := 2.
```

```
?- coq.locate "n" (const C),
   coq.env.const C (some Body) _.

Body = (app [{{S}}, app [{{S}}, {{O}}]]),
C = {{n}}.
```

HOAS of Gallina Terms in ELPI

Elpi

```
type fun name
      -> term
      -> (term -> term)
      -> term.
```

Coq

```
Definition f :=
  fun n : nat => n.
```

```
?- coq.locate "f" (const C),
   coq.env.const C (some Body) _.

Body = (fun `n` {{nat}} c0\ c0),
C = {{f}}.
```

HOAS of Gallina Terms in ELPI

ELPI

```
type prod name
      -> term
      -> (term -> term)
      -> term.
```

Coq

```
Definition P :=
  forall (n : nat), Prop.
```

```
?- coq.locate "P" (const C),
   coq.env.const C (some Body) _.

Body = (prod `n` {{nat}} c0\ sort prop),
C = {{f}}.
```

- `fix`
- `match`

also exists, but we do not need them here

Coq

```
Definition z : uexp := CON cz.
```

ELPI

```
coq.env.add-const "z" (app [CON, cz]) _ _ _.
```

Coq

```
Inductive atm : Set :=
| is_z : uexp -> atm.
```

ELPI

```
Arity = (arity Set),
Constructors =
  (t\ [
     (constructor "is_z"
       (arity (prod _ {{uexp}} _ t)))]),
Decl = (inductive "atm" tt Arity Constructors),
coq.env.add-indt Decl _.
```

Subsection 2

Common Idioms – λProlog beyond Prolog

Concrete Syntax

```
kind lam type.
type lam-abs string -> lam -> lam.
type lam-app lam -> lam -> lam.
type lam-var string -> lam.
```

de Bruijn

```
kind db type.
type db-var int -> db.
type db-abs db -> db.
type db-app db -> db -> db.
```

```
type depth int -> string -> o.

pred lam-to-db i:lam i:int o:db.

lam-to-db (lam-app L R) D (db-app L1 R1) :-
  lam-to-db L D L1,
  lam-to-db R D R1.

lam-to-db (lam-abs V B) D (db-abs B1) :-
  D1 is D + 1,
  depth D V => lam-to-db B D1 B1.

lam-to-db (lam-var V) D (db-var I) :-
  depth N V,
  I is D - N - 1.
```

```
type ctx var -> term -> o.

translate (var V) T :-
  ctx V T.

translate (abs Var Type Body) (fun VarCoq TypeCoq F) :-
  coq.id->name Var VarCoq,
  translate-type Type TypeCoq,
  pi x\ (ctx Var x) => translate Body (F x).
```

```
type ctx var -> term -> o.

translate (var V) T :-
  ctx V T.

translate (abs Var Type Body) (fun VarCoq TypeCoq F) :-
  coq.id->name Var VarCoq,
  translate-type Type TypeCoq,
  pi x\ (ctx Var x) => translate Body (F x).
```

```
kind term type.
type app term -> term -> term.
type abs (term -> term) -> term.
type foo term.
type baz term.

pred build.aux i:list term o:(term -> term).
build.aux [] e\e.
build.aux [T | Ts] (e\ F (app e T)) :-
  build.aux Ts F.

pred build i:list term o:term.
build Ts (F baz) :-
  build.aux Ts F.
```

- Simple representation of AST

- Simple representation of AST
- Powerful idioms for AST manipulation

ELPI for Transpilation

- Simple representation of AST
- Powerful idioms for AST manipulation
- Incremental development via free Variables

ELPI for Transpilation

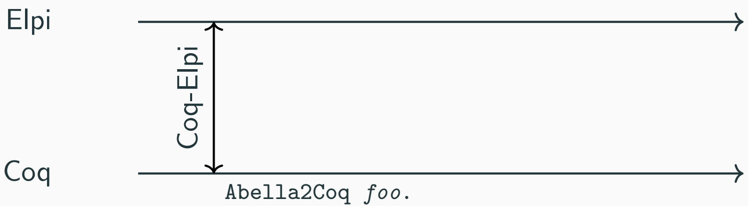
- Simple representation of AST
- Powerful idioms for AST manipulation
- Incremental development via free Variables
- Succinct ($\approx$500 LOC)

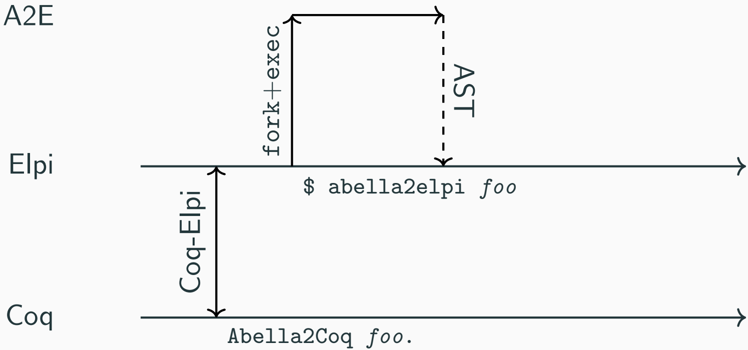
ELPI for Transpilation

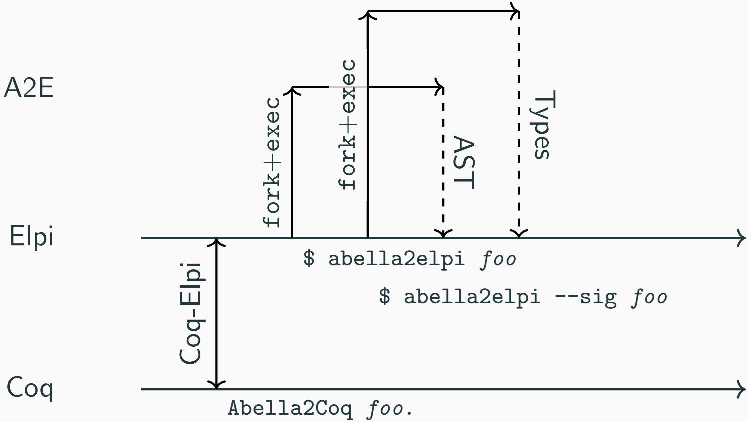
- Simple representation of AST
- Powerful idioms for AST manipulation
- Incremental development via free Variables
- Succinct (≈500 LOC)
- Poor error messages

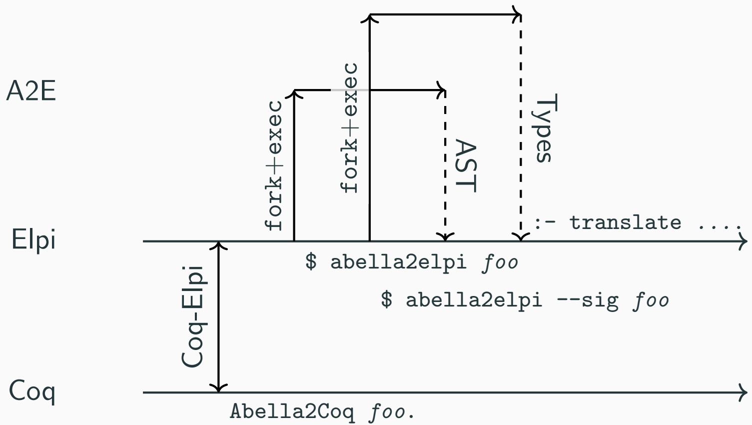
How to situate this in a Coq environment?

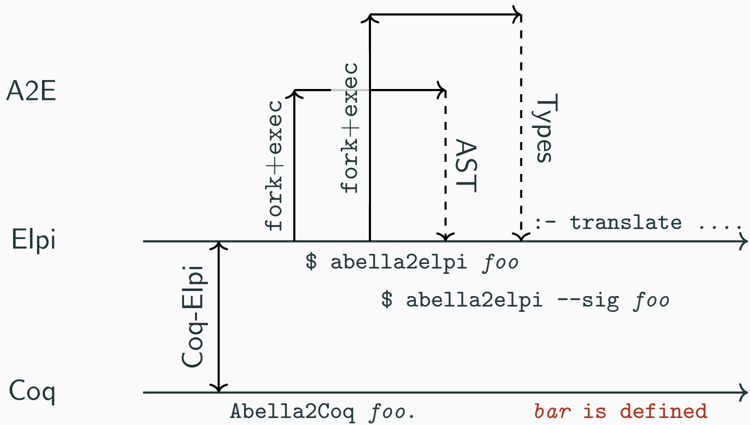
1. How to parse `.sig` and `.mod` files?
   - Re-use the Abella parser
   - Implement a parser in ELPI
2. How facilitate the translation?
   - Write a Coq Plugin that translates
   - Use Coq-ELPI invoke a translation predicate
3. How to invoke the translation from Coq?
   - Manually write a vernacular command
   - Use Coq-ELPI to define a vernacular command


Elpi
Coq-Elpi
Coq
Abella2Coq *foo*.


A2E
Elpi
Coq
fork+exec
AST
Coq-Elpi
$ abella2elpi *foo*
Abella2Coq *foo*.


A2E
Elpi
Coq
fork+exec
fork+exec
AST
Types
Coq-Elpi
$ abella2elpi foo
$ abella2elpi --sig foo
Abella2Coq foo.


A2E
Elpi
Coq
fork+exec
fork+exec
AST
Types
:- translate ....
$ abella2elpi foo
$ abella2elpi --sig foo
Coq-Elpi
Abella2Coq foo.


A2E
Elpi
Coq
fork+exec
fork+exec
AST
Types
:- translate ....
$ abella2elpi foo
$ abella2elpi --sig foo
Coq-Elpi
Abella2Coq foo.
bar is defined

ELPI

- Addition of a Process flexible API

General and Upstream Contributions

ELPI
- Addition of a Process flexible API
- A major mode for GNU Emacs

General and Upstream Contributions

ELPI
- Addition of a Process flexible API
- A major mode for GNU Emacs

Coq-ELPI
- Integration with Coq's `Extra Dependency` system (TBR)

General and Upstream Contributions

ELPI
- Addition of a Process flexible API
- A major mode for GNU Emacs

Coq-ELPI
- Integration with Coq's `Extra Dependency` system (TBR)
- Adding more Elpi predicates to Coq-ELPI

General and Upstream Contributions

ELPI
- Addition of a Process flexible API
- A major mode for GNU Emacs

Coq-ELPI
- Integration with Coq's `Extra Dependency` system (TBR)
- Adding more Elpi predicates to Coq-ELPI

Hybrid
- Fix issues related to newer versions of Coq

Section 3

Practical Examples

Finished Product: *Abella2Coq*

- *Abella2Coq* can installed via OPAM

Finished Product: *Abella2Coq*

- *Abella2Coq* can installed via OPAM
- Translation occurs automatically with a vernacular command.

Finished Product: *Abella2Coq*

- *Abella2Coq* can installed via OPAM
- Translation occurs automatically with a vernacular command.
- `.sig`/`.mod` files are loaded via `Extra Dependency`

```
From Abella2Coq Require Import Abella2Coq.
From A2CExamples Extra Dependency "nat.sig" as nat.
Abella2Coq nat as natural_numbers.
Import natural_numbers.
```

```
From Abella2Coq Require Import Abella2Coq.
From A2CExamples Extra Dependency "nat.sig" as nat.
Abella2Coq nat as natural_numbers.
Import natural_numbers.

...

(* plus (s M) N (s P) :- plus M N P. *)
Lemma add_n_m :
  ∅ ⊢ ∀ n, ∀ m, ∀ o,
    plus n m o → plus (s n) m (s o).
Proof.
  a2c_intros.
  a2c_search.
Qed.
```

```
Lemma add_n_m' :
  ∅ ⊢ ∀ n, ∀ m, ∀ o,
    plus n m o → plus (s n) m (s o).
Proof.
  eapply g_all; intros.
  eapply g_all; intros.
  eapply g_all; intros.
  eapply g_imp.
  eapply g_prog.
  - eapply plus1.
  - eapply g_dyn.
    + eapply elem_self.
    + eapply b_match.
Qed.
```

```
From Abella2Coq Require Import Abella2Coq.
From A2CExamples Extra Dependency "eval.sig" as eval.
Abella2Coq eval.
```

```
From Abella2Coq Require Import Abella2Coq.
From A2CExamples Extra Dependency "eval.sig" as eval.
Abella2Coq eval.

...

Lemma two_step :
   ∅ ⊢ ∀ S, ∀ S′, ∀ S″,
      step S S' ∧ step S' S'' → nstep S S''.
Proof.
  a2c_intros.
  a2c_search.
Qed.
```

Abella's specification logic can be translated via Hybrid into Coq. The use of Coq-ELPI makes this easier.

Abella's specification logic can be translated via Hybrid into Coq. The use of Coq-ELPI makes this easier.

Ideas for future projects

- Adding Coq tactics to simplify reasoning with Hybrid

Abella's specification logic can be translated via Hybrid into Coq. The use of Coq-ELPI makes this easier.

Ideas for future projects

- Adding Coq tactics to simplify reasoning with Hybrid
- Reimplementing the Abella `search` tactic using Elpi

Abella's specification logic can be translated via Hybrid into Coq. The use of Coq-ELPI makes this easier.

Ideas for future projects

- Adding Coq tactics to simplify reasoning with Hybrid
- Reimplementing the Abella `search` tactic using Elpi
- Translating a (fragment) of `.thm` files into Coq proofs

[1] David Baelde et al. "Abella: A system for reasoning about relational specifications". In: *Journal of Formalized Reasoning* 7.2 (2014), pp. 1–89.

[2] Chelsea Battell and Amy Felty. "The Logic of Hereditary Harrop Formulas as a Specification Logic for Hybrid". In: LFMTP '16. Porto, Portugal: Association for Computing Machinery, 2016. ISBN: 9781450347778. DOI: 10.1145/2966268.2966271. URL: https://doi.org/10.1145/2966268.2966271.

[3] Catherine Belleannée, Pascal Brisset, and Olivier Ridoux. "A pragmatic reconstruction of λProlog". In: *The Journal of Logic Programming* 41.1 (1999), pp. 67–102.

[4] Adam Chlipala. *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2022.

[5] Adam Chlipala. "Parametric higher-order abstract syntax for mechanized semantics". In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. Ed. by James Hook and Peter Thiemann. ACM, 2008, pp. 143–156. DOI: 10.1145/1411204.1411226. URL: https://doi.org/10.1145/1411204.1411226.

[6] Alonzo Church. "A formulation of the simple theory of types". In: *The Journal of Symbolic Logic* 5.2 (1940), pp. 56–68. DOI: 10.2307/2266170.

[7] N.G de Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)* 75.5 (1972), pp. 381–392. ISSN: 1385-7258. DOI: https://doi.org/10.1016/1385-7258(72)90034-0. URL: https://www.sciencedirect.com/science/article/pii/1385725872900340.

[8] Cvetan Dunchev, Claudio Sacerdoti Coen, and Enrico Tassi. "Implementing HOL in an Higher Order Logic Programming Language". In: *Logical Frameworks and Meta Languages: Theory and Practice*. LFMTP '16. Porto, Portugal: ACM, June 2016, p. 10. DOI: 10.1145/2966268.2966272. URL: https://inria.hal.science/hal-01394686.

[9] Amy Felty and Alberto Momigliano. *Hybrid: A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax*. 2010. arXiv: 0811.4367 [cs.LO].

[10] Andrew Gacek, Dale Miller, and Gopalan Nadathur. "Nominal abstraction". In: *Information and Computation* 209.1 (2011), pp. 48–73.

[11] Dale Miller. "An overview of a proof theoretical approach to reasoning about computation". In: *Computer Science* 232 (2000), pp. 91–119.

[12] Dale Miller. "Logic programming based on higher-order hereditary harrop formulas". In: *Technical Reports (CIS)* (1988), p. 756.

[13] Dale Miller and Gopalan Nadathur. *Programming with higher-order logic*. Cambridge University Press, 2012.

[14] Alberto Momigliano, Alan J. Martin, and Amy P. Felty. "Two-Level Hybrid: A System for Reasoning Using Higher-Order Abstract Syntax". In: *Electronic Notes in Theoretical Computer Science* 196 (2008), pp. 85–93. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2007.09.019. URL: https://www.sciencedirect.com/science/article/pii/S157106610800039X.

[15] Steven Schäfer, Tobias Tebbi, and Gert Smolka. "Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions". In: *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*. Ed. by Christian Urban and Xingyuan Zhang. Vol. 9236. Lecture Notes in Computer Science. Springer, 2015, pp. 359–374. DOI: 10.1007/978-3-319-22102-1_24. URL: https://doi.org/10.1007/978-3-319-22102-1%5C_24.

[16] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. "Autosubst 2: reasoning with multi-sorted de Bruijn terms and vector substitutions". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*. Ed. by Assia Mahboubi and Magnus O. Myreen. ACM, 2019, pp. 166–180. DOI: 10.1145/3293880.3294101. URL: https://doi.org/10.1145/3293880.3294101.

[17] Enrico Tassi. "Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect)". working paper or preprint. Jan. 2018. URL: https://inria.hal.science/hal-01637063.

[18] Alwen Tiu. "A Logic for Reasoning about Generic Judgments". In: *Electron. Notes Theor. Comput. Sci.* 174.5 (June 2007), pp. 3–18. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2007.01.016. URL: https://doi.org/10.1016/j.entcs.2007.01.016.