Translating Higher-Order Specifications to Coq Libraries Supporting Hybrid Proofs

Nada Habli[1] and Amy P. Felty[1,2]

[1] Department of Mathematics and Statistics, University of Ottawa, Canada
[2] School of Electrical Engineering and Computer Science, University of Ottawa, Canada
nhabl094@uottawa.ca, afelty@eecs.uottawa.ca

Abstract

We describe ongoing work on building an environment to support reasoning in proof assistants that represent formal systems using higher-order abstract syntax (HOAS). We use a simple and general specification language whose syntax supports HOAS. Using this language, we can encode the syntax and inference rules of a variety of formal systems, such as programming languages and logics. We describe our tool, implemented in OCaml, which parses this syntax, and translates it to a Coq library that includes definitions and hints for aiding automated proof in the Hybrid system. Hybrid itself is implemented in Coq, and designed specifically to reason about such formal systems. Given an input specification, the library that is automatically generated by our tool imports the general Hybrid library and adds definitions and hints for aiding automated proof in Hybrid about the specific programming language or logic defined in the specification. This work is part of a larger project to compare reasoning in systems supporting HOAS. Our current work focuses on Hybrid, Abella, Twelf, and Beluga, and the specification language is designed to be general enough to allow the automatic generation of libraries for all of these systems from a single specification.

1 Introduction

The Hybrid system [4] provides support for reasoning about *object languages* (OLs) such as programming languages and other formal systems using *higher-order abstract syntax* (HOAS). In [4], two versions of Hybrid are described, one implemented in Isabelle [8], and one implemented later in the Coq Proof Assistant [1] by fairly directly porting the Isabelle version to Coq. We focus on the Coq version here. Hybrid provides support for encoding syntax, for representing the semantics via inference rules and axioms, and for reasoning about the properties of the OL. For example, reasoning about the metatheory of a programming language allows important properties, such as soundness, to be established formally. Such properties are important for providing assurance that a language can be used to build reliable and secure software systems.

The general Hybrid infrastructure is implemented as two Coq libraries. The first provides an underlying de Bruijn representation of λ-terms parameterized by a set of constants for a particular OL. This layer is hidden from the user. This library includes a set of definitions and lemmas that builds an HOAS layer from this lower level, which is used to encode the syntax of OLs. The only axiom used in the implementation is the law of excluded middle, included by importing Coq's library for classical logic.[1] The reasoning infrastructure has multiple levels also. The inference rules of an OL are defined at the lowest level as logic programming-like clauses (called *prog* clauses here) that are provided as a parameter to an intermediate logic,

[1] This library was originally imported in order to keep the Coq implementation close to the Isabelle one. It is in fact not necessary. See [2] for a constructive version of Hybrid in Coq.

called a *specification logic* (SL). The second Coq library implements the SL. At the highest level is the *reasoning logic*, which is Coq.

This paper presents our tool for supporting reasoning in Hybrid by translating high-level specifications of OLs to Coq libraries. Our specification language has three sections, `Syntax`, `Judgments`, and `Rules`. The first section contains the specification of new constants and their types, representing the basic syntax constructors of the OL. The allowed types are a subset of the types of the simply-typed λ-calculus. In HOAS, object-level binding is encoded directly using meta-level binding, and thus arguments to constructors are allowed to have function types. We restrict to second-order types, which means that the functions appearing as arguments must themselves take arguments of atomic types. Hybrid itself is currently restricted to second-order since the representation of many formal systems does not require more. The declarations in this section are used to generate the set of constants needed for the de Bruijn level, as well as a set of definitions for encoding syntax at the HOAS level.

The declarations in the second section introduce SL-level predicates. These are the predicates used to encode the judgments in the inference rules defining the semantics of the OL. The third section defines the OL inference rules, which are translated to prog clauses. Together, the predicates and clauses instantiate the required parameters of the SL.

We present the specification language and our translation tool informally via an example, which is described in Section 2. In Section 3, we describe the technical details of some of our algorithms and their implementation. This work is part of an ongoing larger project to compare reasoning in a variety of systems that reason using HOAS (see [3], for example). In Section 4, we discuss the current focus of our work on the translation tool in the context of this larger project. In Section 5, we conclude and discuss our longer term goals.

2 An Example: The Polymorphic λ-Calculus

As an example, we consider typing for the polymorphic λ-calculus as defined in [10]. The syntax is defined by the following grammars, and typing is defined by the rules below.

$$\begin{array}{llll} \text{Terms} & M, N & ::= & x \mid \lambda x : T.M \mid M\ M \mid \lambda\alpha.M \mid M[T] \\ \text{Types} & S, T & ::= & \alpha \mid T \to T \mid \forall\alpha.T \end{array}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\, ty_v \qquad \frac{\Gamma \vdash M : S \to T \qquad \Gamma \vdash N : S}{\Gamma \vdash MN : T}\, ty_a \qquad \frac{\Gamma, x : S \vdash M : T}{\Gamma \vdash \lambda x : S.M : S \to T}\, ty_l$$

$$\frac{\Gamma \vdash M : \forall\alpha.T}{\Gamma \vdash M[S] : [S/\alpha]T}\, ty_{ta} \qquad \frac{\Gamma, \alpha \vdash M : T}{\Gamma \vdash \lambda\alpha.M : \forall\alpha.T}\, ty_{tl}$$

Figure 1 encodes the syntax and typing rules, and illustrates the use of our specification language. In the `Syntax` section, the keyword `type` introduces new atomic types for the different syntax classes, which are the polymorphic types (`tp`) and terms (`tm`) in this example. Abstraction in types and terms (defined by constants `all`, `lam`, and `tlam`) is defined using function types. Thus in the HOAS representation, abstraction in the OL will be represented using abstraction in the meta-language, which here is Coq's λ-abstraction.

The typing judgment for the polymorphic λ-calculus is expressed using the `typeof` predicate declared in the `Judgments` section. Here, the keyword `type` will map to the type of propositions of the target system, which for Hybrid is the type of formulas of the SL. Note that we use the

```
Syntax
tp: type.
arr: tp -> tp -> tp.              all: (tp -> tp) -> tp.

tm: type.
app: tm -> tm -> tm.              lam: (tm -> tm) -> tp -> tm.
tapp: tm -> tp -> tm.             tlam: (tp -> tm) -> tm.

Judgments
typeof: tm -> tp -> type.

Rules
ty_a: typeof M (arr S T) -> typeof N S -> typeof (app M N) T.
ty_l: (Pi x. typeof x S -> typeof (M x) T) -> typeof (lam (\x. M x) S) (arr S T).
ty_ta: typeof M (all (\a. T a)) -> typeof (tapp M S) (T S).
ty_tl: (Pi a. typeof (M a) (T a)) -> typeof (tlam (\a. M a)) (all (\a. T a)).
End
```

Figure 1: A Specification for the Polymorphic λ-Calculus

`type` keyword in both sections, borrowing from Twelf [11], where predicates and types are at the same level.

The inference rules appear in the last section, and here again the syntax resembles the syntax of Twelf to some extent. As in Twelf, the contexts that are explicit in the informal presentation of the rules are implicit in the judgment section of the specification. The rules are named, and the arrow is used to separate hypotheses from one another and from the conclusion, which is the last formula before the terminating dot. Binders in the polymorphic λ-calculus are represented using the binding operator of our specification language (backslash). We use tokens starting with uppercase letters for "schematic" variables (used to represent terms and types of the λ-calculus in this example) and tokens starting with lowercase letters for constructors, predicates, rule names, and bound variables.

From this fairly small specification, we generate a library that can be directly loaded into Coq, part of which is shown in Figures 2 and 3. As mentioned earlier, Hybrid is implemented in both Isabelle and Coq, and both implementations are described in [4]. As we present the Coq code in this section, we will often refer to results from [4] that are relevant. For the reader interested in looking up these results, we note that most of the formal definitions and statements in that paper use a pretty-printed version of code that can be viewed as either Isabelle or Coq syntax. (See pages 48–49 for a description of this notation.) In the text of [4], when the implementations diverge, it is explicitly stated. (For example, see Section 2.2.)

The set `Econ` in Figure 2 is the set of constants that serve as a parameter to the de Bruijn representation of terms. Note that there is one for each constructor in the `Syntax` section. The next 3 lines perform this parameter instantiation and are the same for any Hybrid OL library. The last 6 lines of the `Constants` section fill in the Hybrid definitions for the HOAS representation of the 6 constructors. They are defined in terms of their underlying de Bruijn representation. The constant `lambda` is a binding operator defined on top of the de Bruijn representation, and its definition is part of the infrastructure hidden from the user. Types of bound variables in Coq are not explicitly added since they can be inferred. In these Coq definitions, there is no distinction between the types `tp` and `tm` found in the specification. All terms have Coq type `uexp` (the type of de Bruijn terms parameterized by the set `ECon`) and all

```
Require Import sl.

Section encoding.
(*************************************************************************
        Constants
*************************************************************************)
Inductive ECon: Set :=  Carr: ECon | Call: ECon | Capp: ECon |
                        Clam: ECon | Ctapp: ECon | Ctlam: ECon.

Definition uexp: Set := expr ECon.
Definition Var: var -> uexp := (VAR ECon).
Definition Bnd: bnd -> uexp := (BND ECon).

Definition arr:= fun T1 => fun T2 => (APP (APP (CON Carr) T1) T2).
Definition all:= fun T1 => (APP (CON Call) (lambda T1)).
Definition app:= fun T1 => fun T2 => (APP (APP (CON Capp) T1) T2).
Definition lam:= fun T1 => fun T2 => (APP (APP (CON Clam) (lambda T1)) T2).
Definition tapp:= fun T1 => fun T2 => (APP (APP (CON Ctapp) T1) T2).
Definition tlam:= fun T1 => (APP (CON Ctlam) (lambda T1)).
(*************************************************************************
        The atm type and instantiation of oo.
*************************************************************************)
Inductive atm : Set :=
| is_tp : uexp -> atm
| is_tm : uexp -> atm
| typeof : uexp -> uexp -> atm.
Definition oo_ := oo atm ECon.
...
```

Figure 2: A Hybrid Library for Reasoning about the Polymorphic λ-Calculus Part 1

arguments to `lambda` have type (`uexp -> uexp`).

Note that not all Coq functions of type (`uexp -> uexp`) encode object-level λ-terms. Those that do not are often called *exotic* terms. Only functions that behave *uniformly* or *parametrically* on their arguments represent λ-terms. Hybrid includes a predicate `abstr` that rules out exotic terms and identifies exactly those terms that represent OL terms. (See Section 2, pages 52–54 in [4] for a definition of `abstr` as well as other definitions it depends on.) This predicate appears in the Coq code obtained from translating the OL inference rules to prog clauses. (See Figure 3.)

The types `atm` and `oo_` are the Hybrid types of atomic and general formulas, respectively, of the SL. A sequent calculus for the SL is implemented in Hybrid as an inductive predicate defining the type `oo_`. In [4], two sample SLs are given, one for a fragment of second-order intuitionistic logic, and another for an ordered linear logic. The former is used in the work described here. (See Figure 5 on page 68 of [4].) The sequent calculus is analogous to a logic programming interpreter, where the prog clauses can be viewed as a second-order logic program. We note that contexts in the SL are explicitly represented in the inductive definition, while they are implicit in the prog clauses. For readers familiar with Twelf, the prog clauses correspond to a Twelf program, while the SL corresponds to Twelf's meta-level, where OL contexts are represented as meta-level contexts. In Hybrid, both levels are formalized, and thus contexts are explicitly represented and reasoned about at the SL level.

```
(************************************************************************
        Definition of prog
************************************************************************)
Inductive prog : atm -> oo_ -> Prop :=
| tp_arr: forall T1, forall T2,
     prog (is_tp (arr T1 T2)) (Conj (atom_ (is_tp T1)) (atom_ (is_tp T2)))
| tm_lam: forall T1, abstr T1 -> forall T2,
     prog (is_tm (lam T1 T2))
          (Conj (All (fun x1 => (Imp (is_tm x1) (atom_ (is_tm (T1 x1))))))
                (atom_ (is_tp T2)))
...
| ty_a : forall M, forall S, forall T, forall N,
    prog (typeof (app M N) T) (Conj (atom_ (typeof M (arr S T))) (atom_ (typeof N S)))
| ty_l : forall S, forall M, abstr M -> forall T,
    prog (typeof (lam (fun x=>  (M x)) S) (arr S T))
         (All (fun x => (Imp (typeof x S) (atom_ (typeof (M x) T)))))
| ty_ta : forall M, forall T, abstr T -> forall S,
    prog (typeof (tapp M S) (T S)) (atom_ (typeof M (all (fun a => (T a)))))
| ty_tl : forall M, abstr M -> forall T, abstr T ->
    prog (typeof (tlam (fun a => (M a))) (all (fun a => (T a))))
         (All (fun a =>  (atom_ (typeof (M a) (T a))))).
Hint Resolve tp_arr tp_all  tm_app tm_lam tm_tapp tm_tlam  ty_a ty_l
  ty_ta ty_tl : hybrid.
```

Figure 3: A Hybrid Library for Reasoning about the Polymorphic λ-Calculus Part 2

Figure 2 defines `atm` as an inductive set of predicates. Continuing the logic programming analogy, these can be viewed as predicates of a logic program. The binary predicate `typeof` comes directly from the specification. Again, the types of the arguments of this predicate (`tm` and `tp`) in the specification are mapped to `uexp` in the Coq library. As a result, we need to introduce a predicate corresponding to each type in the specification to be used to identify well-formed types and terms of the polymorphic λ-calculus. Here, the `is_tp` and `is_tm` predicates are introduced for this purpose. The last definition in the figure instantiates the `oo_` type, which must be done after the `atm` parameter is defined. The elided part includes other definitions involved in instantiating this type, as well as hints to Coq to help with automating proofs.

Figure 3 defines the prog clauses (or logic program) which serve as the final parameter to the SL. The last 4 clauses are direct translations of the rules in the specification. The `prog` predicate takes two arguments: the conclusion of an inference rule (the head of a logic programming clause) followed by the premise or premises (the body of the logic programming clause). The constructors `Conj`, `Imp`, and `All` are the connectives of the SL, and `atom_` coerces `atm` to `oo_`. `Pi` in the specification language maps to `All`, embedded implication maps to `Imp`, and multiple hypotheses are separated by `Conj`. Note that schematic variables in the specification are implicitly quantified at the outermost level. Explicit quantifiers (`forall` in Coq) are added to each clause of the definition of `prog` as part of the translation.

The other clauses, including the elided ones, are the rules for determining well-formed terms and well-formed types. These are automatically generated from the type declarations in the `Syntax` section, and represent the most complex part of the translation implemented so far.

In Hybrid, as in other logical frameworks such as Twelf, we must be sure that the syntax and inference rules are *adequately* encoded in the meta-language. Proving adequacy involves proving

that there is a one-to-one correspondence between the syntax of the OL and its representation in the meta-language, and that an OL judgment has a proof using the inference rules if and only if the encoded version of the judgment is provable in the logical framework. For an example of how adequacy is proved in Hybrid, see Section 3.2 of [4]. The rules for well-formed terms of the OL are an important component of adequacy proofs in Hybrid. In our example OL, for instance, we must prove that whenever there is a proof in Hybrid that a term of the polymorphic λ-calculus has a particular type, then both the term and the type are well-formed (as defined by the clauses for `is_tp` and `is_tm`).

3 Implementation of the Translation Tool

In this section, we describe the overall structure of the implementation of the translation, which as mentioned, is in OCaml. We have not given a formal definition of the syntax of the specification language, so this description is informal. Using `mllex` and `mlyacc`, the 3 sections of a specification are each parsed to a list of type `(string * exp) list`. In each pair in the list, the first argument is the constructor, predicate, or rule name, and the second argument is an element of the following type:

```
type exp = Type | Id of string | Arrow of exp * exp | App of string * exp list |
           Lambda of string * exp * exp | Pi of string * exp * exp
```

There is a direct mapping of each operator in the specification to a constructor of `exp`. For example, the combination of \ and the dot separating the bound variable from the term maps to `Lambda`. The `type` keyword maps to `Type`, and all other identifiers to `Id`. We unify the syntax of all three sections of the specification, even though the first two do not use `Lambda` and `Pi`. The translation function has the following overall structure, divided here into 4 steps, with details of step 3(e) filled in a bit further in Figure 4.

1. Parse the input file into three lists of declarations: `ds1`, `ds2`, and `ds3` where each one is the parsed input of `Syntax`, `Judgments`, and `Rules`, respectively.

2. Call functions to isolate the following variables:

   (a) `Typelist`: from `ds1`, obtain the list of identifiers (strings) from declarations of the form "`id:type.`" in the specification.
   (b) `Syntax_listName_aux`: from `ds1` and `Typelist`, obtain a list of lists of the remaining identifiers (OL syntax constructor names); use `Typelist` to group them into sublists according to the target types of the constructors (the types just before the terminating dots).
   (c) `Syntax_listExpr_aux`: from `ds1` and `Typelist`, form the list of lists of types of the constructors (expressed as elements of type `exp`), using the same groupings into sublists as above in (b).
   (d) `Rules_listName`: from `ds3`, get the list of the identifiers corresponding to rule names.

3. Call functions to create the following strings using the variables from step 2.

   (a) `string1`: from `Syntax_listName_aux` construct the string for the inductive definition of `ECon`, with one case of the Coq definition for each constructor in `Syntax_listName_aux` with names prepended by `C`. (See `ECon` in Figure 2.)

(b) `string2`: from `Syntax_listName_aux` and `Syntax_listExpr_aux`, construct a string with one line for each constructor, containing a Coq definition for the encoding of syntax for that constructor. Two cases must be considered, depending on whether the constructor's type is first- or second-order. If there is a functional argument, the `lambda` operator is used. (See the 6 definitions at the end of the `Constants` section of Figure 2.)
(c) `string3`: from `Typelist` construct a string for the inductive definition of `atm` containing all the clauses for the well-formedness predicates (those of type `uexp -> atm`, see Figure 2).
(d) `string4`: from `ds2` construct a string containing clauses of the inductive definition of `atm`, one for each predicate in the `Judgments` section. Judgments cannot have function arguments; their types are first-order. We simply count the number of argument types and write "`uexp ->`" for each one, ending the clause with `atm`. (See the last clause of the definition of `atm` in Figure 2.)
(e) `string5`: from `Syntax_listName_aux` and `Syntax_listExpr_aux`, construct a string containing all the prog clauses for well-formedness of OL terms. See Figure 4 for some details of the implementation. (See also Figure 3, which contains 2 of 6 such clauses, with the rest elided.)
(f) `string6`: from `ds3` construct a string containing one prog clause corresponding to each rule in the `Rules` section. We omit the details. (See the last 4 prog clauses in Figure 3.)
(g) `string7`: From `Syntax_listName_aux` and `Rules_listName` it is straightforward to construct the `Hint` string. (See the last line of Figure 3.)

4. Write the following strings to the output file in the appropriate order: strings representing Coq comments, fixed strings (library elements that are the same for all specifications), and the strings obtained from step 3.

4 Extensions

In this section, we discuss the extensions of our tool that we are currently working on, as well as some other near-term goals.

There are a variety of standard lemmas that are useful for reasoning about OLs that can be directly generated from the specification. The next step in our current work is to add capabilities to our tool to automatically generate the statements of these lemmas from the specification. In addition, their proofs are mostly easily automated. Part of our work involves improving Hybrid to include better tactics for automating such proofs. In addition, we envision augmenting the translation to automatically insert parts of a proof script into the Coq libraries. Our current work involves studying the most effective way to combine these two techniques for automating proofs. This approach is used in a variety of other tools such as Krakatoa [6], which automatically generates Coq libraries for Hoare-style reasoning about correctness of Java programs, and uses tactics designed specifically for automating proofs in this domain.

We also have done some preliminary work on extending the specification language to include a declaration section for contexts, used to represent a set of hypotheses. Many theorems require reasoning about contexts, and our previous work on comparing systems [3] focused particularly on this aspect.

Examples of the kinds of "standard lemmas" that we would like to generate and prove partially or fully automatically include lemmas for adequacy and lemmas for dealing with

Loop 1 (outermost): For each type name `tname` in `Typelist`, each corresponding list of constructors `cnames` in `Syntax_listName_aux` and list of expressions representing types `ctypes` in `Syntax_listExpr_aux`, execute Loop 2. Using the first 3 declarations in Figure 1 as an example, the following data is used the first time through Loop 2:

```
tname = "tp"     cnames = ["arr";"all"]
ctypes = [Arrow (Id "tp", Arrow (Id "tp", Id "tp"));
          Arrow (Arrow (Id "tp", Id "tp"), Id "tp")]
```

Loop 2
1. From `tname` and `cnames`, build a list `rnames` of rule names for prog clauses. (In the example, the result is `["tp_arr";"tp_all"]`.)
2. For each element `rname` of `rnames` and the corresponding element `ctype` of `ctypes`, execute Loop 3.

Loop 3
1. Count the number of arguments in `ctype` by finding the number of external arrows (all those except arrows in function types of arguments). Create a list `args` of pairs containing variables `T1`, `T2`, ..., `Tn`, one for each argument and arity of the argument (0 for non-functional arguments). For the first elements of `ctypes`, we get `[("T1",0);("T2",0)]` and for the second element, we get `[("T1",1)]`.
2. Using `rname`, `ctype`, and the corresponding element of `args`, build a string by concatenating the following substrings:
   - `"| " ^ rname ^ ":"`
   - For each `("Ti",m)` in `args`, add `"forall Ti,"`. If `m` > 0, add `"abstr Ti ->"`.
   - `"prog (is_" ^ tname ^ "(" ^ cname`
   - For each pair in `args` write the first element followed by a space. At the end, add `"))"`.
   - Create a string of the form `(Conj` s_1 `(Conj` $s_2 \ldots$ `(Conj` s_{n-1} s_n`)`$\cdots$`))` where n is the number of elements of `args`. If $n = 1$, the string is just s_1 with no `Conj`.
   - If the i^{th} element of `args` is `("Ti",0)`, s_i is `"(atom_ (is_" ^ t ^ " Ti))"` where `t` is the corresponding identifier in `ctype` (always `"tp"` in this example).
   - If the i^{th} element of `args` is `("Ti",m)` where `m` > 0, then create variables `x1`, ..., `xm`. Form s_i as follows: for each `xj`, add the substring `"(All (fun xj => (Imp (is_" ^ tj ^ "xj)"`; end s_i with `"(atom_ (is_" ^ t ^ " (Ti x1...xm)...))"` where `tj` and `t` are the appropriate types in `ctype`.

   For our example, from `tname`, the first elements of `cnames` and `ctypes`, and the first list `args` above, the output string we obtain is the first clause of the inductive definition of `prog` in Figure 3.

Figure 4: Building `string5` from Step 3(e).

explicit contexts, as well as a variety of others. For example, the adequacy lemma mentioned in Section 2 can be automatically generated. Many proofs in Hybrid proceed by induction over the SL with inversion over both the definitions of the SL and the prog clauses of the OL. In addition to the inversion lemmas automatically generated from a Coq inductive definition, we state and prove specialized inversion lemmas that can greatly simplify Hybrid proofs. The first lemma in Figure 5 is an example of such a lemma, one whose statement and proof can be automatically generated. Note that `seq_` is Hybrid's predicate for SL sequents. It takes 3 arguments: the height of a proof, a context of assumptions, and the formula to be proved. The `proper` predicate appearing in the lemma is important for adequacy (see [4]).

Context weakening is a general lemma that follows from the definition of the SL, and it

```
Lemma ty_l_inv : forall (i:nat) (Psi:list atm) (M:uexp->uexp) (T1 T2:uexp),
  (forall x : uexp, proper x ->
    seq_ i Psi (Imp (typeof x T1) (atom_ (typeof (M x) T2)))) ->
  exists j:nat, (i=j+1 /\
    forall x : uexp, proper x ->
      seq_ j (typeof x T1::Psi) (atom_ (typeof (M x) T2))).

Lemma simple_strengthen : forall (i:nat) (T x y:uexp) (Gamma:list atm),
  seq_ i (is_tm x::is_tp y::Gamma) (atom_ (is_tp T)) ->
  seq_ i (is_tp y::Gamma) (atom_ (is_tp T)).
```

Figure 5: Example OL Lemmas

is stated and proved in the SL library. Context "strengthening" on the other hand, where assumptions that are irrelevant to the proof of a particular judgment are removed, depends on the OL. A simple example of a strengthening lemma is given in Figure 5. For several examples that occur in the context of a case study, see [3] (e.g., the occurrence of strengthening in the proof of Theorem 2). Part of our extension to the specification language will include the capability to specify at a high level what kinds of strengthening lemmas are desired, and then automatically generate and prove or partially prove them as part of the translation.

Our specifications can be translated to libraries for other systems supporting reasoning with HOAS, although we have not yet done so. Much of the work done here, however, can be directly reused, including, of course, the parsing of a specification to its internal representation in OCaml, as well as much of the overall structure of the OCaml functions we have defined to perform the translation. The systems we are currently targeting are Abella [5], Twelf [11], and Beluga [9]. A common characteristic of all of these systems is multi-level reasoning. A straightforward modification of the translation is all that should be required to obtain a basic input library for each of these systems. We mentioned earlier that our specification language adopts several features of Twelf directly. In fact, translation to Twelf will be the most straightforward to implement. Adding more significant support for each of these systems, such as including specialized lemmas, will require further effort.

5 Conclusion and Future Work

We have described our translation tool, which provides support for reasoning in Hybrid about object languages expressed using HOAS. We presented the specification language, described the translation to a Hybrid library, and discussed the implementation as well as several extensions planned for the near term.

In the longer term, we would also like to examine translations to more systems, in order to facilitate a more fuller comparison of reasoning in systems supporting HOAS. Such work may require extending the specification language. The work presented here can be considered as a variant of the Ott project [12] tailored specifically to the needs of HOAS. Ott contains constructs for specifying binders, and one of our longer term goals is to integrate our tool with Ott. In the present work, we chose to start with a smaller simpler language targeted to the needs of Hybrid, Abella, Twelf, and Beluga. Another approach is to consider using the higher-order logic programming language λProlog [7] as both the specification language as well as the implementation language for the translation.

Finally, we mentioned in Section 1 that both Hybrid and the specification language restrict the definition of syntax of OLs to second-order types. Another long-term goal is to generalize Hybrid to higher-order, which will then require extending our specification language and translation.

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[2] Venanzio Capretta and Amy P. Felty. Combining de Bruijn indices and higher-order abstract syntax in Coq. In *TYPES*, pages 63–77, 2006.

[3] Amy Felty and Brigitte Pientka. Reasoning with higher-order abstract syntax and contexts: A comparison. In *International Conference on Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2010.

[4] Amy P. Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *Journal of Automated Reasoning*, 48(1):43–105, 2012.

[5] Andrew Gacek. The Abella interactive theorem prover (system description). In *4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 154–161. Springer, 2008.

[6] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa Tool for Certification of Java/JavaCard Programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):86–106, 2004.

[7] Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, 2012.

[8] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[9] Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *5th International Joint Conference on Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 15–21. Springer, 2010.

[10] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[11] Carsten Schürmann. The Twelf proof assistant. In *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*, pages 79–83. Springer, 2009.

[12] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool suppor for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010.