Harpoon: Mechanizing Metatheory Interactively
(System Description)

Jacob Errington[1 [0000−0002−4307−7505]], Junyoung Jang[1 [0000−0001−6338−2155]], and Brigitte Pientka[1 [0000−0002−2549−4276]]

McGill University, Montreal, Canada
{jacob.errington, junyoung.jang}@mail.mcgill.ca, bpientka@cs.mcgill.ca

Abstract. Beluga is a proof checker that provides sophisticated infrastructure for implementing formal systems with the logical framework LF and proving metatheoretic properties as total, recursive functions transforming LF derivations. In this paper, we describe Harpoon, an interactive proof engine built on top of Beluga. It allows users to develop proofs interactively using a small, fixed set of high-level *actions* that safely transform a subgoal. A sequence of actions elaborates into a (partial) *proof script* that serves as an intermediate representation describing an assertion-level proof. Last, a proof script translates into a Beluga program which can be type-checked independently. Harpoon is available on GitHub. We have used Harpoon to replay a wide array of examples covering all features supported by Beluga. In particular, we have used it for normalization proofs, including the recently proposed POPLMark reloaded challenge.

1 Introduction

Mechanizing formal systems and proofs about them plays an important role in establishing trust in programming languages and verifying software systems in general. Key questions in this setting are how to represent variables, (simultaneous) substitutions, assumptions, and derivations that depend on assumptions. Higher-order abstract syntax (HOAS) provides an elegant and unifying answer to these questions, relieving users from having to write boilerplate code.

Beluga is a proof checker with built-in support for HOAS encodings of formal systems based on the logical framework LF [13]. Metatheoretic inductive proofs are implemented as recursive, dependently-typed functions that manipulate and transform HOAS representations [21,4,25]. In this paper, we describe the interactive proof engine Harpoon which is built on top of Beluga. A Harpoon user modularly and incrementally develops a metatheoretic proof by solving independent subgoals via a fixed set of high-level *actions*. An action eliminates the subgoal on which it is executed, filling it with a proof that possibly contains new subgoals to be resolved. The actions we support are: introduction of assumptions, case-analysis, inductive reasoning, and both forward and backward reasoning styles.

While our fixed set of actions is largely inspired by similar systems such as Twelf [20,28,27] and Abella [11], Harpoon advances the state of the art in interactively developing mechanized proofs about HOAS representations in two ways: 1. We treat subgoals as first-class and characterize them using contextual types that pair their goal types together with the contexts in which they are meaningful; a contextual substitution property guarantees that each step of proof development correctly refines the partial proof under construction [8]. 2. Rather than simply record the sequence of actions given by the user, we elaborate this sequence into an assertion-level proof [15], represented as what we call a *proof script*. The proof script is what we record as output of an interactive session. It can be both typechecked directly and translated into a Beluga program.

We have used Harpoon (see https://beluga-lang.readthedocs.io/) on a wide range of representative examples from the Beluga library: normalization proofs for the simply-typed lambda calculus [6], benchmarks for reasoning about binders [9,10], and the recent POPLMark Reloaded challenge [1]. These examples involve numerous concerns that arise in proof development, and cover all the domain-specific abstractions that Beluga provides. Our experience shows that Harpoon lowers the entry barrier for users: they only need to understand how to represent formal systems and derivations using HOAS encodings and can then manipulate the HOAS representations directly via the high-level actions which correspond closely to how proofs are developed on paper. As such, we believe that Harpoon eases the task of proving metatheoretic statements.

2 Proof Development in Harpoon

We introduce the main features of Harpoon by interactively developing the proof of two lemmas that play a central role in the proof of weak normalization of the simply-typed lambda calculus. For a more detailed description, see [6].

2.1 Initial setup: encoding the language

We begin by defining the simply-typed lambda-calculus in the logical framework LF [13] using an intrinsically typed encoding. In typical HOAS style, lambda abstraction takes an LF function representing the abstraction of a term over a variable. There is no case for variables, as they are treated implicitly. We remind the reader that this is a weak, representational function space – there is no case analysis or recursion, so only genuine lambda terms can be represented.

```
LF tp : type =
 | unit: tp
 | arr : tp → tp → tp;
```

```
LF tm : tp → type =
 | lam : (tm T1 → tm T2) → tm (arr T1 T2)
 | app : tm (arr T1 T2) → tm T1 → tm T2;
```

Free variables such as `T1` and `T2` are implicitly universally quantified (see [23]) and programmers subsequently do not supply arguments for implicitly quantified parameters when using a constructor.

Next, we define a small-step operational semantics for the language. For simplicity, we use a call-by-name reduction strategy and do not reduce under lambda-abstractions. Note that we use LF application to encode the object-level substitution in the `s_beta` rule.

```
LF step : tm T → tm T → type =
 | s_app : step M M'
       → step (app M N) (app M' N)
 | s_beta: step (app (lam M) N) (M N);
```

```
LF steps : tm T → tm T → type =
 | next : step M M' → steps M' N
       → steps M N
 | refl: steps M M;
```

Using this definition, we define a notion of termination: a term halts if it reduces to a value. This is captured by the constructor `halts/m`.

```
LF val : tm T → type = v_lam: val (lam M);
LF halts : tm T → type = halts/m : val V → steps M V → halts M;
```

2.2 Termination Property: `intros`, `split`, `unbox`, and `solve`

As the first short lemma, we show the Termination property: if `M'` is known to halt and `steps M M'`, then `M` also halts. We start our interactive proof session by loading the signature and defining the name of the theorem and the statement that we want to prove.

```
Name of theorem: halts_step
Statement of theorem: [ ⊢ step M M'] → [ ⊢ halts M'] → [ ⊢ halts M]
```

We pair each LF object such as `step M M'` together with the LF context in which it is meaningful [21,26,19]. We refer to such an object as a *contextual object* and embed contextual types, written as `_ ⊢ _` , into Beluga types using the "box" syntax. In this example, the LF context, written on the left of ⊢ , is empty, as we consider closed LF objects. As before, the free variables `M` and `M'` are implicitly quantified at the outside. They themselves stand for contextual objects and have contextual type `( ⊢ tm T)`. The *theorem statements* are hence *statements about contextual LF objects* and directly correspond to Beluga types.

The proof begins with a single subgoal whose type is simply the statement of the theorem under no assumptions. Since this subgoal has a function type, Harpoon will automatically apply the **`intros`** action, which introduces assumptions as follows: First, the (implicitly) universally quantified variables `M`, `M'` are added to the *meta-context*. This context collects parameters introduced by universal quantifiers. This is in contrast with the *computational context*, which collects assumptions introduced by the simple function space. In particular, the second phase of the **`intros`** action adds the assumptions `s : [⊢ step M M']` and `h : [⊢ halts M']` to the computational context. Observe that since `M` and `M'` have type `tm T`, **`intros`** also adds `T` to the meta-context, although it is implicit in the definitions of `step` and `halts` and is not visible at all in the theorem statement (see the meta-context Fig. 1 step 1).

The proof proceeds by inversion on `h`. Using the **`split`** action, we add the two new assumptions `S:(⊢ steps M' M2)` and `V:(⊢ val M2)` to the meta-context

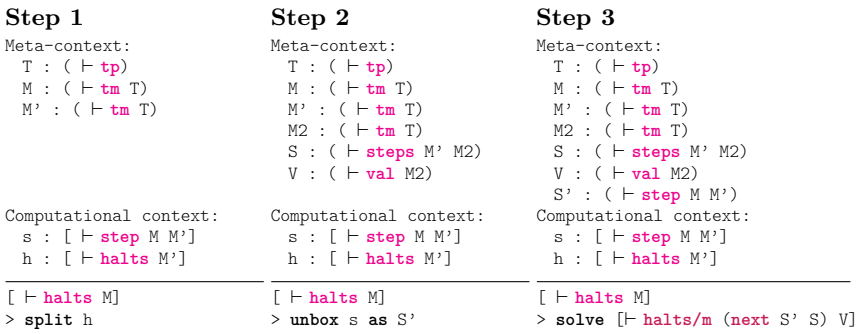

Fig. 1. Interactive session of the proof for the `halts_step` lemma.

(see Fig. 1, step 1.). To build a proof for `[⊢ halts M]`, we need to show that there is a step from `M` to some value `M2`. To build such a derivation, we use first the **`unbox`** action on the computation-level assumption `s` to obtain an assumption `S'` in the meta-context which is accessible to the LF layer (inside a box) (see Fig. 1, step 2.). Finally, we can finish the proof by supplying the term `[ ⊢ halts/m (next S' S) V]` with the **`solve`** action (see Fig. 1, step 3). This is similar to the `exact` tactic in Coq.

The resulting proof script is given below. Assertions are written in boldface and curly braces denote new scopes, listing the full meta-context and the full computational context. Using an erasure we can then generate a translated program in the external syntax, i.e. the syntax a user would use when implementing the proof directly, rather than the internal syntax. It is hence much more compact than the actual proof script. This program can then be seamlessly combined with hand-written Beluga programs and can also independently type-checked.

Theorem `halts_step`: `[ ⊢ step M M'] → [ ⊢ halts M'] → [ ⊢ halts M]`

Proof Script

```
intros
{ T : ( ⊢ tp), M : ( ⊢ tm T), M' : ( ⊢ tm T)
| s : [ ⊢ step M M'], h : [ ⊢ halts M']
; split h as
  case halts/m:
  { T : ( ⊢ tp), M : ( ⊢ tm T), M' : ( ⊢ tm T),
    M2 : ( ⊢ tm T), S : ( ⊢ steps M' M2),
    V : ( ⊢ val M2)
  | s : [ ⊢ step M M'], h : [ ⊢ halts M']
  ; by s as S' unboxed
  ; solve [ ⊢ halts/m (next S' S) V]
  }
}
```

Erased program (external syntax)

```
fn s => fn h =>
  let [ ⊢ halts/m S V] = h in
  let [ ⊢ S'] = s in
    [ ⊢ halts/m (next S' S) V]
```

2.3 Setup continued: reducibility

We now consider one of the key lemmas in the weak normalization proof, called the backwards closed lemma, i.e. if `M'` is reducible at some type `T` and `M` steps to

M', then M is also reducible at T. We begin to define a set of terms *reducible* at a type T. All reducible terms are required to halt, and reducible terms at an arrow type are required to produce reducible output given reducible input. Concretely, a term M is reducible at type (arr T1 T2), if for all terms N:tm T1 where N is reducible at type T1, then (app M N) is reducible at type T2. Reducibility cannot be directly encoded on the LF layer, as it is not merely describing the syntax of an expression or derivation. Instead, we encode the set of reducible terms using the stratified type Reduce which is recursively defined on the type T in Beluga (see [16]). Note that we write { } for explicit universal quantification over contextual objects.

```
stratified Reduce : {T : (⊢ tp)} [⊢ tm T] → ctype =
  | Unit: [⊢ halts M] → Reduce [⊢ unit] [⊢ M]
  | Arr : [⊢ halts M]
        → ({N:(⊢ tm T1)} Reduce [⊢ T1] [⊢ N] → Reduce [⊢ T2] [⊢ app M N])
        → Reduce [⊢ arr T1 T2] [⊢ M];
```

2.4 Backwards Closed Property: msplit, suffices, and by

We can now state the backwards closed lemma formally as follows: if M' is reducible at some type T and M steps to M', then M is also reducible at T. We prove this lemma by induction on T. This is specified by referring to the position of the induction variable in the statement.

```
Name of theorem: bwd_closed
Statement of theorem:
   {T : (⊢ tp)} {M : (⊢ tm T)} {M' : (⊢ tm T)}
   [⊢ step M M'] → Reduce [⊢ T] [⊢ M'] → Reduce [⊢ T] [⊢ M]
Induction order: 1
```

After Harpoon automatically introduces the metavariables T, M, and M' together with an assumption s : [⊢ step M M'] and r : Reduce [⊢ T] [⊢ M'], we use **msplit** T to split the proof into two cases (see Fig. 2, step 1). Whereas **split** case analyzes a Beluga type, **msplit** considers the cases for a (contextual) LF type. In reality, **msplit** is implemented in terms of the **split** action.

The case for T = unit is straightforward (see Fig. 2, steps 2 and 3). First, we use the **split** action to invert the premise r : Reduce [⊢ unit] [⊢ M']. Then, we use the **by** action to invoke the halts_step lemma (see Sec. 2.2) to obtain an assumption h : [⊢ halts M]. We **solve** this case by supplying the term Unit h (see Fig. 2 step 3).

In the case for T = arr T1 T2, we begin similarly by inversion on r using the **split** action (see Fig. 3 step 4). We observe that the goal type is Reduce [⊢ arr T1 T2] [⊢ M], which can be produced by using the Arr constructor if we can construct a proof for each of the user-specified types, [⊢ halts M] and {N:(⊢ tm T1)} Reduce [⊢ T1] [⊢ N] → Reduce [⊢ T2] [⊢ app M N]. Such *backwards reasoning* is accomplished via the **suffices** action. The user supplies a term representing an implication whose conclusion is compatible with the current goal and proceeds to prove its premises as specified (see Fig.3 step 5).

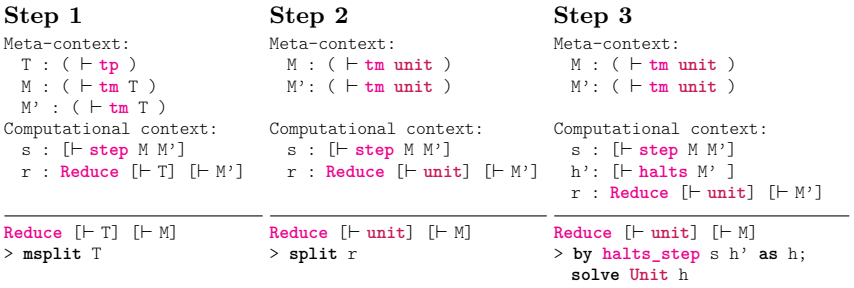

Fig. 2. Backwards Closed Lemma. Step 1: Case analysis of the type `T`; Steps 2 and 3: Base case (`T` = `unit`).

To prove the first premise, we apply the `halts_step` lemma (see Fig. 3 step 6). As for the second premise, Harpoon first automatically introduces the variable `N:(⊢ tm T1)` and the assumption `r1:Reduce [⊢ T1] [⊢ N]`, so it remains to show `Reduce [⊢ T2] [⊢ app M N]`. We deduce `r':Reduce [⊢ T2] [⊢ app M' N]` using the assumption `rn`. Using `s:[⊢ step M M']`, we build a derivation `s':[⊢ step (app M N) (app M' N)]` using `s_app`. Finally, we appeal to the induction hypothesis. Using the **by** action, we refer to the recursive call to complete the proof (see Fig. 3 step 7). The resulting proof script (of around 70 lines) can again be translated into a compact program.

Note that Harpoon allows users to use underscores to stand for arguments that are uniquely determined (see Harpoon Proof 3 step 7). We enforce that these underscores stand for uniquely determined objects in order to guarantee that the contexts and the goal type of every subgoal are closed. This ensures modularity: solving one subgoal does not affect any other open subgoals. As a consequence, users are not restricted in their proof development. As they would on paper, users can work on goals in any order, mix forward and backward reasoning, erase wrong parts, and replace them by correct steps.

Using the explained actions, one can now prove the fundamental lemma and the weak normalization theorem. For a more detailled description of this proof in Beluga see [5,6].

Additional actions. Harpoon supports some additional features not discussed in this paper; see https://beluga-lang.readthedocs.io/ for a complete list of actions. In general, these actions add no expressive power, but enable more precise expression of a user's intent. For example, the **invert** action splits on the type of a given term, ensuring that there is a unique case to consider. It is implemented simply as the **split** action followed by an additional check.

3 Implementation of Harpoon

Harpoon is a front end that allows users to construct a proof for a theorem statement represented as a Beluga type. Types in Beluga include universal

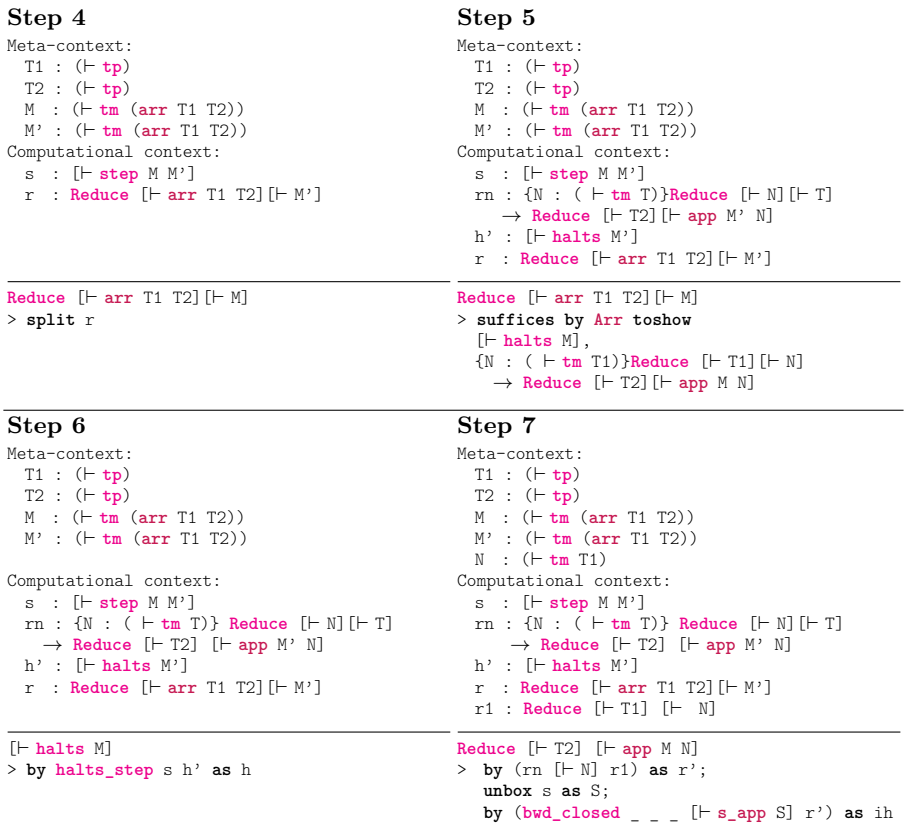

Fig. 3. Backwards Closed Lemma: Step Case

quantification over contextual types (dependent function space, written with curly braces), implications (simple function space), boxed contextual types, and stratified/recursive types (written as $\mathbf{c}\ \overrightarrow{C}$ where C stands for a contextual object). In addition, Beluga supports quantification over LF contexts and even LF substitutions relating two LF contexts. We omit these below for simplicity, although they are also supported in Harpoon. In essence, Beluga types correspond to statements in first-order logic over a domain consisting of contextual objects, LF contexts, and LF substitutions. We can view $\mathbf{c}\ \overrightarrow{C}$ and $[\Psi \vdash A]$ as atomic propositions.

$$\begin{array}{lll} \text{Types} & \tau & ::= \mathbf{c}\ \overrightarrow{C} \mid [\Psi \vdash A] \mid \{X{:}(\Psi \vdash A)\}\,\tau \mid \tau_1 \to \tau_2 \\ \text{Meta-Context} & \Delta & ::= \cdot \mid \Delta, X{::}(\Psi \vdash A) \\ \text{Context} & \Gamma & ::= \cdot \mid \Gamma, x{:}\tau \end{array}$$

Users construct a natural deduction proof for a theorem statement where Γ, the *computation context*, contains hypotheses introduced from the simple function space and where Δ, the *meta-context*, holds parameters introduced

from the universal quantifier (curly-brace syntax) or by lifting an assumption $[\Psi \vdash A]$ from Γ (box-elimination rule).

A subgoal in HARPOON is a typed hole in the proof that remains to be filled by the user. Such a hole is represented by a *subgoal variable*, the type of which is a contextual type $(\Delta; \Gamma \vdash \tau)$ that captures the typechecking state at the point the variable occurs [19,3]: it remains to construct a proof for τ with the parameters from Δ and the assumptions from Γ. Subgoal variables in the proof script are collected into a *subgoal context* and substitution of subgoal variables is type-preserving [8]. Interactive actions are implemented with subgoal substitutions, so the correctness of interactive proof refinement is a consequence of the subgoal substitution property. Note that a subgoal's type cannot itself contain subgoals – the subgoal type must be fully determined, so solving one subgoal cannot affect any other subgoal. Furthermore, subgoal variables may be introduced only in positions where we must construct a normal term (written e); these are terms that we must *check* against a given type. This given type becomes part of the subgoal's type. Subgoal variables stand thus in contrast with ordinary variables, which are neutral terms (written i). (See [14,26,16] for examples of this so-called *bi-directional* characterization of normal and neutral proof terms in BELUGA.)

An action is executed on a subgoal to eliminate it, while possibly introducing new subgoals. Actions emphasize the bi-directional nature of interactive proof construction: some demand normal terms e and others demand neutral terms i. To execute an action, the system synthesizes a proof script fragment from it, and substitutes that fragment for the current subgoal. Any subgoal variables present in the fragment become part of the subgoal context, and the user will have to solve them later. When no subgoals remain, the proof script is closed and can be translated straightforwardly to a BELUGA program in internal (fully elaborated) syntax. We employ an erasure to display the program to the user. These are the essential actions for proof development, omitting our so-called "administrative" actions (such as `undo`):

$$\text{Actions } \alpha ::= \texttt{intros} \mid \texttt{solve}\ e \mid \texttt{by}\ i\ \texttt{as}\ x \mid \texttt{unbox}\ i\ \texttt{as}\ X \mid \texttt{split}\ i \mid \texttt{suffices}\ i\ \texttt{by}\ \overrightarrow{\tau}$$

`intros` introduces all assumptions from function types in the current goal; `solve` closes the current subgoal with a given a normal term, introducing no new subgoals. This action trivially makes HARPOON complete, as a full BELUGA program could be given via `solve` to eliminate the initial subgoal of any proof. The action `by` enables introducing an intermediate result, often from a lemma or an induction hypothesis, demanding a neutral term i and binding it to a given name; `unbox` is the same as `by`, but it binds the result as a variable in the *meta-context*; `split` considers a covering set of cases for a neutral term (typically a variable) and generates possible induction hypotheses based on the specified induction order, (for details on coverage, see [24]); `suffices` allows programmers to reason backwards by supplying a neutral term i of function type and the types $\overrightarrow{\tau}$ of arguments to construct for this function.

4 Empirical evaluation of Harpoon

We give a summary of representative case studies that we replayed using Harpoon in Table 1. In porting these proofs to Harpoon, we use **solve** e only when e is atomic, i.e. it describes either a contextual LF term or a constant applied to all its arguments (either $e = M$, $e = [C]$ or $e = c\ \overrightarrow{C}\ e_1 \dots e_n$). We list in the table the number of commands used to complete the proof and what particular features made the selected case study interesting for testing Harpoon. The first

Case study	Main feature tested
MiniML value soundness	Automatic solving of trivial goals
MiniML compilation completeness	Unboxing program variables
STLC type preservation	Automatic solving of trivial goals
STLC type uniqueness [22]	Open term manipulation: contexts, parameter variables
STLC weak normalization [6]	Case analysis on LF contexts, substitution variables, parameter variables, and inductive and stratified types.
STLC strong normalization [1]	Larger development (310 commands), all forms of case analysis as above.
STLC alg. equality completeness [6]	Larger development (180 commands), all forms of case analysis as above.

Table 1. Summary of proofs ported to Harpoon from Beluga.

four examples proceed by straightforward induction, but the remaining examples are less direct since they feature logical relations. The STLC strong normalization and algorithmic equality completeness examples are larger developments, totalling 38 and 26 theorems respectively. Crucially, these case studies make use of Beluga's domain-specific abstractions, by splitting on contexts, reasoning about object-language variables, and exploiting the built-in equational theory of substitutions. We have since used Harpoon to replay the meta-theoretic proofs about Standard ML from [18].

This evaluation gives us confidence in the robustness and expressive power of Harpoon.

5 Related work

There are several approaches to specify and reason about formal systems.

Beluga and hence Harpoon belong to the lineage of the Twelf system [20], which also implements the logical framework LF. Metatheoretic proofs in Twelf are implemented as *relations*. Totality checking then ensures that these relations correspond to actual proofs. As Twelf is limited to proving Π_1 formulas ("forall-exists" statements), normalization proofs using logical relations cannot be directly encoded. Although Harpoon's actions are largely inspired by the internal actions of Twelf's (experimental) fully-automated metatheorem prover [28,27], Harpoon supports user interaction, more expressive theorem statements, and

generation of proof witnesses, in the form of both the generated proof script and Beluga program resulting from translation.

The Abella system [11] also provides an interactive theorem prover for reasoning about specifications using HOAS. First, its theoretical basis is quite different from Beluga's: Abella's reasoning logic extends first-order logic with a ∇ quantifier [12] that is used to express properties about variables. Second, Abella's interactive mode provides a fixed set of *tactics*, similar to the actions we describe in this paper. However, these tactics only loosely connect to the actual theoretical foundation of Abella and no proof terms are generated as witnesses by the Abella system.

We can also reason about formal systems in general purpose proof assistants such as Coq. The general philosophy in such systems is that users should be in the position of writing complex domain-specific tactics to facilitate proof construction using languages such as LTac [7] or MTac(2) [29,17]. Although this is an extremely flexible approach, we believe that the tactic-centric view often obscures the actual line of reasoning in the proof. The proofs themselves can often be illegible and incomprehensible. Further, strong static guarantees about interactive proof construction are lacking; for example, *dynamic* checks enforce variable dependencies. In contrast, our goal is to enable mechanized proof development in a style close to that of a proof on paper. Thus we provide a fixed set of tactics suitable for a wide array of proofs, so users can concentrate on proof development instead of tactic development. As such, our work draws inspiration from [2] where the authors describe high-level actions within the tutorial proof checker Tutch. Our work extends and adapts this view to the mechanization of inductive metatheoretic proofs based on HOAS representations.

6 Conclusion

We have presented Harpoon, an interactive command-driven front-end of Beluga for mechanizing meta-theoretic proofs based on high-level actions. The sequence of interactive actions is elaborated into a proof script behind the scenes that represents an assertion-level proof. Last, proof scripts can soundly be translated to Beluga programs. We have evaluated Harpoon on several case-studies, ranging from purely syntactic arguments to proofs by logical relations. Our experience is that Harpoon lowers the entry barrier for users to develop meta-theoretic proofs about HOAS encodings.

In the future, we aim to extend Harpoon with additional high-level actions that support further automation. A natural first step is to support an action `trivial` which would attempt to automatically close an open sub-goal.

Acknowledgments. Jacob Errington and Junyoung Jang acknowledge support from Fonds de Recherche du Québec – Nature et technologies (FRQNT). Brigitte Pientka acknowledges support from National Science and Engineering Research Council (NSERC).

References

1. Abel, A., Allais, G., Hameer, A., Pientka, B., Momigliano, A., Schäfer, S., Stark, K.: POPLMark Reloaded: Mechanizing Proofs by Logical Relations. J. Funct. Program. **29**, e19 (2019). https://doi.org/10.1017/S0956796819000170
2. Abel, A., Chang, B.Y.E., Pfenning, F.: Human-readable machine-verifiable proofs for teaching constructive logic. In: Egly, U., Fiedler, A., Horacek, H., Schmitt, S. (eds.) Proceedings of the Workshop on Proof Transformation and Presentation and Proof Complexities (PTP'01). pp. 33–48. Siena, Italy (2001), http://www2.tcs.ifi.lmu.de/~abel/ptp01.pdf
3. Boespflug, M., Pientka, B.: Multi-level contextual modal type theory. In: Nadathur, G., Geuvers, H. (eds.) 6th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP'11). Electronic Proceedings in Theoretical Computer Science (EPTCS), vol. 71, pp. 29–43 (2011)
4. Cave, A., Pientka, B.: Programming with binders and indexed data-types. In: 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12). pp. 413–424. ACM Press (2012)
5. Cave, A., Pientka, B.: First-class substitutions in contextual type theory. In: 8th ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13). pp. 15–24. ACM Press (2013)
6. Cave, A., Pientka, B.: Mechanizing Proofs with Logical Relations – Kripke-style. Mathematical Structures in Computer Science **28**(9), 1606–1638 (2018). https://doi.org/10.1017/S0960129518000154
7. Delahaye, D.: A tactic language for the system Coq. In: Parigot, M., Voronkov, A. (eds.) 7th International Conference on Logic for Programming and Automated Reasoning (LPAR'00). Lecture Notes in Computer Science, vol. 1955, pp. 85–95. Springer (2000). https://doi.org/10.1007/3-540-44404-1_7
8. Errington, J.: Mechanizing metatheory interactively. Master's thesis, McGill University (2020)
9. Felty, A.F., Momigliano, A., Pientka, B.: Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions. Math. Struct. in Comp. Science **28**(9), 1507–1540 (2018). https://doi.org/10.1017/S0960129517000093
10. Felty, A.P., Momigliano, A., Pientka, B.: The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2 - a survey. Journal of Automated Reasoning **55**(4), 307–372 (2015). https://doi.org/10.1007/s10817-015-9327-3
11. Gacek, A.: The Abella interactive theorem prover (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) 4th International Joint Conference on Automated Reasoning. Lecture Notes in Artificial Intelligence, vol. 5195, pp. 154–161. Springer (2008)
12. Gacek, A., Miller, D., Nadathur, G.: Combining generic judgments with recursive definitions. In: Pfenning, F. (ed.) 23rd Symposium on Logic in Computer Science. IEEE Computer Society Press (2008)
13. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM **40**(1), 143–184 (January 1993)
14. Heilala, S., Pientka, B.: Bidirectional decision procedures for the intuitionistic propositional modal logic is4. In: Pfenning, F. (ed.) 21st International Conference on Automated Deduction (CADE'07). pp. 116–131. Lecture Notes in Computer Science (LNCS 4603), Springer (2007)

15. Huang, X.: Reconstruction proofs at the assertion level. In: Bundy, A. (ed.) Proceedings of the 12th International Conference on Automated Deduction (CADE-12). Lecture Notes in Computer Science, vol. 814, pp. 738–752. Springer (1994). https://doi.org/10.1007/3-540-58156-1_53
16. Jacob-Rao, R., Pientka, B., Thibodeau, D.: Index-stratified types. In: Kirchner, H. (ed.) 3rd International Conference on Formal Structures for Computation and Deduction (FSCD'18). pp. 19:1–19:17. LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (January 2018)
17. Kaiser, J., Ziliani, B., Krebbers, R., Régis-Gianas, Y., Dreyer, D.: Mtac2: typed tactics for backward reasoning in coq. Proc. ACM Program. Lang. **2**(ICFP), 78:1–78:31 (2018). https://doi.org/10.1145/3236773
18. Lee, D.K., Crary, K., Harper, R.: Towards a mechanized metatheory of Standard ML. In: 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07). pp. 173–184. ACM Press (2007)
19. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Transactions on Computational Logic **9**(3), 1–49 (2008)
20. Pfenning, F., Schürmann, C.: System description: Twelf — A Meta-Logical Framework for Deductive Systems. In: Ganzinger, H. (ed.) 16th International Conference on Automated Deduction (CADE-16). pp. 202–206. Lecture Notes in Artificial Intelligence (LNAI 1632), Springer (1999)
21. Pientka, B.: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08). pp. 371–382. ACM Press (2008)
22. Pientka, B.: Programming inductive proofs: a new approach based on contextual types. In: Siegler, S., Wasser, N. (eds.) Verification, Induction, Termination Analysis - Festschrift for Christoph Walther on the Occasion of his 60th Birthday. pp. 1–16. Lecture Notes in Computer Science (LNCS 6463), Springer (2010)
23. Pientka, B.: An insider's look at LF type reconstruction: Everything you (n)ever wanted to know. Journal of Functional Programming **1**(1–37) (2013)
24. Pientka, B., Abel, A.: Structural recursion over contextual objects. In: Altenkirch, T. (ed.) 13th International Conference on Typed Lambda Calculi and Applications (TLCA'15). pp. 273–287. Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl (2015)
25. Pientka, B., Cave, A.: Inductive Beluga: Programming Proofs (System Description). In: Felty, A.P., Middeldorp, A. (eds.) 25th International Conference on Automated Deduction (CADE-25). pp. 272–281. Lecture Notes in Computer Science (LNCS 9195), Springer (2015)
26. Pientka, B., Dunfield, J.: Programming with proofs and explicit contexts. In: ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08). pp. 163–173. ACM Press (2008)
27. Schürmann, C.: Automating the Meta Theory of Deductive Systems. Ph.D. thesis, Department of Computer Science, Carnegie Mellon University (2000), CMU-CS-00-146
28. Schürmann, C., Pfenning, F.: Automated theorem proving in a simple meta-logic for LF. In: Kirchner, C., Kirchner, H. (eds.) Proceedings of the 15th International Conference on Automated Deduction (CADE-15). pp. 286–300. Springer-Verlag Lecture Notes in Computer Science (LNCS) 1421, Lindau, Germany (Jul 1998)
29. Ziliani, B., Dreyer, D., Krishnaswami, N.R., Nanevski, A., Vafeiadis, V.: Mtac: A monad for typed tactic programming in Coq. Journal of Functional Programming **25** (2015)