Mechanizing Metatheory Interactively

Jacob Errington Junyoung Clare Jang Brigitte Pientka
McGill University, Montreal, Canada
{jacob.errington,junyoung.jang}@mail.mcgill.ca bpientka@cs.mcgill.ca

Beluga is a proof environment based on the logical framework LF that provides infrastructural support for representing formal systems and proofs about them. As a consequence, meta-theoretic proofs are precise and compact. However, programmers write proofs as total recursive programs. This can be challenging and cumbersome.

We present the design and implementation of Harpoon, an interactive proof environment built on top of Beluga. Harpoon users develop proofs using a small, fixed set of tactics. Behind the scenes, the execution of tactics elaborates a proof script that reflects the subgoal structure of the proof. We model incomplete proofs using contextual variables to represent holes. We give a sound translation of proof scripts into Beluga programs which allows us to execute them. Proof scripts and programs seamlessly interact and can be used interchangeably.

We have used Harpoon for examples ranging from simple type safety proofs for MiniML to normalization proofs including the recently proposed POPLMark reloaded challenge.

1 Introduction

Mechanizing metatheory about formal systems such as programming languages and logics plays an important role in establishing trust in formal developments. One key question in this endeavour is how to represent variables, (simultaneous) substitution, assumptions, derivations that depend on assumptions, and the proof state, as our choices may impact how easy or how cumbersome it will be to develop proofs about formal systems.

Beluga [25, 23] is a proof environment which provides sophisticated infrastructure for implementing formal systems based on the logical framework LF [13]. This allows programmers to uniformly specify syntax, inference rules, and derivation trees using higher-order abstract syntax (HOAS) and relieves users from having to build custom-support to manage variable binding, renaming, and substitution. Following the Curry-Howard correspondence, Beluga users develop inductive metatheoretic proofs about formal systems by writing a total recursive dependently-typed program by pattern matching on derivation trees. Proof checking then amounts to type checking the user's program. Beluga hence follows in the foot steps of proof checkers such as Automath [16], Agda [17], and specifically Twelf [19].

While writing a proof as a dependently typed program is a beautiful idea, it also can be challenging and cumbersome. This limits the wide spread use of dependently-typed programming languages for mechanizing proofs in general. Hence, many proof assistants in this domain provide some form of interaction: for example, Agda [17] supports leaving holes (questionmarks) and writing partial programs which can later be refined using a fixed limited set of interactions. However a clear specification and theoretical foundation of how these interactions transform programs is largely missing. In Coq [2] users interactively develop a proof using *tactics*. Behind the scenes, a sequence of tactic applications is elaborated into a dependently-typed program.

Submitted to:
LFMTP 2020

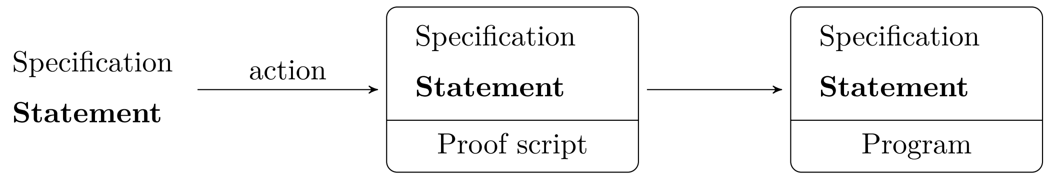

Figure 1: Harpoon Design Overview

Ideally, applying successfully a tactic to a proof state should only result in a new valid, consistent proof state, but this isn't always the case: user-defined tactics for Coq constructed in the Ltac language [8] are mostly unconstrained; it is Coq's typechecker that verifies post hoc that the program generated by the tactics is valid. A common additional caveat of tactic languages is that often, the resulting proof script is brittle and unreadable.

In this paper, we present the design and implementation of Harpoon, an interactive proof environment built on top of Beluga, where programmers develop proofs by a fixed set of tactics. The user invokes a tactic in the context of a subgoal in order to transform, split, or solve it. Our fixed set of tactics is largely inspired by similar systems such as Twelf [27] or Abella [11] supporting introduction of assumptions, case-analysis, and inductive reasoning, as well as both forward and backward reasoning styles. As Harpoon is built on top of Beluga, its tactics can also refer to a Beluga programs to provide an explicit proof witness to justify a proof step. The ability to seamlessly mix programming with command-driven interactive theorem proving is particularly useful when appealing to a lemma and switching between proving and programming. Finally, successful tactic application is guaranteed to transform a valid proof state into another valid proof state. Harpoon's command-driven front-end generates automatically as a result a proof script that retains the subgoal structure. We think of a proof script as an intermediate proof representation language to facilitate translation to other formats, such as into (executable) Beluga programs as shown in this paper or perhaps eventually into a human-readable proof format. Our specific contributions are the following:

- We present the design and implementation of Harpoon, an interactive command-driven front-end of Beluga for mechanizing metatheoretic proofs. Starting from a user-specified theory (including both its syntax and its judgments), users interactively develop metatheoretic proofs using tactics. In tutorial style, we develop a short proof in Sec. 2 by way of giving a whirlwind tour of the main supported tactics in Harpoon.

- We define in Sec. 3.2 a proof script language that reflects the proof structure laid out by the user and clearly separates forwards and backwards reasoning. We describe formally the relation between interactive tactics and proof scripts in Sec. 3.3 and prove soundness of interactive proof construction. We give a type-preserving translation from proof scripts to Beluga programs in Sec. 3.4. This guarantees that proof scripts actually represent proofs and allows proof scripts to be not only typechecked, but also executed. Fig. 1 summarizes the connection among tactics, proof scripts, and Beluga programs.

- We characterize and reason about incomplete programs using contextual types. A variable of such a type represents a hole in the proof, i.e. a statement to prove together with a set of available assumptions. Our formalism of incomplete proofs is such that holes are independent of each other and may be solved in any order. We show that incremental proof development amounts of successively applying contextual substitutions to eliminate holes, while possibly introducing new ones.

- Harpoon is implemented as part of Beluga and is available at https://beluga-lang.readthedocs.io/. We have used it for a range of representive examples from the Beluga library, in particular type safety proofs for MiniML, 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 cover a wide range of aspects that arise in the proof development such as complex reasoning with and about contexts, context schemas, and substitutions.

2 Proof Development in Harpoon

We introduce the main features of Harpoon by considering two lemmas that play a central role in proof of weak normalization of the simply-typed lambda calculus. First, the Termination Property states that if well-typed term `M'` halts and `M` reduces to `M'`, then `M'` halts. Second, the Backwards Closed Property states that if a well-typed term `M'` is reducible and `M` reduces to `M'`, then `M` is also reducible.

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

Free variables such as `T1` and `T2` are implicity universally quantified (see [21]) and programmers subsequently do not supply arguments for implicitly quantified parameters when using a constructor.

With the syntax out of the way, 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.

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

Notice in particular that we use LF application to encode the object-level substitution in the `s_beta` rule. We define a predicate `val: tm T → type` on well-typed terms expressing what it means to be a value: `v_lam: val (lam M)`. Last, we define a notion of termination: a term halts if it reduces to a value. This is captured by the constructor `halts/m`.

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

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

As the first short lemma, we show the Termination Property, that 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 (empty name to finish): halts_step
Statement of theorem: [ ⊢ step M M’] → [ ⊢ halts M’] → [ ⊢ halts M]
```

Beluga is a proof environment in which an encoded theory is clearly separated from its metatheory. LF objects encoding the syntax or judgments from a theory are embedded within Beluga using the "box" syntax `[ ⊢ ]`. Furthermore, we embed such LF objects together with the LF context in which they are meaningful [20, 24, 15]. We call such an object paired with its context a *contextual object*. In this example, the LF context, written on the left of ⊢, is empty as we consider closed LF objects.

Whereas a judgment of an encoded theory is represented as an LF type, a metatheoretic statement is represented as a Beluga type. As is often the case, implications are modelled using simple functions written with → . As before, the free variables `M` and `M'` are implicitly bound by Π-types at the outside, which correspond to universal quantification. In terms of expressiveness, Beluga is comparable to a first-order logic with fixed points together with LF as an index domain.

With theorem configuration out of the way, 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` tactic: first, the (implicitly) universally quantified variables `M`, `M'` are added to the metacontext; second, the assumptions `s : [⊢ step M M’]` and `h : [⊢ halts M’]` are added to the computational context. Observe that since `M` and `M'` have type `tm T`, `intros` also adds `T` to the metacontext, although it is implicit in the definitions of `step` and `halts` and is not visible. (See Harpoon example 1 Step 1.)

Step 1

```
Meta-context:
  T : ( ⊢ tp)
  M : ( ⊢ tm T)
  M’ : ( ⊢ tm T)



Computational context:
  s : [ ⊢ step M M’]
  h : [ ⊢ halts M’]
_________________________
[ ⊢ halts M]
> split h
```

Step 2

```
Meta-context:
  T : ( ⊢ tp)
  M : ( ⊢ tm T)
  M’ : ( ⊢ tm T)
  M2 : ( ⊢ tm T)
  S : ( ⊢ mstep M’ M2)
  V : ( ⊢ val M2)

Computational context:
  s : [ ⊢ step M M’]
  h : [ ⊢ halts M’]
_________________________
[ ⊢ halts M]
> unbox s as S’
```

Step 3

```
Meta-context:
  T : ( ⊢ tp)
  M : ( ⊢ tm T)
  M’ : ( ⊢ tm T)
  M2 : ( ⊢ tm T)
  S : ( ⊢ mstep M’ M2)
  V : ( ⊢ val M2)
  S’ : ( ⊢ step M M’)
Computational context:
  s : [ ⊢ step M M’]
  h : [ ⊢ halts M’]
_________________________
[ ⊢ halts M]
> solve
  [⊢ halts/m (next S’ S) V]
```

Harpoon example 1: Interactive session of the proof for the `halts_step` lemma.

The proof proceeds by inversion on `h`. Using the **`split`** tactic, we add the two new assumptions `S:(⊢ steps M’ M2)` and `V:(⊢ val M2)` to the metacontext. (See Harpoon example 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** tactic on the computation-level assumption `s` to obtain an assumption `S'` in the metacontext which is accessible to the LF layer. (See Harpoon example 1 Step 2.) Finally, we can finish the proof by supplying the term `[ ⊢ halts/m (next S' S) V]` (See Harpoon example 1 Step 3.)

2.3 Setup continued: reducibility

We now proceed to define a set of terms *reducible* at a type `T`. All reducible terms are required to halt, and terms reducible 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 at the LF layer, as it is not merely describing the syntax tree 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 [14]). Note that we write `{ }` for explicit universal quantification.

```
stratified Reduce : {T : (⊢ tp)} [⊢ tm T] → type =
  | 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: tactics `msplit`, `suffices`, and `by`

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 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 Harpoon Proof 2 Step 1). Whereas **split** case analyzes a Beluga type, **msplit** considers the cases for a (contextual) LF type. In reality, **msplit** is syntactic sugar for a more verbose use the ordinary **split** tactic.

The case for `T` = `b` is straightforward (see Harpoon Proof 2 Step 2 and 3). First, we use the **split** tactic to invert the premise `r : Reduce [⊢ b] [⊢ M']`. Then, we use the **by** tactic 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` (Harpoon Proof 2 Step 3).

In the case for `T` = `arr T1 T2`, we begin similarly by inversion on `r` using the **split** tactic (Harpoon Proof 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** tactic. 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 Harpoon Proof 3 Step 5).

Step 1

```
Meta-context:
  T : ( ⊢ tp )
  M : ( ⊢ tm T )
  M' : ( ⊢ tm T )
Computational context:
  s : [⊢ step M M']
  r : Reduce [⊢ T] [⊢ M']
────────────────────────────
Reduce [⊢ T] [⊢ M]
> msplit T
```

Step 2

```
Meta-context:
  M : ( ⊢ tm b )
  M': ( ⊢ tm b )

Computational context:
  s : [⊢ step M M']
  r : Reduce [⊢ b] [⊢ M']
────────────────────────────
Reduce [⊢ b] [⊢ M]
> split r
```

Step 3

```
Meta-context:
  M : ( ⊢ tm b )
  M': ( ⊢ tm b )

Computational context:
  s : [⊢ step M M']
  h': [⊢ halts M' ]
  r : Reduce [⊢ b] [⊢ M']
────────────────────────────
Reduce [⊢ b] [⊢ M]
> by halts_step s h' as h;
  solve Unit h
```

Harpoon example 2: Backwards closed lemma. Step 1: Case analysis of the type `T`; Steps 2 and 3: Base case (`T` = `b`).

Step 4

```
Meta-context:
  T1 : (⊢ tp)
  T2 : (⊢ tp)
  M  : (⊢ tm (arr T1 T2))
  M' : (⊢ tm (arr T1 T2))
Computational context:
  s  : [⊢ step M M']
  r  : Reduce [⊢ arr T1 T2][⊢ M']
────────────────────────────────────
Reduce [⊢ arr T1 T2][⊢ M]
> split r
```

Step 5

```
Meta-context:
  T1 : (⊢ tp)
  T2 : (⊢ tp)
  M  : (⊢ tm (arr T1 T2))
  M' : (⊢ tm (arr T1 T2))
Computational context:
  s  : [⊢ step M M']
  rn : {N : ( ⊢ tm T)} Reduce [⊢ N][⊢ T]
     → Reduce [⊢ T2][⊢ app M' N]
  h' : [⊢ halts M']
  r  : Reduce [⊢ arr T1 T2][⊢ M']
────────────────────────────────────
Reduce [⊢ arr T1 T2][⊢ M]
> suffices by Arr
1> [⊢ halts M]
2> {N : ( ⊢ tm T1)}Reduce [⊢ T1][⊢ N]
   → Reduce [⊢ T2][⊢ app M N]
```

Harpoon example 3: Backwards closed lemma: Step Case

To prove premise `1>`, we apply the `halts_step` lemma (Harpoon Proof 4 Step 6). As for premise `2>`, 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`** tactic, we write out and refer to the recursive call to complete the proof (Harpoon Proof 4 Step 7).

Note that Harpoon allows users to use underscores to stand for arguments that are uniquely determined (see Harpoon Proof 4 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 be closed. This ensures modularity: solving one subgoal does not affect any other open subgoals.

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

Step 6

```
Meta-context:
  T1 : (⊢ tp)
  T2 : (⊢ tp)
  M  : (⊢ tm (arr T1 T2))
  M’ : (⊢ tm (arr T1 T2))

Computational context:
  s  : [⊢ step M M’]
  rn : {N : ( ⊢ tm T)} Reduce [⊢ N][⊢ T]
   → Reduce [⊢ T2] [⊢ app M’ N]
  h’ : [⊢ halts M’]
  r  : Reduce [⊢ arr T1 T2][⊢ M’]
──────────────────────────────────────
[⊢ halts M]
> by halts_step s h’ as h
```

Step 7

```
Meta-context:
  T1 : (⊢ tp)
  T2 : (⊢ tp)
  M  : (⊢ tm (arr T1 T2))
  M’ : (⊢ tm (arr T1 T2))
  N  : (⊢ tm T1)
Computational context:
  s  : [⊢ step M M’]
  rn : {N : ( ⊢ tm T)} Reduce [⊢ N][⊢ T]
    → Reduce [⊢ T2] [⊢ app M’ N]
  h’ : [⊢ halts M’]
  r  : Reduce [⊢ arr T1 T2][⊢ M’]
  r1 : Reduce [⊢ T1] [⊢ N]
──────────────────────────────────────
Reduce [⊢ T2] [⊢ app M N]
> by (rn [⊢ N] r1) as r’;
  unbox s as S; by [⊢ s_app S] as s’;
  by (bwd_closed [⊢ T2] _ _ s’ r’)
  as ih
```

Harpoon example 4: Backwards closed lemma: Step Case – continued

2.5 Additional features

Our implementation of Harpoon supports several features not discussed in this section. Two additional tactics are variants on `split`. First, the `invert` tactic splits on the type of a given term, but checks that the split produces a unique case. Second, the `impossible` tactic verifies that the split produces no cases, so the supplied term’s type is empty.

The `strengthen` tactic can be used to strengthen the contextual type of a given declaration according to a type subordination analysis [31]. This tactic is essential in the completeness proof for algorithmic equality [6].

We also support a number of tactics that do not contribute to the elaboration of the proof, called *administrative tactics*. Many of these are for navigating and listing theorems and subgoals. Besides navigation commands, we include an `undo` tactic for rolling back previous steps in a proof.

Our implementation also performs some rudimentary automation to detect available assumptions that match the current goal type. Already, this is quite convenient as it automatically eliminates certain trivial subgoals from proofs.

3 A Logical Foundation for Interactive Theorem Proving

In this section we give a logical foundation for interactive command-driven theorem proving in Beluga. In particular, we describe interactive commands and their relationship to proof scripts which in turn can be compiled to Beluga programs.

3.1 Background: Beluga’s Programming Language

We begin by describing Beluga’s programming language where we can describe (inductive) proofs as total recursive programs. From a logical perspective, Beluga programs provide witnesses for first-order inductive proofs over a specific index domain. In general, this index domain can be natural numbers, strings, or lists [7, 32], although in BELUGA, this index domain consists of first-class contexts and (contextual) LF objects (see [4]). Keeping this in mind, we keep the index domain abstract in the description of BELUGA below. We abstractly refer to terms and types in the index language by *index term* C and *index type* U.

$$\begin{array}{llll} \text{Index type } U ::= \ldots & \text{Index context} & \Delta & ::= \cdot \mid \Delta, X{:}U \\ \text{Index term } C ::= \ldots & \text{Index substitution} & \theta & ::= \cdot \mid \theta, C/X \end{array}$$

Variables occurring in index terms are declared in an index context Δ. We use index substitutions to model the runtime environment of index variables. Looking up X in the substitution θ returns the index term C to which X is bound at runtime. The index context Δ captures the information that is statically available and is used during type checking.

In the previous example from Sec. 2, the index domain included the definitions for `tp`, `tm A`, `step M M`, and `steps M M`. Recall that to make statements about those index domain objects, we paired the objects (and type) together with the context in which they were meaningful. In our grammar above, U refers to such a contextual type and C denotes a contextual object, for example `(` $\vdash$ `arr unit unit)` is the contextual type of `(` $\vdash$ `lam` λ`x. x)`. Contextual objects may contain index variables that are declared in Δ. For example, `(` $\vdash$ `steps M M)` is meaningful in the index context $\Delta =$ `A:(` $\vdash$ `tp), M:(` $\vdash$ `tm A)`.

We do not describe here in full the index language, since it has been described elsewhere and is not crucial for the understanding or our design of HARPOON; the interested reader is referred to [30, 14]. Instead we list several relevant properties of the index language to be compatible with our current presentation.

Type checking index terms. $\Delta \vdash C \Longleftarrow U$

Substitution principle. If $\Delta \vdash \theta \Longleftarrow \Delta'$ and $\Delta' \vdash C \Longleftarrow U$ then $\Delta \vdash [\theta]C \Longleftarrow [\theta]U$.

Coverage. $\mathsf{cov}\,(\Delta; U) = \overrightarrow{(C_k; \theta_k; \Delta_k; \Gamma_k)}$ computes a covering set for U in the metacontext Δ such that for each k, the index pattern C_k satisfies $\Delta_k \vdash C_k \Longleftarrow [\theta_k]U$. Moreover, it computes any well-founded recursive calls and includes them as part of Γ_k (see [22]).

Below we describe the core fragment of BELUGA. We do so in a bidirectional way, separating terms that we check against a given type from those for which we synthesize a type. To keep the presentation simple, we model (co)inductive and stratified types as constants. Types are simple functions (implications), written as $\tau_1 \to \tau_2$; dependent functions (universal quantification over elements in the index domain), written as $\Pi X{:}U.\tau$; boxed types, written as $[U]$; and constants $\mathbf{b}\ \vec{C}$ used to model (co)inductive and stratified types. Here $\mathbf{b}$ stands for an indexed type family and recall that U stands for a type from the index domain.

$$\begin{array}{lll} \text{Base Types} & \beta & ::= \mathbf{b}\ \vec{C} \mid [U] \\ \text{Types} & \tau & ::= \beta \mid \Pi X{:}U.\tau \mid \tau_1 \to \tau_2 \\ \text{Checkable Terms} & e & ::= \bar{g} \mid i \mid [C] \mid \mathtt{mlam}\ X \Rightarrow e \mid \mathtt{fn}\ x \Rightarrow e \mid \mathtt{case}\ i\ \mathtt{of}\ \overrightarrow{p_k \Rightarrow e_k} \\ & & \quad \mid \mathtt{let}\ x = i\ \mathtt{in}\ e \mid \mathtt{let\,box}\ X = i\ \mathtt{in}\ e \\ \text{Synthesizable Terms} & i & ::= x \mid \mathbf{c} \mid i\ C \mid i\ e \mid (e : \tau) \\ \text{Patterns} & p & ::= [C] \mid \mathbf{c}\ \vec{p} \mid x \\ \text{Context} & \Gamma & ::= \cdot \mid \Gamma, x{:}\tau \\ \text{Subgoal context} & \Omega & ::= \cdot \mid \Omega,\ g{:}(\Delta;\Gamma \vdash \tau) \mid \Omega,\ \bar{g}{:}(\Delta;\Gamma \vdash \tau) \end{array}$$

$\boxed{\Omega \mid \Delta;\Gamma \vdash e \Longleftarrow \tau}$ BELUGA term e checks against type τ

$$\frac{\Omega \mid \Delta, X{:}U;\Gamma \vdash e \Longleftarrow \tau}{\Omega \mid \Delta;\Gamma \vdash \texttt{mlam}\, X \Rightarrow e \Longleftarrow \Pi X{:}U.\tau} \quad \frac{\Omega \mid \Delta;\Gamma, x:\tau_1 \vdash e \Longleftarrow \tau_2}{\Omega \mid \Delta;\Gamma \vdash \texttt{fn}\, x \Rightarrow e \Longleftarrow \tau_1 \to \tau_2} \quad \frac{\Omega \mid \Delta;\Gamma \vdash i \Longrightarrow \tau}{\Omega \mid \Delta;\Gamma \vdash i \Longleftarrow \tau}$$

$$\frac{\begin{array}{c}\Omega \mid \Delta;\Gamma \vdash i \Longrightarrow \beta \quad \mathsf{cov}\,(\Delta;\Gamma;\beta) = \overrightarrow{(p_k, \theta_k, \Delta_k, \Gamma_k)} \\ \text{for all } k.\ \Omega_k \mid \Delta_k;\Gamma_k \vdash e_k \Longleftarrow [\theta_k]\tau\end{array}}{\Omega, \overrightarrow{\Omega_k} \mid \Delta;\Gamma \vdash \texttt{case}\, i\, \texttt{of}\, \overrightarrow{p_k \Rightarrow e_k} \Longleftarrow \tau} \quad \frac{}{g : (\Delta;\Gamma \vdash \tau) \mid \Delta;\Gamma \vdash g \Longleftarrow \tau}$$

$$\frac{\Omega_1 \mid \Delta;\Gamma \vdash i \Longrightarrow \tau' \quad \Omega_2 \mid \Delta;\Gamma, x:\tau' \vdash e \Longleftarrow \tau}{\Omega_1, \Omega_2 \mid \Delta;\Gamma \vdash \texttt{let}\, x = i\, \texttt{in}\, e \Longleftarrow \tau} \quad \frac{\Omega_1 \mid \Delta;\Gamma \vdash i \Longrightarrow [U] \quad \Omega_2 \mid \Delta, X{:}U;\Gamma \vdash e \Longleftarrow \tau}{\Omega_1, \Omega_2 \mid \Delta;\Gamma \vdash \texttt{let}\,\texttt{box}\, X = i\, \texttt{in}\, e \Longleftarrow \tau}$$

$\boxed{\Omega \mid \Delta;\Gamma \vdash i \Longrightarrow \tau}$ BELUGA term i synthesizes type τ

$$\frac{\Gamma(x) = \tau}{\cdot \mid \Delta;\Gamma \vdash x \Longrightarrow \tau} \quad \frac{\mathrm{Sig}(\mathbf{c}) = \tau}{\cdot \mid \Delta;\Gamma \vdash \mathbf{c} \Longrightarrow \tau} \quad \frac{\Omega \mid \Delta;\Gamma \vdash e \Longleftarrow \tau}{\Omega \mid \Delta;\Gamma \vdash (e : \tau) \Longrightarrow \tau}$$

$$\frac{\Omega \mid \Delta;\Gamma \vdash i \Longrightarrow \Pi X{:}U.\,\tau \quad \Delta \vdash C \Longleftarrow U}{\Omega \mid \Delta;\Gamma \vdash i\, C \Longrightarrow [C/X]\tau} \quad \frac{\Omega_1 \mid \Delta;\Gamma \vdash i \Longrightarrow \tau_1 \to \tau_2 \quad \Omega_2 \mid \Delta;\Gamma \vdash e \Longleftarrow \tau_1}{\Omega_1, \Omega_2 \mid \Delta;\Gamma \vdash i\, e \Longrightarrow \tau_2}$$

$\boxed{\vdash \Omega\ \mathsf{sgctx}}$ Ω is a valid subgoal context

$$\frac{}{\vdash \cdot\ \mathsf{sgctx}} \quad \frac{\vdash \Omega\ \mathsf{sgctx} \quad \vdash \Delta\ \mathsf{mctx} \quad \Delta \vdash \tau\ \mathsf{type} \quad \Delta \vdash \Gamma\ \mathsf{ctx}}{\vdash (\Omega, \hat{g} : (\Delta;\Gamma \vdash \tau))\ \mathsf{sgctx}}\ \text{where } \hat{g} \in \{g, \bar{g}\}$$

Figure 2: BELUGA's bidirectional type system, and well-formedness of subgoal contexts.

Synthesizable terms include variables, constants, and simple and dependent function eliminations. All synthesizable terms are checkable. Conversely, one uses a type annotation to embed a checkable term as a synthesizable term. This embedding notably enables using a contextual object as a `case` scrutinee.

Checkable terms include simple and dependent function abstraction (`fn` and `mlam` resp.), boxed index objects $[C]$, and a `case` expression. We also include for convenience two different let-expressions, $\texttt{let}\, x = i\, \texttt{in}\, e$ and $\texttt{let}\,\texttt{box}\, X = i\, \texttt{in}\, e$, although both could be defined given the other terms in the language.

Last, the syntax of checkable expressions contains contextual variables $\bar{g}$ following [15, 3], which we call *subgoal variables*. A subgoal variable represents a typed hole in the program that remains to be filled by the programmer. It captures in its type $(\Delta;\Gamma \vdash \tau)$ the typechecking state at the point it occurs: it remains to construct a term of type τ in the index context Δ with the assumptions in Γ. These subgoal variables are collected in a subgoal context Ω. Algorithmically, we understand a subgoal context Ω not as an input to the typing judgments in Fig. 2 but rather as an output: the set of holes in the program is computed by the judgment. This explains why we must *check* a subgoal variable against a type τ. Observe that subgoal variables appear only in the *term* language: this ensures that subgoals cannot refer to each other. Since subgoals are independent of each other, they may be solved in any order by the user. An expression is called *complete* if it is free from subgoals, and *incomplete* otherwise.

Most of the typing rules in Fig. 2 are as expected. To typecheck a `case` expression, we infer the type of the expression that we want to analyze, then generate a covering set consisting of the pattern and the refinement substitution θ. We then verify that the given set of patterns matches covering set using the primitive $\mathsf{cov}\,(\Delta;\Gamma;\beta)$ which in turn relies on coverage for index objects $\mathsf{cov}\,(\Delta;U)$. Similar to the coverage primitive for index types, the coverage primitive for computation-level base types also generates well-founded recursive calls and includes them as part of Γ_k. We can think of Γ_k as an extension of Γ which includes any program variables from the pattern and any well-founded recursive calls. Finally, we type check each branch taking into account the refinement constraints that might arise from the pattern in addition to the context Γ_k. We omit the rules for checking patterns since pattern typing mirrors expression typing. As for the subgoal context, every rule's conclusion merely collects all premise subgoal contexts to propagate the subgoals downwards. Note that all subgoal variables are distinct and occur exactly once.

We omit kinding rules for types and well-formedness rules for Δ and Γ, but to emphasize that each subgoal type cannot depend on other subgoal types, we include the well-formedness rules for the subgoal context Ω in Fig. 2.

3.2 Harpoon Script Language

To build proofs interactively, we introduce interactive commands, called *actions*, which are typed by the user into the Harpoon interactive prompt. An action is executed on a particular subgoal and eliminates or transforms it, while possibly introducing new subgoals.

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

We consider here a subset of the tactics we support in our implementation of Harpoon: `intros` introduces a series of assumptions; `solve` provides an explicit proof witness/term to close the current subgoal; `by` allows programmers to refer to a lemma, introduce an intermediate result, or use an induction hypothesis, and bind the result to an intermediate program variable; `unbox` is the same as `by` , but it binds the result as an *index* variable; `split` generates a covering set of cases to consider; `suffices` allows programmers to reason backwards via a lemma or a constructor.

Behind the scenes, the interactive execution of tactics incrementally builds a (partial) proof script. This script reflects the structure of the proof very closely, and the core constructs of the proof script language closely resemble the syntax of actions.

$$\begin{array}{lll}
\text{Proof Script} & P & ::= g \mid D \mid \textbf{by}\; i\; \textbf{as}\; x; P \mid \textbf{unbox}\; i\; \textbf{as}\; X; P \\
\text{Directives} & D & ::= \textbf{solve}\; e \mid \textbf{intros}\; \{\Delta;\Gamma \vdash P\} \mid \textbf{split}\; i\; \textbf{as}\; \overrightarrow{\{\Delta_k;\Gamma_k \vdash P_k\}} \\
& & \mid \textbf{suffices by}\; i\; \textbf{to show}\; \overrightarrow{(k{>}\; \tau_k\; \textbf{as}\; P_k)}
\end{array}$$

We give the typing rules for partial Harpoon proof scripts in Fig. 3. In its simplest form, a proof script P is either a subgoal variable g or a directive D that describes how to prove a given goal. The understanding of subgoal variables here is the same as in the previous section: it is a contextual variable of type $(\Delta;\Gamma \vdash \tau)$, representing the goal τ together with its index domain context Δ and assumptions Γ. To distinguish subgoal variables that stand for proofs from those that stand for programs, we write g rather than $\bar{g}$. As before, subgoal variables cannot depend on other subgoal variables.

We extend a proof script using **by** or **unbox** to introduce new assumptions. The **unbox** construct is used to introduce a new index variable by unboxing the result of a given Beluga term i, often an assumption from Γ. The **by** construct is used both for invoking a lemma, introducing an intermediate result, and for appealing to an induction hypothesis, extending Γ with a new variable representing the invocation. Checking that appeals to induction hypotheses are well-founded proceeds simply by looking up the appeal in Γ, as splitting additionally generates valid appeals and extends Γ with them.

$\boxed{\Omega \mid \Delta;\Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau}$ Partial proof script P corresponding to theorem τ

$$\frac{\Omega_1 \mid \Delta;\Gamma \vdash i \Longrightarrow \tau' \quad \Omega_2 \mid \Delta;\Gamma, x{:}\tau' \vdash_{\mathbf{P}} P \Longleftarrow \tau}{\Omega_1,\Omega_2 \mid \Delta;\Gamma \vdash_{\mathbf{P}} \mathbf{by}\ i\ \mathbf{as}\ x; P \Longleftarrow \tau} \qquad \frac{\Omega_1 \mid \Delta;\Gamma \vdash i \Longrightarrow [U] \quad \Omega_2 \mid \Delta, X{:}U;\Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau}{\Omega_1,\Omega_2 \mid \Delta;\Gamma \vdash_{\mathbf{P}} \mathbf{unbox}\ i\ \mathbf{as}\ X; P \Longleftarrow \tau}$$

$$\frac{}{g : (\Delta;\Gamma \vdash \tau) \mid \Delta;\Gamma \vdash_{\mathbf{P}} g \Longleftarrow \tau} \qquad \frac{\Omega \mid \Delta;\Gamma \vdash_{\mathbf{D}} D \Longleftarrow \tau}{\Omega \mid \Delta;\Gamma \vdash_{\mathbf{P}} D \Longleftarrow \tau}$$

$\boxed{\Omega \mid \Delta;\Gamma \vdash_{\mathbf{D}} D \Longleftarrow \tau}$ Directive D establishes theorem τ

$$\frac{\Omega \mid \Delta;\Gamma \vdash e \Longleftarrow \tau}{\Omega \mid \Delta;\Gamma \vdash_{\mathbf{D}} \mathbf{solve}\ e \Longleftarrow \tau} \qquad \frac{g : (\Delta';\Gamma' \vdash \beta) \mid \Delta;\Gamma \vdash \tau \rightsquigarrow e \quad \Omega \mid \Delta';\Gamma' \vdash_{\mathbf{P}} P \Longleftarrow \beta}{\Omega \mid \Delta;\Gamma \vdash_{\mathbf{D}} \mathbf{intros}\ \{\Delta';\Gamma' \vdash P\} \Longleftarrow \tau}$$

$$\frac{\Omega \mid \Delta;\Gamma \vdash i \Longrightarrow \beta \quad \mathsf{cov}\,(\Delta;\Gamma \vdash \beta) = \overrightarrow{(\_;\theta_k;\Delta_k;\Gamma_k)} \quad \forall k.\ \Omega_k \mid \Delta_k;\Gamma_k \vdash_{\mathbf{P}} P_k \Longleftarrow [\theta_k]\tau}{\Omega,\ \bigcup_k \Omega_k \mid \Delta;\Gamma \vdash_{\mathbf{D}} \mathbf{split}\ i\ \mathbf{as}\ \overrightarrow{\{\Delta_k;\Gamma_k \vdash P_k\}} \Longleftarrow \tau}$$

$$\frac{\begin{array}{c}\Omega \mid \Delta;\Gamma \vdash i \Longrightarrow \Pi\Delta'.\ \tau'_n \to \ldots \to \tau'_1 \to \tau'_0 \quad \Delta \vdash (\mathsf{id}_\Delta, \theta) : (\Delta, \Delta') \quad \Delta \vdash [\theta]\tau'_0 = \tau_0 \\ \text{for all } k \in [1,n] \quad \Delta \vdash [\theta]\tau'_k = \tau_k \quad \Omega_k \mid \Delta;\Gamma \vdash_{\mathbf{P}} P_k \Longleftarrow \tau_k\end{array}}{\Omega,\ \bigcup_k \Omega_k \mid \Delta;\Gamma \vdash_{\mathbf{D}} \mathbf{suffices\ by}\ i\ \mathbf{to\ show}\ \overrightarrow{(k\!>\ \tau_k\ \mathbf{as}\ P_k)} \Longleftarrow \tau_0}$$

Figure 3: The type system for Harpoon proofs and directives

There are four different directives we can use in a proof. The simplest directive, **solve** e, merely ends a proof script by giving a proof term e as a witness of the appropriate type. To introduce hypotheses into the index context Δ and the context Γ, we use **intros** $\{\Delta';\Gamma' \vdash P'\}$ where $\Delta';\Gamma'$ are extensions of Δ and Γ. The new goal type τ' and the extended contexts $\Delta';\Gamma'$ are computed from the current subgoal by *unrolling* it as in Fig. 4. At the same time as we are unrolling the type (formula) and building a partial proof script, we also build a partial program witness. This links already proof scripts to programs and we elaborate the full translation in Sec. 3.4. By construction, the resulting partial program expression is well-typed. The unrolling of a type stops once we reach a base type β, i.e. it is either $[U]$ (a contextual LF type) or a stratified or recursive type $\mathbf{b}\ \overrightarrow{C}$.

The directive **split** breaks up the proof into cases, one for each constructor of the type τ' of the term i being split on. The cov primitive computes a covering set of cases and generates well-founded recursive calls based on the user-defined termination measure (see [22]). Each computed

$\boxed{g : (\Delta';\Gamma' \vdash \tau') \mid \Delta;\Gamma \vdash \tau \rightsquigarrow e}$ Beluga type τ unrolls to τ' in the extended meta-context Δ' and computation context Γ'.

$$\frac{}{g : (\Delta;\Gamma \vdash \beta) \mid \Delta;\Gamma \vdash \beta \rightsquigarrow g} \qquad \frac{g : (\Delta';\Gamma' \vdash \beta) \mid \Delta, X{:}U;\Gamma \vdash \tau \rightsquigarrow e}{g : (\Delta';\Gamma' \vdash \beta) \mid \Delta;\Gamma \vdash \Pi X{:}U.\tau \rightsquigarrow \texttt{mlam}\, X \Rightarrow e}$$

$$\frac{g : (\Delta';\Gamma' \vdash \beta) \mid \Delta;\Gamma, x{:}\tau_1 \vdash \tau_2 \rightsquigarrow e}{g : (\Delta';\Gamma' \vdash \beta) \mid \Delta;\Gamma \vdash \tau_1 \to \tau_2 \rightsquigarrow \texttt{fn}\, x \Rightarrow e}$$

Figure 4: Unrolling a Beluga type. By design, this judgment closely mirrors typing and a soundness property holds: if $g : (\Delta';\Gamma' \vdash \beta) \mid \Delta;\Gamma \vdash \tau \rightsquigarrow e$, then $g : (\Delta';\Gamma' \vdash \beta) \mid \Delta;\Gamma \vdash e \Longleftarrow \tau$.

4-tuple contains the pattern p_k (unused here, but used and explained in Sec. 3.4), a refinement substitution θ_k such that $\Delta_k \vdash \theta_k : \Delta$, and contexts Δ_k and Γ_k. The proof is then decomposed into multiple branches, one for each k. Each branch may introduce new assumptions, namely subderivations, and may refine other assumptions via the substitution θ_k. It is also possible for **split** to produce no cases, which corresponds to an elimination principle for empty types.

Last, the **suffices** directive reasons backwards by introducing new proof obligations based on what it takes to establish the current goal. For simplicity, we only consider here types of the form $\Pi\Delta'.\tau'_n \to \ldots \to \tau'_1 \to \tau'_0$. If the current goal type $\Delta \vdash \tau_0$ is an instance of the target type τ'_0, i.e. there exists a substitution θ s.t. $\Delta \vdash \theta : \Delta'$ and $[\theta]\tau_0 = \tau_0$, then the proof is complete if we can construct, for each k, a P_k fullfilling the stated proof obligation $[\theta]\tau'_k$. In practice, θ is computed by unification given both the goal type τ_0 and the target type τ'_0.

3.3 Interactive Proof Development in Harpoon

We describe typing of actions and their elaboration into partial proof scripts in Fig. 5. By design, this process is immediate: each action is simply elaborated into its corresponding construct in the proof script language, using subgoal variables where appropriate to explicitly represent outstanding proof obligations.

Multiple actions can be sequenced to form an interactive *session* $\bar{\alpha}$. A session is an idealized representation of how the user interacts with the proof assistant.

$$\text{Session } \bar{\alpha} ::= \cdot \mid \alpha, \bar{\alpha}$$

We describe typing for sessions also in Fig. 5. Each action in a session is executed on a subgoal variable to eliminate it. This elimination depends on whether the variable stands for a proof, or for a program. In the former case, we obtain by action typing a partial proof P' that we simply substitute for g. In the latter case, we translate the proof P' to an expression e (see Sec. 3.4) before substituting e for $\bar{g}$. Since the partial proof script generated by the execution of an action is well-typed and both forms of subgoal substitution preserve types, we have that sessions preserve types.

Theorem 1 (Session Soundness)**.** *1. If* $\Delta;\Gamma \vdash \alpha : \tau \longrightarrow \Omega \vdash P$*, then* $\Omega \mid \Delta;\Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau$*.*

2. If $\Omega \mid \Delta;\Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau$ *and* $\Omega \vdash P \xrightarrow{\bar{\alpha}} \Omega' \vdash P'$*, then* $\Omega' \mid \Delta;\Gamma \vdash_{\mathbf{P}} P' \Longleftarrow \tau$*.*

Proof. 1. By a simple case analysis on the given derivation. 2. By straightforward induction using (1) and the subgoal substitution property. □

$\boxed{\Delta;\Gamma \vdash \alpha : \tau \longrightarrow \Omega \vdash P}$ Command α applied to subgoal $(\Delta;\Gamma \vdash \tau)$ produces an incomplete proof script P with open subgoals Ω.

$$\frac{\Omega \mid \Delta;\Gamma \vdash e \Longrightarrow \tau}{\Delta;\Gamma \vdash \texttt{solve}\; e : \tau \longrightarrow \Omega \vdash \textbf{solve}\; e}$$

$$\frac{g : (\Delta';\Gamma' \vdash \beta) \mid \Delta;\Gamma \vdash \tau \rightsquigarrow e}{\Delta;\Gamma \vdash \texttt{intros} : \tau \longrightarrow g : (\Delta';\Gamma' \vdash \beta) \vdash \textbf{intros}\; \{(\Delta';\Gamma' \vdash \beta) \vdash g\}}$$

$$\frac{\Omega \mid \Delta;\Gamma \vdash i \Longrightarrow \tau'}{\Delta;\Gamma \vdash \texttt{by}\; i\; \texttt{as}\; x : \tau \longrightarrow \Omega, g : (\Delta;\Gamma, x : \tau' \vdash \tau) \vdash \textbf{by}\; i\; \textbf{as}\; x; g}$$

$$\frac{\Omega \mid \Delta;\Gamma \vdash i \Longrightarrow \beta \quad \mathsf{cov}\,(\Delta;\Gamma;\beta) = \overrightarrow{(\_;\theta_k;\Delta_k;\Gamma_k)}}{\Delta;\Gamma \vdash \texttt{split}\; i : \tau \longrightarrow \Omega, \overrightarrow{g_k : (\Delta_k;\Gamma_k \vdash [\theta_k]\tau)} \vdash \textbf{split}\; i\; \textbf{as}\; \overrightarrow{\{\Delta_k;\Gamma_k \vdash g_k\}}}$$

$$\frac{\Omega \mid \Delta;\Gamma \vdash i \Longrightarrow \Pi\Delta'.\tau'_n \to \ldots \to \tau'_1 \to \tau'_0 \quad \Delta \vdash \theta : (\Delta, \Delta') \quad \Delta \vdash [\theta]\tau'_k = \tau_k \quad \Delta \vdash [\theta]\tau'_0 = \tau_0}{\Delta;\Gamma \vdash \texttt{suffices}\; i\; \texttt{by}\; \overrightarrow{\tau_k} : \tau_0 \longrightarrow \Omega, \overrightarrow{g_k : (\Delta;\Gamma \vdash \tau_k)} \vdash \textbf{suffices by}\;\; i\; \textbf{to show}\; \overrightarrow{k\!>\; g_k}}$$

$\boxed{\Omega \vdash P \xrightarrow{\bar{\alpha}} \Omega' \vdash P'}$ Sequence of actions $\bar{\alpha}$ transforms proof script P with subgoals Ω into proof script P' with subgoals Ω'.

$$\frac{}{\Omega \vdash P \xrightarrow{\cdot} \Omega \vdash P} \qquad \frac{\Delta;\Gamma \vdash \alpha : \tau \longrightarrow \Omega_2 \vdash P' \quad \Omega_1, \Omega_2 \vdash [P'/g]P \xrightarrow{\bar{\alpha}} \Omega_3 \vdash Q}{\Omega_1, g : (\Delta;\Gamma \vdash \tau) \vdash P \xrightarrow{\alpha,\bar{\alpha}} \Omega_3 \vdash Q}$$

$$\frac{\Delta;\Gamma \vdash \alpha : \tau \longrightarrow \Omega_2 \vdash P' \quad \Omega_2 \mid \Delta;\Gamma \vdash_{\mathbf{P}} P' \rightharpoonup e \Longleftarrow \tau \quad \Omega_1, \Omega_2 \vdash [e/\bar{g}]P \xrightarrow{\bar{\alpha}} \Omega_3 \vdash Q}{\Omega_1, \bar{g} : (\Delta;\Gamma \vdash \tau) \vdash P \xrightarrow{\alpha,\bar{\alpha}} \Omega_3 \vdash Q}$$

Figure 5: Typechecking interactive actions and elaboration into partial proof scripts.

3.4 Translation

The translation in Fig. 6 from proofs to Beluga programs is straightforward: **unbox** i **as** X becomes a `let box` expression and **by** i **as** x becomes a `let` expression. As for subgoal variables, we replace g with $\bar{g}$, illustrating that outstanding *proof* obligations have been transformed into program holes. The subgoal context in the translation judgment, being again an output, represents the subgoal variables present in the output term. Hence, it contains only program subgoal variables, of the form $\bar{g}$.

The translation of directives is similarly direct. For **intros**, we already built an incomplete expression e when we were unrolling the type τ, so it suffices to translate P to an expression e' and perform a substitution. The soundness of unrolling and the subgoal substitution property ensure that this preserves types. The **split** directive translates to a case-expression in Beluga, making use of the patterns produced by cov. Finally the **suffices** directive translates into a function application. It is easy to show that the translation is total and type-preserving.

$\boxed{\Omega \mid \Delta;\Gamma \vdash_{\mathbf{P}} P \rightharpoonup e \Longleftarrow \tau}$ Proof P is translates to Beluga term e

$$\frac{\Omega \mid \Delta;\Gamma \vdash_{\mathbf{D}} D \rightharpoonup e \Longleftarrow \tau}{\Omega \mid \Delta;\Gamma \vdash_{\mathbf{P}} D \rightharpoonup e \Longleftarrow \tau} \qquad \frac{\Omega_1 \mid \Delta;\Gamma \vdash i \Longrightarrow \tau' \quad \Omega_2 \mid \Delta;\Gamma, x:\tau' \vdash_{\mathbf{P}} P \rightharpoonup e \Longleftarrow \tau}{\Omega_1,\Omega_2 \mid \Delta;\Gamma \vdash_{\mathbf{P}} \mathbf{by}\ i\ \mathbf{as}\ x; P \rightharpoonup \mathtt{let}\ x = i\ \mathtt{in}\ e \Longleftarrow \tau}$$

$$\frac{}{\bar{g} : (\Delta;\Gamma \vdash \tau) \mid \Delta;\Gamma \vdash_{\mathbf{P}} g \rightharpoonup \bar{g} \Longleftarrow \tau} \qquad \frac{\Omega_1 \mid \Delta;\Gamma \vdash i \Longrightarrow [U] \quad \Omega_2 \mid \Delta, X{:}U;\Gamma \vdash_{\mathbf{P}} P \rightharpoonup e \Longleftarrow \tau}{\Omega_1,\Omega_2 \mid \Delta;\Gamma \vdash_{\mathbf{P}} \mathbf{unbox}\ i\ \mathbf{as}\ X; P \rightharpoonup \mathtt{let\,box}\ X = i\ \mathtt{in}\ e \Longleftarrow \tau}$$

$\boxed{\Delta;\Gamma \vdash_{\mathbf{D}} D \rightharpoonup e \Longleftarrow \tau}$ Directive D translates to Beluga term e

$$\frac{\Omega \mid \Delta;\Gamma \vdash e \Longleftarrow \tau}{\Omega \mid \Delta;\Gamma \vdash_{\mathbf{D}} \mathbf{solve}\ e \rightharpoonup e \Longleftarrow \tau} \qquad \frac{g : (\Delta';\Gamma' \vdash \tau') \mid \Delta;\Gamma \vdash \tau \rightsquigarrow e \quad \Omega \mid \Delta';\Gamma' \vdash_{\mathbf{P}} P \rightharpoonup e' \Longleftarrow \tau'}{\Omega \mid \Delta;\Gamma \vdash_{\mathbf{D}} \mathbf{intros}\ \{\Delta';\Gamma' \vdash P\} \rightharpoonup [e'/g]e \Longleftarrow \tau}$$

$$\frac{\Omega \mid \Delta;\Gamma \vdash i \Longrightarrow \beta \quad \mathsf{cov}\,(\Delta;\Gamma;\beta) = \overrightarrow{(p_k;\theta_k;\Delta_k;\Gamma_k)} \quad \text{for all } k.\ \Omega_k \mid \Delta_k;\Gamma_k \vdash_{\mathbf{P}} P_k \rightharpoonup e_k \Longleftarrow [\theta_k]\tau}{\Omega, \bigcup_k \Omega_k \mid \Delta;\Gamma \vdash_{\mathbf{D}} \mathbf{split}\ i\ \mathbf{as}\ \overrightarrow{\{\Delta_k;\Gamma_k \vdash P_k\}} \rightharpoonup \mathtt{case}\ i\ \mathtt{of}\ \overrightarrow{p_k \Rightarrow e_k} \Longleftarrow \tau}$$

$$\frac{\begin{array}{c}\Omega \mid \Delta;\Gamma \vdash i \Longrightarrow \Pi\Delta'.\tau'_n \to \ldots \to \tau'_1 \to \tau'_0 \quad \Delta \vdash \theta : \Delta,\Delta' \\ \Delta \vdash [\theta]\tau'_k = \tau_k \quad \Omega_k \mid \Delta;\Gamma \vdash_{\mathbf{D}} P_k \rightharpoonup e_k \Longleftarrow \tau_k\end{array}}{\Omega, \bigcup_k \Omega_k \mid \Delta;\Gamma \vdash_{\mathbf{D}} \mathbf{suffices}\ i\ \mathbf{by}\ \overrightarrow{\tau_k\ \mathbf{as}\ P_k} \rightharpoonup i\ \overrightarrow{C_j}\ \overrightarrow{e_k} \Longleftarrow \tau_0}$$

where $\theta = C_1/X_1, \ldots C_m/X_m$

Figure 6: The translation from a Harpoon proof script to a Beluga program.

4 Evaluation

One should be able to use Harpoon to prove anything that one could prove in Beluga. A proper completeness theorem is for now too complex[1], so instead we have replicated a number of case studies originally proven as functional programs in Beluga.

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	Open term manipulation
STLC weak normalization	Advanced splitting
STLC strong normalization [1]	Large development
STLC alg. equality completeness [6]	Large development

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

The first four examples are purely syntactic arguments that proceed by straightforward induction. The remaining examples involve more sophisticated features from Beluga's computation language such as inductive and stratified types used to encode logical relations.

[1] Beluga's support for deep pattern matching complicates a potential completeness proof.

Recreating these case studies in Harpoon provided us with insight as to future work regarding automation: proofs for the syntactic examples tend to proceed by case analysis on the induction variable, inverting any other assumptions when possible, invoking available induction hypotheses, and applying a few inference rules. This recipe could be (partially) automated.

The STLC strong normalization and algorithmic equality completeness examples are larger developments, totalling 38 and 26 theorems respectively. Further, these case studies make use of Beluga's first-class substitutions, contexts, and variables. In particular, these case studies both involve splitting on contexts, reasoning about object-language variables, and exploiting the built-in equational theory of substitutions.

5 Related work & conclusion

The Hazelnut system is similar to Harpoon in that its metatheory formally describes partial programs and the user interactions that construct such a program [18]. Whereas Hazelnut concentrates on programming, Harpoon is an interface to Beluga, a proof assistant. Hazelnut's edit actions construct a simply-typed program by successively filling holes, and types in Hazelnut may also contain holes that are refined by edit actions.

Abella is similar to the Beluga project more broadly in that it is a domain-specific language using HOAS for mechanizing metatheory [11, 12]. Its theoretical basis differs from Beluga's, however, as it extends first-order logic with a ∇ quantifier to express properties about variables. Contexts and simultaneous substitutions are expressed as inductive definitions, but since they are not first-class one must separately establish properties about them, regarding e.g. substitution composition and context well-formedness. Interactive proof development in Abella follows the traditional model: the proof state is manipulated using tactics drawn from a fixed set. No proof object that witnesses the theorem is produced.

The tactic languages of VeriML [28, 29] and Coq stand in contrast to Harpoon's, as theirs may be extended. In VeriML, one uses an ML-like computation language to define tactics that manipulate objects from its underlying logic language. This computation language is very expressive, including such features as nontermination and mutable references. However, its management of metatheoretic concerns such as substitutions and contexts is lower-level: one must explicitly model them, e.g. using lists. As for Coq, its Ltac language [8] is quite unrestricted in its power, so Coq's typechecker must verify separately any term built by an Ltac program.

In conclusion, we have presented Harpoon, an interactive command-driven front-end of Beluga for mechanizing metatheoretic proofs. Users develop proofs using interactive actions that elaborate a proof script behind the scenes. This elaboration's metatheory shows that all intermediate partial proofs are well-typed with respect to a context of outstanding subgoals to resolve and that proof scripts can soundly be translated to Beluga programs. This development relies crucially on reasoning about partial programs, which we represent as containing contextual variables, called subgoal variables, that capture the current typechecking state. We have evaluated Harpoon on a number of case-studies, ranging from purely syntactic arguments to logical relations.

In the future, we aim to improve the automation capabilities of Harpoon. At first, we wish to add a built-in form of proof search to assist in using the `solve` command, perhaps replacing it entirely. In the long term, we hope to apply insights gained from work on Cocon [26] to enable users to define custom tactics together with correctness guarantees about them.

References

[1] Andreas Abel, Guillame Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer & Kathrin Stark (2019): *POPLMark Reloaded: Mechanizing Proofs by Logical Relations. J. Funct. Program.* 29, p. e19, doi:10.1017/S0956796819000170. Available at https://doi.org/10.1017/S0956796819000170.

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

[3] Mathieu Boespflug & Brigitte Pientka (2011): *Multi-level Contextual Modal Type Theory.* In Gopalan Nadathur & Herman Geuvers, editors: *6th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP'11)*, *Electronic Proceedings in Theoretical Computer Science (EPTCS)* 71, pp. 29–43.

[4] Andrew Cave & Brigitte Pientka (2012): *Programming with binders and indexed data-types.* In: *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, ACM Press, pp. 413–424.

[5] Andrew Cave & Brigitte Pientka (2013): *First-class substitutions in contextual type theory.* In: *8th ACM SIGPLAN International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'13)*, ACM Press, pp. 15–24.

[6] Andrew Cave & Brigitte Pientka (2018): *Mechanizing Proofs with Logical Relations – Kripke-style. Mathematical Structures in Computer Science* 28(9), p. 1606–1638, doi:10.1017/S0960129518000154.

[7] James Cheney & Ralf Hinze (2003): *First-Class Phantom Types.* Technical Report CUCIS TR2003-1901, Cornell University.

[8] David Delahaye (2000): *A Tactic Language for the System Coq.* In Michel Parigot & Andrei Voronkov, editors: *7th International Conference on Logic for Programming and Automated Reasoning (LPAR'00)*, *Lecture Notes in Computer Science* 1955, Springer, pp. 85–95, doi:10.1007/3-540-44404-1_7. Available at https://doi.org/10.1007/3-540-44404-1_7.

[9] Amy F. Felty, Alberto Momigliano & Brigitte Pientka (2018): *Benchmarks for Reasoning with Syntax Trees Containing Binders and Contexts of Assumptions. Math. Struct. in Comp. Science* 28(9), pp. 1507–1540, doi:10.1017/S0960129517000093.

[10] 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. Journal of Automated Reasoning* 55(4), pp. 307–372, doi:10.1007/s10817-015-9327-3.

[11] Andrew Gacek (2008): *The Abella Interactive Theorem Prover (System Description).* In: *4th International Joint Conference on Automated Reasoning*, *Lecture Notes in Artificial Intelligence* 5195, Springer, pp. 154–161.

[12] Andrew Gacek, Dale Miller & Gopalan Nadathur (2012): *A Two-Level Logic Approach to Reasoning About Computations. Journal of Automated Reasoning* 49(2), pp. 241–273.

[13] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics. Journal of the ACM* 40(1), pp. 143–184.

[14] Rohan Jacob-Rao, Brigitte Pientka & David Thibodeau (2018): *Index-Stratified Types.* In H. Kirchner, editor: *3rdd International Conference on Formal Structures for Computation and Deduction (FSCD'18)*, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 19:1–19:17.

[15] Aleksandar Nanevski, Frank Pfenning & Brigitte Pientka (2008): *Contextual modal type theory. ACM Transactions on Computational Logic* 9(3), pp. 1–49.

[16] R.P. Nederpelt, J.H. Geuvers & R.C. de Vrijer, editors (2004): *Selected Papers on Automath. Studies in Logic and the Foundations of Mathematics* 133, Elsevier.

[17] Ulf Norell (2007): *Towards a practical programming language based on dependent type theory.* Ph.D. thesis, Department of Computer Science and Engineering, Chalmers University of Technology. Technical Report 33D.

[18] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich & Matthew A. Hammer (2017): *Hazelnut: A Bidirectionally Typed Structure Editor Calculus. SIGPLAN Not.* 52(1), p. 86–99, doi:10.1145/3093333.3009900. Available at https://doi.org/10.1145/3093333.3009900.

[19] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf — A Meta-Logical Framework for Deductive Systems.* In H. Ganzinger, editor: *16th International Conference on Automated Deduction (CADE-16)*, Lecture Notes in Artificial Intelligence (LNAI 1632), Springer, pp. 202–206.

[20] Brigitte Pientka (2008): *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)*, ACM Press, pp. 371–382.

[21] Brigitte Pientka (2013): *An insider's look at LF type reconstruction: Everything you (n)ever wanted to know. Journal of Functional Programming* 1(1–37).

[22] Brigitte Pientka & Andreas Abel (2015): *Structural Recursion over Contextual Objects.* In Thorsten Altenkirch, editor: *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl, pp. 273–287.

[23] Brigitte Pientka & Andrew Cave (2015): *Inductive Beluga:Programming Proofs (System Description).* In Amy P. Felty & Aart Middeldorp, editors: *25th International Conference on Automated Deduction (CADE-25)*, Lecture Notes in Computer Science (LNCS 9195), Springer, pp. 272–281.

[24] Brigitte Pientka & Joshua Dunfield (2008): *Programming with proofs and explicit contexts.* In: *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, ACM Press, pp. 163–173.

[25] Brigitte Pientka & Joshua Dunfield (2010): *Beluga: a Framework for Programming and Reasoning with Deductive Systems (System Description).* In Jürgen Giesl & Reiner Haehnle, editors: *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), Springer, pp. 15–21.

[26] Brigitte Pientka, David Thibodeau, Andreas Abel, Francisco Ferreira & Rébecca Zucchini (2019): *A Type Theory for Defining Logics and Proofs.* In: *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, IEEE, pp. 1–13, doi:10.1109/LICS.2019.8785683. Available at https://doi.org/10.1109/LICS.2019.8785683.

[27] Carsten Schürmann & Frank Pfenning (1998): *Automated Theorem Proving in a Simple Meta-Logic for LF.* In Claude Kirchner & Hélène Kirchner, editors: *Proceedings of the 15th International Conference on Automated Deduction (CADE-15)*, Springer-Verlag Lecture Notes in Computer Science (LNCS) 1421, Lindau, Germany, pp. 286–300.

[28] Antonis Stampoulis & Zhong Shao (2010): *VeriML: typed computation of logical terms inside a language with effects.* In Paul Hudak & Stephanie Weirich, editors: *15th ACM SIGPLAN International Conference on Functional Programming (ICFP'10)*, ACM, pp. 333–344.

[29] Antonis Stampoulis & Zhong Shao (2012): *Static and user-extensible proof checking.* In John Field & Michael Hicks, editors: *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, ACM, pp. 273–284, doi:10.1145/2103656.2103690. Available at https://doi.org/10.1145/2103656.2103690.

[30] David Thibodeau, Andrew Cave & Brigitte Pientka (2016): *Indexed Codata.* In Jacques Garrigue, Gabriele Keller & Eijiro Sumii, editors: *21st ACM SIGPLAN International Conference on Functional Programming (ICFP'16)*, ACM, pp. 351–363.

[31] Roberto Virga (1999): *Higher-order rewriting with dependent types.* Ph.D. thesis, PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University.

[32] Hongwei Xi, Chiyan Chen & Gang Chen (2003): *Guarded Recursive Datatype Constructors*. In: *30th ACM Symposium on Principles of Programming Languages (POPL'03)*, ACM Press, pp. 224–235, doi:10.1145/604131.604150.