ABELLA2COQ: Translating Abella Specifications into Coq

Florian Guthmann
Philip Kaludercic
University of Erlangen-Nuremberg
Erlangen, Germany

Johannes Lindner
Technical University of Vienna
Vienna, Austria

Tadeusz Litak
University of Erlangen-Nuremberg
Erlangen, Germany

The Abella proof assistant is based on a "two-level approach" that distinguishes between a specification logic, based on λProlog, and a reasoning logic. Our contribution is ABELLA2COQ, a tool translating Abella specifications into Coq via the Hybrid framework supporting the use of Higher-Order Abstract Syntax (HOAS). Our implementation crucially involves the Coq-Elpi plugin, and in fact can be seen as an important case study of its usage. After providing the necessary background to understand our work, we give a high-level overview of the translation process and give examples of its application. We conclude with a preliminary discussion indicating how one can reason in Coq about translated specifications in the absence of automatic translation of Abella's reasoning logic.

1 Introduction

When mechanising structures with bindings in a general proof assistants like Coq [17], the administrative overhead of reasoning about said bindings is frequently burdensome. The POPLmark challenge[1] and the subsequent ORBI survey [8] give an overview of some of the approaches (HOAS/weak HOAS, de Bruijn, locally nameless...) that researchers in this area have developed. While Coq's underlying type theory does not lend itself naturally to, e.g., nominal extensions or even a straightforward form of unrestricted HOAS (see Section 3.2.1 regarding the latter point), several other POPLmark challenge approaches have been in fact pursued in Coq itself. We are particularly interested here in the Hybrid framework [2, 7, 13] supporting HOAS representation of object logics. There is also a Coq-Elpi plugin providing an implementation of λProlog, a logic-programming language natively supporting HOAS representations, and it is also going to play a major role in this paper.

Specialised proof assistants take a different route by providing specialised primitives and logic foundations suitably tailored for some of the approaches in question. The ORBI survey [8] presents the details of some of these systems (Twelf, Beluga, Abella and Hybrid). Their use streamlines proof writing, at the cost of lacking *quality of life* functionality (tooling, proof automation etc.) that a mature proof assistant like Coq provides. Such domain-specific proof assistants are also as a rule less widely known and less accessible for broader public. We are here focusing on Abella [1], an interactive theorem prover based on lambda-tree syntax. As the authors indicate

> A reader familiar with the Coq system will also find many points of similarity in the development of proofs in Abella, although we caution that there are significant differences as well [1, page 5].

There are several interesting legacy formalizations in Abella (π-calculus, process calculi, etc.), and providing an automated translation seems a good case study of Coq libraries and plugins such as those mentioned in the previous paragraph.

[1] "[...] a concrete set of benchmarks intended both for measuring progress and for stimulating discussion and collaboration in mechanizing the metatheory of programming languages.", see `https://www.seas.upenn.edu/~plclub/poplmark/`

Submitted to:
LFMTP 2024

This turns out to be a non-trivial task, since Abella's two-level logical foundations (*hereditary Harrop formulas* [10] as specification logic and *Logic $\mathscr{G}$* [9] as reasoning logic) and Coq's *Calculus of Constructions* [5]—or, more specifically, the Predicative Calculus of Cumulative Inductive Constructions (pCuIC) [18]—differ sufficiently. In this paper, we restrict our attention to translating only the *specification logic* of Abella into Coq using our ABELLA2COQ tool.

The structure of the paper is as follows. Section 2 briefly describes λProlog and its relationship with Abella. Section 3 explains the difficulties involved in translating Abella's specification to Coq and introduces the Hybrid framework with its implementation of hereditary Harrop formulas as an adequate translation target. Section 4 presents the basic usage of our tool, the translation pipeline and the role played by the Coq-Elpi plugin. Section 5 discusses our early attempts at reasoning about translated specifications in the absence of automatic translation of $\mathscr{G}$. In this context, we supplement the running example of translated simply typed lambda calculus with a translation of (finite) π-calculus. Section 6 concludes with future work.

The source code, examples and documentation are available at

`https://git8.cs.fau.de/software/abellatocoq` .

2 A Quick, Dirty but Mostly Accurate Introduction to λProlog

The specification logic of Abella is a fragment of λProlog, which in turn is a dialect of the logic programming language *Prolog*. The theoretical foundations of Prolog are those of *(first-order) Horn clauses* [11], which are FOL formulas given by the grammar

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

starting with H, where x is a function constant applied to the terms $t_1 \ldots t_n$ and X is a predicate.

For λProlog, *hereditary Harrop formulas* (HHF) serve as the logical foundation. This extends the query grammar of Horn clauses by

$$G ::= \cdots \mid \forall x.G \mid H \implies G$$

In effect, this allows λProlog to introduce hypothetical judgements while resolving a query. The duration of their effect is comparable to the visibility of dynamically scoped variables in Lisp.

```
sig type-uniq.

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

(a) Contents of the file `stlc.sig`

```
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.
```

(b) Contents of the file `stlc.mod`

Figure 1: An example of a relation `of` that relates simply-typed λ-expressions to their types.

Further differences between λProlog and Prolog include:

- All predicates are statically typed and new types and there constructors are definable.
- Lambda terms ($\lambda x.t$ is written `x\t`) handled by *higher-order unification* (i.e. unification up to $\alpha\beta\eta$-conversion).
- A module-signature requires the programmer to write the types of predicates and constructors in one file, and the relations in another.
- The syntax requires fewer parentheses, in a style comparable to ML-family languages.

Figure 1 exemplifies these points, by formalising the theory of simply typed lambda terms:

The `.sig` file declares two types `tm` (λ-terms) and `ty` (λ-types). The former has two constructors, `app` (application) that represents the application of two terms and `abs` (λ-abstraction) that takes a λProlog-function mapping terms to terms, and raises this to a term. Lastly, we declare a relation `of`, will relate a term to a type. By mapping to the propositional type `o` (read "omicron"), λProlog knows that `of` is not a type constructor, as was the case with `app`, `abs` and `arrow`.

Given this typing information, the `.mod` file will seem more familiar to a Prolog programmer. In Figure 1b we see the definition of `of`, with two cases for the two constructors of the first argument of type `tm`. Considering the second rule first, one might conventionally write as the typing relation

$$mn : t \dashv m : u \to t, n : u$$

where λProlog determines the type u. The first rule involves hypothetical reasoning: For an λ-abstraction to be of type $t \to u$, we introduce a new fresh *constant* x using `pi` (understood as a function introducing universal quantification) and postulate that this is of type t using `=>`. Recall that `R` is a function of type `tm -> tm`. Applying the fresh x to R will result in a term that λProlog will now attempt to unify with the target type u. Note that this allows us to avoid having to specify a set of base types. Instead λProlog will introduce them as necessary during the unification process.

When loaded into a λProlog-system, we could issue queries like:

```
:- of (abs x\x) (arrow X X).
X = t0.
:- of (abs x\x) T.
T = arrow t1 t1.
:- of (abs x\app x x) T.
false.
```

The first query, read as "Does $\vdash \lambda x.x : t \to t$ for some base type t hold" is satisfiable by any fresh identifier `t0`. Note that the lambda abstraction $\lambda x.x$ represented by the application of the lambda abstraction `x\x` to `abs` encodes bindings in the object theory (in this case the theory of the simply typed lambda calculus formalised inside the λProlog-system) using the existing binding system of the meta-language (in this case the λProlog-system itself). This approach goes back to Church [4], and has since been commonly referred to as *higher-order abstract syntax* (HOAS) [15].

The second and third queries demonstrate, in classical Prolog fashion how a declarative description of typing is sufficient for the system to decide is well typed by constructing a witness to the claim or by indicating that the unification failed ("`false`").

For more information and background on λProlog, consult [12, 14, 6].

```
% Self application cannot be simply-typed
Theorem of_self_app_absurd : forall T,
  {of (abs (x\ app x x)) T} -> false.
intros. case H1. case H2.
  case H3. case H6.
    case H5. case H4. case H8.
      case H7.
      case H9.
    case H7.
  case H4. case H3. case H5.
```

Figure 2: Proof of $\lambda x.xx$ not having a type, taken from `https://abella-prover.org/examples/lambda-calculus/eval.html`.

Abella, and its relation to λProlog For Abella, λProlog forms the bottom level of the "two-level approach". On top of this, a reasoning layer allows for the formalisation of theorems and their proofs. To this end, Abella provides a tactic-based, interactive proof system, which would be familiar in principle to anyone with experience using Coq. Using the example from Figure 1, such as the above claim that $\lambda x.xx$ should not have a type, see Figure 2. Note that the proof relies on `case` distinctions, where Abella discards impossible branches. This would not be automatically detected by Coq, instead requiring an "explicit" proof of a contradiction.

3 An Overview of the Translation Problem

Section 3.1 explains the impossibility of translating directly Abella's specification to Coq without breaking the crucial properties of its underlying type theory pCuIC [18]. Section 3.2 explains how the Hybrid framework [13, 7] allows us to bypass the problem. More specifically, as Hybrid distinguishes between the object logic (OL) and the specification logic (SL), section 3.2.1 illustrates the OL encoding using the running example of STLC, whereas section 3.2.2 presents the Hybrid implementation of HHF by Battell & Felty [2] as our choice for SL.

3.1 A Naive Translation, its Limits and Transcendence

Any translation of signature files such as the one given in Figure 1a must distinguish between predicates (mappings to `o`, the built-in propositional type) and the remaining types. A naive translation would interpret the latter as constructors of an inductive data type. This will quickly run into an issue, when considering `abs` as a constructor for `tm`, since the first argument is a function from `tm` to `tm`. Coq forbids "non-strictly positive" occurrences of `tm`, which prevents the generated code from being type-checked. While an attribute like `#[bypass_check(positivity)]`[2] could resolve that issue, it is undesirable, as this could allow for subversion of the strong normalisation property [3, p. 56].

This demonstrates the insufficiency of this approach. Instead of embedding the λProlog-program directly into Coq, our translation targets its underlying calculus of hereditary Harrop formulas,

[2] `https://coq.inria.fr/doc/V8.18.0/refman/proof-engine/vernacular-commands.html#coq:attr.bypass_check(positivity)`

3.2 The Hybrid Framework

To avoid the need of having to implement HHF by hand, we make use of the Hybrid framework [13, 7] supporting the representation of formal systems in higher-order abstract syntax within Coq. As such, it employs a two-level approach akin to Abella. They distinguish between the formal system we want to reason *about* ("object logic" or "OL") and the logic we reason *in* ("specification logic" or "SL"). The framework is parametric over both, allowing great flexibility in reasoning all while remaining in the setting of Coq. SLs and OLs are therefore implemented as inductive types.

Given an object logic and a specification logic (in the appropriate formats), the role of Hybrid is in relating the two. Internally, terms are represented via an inductive type `expr` (given in Figure 3) that uses de Bruijn-indices to encode λ-terms. Here, free variables are represented by the constructor `VAR`, bound variables by `BND`, where `var` and `bnd` are set to `nat`. The constructor `CON` allows encoding connectives (i.e. elements of the signature) of the object logic. Those connectives are given by the type `con`, over which `expr` is parametric. Finally application of terms and λ-abstraction are given by the constructors `APP` and `ABS`. Notice how `ABS` does not need to take the variable being bound as an argument since occurences of variables in de Bruijn-terms specify how many binders to step over to find the relevant λ-abstraction. Translating the representation of the lambda term $\lambda f.\ \lambda x.\ f\ x$ into this signature therefore results in `ABS (ABS (APP (BND 1) (BND 0)))`. This representation then begets the question of how one can fit higher-order terms of our object logic into this signature. Fortunately, Hybird provides the wrapper `lambda : (expr → expr) → expr` for just that.

```
Inductive expr (con : Set) : Set :=
  CON : con -> expr
| VAR : var -> expr
| BND : bnd -> expr
| APP : expr -> expr -> expr
| ABS : expr -> expr.
```

Figure 3: Terms in Hybrid

3.2.1 Object Logics

To illustrate the encoding of object logics we will consider the simply typed lambda calculus. As shown in Figure 4, we provide a signature `con` as an inductive type that includes constructors for both terms and types: λ-abstraction with `cABS`, application of terms with `cAPP` and function types with `cARROW`. The definitions for `App`, `Abs` and `Arrow` illustrate how we can then translate terms into the HOAS representation provided by Hybrid.

The above lambda expression $(\lambda f.\ \lambda x.\ f\ x)$ can then be represented in our encoding as follows: `Abs (fun f => Abs (fun x => App f x))`. Note that in this example "`fun`" denotes the built-in lambda abstraction of Coq.

Now that we can encode expressions of an object logic, we can consider its relations or predicates. They are given by two parameters (`atm` and `prog`), both given as inductive types. In `atm`, each constructor corresponds to a relation we want to reason about and encodes its type. In Figure 5, we give two binary relations over terms of the simply typed λ-calculus, namely typing (`of`) and evaluation (`eval`).

```
Inductive con: Set :=
| cABS   : con
| cAPP   : con
| cARROW : con.

Definition App (e1 e2 : expr con) : expr con  :=
  (APP (APP (CON cAPP) e1) e2).
Definition Abs (f : expr con -> expr con) : expr con :=
    (APP (CON cABS) (lambda f)).
Definition Arrow (A B : expr con) : expr con :=
     (APP (APP (CON cARROW) A) B).
```

Figure 4: Simply Typed λ-Calculus as an object logic

```
type of   tm -> ty -> o.
type eval tm -> tm -> o.
```

(a) Abella

```
Inductive atm : Set :=
  | of : expr con -> expr con -> atm.
  | eval : expr con -> expr con -> atm
```

(b) Hybrid

Figure 5: Typing and evaluation relations of simply typed λ-calculus

With this, we only need to state when these relations hold, i.e. we need to translate the predicate clauses. Since we need a specification logic to define this, we will work in a SL that provides lifting of object terms ($\langle$`A`$\rangle$), conjunction (`A , B `), implication (`A --> B`), universal quantification ($\forall$ `fun x => P`) and truth ($\top$). The concrete structure of this specification logic will be given in Section 3.2.2. In Figure 6 we showcase how this allows for a relatively immediate translation of clauses into the inductive `Property` `prog`. One thing we need to be careful about however are terms of a function type. Notice how whenever we encounter a term of type `expr con` $\rightarrow$ `expr con`, e.g., `R` in the constructor `of0`, we add a precondition `abstr R` to the constructors type. This stems from the HOAS encoding that Hybrid provides. When using such an encoding, we delegate the tedious handling of binders to the function type of the host language. However, this introduces another problem, namely that of *exotic terms*. In a expression like `Abs F` we can instantiate `F` to any term in the function space `expr con` $\rightarrow$ `expr con`. This unfortunately allows for terms that do not reduce to an expression. These exotic terms must be ruled out to allow for reasoning over expressions. In Hybrid this is done via the predicate `abstr : (expr con` $\rightarrow$ `expr con)` $\rightarrow$ `Prop`.

3.2.2 Specification Logics

As stated before, Hybrid is parametric over the specification logic. For our purposes, we focus on the Hybrid implementation of HHF by Battell & Felty [2]. That gives us a type of propositions, denoted as `oo` and given in Figure 7. We use the notations "$\langle$`A`$\rangle$", "`A, B`", "`A --> B`" and "$\forall$ `F`" for `(atom A)`, `(Conj A B)`, `(Imp A B)` and `(All F)`, respectively. In particular, this definition allows lifting of object level terms (`atm`) into propositions and quantification over Hybrid expressions. This corresponds to

```
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.

eval (abs R) (abs R).
eval (app M N) V :-
    eval M (abs R),
    eval (R N) V.
```

(a) Abella

```
Reserved Notation "H :- B" (at level 65, no associativity).
Inductive prog : atm -> oo -> Prop :=
| of0 : forall (R : expr con -> expr con) (T U : expr con),
    abstr R ->
    of (Abs R) (Arrow T U) :-
        ∀ (fun x => ⟨of x T⟩ --> ⟨of (R x) U⟩)
| of1 : forall (M N T U : expr con),
    of (App M N) T :-
        ⟨of M (Arrow U T)⟩ ,
        ⟨of N U⟩
| eval0 : forall (R : expr con -> expr con),
    abstr R ->
    eval (Abs R) (Abs R) :- ⊤
| eval1 : forall (M N V : expr con) (R : expr con -> expr con),
    abstr R ->
    eval (App M N) V :-
        ⟨eval M (Abs R)⟩,
        ⟨eval (R N) V⟩
  where "H :- B" := (prog H B).
```

Figure 6: Predicate clauses

formulas of HHF given in Section 2. Note that [2] do not provide a constructor for disjunction, though we have not needed this for encoding object logics so far.

In addition to a type of propositions, a specification logic must provide a calculus to reason in. In our case, we will use the provided sequent calculus for HHF, which fits our purposes. Figure 8 gives a partial definition of this sequent calculus. We will spare the reader the details of this definition and just focus on its relation to the structure of the object logic.

4 The ABELLA2COQ Tool and its Implementation

This section introduces our ABELLA2COQ tool. The following snippet demonstrates the basic usage:

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

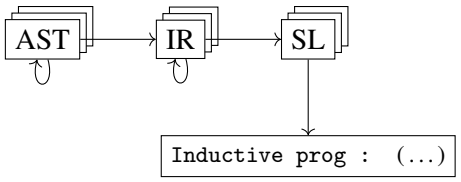
```
Inductive oo : Type :=
  | atom : atm -> oo
  | T : oo
  | Conj : oo -> oo -> oo
  | Imp : oo -> oo -> oo
  | All : (expr con -> oo) -> oo
  | Allx : (X -> oo) -> oo
  | Some : (expr con -> oo) -> oo.
```

Figure 7: Propositions of hereditary Harrop formulas

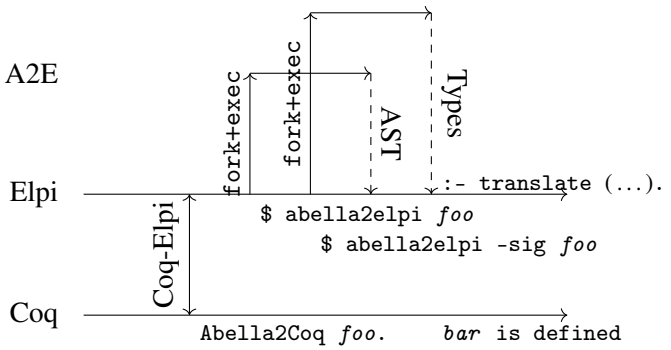
```
Inductive grseq : context -> oo -> Prop :=
  | g_prog :
      forall (L : context) (G : oo) (A : atm),
      (prog A G) -> grseq L G
      -> grseq L ⟨ A ⟩
(...)
```

Figure 8: Sequent Calculus rule from `Hsl.v` [2]

After importing the `Abella2Coq` library, we use Coq's dependency system to specify the external file, by pointing to the signature file of the `.sig`/`.mod` file pair. This reference is then passed as an argument to the vernacular `Abella2Coq` command, implemented as part of the library of the same name. This triggers the automatic translation process, instantiating the object logic for Hybrid. `Abella2Coq` encapsulates the resulting definitions within a Coq module.



(a) Processing of the intermediate representations



(b) The global process

Figure 9: An overview of the translation pipeline

The translation uses Enrico Tassi's Coq-Elpi [16], a Coq plugin based on the λProlog dialect Elpi. This allowed us to

- streamline the development of the plugin,

- use λProlog's reasoning to facilitate the translation, and
- use Coq-Elpi's integration into Coq to inspect and modify the environment.

The reader may wonder whether our translation could target (Coq-)Elpi directly. One should, however keep in mind that Coq-Elpi provides only a metaprogramming interface, akin to Ltac or other tactic languages. As such, we cannot state theorems in Coq-Elpi or Coq theorems about Coq-Elpi programs.

As sketched in Figure 9b, the process goes through a number of steps:

1. The `Abella2Coq` library instantiates Elpi using Coq-Elpi, if necessary.
2. The `Abella2Coq` command invokes a `main` predicate written in Elpi, which dispatches the next steps.
3. After resolving the external dependency passed to `Abella2Coq`, `main` invokes two system processes to parse the necessary files. These processes run a program called `abella2elpi`, which re-uses the Abella parser to generate terms that Elpi can easily parse. The first process returns an abstract syntax tree of the `.mod` files, roughly but insufficiently corresponding to the intended HHF target. The second process just returns the signatures of all types and predicates defined in the `.sig` file.
4. With the data avaliable, `main` delegates the further work to a predicate `translate`. It translates the AST into a bespoke Coq-Elpi HOAS representation of the Hybrid definitions, by iterating over a series of intermediate representations.
5. Finally, `translate` uses Coq-Elpi to inject the generated inductive definitions into Coq.

Step 3 required upstream work on Coq-Elpi. Specifically we required a robust mechanism to invoke an external process and distinguish between the standard input, output and error streams. Furthermore, Coq-Elpi lacked integration into Coq's `External Dependency` system. We would like to express our gratitude to Enrico Tassi for his collaboration in implementing the necessary functionality.

5 Reasoning About the Translated Specifications

This section provides a preliminary discussion of how one can reason in Coq about translated specifications in the absence of automatic translation of Abella's reasoning logic $\mathscr{G}$. Section 5.1 demonstrates an approach based on "vanilla" Ltac relying on `eauto`. Section 5.2 illustrates this further on another example (finite π-calculus). Section 5.3 discusses potential improvements involving Coq-Elpi.

5.1 A Sketch of an Automated *Proof Search* Command

When reasoning in Abella, a proof branch is often closed using the `search` tactic, as one would use `reflexivity` or `contradiction` in Coq. Roughly speaking, this corresponds to a λProlog query to try and find a unification.

Figure 10 demonstrates a simplified version of this command[3]. Figure 11 gives the two tactics, which attempt to automate common patterns when reasoning via Hybrid:

`a2c_intros` This tactic intends to be used like Coq's usual intros tactic without any arguments, trying not only to pull as much as possible into the context, including implications and universal quantification inside the specification logic.

[3]The type of `of_app` requires a near lethal amount of notation and type coercion that is not further interesting for this discussion

```
Lemma of_app :
  ∅ ⊢ ∀ S, ∀ S', ∀ T, ∀ T',
      of S T ∧ of S' (arrow T T') → 
      of (app S' S) T'.
Proof.
  a2c_intros.
  a2c_search.
Qed.
```

Figure 10: An example of a subject reduction proof in Coq, using definitions imported from an Abella specification

```
Tactic Notation "a2c_intros" :=
  repeat (intros; try eapply g_all; try eapply g_imp);
    (* the tactic finishes by invoking Hybrid's hint database. *)
    eauto with hybrid.

Tactic Notation "a2c_search" :=
  progress eauto using
      grseq, bcseq,
      prog,
      context_swap,
      elem_sub,
      elem_inv,
      elem_self,
      elem_rep;
    eauto with hybrid.
```

Figure 11: Definitions of some auxilary tactics to reason on the definitions generated for Hybrid

`a2c_search` just consists of a combination of attempting to apply constructors from Hybrid that operate on the sequent and the context along with the inductive type generated by the translation predicate (prog). This makes it clear, why this approach only manages to emulate Prolog-level reasoning, as this just reduces to an elaborate (e)auto invocation

As the Coq documentation explains, `eauto`/`auto` that use a *Prolog-like resolution procedure* to find a solution. Hence it fails when the resolution would require λProlog search techniques, which are usually involved in any interesting proofs.

5.2 Another Example: Finite π-calculus

For a more involved example we consider Alwen Tiu's formalisation of the finite π-calculus [19] in Abella, which gives a specification of terms and a one-step relation. In Figure 12 we give an abridged version of the inductive `Property` `prog` that results from our translation.

```
Reserved Notation "H :- B" (at level 65, no associativity).
Inductive prog : atm -> oo_ -> Prop :=
    oneb0 : forall (X : expr con) (M : expr con -> expr con),
        abstr M ->
        (oneb (inp X M) (dn X) M) :- ⊤
  | one1 : forall X Y P : expr con,
        (one (out X Y P) (up X Y) P) :- ⊤
  | one2 : forall P : expr con,
        (one (taup P) tau P) :- ⊤
  | one3 : forall X P A Q : expr con,
          (one (matchp X X P) A Q) :- ⟨one P A Q⟩
  | oneb4 : forall (X P : expr con) (A M : expr con -> expr con),
        abstr A ->
        abstr M ->
        (oneb (matchp X X P) A M) :- ⟨oneb P A M ⟩
  | one5 : forall P Q A R : expr con,
        (one (plus P Q) A R) :- ⟨one P A R⟩
  | one6 : forall P Q A R : expr con,
        (one (plus P Q) A R) :- ⟨one Q A R⟩
  (...)
  where "H :- B" := (prog H B).
```

Figure 12: One-step relation of the finite π-calculus

With this we can already prove some simple theorems. For instance, consider the idempotence of the non-deterministic sum operator `plus`:

```
Theorem plus_idem : ∅ ⊢ ∀ P, ∀ a, ∀ R, one P a R → one (plus P P) a R.
  a2c_intros.
  a2c_search.
Qed.
```

Note the remarkable similarity to the same proof in Abella:

```
Theorem plus_idem : forall P A R, {one P A R} -> {one (plus P P) A R}.
  intros.
  search.
```

Incidentally, Tiu's development provides a good example of limitations of our strategy. Many important definitions, e.g., that of the satisfaction relation of the modal logic associated with π-calculus or that of a bisimulation, are not only contained in `.thm` files, but include features not covered by our approach, such as the `nabla` quantifier or Abella's `CoDefine` vernacular.

The reader can find further examples under the `examples/` directory in the project repository.

5.3 Towards *proof search* with Coq-Elpi

We argue that using Elpi could improve `a2c_search` and related tactics. As Elpi is a λProlog-dialect, it provides the necessary search facilities that could be adequately re-used. As in the case with Figure 10, this should smooth over the overhead introduced by Hybrid, simplifying common operations. The exception is that the procedure would take on the form of a λProlog meta-interpreter: While Hybrid embeds HHF *inside* Coq, Elpi would help automate part of the reasoning *inside* of Hybrid.

Coq-Elpi provides an API for defining tactics in Elpi [4], which we are going to use in the following. To implement a tactic, we start by issuing the `Elpi Tactic` vernacular. Hereafter, we can simply give a block of Elpi code, introduced by `Elpi Accumulate`, implementing a clause for the `solve` predicate. As an example, we can reimplement[5] the basic tactic `constructor`, which given a goal of an inductive type determines a suitable constructor and applies it:

```
Elpi Tactic constructor.
Elpi Accumulate lp:{{
  solve (goal Ctx Tr Ty Pr Args as G) GL :-
    coq.safe-dest-app Ty (global (indt GR)) _,
    coq.env.indt GR _ _ _ _ Ks Kt,
    std.exists2 Ks Kt
      (k\ t\ coq.saturate t (global (indc k)) P,
            refine P G GS).
}}.
```

In this definition, we make use of Coq-Elpi's metaprogramming capabilities to query the Coq environment and act accordingly. Given a goal of type `Ty`, we first check that `Ty` is in fact inductive. If it is, we proceed by extracting its constructors `Ks` along with their respective types `Kt`. We then ask Elpi to select a constructor for which the application to the current goal typechecks. If it does, we refine and terminate.

A simple non-backtracking search tactic could then be obtained by

```
Ltac nb_search := repeat (elpi constructor).
```

To emulate Abella's `search` tactic one would need to extend the above to allow for backtracking. With sufficient effort this appears possible to implement, though we consider it to be outside the scope of this work.

6 Future Work

The resulting tool represents a first step towards the goal of automatically translating entire Abella developments into Coq. Ideas on how to extend ABELLA2COQ include:

1. Providing a full alternative to Abella's `search` tactic, by re-using the proof search that Elpi provides, as explained in Section 5.1.
2. Implementing tactics in Coq to simplify using Hybrid, as sketched in Figure 10 by `a2c_intros` and `a2c_search`.

[4] `https://lpcic.github.io/coq-elpi/tutorial_coq_elpi_tactic.html`

[5] Adapted from `https://github.com/LPCIC/coq-elpi/blob/master/apps/eltac/theories/constructor.v`

3. Support handling specifications declared in `.thm` files, as opposed to `.mod` and `.sig`, as is traditionally the case with λProlog. Currently this would be easy to implement, up until instances where definitions would involve unsupported constructs like `nabla`.
4. *Most importantly*: Translating an Abella proof like Figure 2 into Coq. As with HHF, we do not expect a direct translation of Logic $\mathscr{G}$ into pCuIC to be possible. Again, the ∇ quantifier appears as a major hurdle.

References

[1] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu & Yuting Wang (2014): *Abella: A system for reasoning about relational specifications*. Journal of Formalized Reasoning 7(2), pp. 1–89.

[2] Chelsea Battell & Amy Felty (2016): *The Logic of Hereditary Harrop Formulas as a Specification Logic for Hybrid*. LFMTP '16, Association for Computing Machinery, New York, NY, USA. Available at `https://doi.org/10.1145/2966268.2966271`.

[3] Adam Chlipala (2013): *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press.

[4] Alonzo Church (1940): *A formulation of the simple theory of types*. The Journal of Symbolic Logic 5(2), p. 56–68, doi:10.2307/2266170.

[5] Thierry Coquand & Gérard Huet (1988): Information and Computation 76(2), pp. 95–120. Available at `https://doi.org/10.1016/0890-5401(88)90005-3`.

[6] Amy Felty: *A Tutorial on Lambda Prolog and its Applications to Theorem Proving*. Available at `https://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/felty-tutorial-lprolog97.pdf`.

[7] Amy P. Felty & Alberto Momigliano (2012): *Hybrid - A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax*. J. Autom. Reason. 48(1), pp. 43–105. Available at `https://doi.org/10.1007/s10817-010-9194-x`.

[8] Amy P. Felty, Alberto Momigliano & Brigitte Pientka (2015): *The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations - Part 2 - A Survey*. J. Autom. Reason. 55(4), pp. 307–372. Available at `https://doi.org/10.1007/s10817-015-9327-3`.

[9] Andrew Gacek, Dale Miller & Gopalan Nadathur (2011): *Nominal abstraction*. Inf. Comput. 209(1), pp. 48–73. Available at `https://doi.org/10.1016/j.ic.2010.09.004`.

[10] Ronald Harrop (1956): *On disjunctions and existential statements in intuitionistic systems of logic*. Mathematische Annalen 132(4), pp. 347–361.

[11] Alfred Horn (1951): *On Sentences Which are True of Direct Unions of Algebras*. J. Symb. Log. 16(1), pp. 14–21. Available at `https://doi.org/10.2307/2268661`.

[12] Dale Miller & Gopalan Nadathur (2012): *Programming with Higher-Order Logic*. Cambridge University Press. Available at `http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/programming-higher-order-logic?format=HB`.

[13] Alberto Momigliano, Alan J. Martin & Amy P. Felty (2008): *Two-Level Hybrid: A System for Reasoning Using Higher-Order Abstract Syntax*. Electronic Notes in Theoretical Computer Science 196, pp. 85–93. Available at `https://doi.org/10.1016/j.entcs.2007.09.019`. Proceedings of LFMTP 2007.

[14] Gopalan Nadathur & Dale Miller (1988): *An Overview of Lambda-PROLOG*, pp. 810–827.

[15] Frank Pfenning & Conal Elliott (1988): *Higher-Order Abstract Syntax*. In Richard L. Wexelblat, editor: Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988, ACM, pp. 199–208. Available at `https://doi.org/10.1145/53990.54010`.

[16] Enrico Tassi (2018): *Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect)*. In: *The Fourth International Workshop on Coq for Programming Languages*, Los Angeles (CA), United States. Available at `https://inria.hal.science/hal-01637063`.

[17] The Coq Development Team (2023): *The Coq Reference Manual – Release 8.18.0*. `https://coq.inria.fr/doc/V8.18.0/refman`.

[18] Amin Timany & Matthieu Sozeau (2018): *Cumulative Inductive Types In Coq*. In Hélène Kirchner, editor: *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*, *LIPIcs* 108, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 29:1–29:16. Available at `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2018.29`.

[19] Alwen Tiu: *Formalisation of modal logics for the pi-calculus*. `https://github.com/alwentiu/abella/tree/master/pic`.