Beluga

Sep 12, 2023

Contents

Beluga is a language for encoding and reasoning about formal systems specified by inference rules, e.g. lambda calculi and type systems. It uses contextual modal logic as its foundation. Object-language binding constructs are encoded using higher-order abstract syntax. That is, functions are used to encode binders. Terms are paired with the contexts in which they are meaningful to form *contextual objects*, enabling reasoning about open terms. Proofs in Beluga are represented by recursive programs according to the Curry-Howard correspondence.

Harpoon is an interactive proof development REPL built on top of Beluga. Users develop proofs by successively eliminating subgoals using a small, fixed set of tactics. Behind the scenes, the execution of tactics builds a proof script that can be machine-translated into a traditional Beluga program.

The Beluga project is developed at the Complogic group at McGill university, led by Professor Brigitte Pientka. It is implemented in OCaml.

CHAPTER 1

Getting Started

1.1 Installation

We support only installation on macOS, Linux, and WSL.

Any of the following methods uses `opam` 2. Please ensure that you have that version of `opam`. You can find the installation instruction here.

We use recent versions of OCaml, so check which are supported on our continuous integration before creating an opam switch.

1.1.1 Install using `opam`

The following command will install `beluga` under the switch you created.

```
opam install beluga
```

Now `beluga` and `harpoon` binaries are installed in `$OPAM_SWITCH_PREFIX/bin`, and these command work:

```
beluga --help
harpoon --help
```

1.1.2 Install from the source

First, clone the repository.

```
git clone https://github.com/Beluga-lang/Beluga
cd Beluga
```

Now, build the source with the following commands:

```
make setup-install
make install
```

This will place `beluga` and `harpoon` binaries in `$OPAM_SWITCH_PREFIX/bin`. They can be copied wherever you like, or you can add this directory to your `PATH` environment variable.

1.2 Harpoon Example

You can try the following example to check Harpoon works:

```
LF tp : type =
  | i : tp
  | arr : tp -> tp -> tp
;

LF eq : tp -> tp -> type =
  | eq_i : eq i i
  | eq_arr : eq A1 A2 -> eq B1 B2 -> eq (arr A1 B1) (arr A2 B2)
;
```

First, save this code as a file `tp-refl.bel`. Next, run the following command to load the Harpoon session.

```
harpoon --sig tp-refl.bel
```

Here, `--sig` option represents a *signature* used for proofs. Now, Harpoon will print a session wizard:

```
## Type Reconstruction begin: tp-refl.bel ##
## Type Reconstruction done:  tp-refl.bel ##
Configuring theorem #1
  Name of theorem (:quit or empty to finish):
```

The session wizard will ask for the name of theorem, the actual statement, and the induction order. After giving `tp-refl`, `{A : [|- tp]} [|- eq A A]`, and `1`, the session wizard will print this:

```
## Type Reconstruction begin: stlc.bel ##
## Type Reconstruction done:  stlc.bel ##
Configuring theorem #1
  Name of theorem (:quit or empty to finish): halts_step
  Statement of theorem: [|- step M M'] -> [|- halts M'] -> [|- halts M]
  Induction order (empty for none):
Configuring theorem #2
  Name of theorem (:quit or empty to finish):
```

Users can give any numbers of theorems they want. Here, for the purpose of this example, we will finish the session wizard, by typing the enter key. Then, Harpoon will display an interactive session:

```
Assumptions
  Meta-assumptions:
    A : ( |- tp)
are automatically introduced for the subgoal of type
  {A : ( |- tp)} [ |- eq A A]


Theorem: tp-refl
intros
```

(continues on next page)

(continued from previous page)

```
Meta-context:
  A : ( |- tp)
Computational context:


------------------------------------------------------------------------------
[ |- eq A A]

>
```

Now we can use interactive tactics to prove the goal (the type under the line). First, by applying `split [|- A]`, we split the type into cases.

```
Theorem: tp-refl
intros
Meta-context:
  A : ( |- tp)
Computational context:


------------------------------------------------------------------------------
[ |- eq A A]

> split [|- A]
```

This will generate two subgoals, and you will notice that the label (the string on the second line) is changed so that we can see which subgoal we are in.

```
Theorem: tp-refl
intros <- split [ |- X1] (case arr)
Meta-context:
  X : ( |- tp)
  X1 : ( |- tp)
Computational context:


------------------------------------------------------------------------------
[ |- eq (arr X X1) (arr X X1)]

>
```

To prove this, we need `[|- eq X X]` and `[|- eq X1 X1]`. We can get these by induction.

```
Theorem: tp-refl
intros <- split [ |- X1] (case arr)
Meta-context:
  X : ( |- tp)
  X1 : ( |- tp)
Computational context:


------------------------------------------------------------------------------
[ |- eq (arr X X1) (arr X X1)]

> by tp-refl [|- X] as EQ_X unboxed
```

```
Theorem: tp-refl
intros <- split [ |- X1] (case arr)
Meta-context:
  X : ( |- tp)
  X1 : ( |- tp)
  EQ_X : ( |- eq X X)
Computational context:


------------------------------------------------------------------------------
[ |- eq (arr X X1) (arr X X1)]

> by tp-refl [|- X1] as EQ_X1 unboxed
```

With these two, we are able to use `eq_arr`.

```
Theorem: tp-refl
intros <- split [ |- X1] (case arr)
Meta-context:
  X : ( |- tp)
  X1 : ( |- tp)
  EQ_X : ( |- eq X X)
  EQ_X1 : ( |- eq X1 X1)
Computational context:


------------------------------------------------------------------------------
[ |- eq (arr X X1) (arr X X1)]

> solve [|- eq_arr EQ_X EQ_X1]
```

This will solve the subgoal, and Harpoon will subsequently show the next case, which can be solved directly with `eq_i`.

```
Theorem: tp-refl
intros <- split [ |- FREE MVar 1] (case i)
Meta-context:

Computational context:


------------------------------------------------------------------------------
[ |- eq i i]

> solve [|- eq_i]
```

After solving all subgoals, Harpoon will print the proof script as well as its translation as a Beluga program, and save the proof script (You can check it by `cat tp-refl.bel`) and type-check the signature file again.

```
Subproof complete! (No subgoals left.)
Full proof script:
  intros
  { A : ( |- tp)
  |
  ; split [ |- A] as
    case arr:
    { X : ( |- tp), X1 : ( |- tp)
```

(continues on next page)

(continued from previous page)

```
    |
    ; by tp-refl [ |- X] as EQ_Y unboxed;
      by tp-refl [ |- X1] as EQ_X unboxed;
      solve [ |- eq_arr EQ_Y EQ_X]
    }
    case i:
    {
    |
    ; solve [ |- eq_i]
    }
  }
Translation generated program:
  mlam A =>
  case [ |- A] of
  | [ |- arr X X1] =>
    let [ |- EQ_Y] = tp-refl [ |- X] in
    let [ |- EQ_X] = tp-refl [ |- X1] in [ |- eq_arr EQ_Y EQ_X]
  | [ |- i] =>
    [ |- eq_i]

No theorems left. Checking translated proofs.
- Translated proofs successfully checked.
Proof complete! (No theorems left.)
## Type Reconstruction begin: t/harpoon/tp-refl.bel ##
## Type Reconstruction done:  t/harpoon/tp-refl.bel ##
```

Once the proof is completed, Harpoon will restart the session wizard, and we can choose whether to prove more theorems or `:quit`.

```
Configuring theorem #1
  Name of theorem (:quit or empty to finish): :quit
Harpoon terminated.
```

That's it! If you want to know more details including how to write the signature file and what kinds of tactics do we provide, please read the *common elements* and *interactive proving with harpoon* section of this page. For additional examples, you can check out the test directory in our github repository.

CHAPTER 2

Common elements

2.1 The Logical Framework LF

Languages are encoded for study in Beluga using the Logical Framework LF[1]. New syntactic categories are defined using the `LF` toplevel definition, which defines a new LF (indexed) type together with its constructors.

2.1.1 Basic example: natural numbers

```
LF nat : type =
  | zero : nat
  | succ : nat -> nat
;
```

The first line defines a new simple LF type `nat` and the following lines define its constructors, `zero` and `succ`. The number three is written `succ (succ (succ zero))` in this encoding.

One can define relations on natural numbers by defining *indexed types*. For example, we can encode a less-than-equals relation on `nat` as follows.

```
LF le : nat -> nat -> type =
  | le_z : le zero N
  | le_s : le N M ->
           % ------------------
           le (succ N) (succ M)
;
```

Terms of this type encode *proofs*. As a concrete example, a term of type `le (succ zero) (succ (succ zero))` would represent a proof that 1 is less than 2.

First, observe the presence of *free variables* `M` and `N`. Any free variable is automatically Pi-quantified at the very front of the constructor being defined. Thus, the internal type of `le_s` is really `{N : nat} {M : nat} le`

[1] TODO cite LF paper

`N M -> le (succ N) (succ M)`. (Beluga uses curly braces to write Pi-types.) We call such automatically quantified variables *implicit parameters*. The types of implicit parameters are determined by *type reconstruction*. It is not possible for the user to explicitly provide an instantiation for an implicit parameter. Instantiations for implicit parameters are automatically found via unification, based on the types of provided arguments. In other words, when one writes a term `le_s Q` for some argument proof `Q`, it is via the type of `Q` that the instantiations for `N` and `M` are found.

Next, the rule `le_s` has been suggestively written with a commented line to illustrate that one would ordinarily write this as an inference rule. In Beluga, we use `LF` declarations to uniformly represent the syntax and the inference rules of object languages. (Semantic predicates about encodings are defined using *Inductive Types*.)

One can state a theorem about such an encoding using *Contextual LF* and one can prove them by writing a functional program or *interactively*.

2.1.2 HOAS example: lambda calculus

Beluga is a domain-specific language for reasoning about formal systems, and one of the simplest such systems is the lambda calculus. Unlike in the natural numbers, lambda-terms contain *binders*. The philosophy of LF is to represent binders using higher-order abstract syntax (HOAS). That is, the functions of the metalanguage (LF) are used to represent the binding structure of the object-language (lambda calculus). Here is how we define untyped lambda terms using HOAS.

```
LF tm : type =
  | lam : (tm -> tm) -> tm
  | app : tm -> tm -> tm
;
```

One immense benefit of HOAS is that the object-language inherits the substitution properties of the metalanguage. In practice, this means that one needs not define substitution, but rather simply use LF function application. For example, consider the following encoding of a small-step, call-by-name operational semantics for the lambda calculus.

```
LF step : tm -> tm -> type =
  | e_app : step M M' ->
            % ----------------------
            step (app M N) (app M' N)

  | beta : step (app (lam M) N) (M N)
;
```

First, observe that `step` is not a simple type. It is indexed by two terms, so we understand it as a binary relation between terms.

Finally, the rule `beta` demonstrates HOAS in action. We use LF function application to implement the beta reduction of the lambda calculus. The type of the variable `M` in this constructor is inferred by type reconstruction as `tm -> tm`, given that it appears as the first argument to the constructor `lam`.

To complete the example encoding of the lambda calculus, we will now turn our attention to a simple type assignment system for this language. First, we will define the syntax of types.

```
LF tp : type =
  | base : tp
  | arr : tp -> tp -> tp
;
```

Second, we define the typing judgment as an indexed type. In this case, we understand `oft` as relating a term `tm` to a type `tp`.

```
LF oft : tm -> tp -> type =
  | t_app : oft M (arr A B) -> oft N A ->
            % ------------------------
            oft (app M N) B

  | t_lam : ({x : tm} oft x A -> oft (M x) B) ->
            % ---------------------------------
            oft (lam M) (arr A B)
;
```

We will concentrate on the rule `t_lam`. Here, the variable `M` is understood as the body of the lambda-abstraction, and it has type `tm -> tm`. The premise of this rule reads "for any term `x`, if `x` is of type `A`, then `M x` is of type `B`". This precisely captures the parametric reasoning used on paper when proving that a lambda-abstract has an arrow-type. Here it is necessary to explicitly write a Pi-type for `x` as leaving it implicit would have it incorrect quantified at the level above.

To *reason* about these definitions, one would formulate a theorem and prove it. Theorems are stated and proven in Beluga's computation language. Whereas LF is used as a metalanguage for encoding various formal systems, Beluga's computation language is used as a metalanguage for *Contextual LF*. To prove a theorem, one either writes a functional program in Beluga or *uses Harpoon*.

2.2 Contextual LF

In contrast with languages such as Coq or Agda that feature full dependent types, Beluga has an *indexed type system*. This is a restricted form of dependent types in which one quantifies only over a specific *index domain*. The index domain of Beluga is Contextual LF.

At a high level, a Contextual LF object (or type) is an LF term (or type) together with an LF context in which the term (or type) makes sense. For example, consider the following syntax of types for the simply-typed lambda calculus together with a structural equality relation on types.

```
LF tp : type =
  | base : tp
  | arr : tp -> tp -> tp
;

LF eq : tp -> tp -> type =
  | eq_base : eq base base
  | eq_arr : eq A1 A2 -> eq B1 B2 ->
             % ------------------------
             eq (arr A1 B1) (arr A2 B2)
;
```

The contextual object that encodes a closed proof that the `base` type is equal to the `base` type is `|- eq_base`. We write this sometimes with parentheses to make reading easier, e.g. `( |- eq_base )`.

Open terms can be represented by using a nonempty LF context. For example,

```
x : eq A B |- eq_arr x eq_base
```

is a contextual object of type `x : eq A B |- eq (arr A base) (arr B base)`.

2.2.1 Computation types

Contextual LF objects and types may be *boxed* into **computation types**. These types are primarily used for encoding theorems about a formal system. For example, the type of a theorem expressing that equality is reflexive would be

```
{A : |- tp} [ |- eq A A]
```

We read this as "for all (closed) types `A` , `A` is equal to itself." This is a **computation type**. It is composed of two parts in this example: a *PiBox-type* and a *box-type*. The PiBox is used as a quantifier; we will go into more depth on this in the following section. The syntax `[ |- eq A A ]` is a *boxed contextual type*: any contextual type may be surrounded by a box `[ ]` to embed it as a base computation type. Similarly, contextual objects may be boxed to form computation expressions.

Another example theorem `trans`, that equality is transitive, can be expressed as

```
[ |- eq A B] -> [ |- eq B C] -> [ |- eq A C]
```

Similar to in a pure LF type (see *here*), free metavariables appearing in a computation type are automatically abstracted over using PiBox at the front. So really, the above example type is elaborated into

```
{A : |- tp} {B : |- tp} {C : |- tp}
[ |- eq A B] -> [ |- eq B C] -> [ |- eq A C]
```

But again as in a pure LF type, these quantifiers are implicit and there is no way for a programmer to explicitly supply an instantiation for them. The instantiations are found when the theorem `trans` is used.

This example illustrates the simple function space, written with the arrow `->`. In Beluga, the dependent function space (using the PiBox syntax) is separate from the simple function space (using the arrow syntax).

To recap, the formal grammar of computation types is the following.

where `B` denotes an *inductive type* and `U` denotes a contextual type. The remaining two forms are the simple and dependent function space, respectively.

2.2.2 Metavariables

The syntax `{A : |- tp} ...` expresses what's called a PiBox-type. This example quantifies over the contextual type `|- tp` and binds `A` as a **metavariable**. Whenever a metavariable is used, as `A` is used for example in `eq A A`, it is given a *substitution*. This substitution mediates between the LF context of the metavariable and the LF context at the use site. If no explicit substitution is provided, an identity substitution is assumed. An identity substitution is sufficient in the example because the context of both the metavariable and its use site is the empty context.

For example, suppose we have the metavariable `X : (x : tm, y : tm |- tp)`. (Perhaps `X` refers to `x, y |- arr x y`.) Then, `X[base, base] : |- tp`. Here we use an explicit substitution to instantiate the bound variables `x` and `y`.

2.2.3 Context variables and schemas

Beluga provides a mechanism for abstracting over and quantifying over contexts. An abstract context is referred to by a **context variable**. A context variable is a special form of metavariable.

Whereas kinds classify types, contexts are classified by **schemas**. A schema essentially lists the possible types of the variables occurring in the context. The following declaration defines a schema `ctx` that can contain only types.

```
schema ctx = tp;
```

Before we can elaborate an example demonstrating context variables, first consider the following syntax of terms for the simply typed lambda calculus as well as a typing judgment for this language.

```
LF tm : type =
  | app : tm -> tm -> tm
  | lam : tp -> (tm -> tm) -> tm
;

LF oft : tm -> tp -> type =
  | t_app : oft M (arr A B) -> oft N A ->
            % --------------------------
            oft (app M N) B

  | t_lam : ({x : tm} oft x A -> oft (M x) B) ->
            % ----------------------------------
            oft (lam A M) (arr A B)
```

This syntax of terms includes a type annotation on lambda abstractions, and the typing judgment ensures that it is the type given in the annotation that is used as the parameter `x`'s type in the premise of the `t_lam` rule.

This language admits *type uniqueness*. That is, given two typing derivations for the same term, the types assigned to that term must be equal. We can state this theorem as a computation type in Beluga as follows.

```
(g : ctx) [g |- oft M A1[]] -> [g |- oft M A2[]] -> [|- eq A1 A2]
```

First, notice the syntax `(g : ctx) ...`. This is called *implicit context quantification*. Unlike for ordinary implicit metavariables such as `M`, the schema of an implicit context variable cannot be inferred by type reconstruction. Therefore, one must use implicit context quantification to explicitly specify the schema of the context variable.

Second, notice that the metavariables `A1` and `A2`, referring to types, are associated with the substitution `[]` in the assumptions of the theorem. Type reconstruction is in some sense a greedy algorithm, so had these substitutions been left out, the type of `A1`, upon appearing in `g |- oft M A1`, would be `g |- tp`. But this makes no sense because types ought to be *closed* in the simply-typed lambda calculus. To specify that the metavariables `A1` and `A2` must be closed, we associate them with a *weakening substitution* `[]`. This way, type reconstruction will correct infer that the context of these metavariables is empty.

Confusingly, the reported error had the weakening substitutions been omitted would be relating to the occurrences of `A1` and `A2` in `|- eq A1 A2`. Here, the implicit identity substitution would be ill-typed. Recall that the type of `A1`, for instance, would have been inferred as `g |- tp` and the identity subsitution would need to send the metavariable's context `g` to the empty context, which it does not. In general, when dealing with ill-typed substitution errors, it is worth paying close attention to *every* occurrence of any relevant metavariables.

2.2.4 Unboxing

When one has a computational variable referring of a boxed contextual type, one frequently likes to promote this variable to a metavariable. This process is called *unboxing*. For example, suppose we have the assumption `x : [|- tp]`.

- In Beluga, one writes `let [_ |- X] = x in ...` in order to unbox `x` as the metavariable `X`.
- In Harpoon, one uses the `unbox` *tactic* for this: `unbox x as X`.

2.3 LF Subordination

LF declarations in Beluga may or may not refer to earlier declarations. Therefore, it is possible to ascertain that certain derivations cannot depend on certain contextual assumptions, and to eliminate these assumptions. As a concrete

example, consider this signature.

```
LF unit1 : type =
  | u1 : unit1
;

LF unit2 : type =
  | u2 : unit2
;

schema ctx = block a1 : unit1, a2 : unit2 ;
```

The schema `ctx` consists of blocks (pairs) consisting of a `unit1` and a `unit2`. Now suppose we have a derivation of type `g, b : block a1 : unit1, a2 : unit2 |- unit1` where `g : ctx`. This derivation builds a `unit1`, and by looking at the definition of that type, we can see that there is no way any `unit2` could be involved in the construction of that derivation.

2.3.1 Strengthening

Beluga recognizes as legitimate any program that would drop such irrelevant assumptions: this is called **strengthening**. In Harpoon this is accomplished by using the `strengthen` *tactic*. In a Beluga program, one uses pattern matching together with an explicit substitution that witnesses the strengthening. Suppose in the below example that `x : [g, b : block a1 : unit1, a2 : unit2 |- unit1]`. To strengthen this, one would use a construction as follows.

```
let [g, b : block a1 : unit1, a2 : unit2 |- X[.., b.1]] = x in ...
```

The synthesized type for `X` would be `g, x1 : unit1 |- unit1`. This is somewhat confusing because the substitution that witnesses the strengthening here is in fact a substitution that *weakens* `X`.

Beluga's subordination analysis over the loaded signature accounts for transitive dependencies between LF declarations. Beluga will consider strengthening notably during coverage checking as well as when synthesizing the type of unknowns, written as _ in LF terms.

2.4 Inductive Types

Attention: This page is a stub.

CHAPTER 3

Interactive Proving with Harpoon

Harpoon is an interactive prover for Beluga. Users of Harpoon develop proofs in a REPL using a small set of built-in tactics. These correspond to the main proof strategies one uses in developing metatheory proofs, e.g. inversion or appeal to a lemma or induction hypothesis.

3.1 Prover Structure

Harpoon is structured in a hierarchical fashion: the prover maintains a number of *sessions*, each contains a number of *theorems*, each of which contains a number of *subgoals*. A theorem with no subgoals remaining is *complete*, and similarly, a session with no theorems remaining is *complete*.

3.1.1 Session

Harpoon organizes a set of mutually inductive theorems into a *session*. Often, there is only one session at a time in Harpoon, but more than one simultaneous session is possible. For example, you might want to start a second session if you decide to prove a lemma while in the middle of another proof.

Changing sessions

One can switch between theorems by using the *select* command. Selecting a theorem belonging to another session will cause a *session switch*, which is a somewhat delicate process.

In order to prevent incomplete theorems in different sessions from referring to each other, all theorems belonging to other sessions are not in scope. When switching sessions, the theorems in the active session are moved out of scope, and the theorems in the destination session are brought into scope. Crucially, this is to prevent undesirable circularities between theorems.

It is important to be aware of this limitation, as it means that lemmas must be proven before they can be used.

Session configuration wizard

The series of interactive prompts that appear when Harpoon is started is called the *session configuration wizard*. This wizard appears when there are no sessions in the prover, and unless there were incomplete proofs in the loaded signature, there will be no sessions when Harpoon starts.

The wizard prompts for three things about each theorem:

1. The name of the theorem. This is how the theorem is referred to for induction hypotheses, and how other theorems can refer to this theorem.
2. The statement of the theorem. This is a Beluga computational type. (See *Contextual LF*.)
3. The induction order of the theorem. Non-inductive theorems can simply leave this blank. Inductive theorems specify the order numerically, by giving the position of the parameter to induct on. Only explicit parameters are counted, and counting proceeds from left to right.

Note: Implicit context quantifiers are not counted for the numeric induction order. That is, in

```
(g : ctx) [g |- oft M A] -> [g |- step M M'] -> [g |- oft M' A]
```

the position of the parameter of type `[g |- step M M']` is still `2`.

For example, here is how one might configure a type preservation proof for the simply-typed lambda calculus.

```
Name of theorem: tps
Statement of theorem: [|- oft M A] -> [|- step M M'] -> [|- oft M' A]
Induction order: 2
```

Once a theorem is configured, the wizard will repeat so you can configure additional theorems to be proven mutually. To end the wizard, use an empty theorem name or the special name `:quit`. Ending the wizard without configuring any theorems will abort the creation of the new session. If there are no other sessions in Harpoon, then Harpoon will exit.

Loading goals from a file

Harpoon proofs are recorded into signature files as a **proof script**. This is a structured language whose core constructs close resembles the syntax of *Proof tactics*, which the following section in this document will describe. In a signature file, a proof named `foo` is declared as follows.

```
proof foo : tau = P;
```

where `tau` is a *computation type* and `P` is the body of the proof.

However, when a proof script is saved back using *session commands*, this `P` contains some subgoals left. When harpoon loads a signature file with such incomplete proofs, harpoon will load the proofs into a session so that a user doesn't need to repeat the session wizard. It is still possible to open the session wizard using *session commands* in a case when a user want to use it to start a new proof.

3.1.2 Subgoal

Proofs are developed by applying a tactic to a subgoal. If the tactic is successful, the subgoal it is applied to is *solved* and removed from its theorem. New subgoals may be introduced by the tactic. These subgoals are understood as children of the subgoal that was eliminated.

Subgoal prompt

To apply a tactic, one types the corresponding command into the *subgoal prompt*. This prompt is the main point of interaction with Harpoon. Consider this example subgoal prompt from the beginning of a type preservation proof for the simply-typed lambda calculus.

```
intros
Meta-context:
  M : ( |- tm) (not in scope)
  A : ( |- tp) (not in scope)
  M' : ( |- tm) (not in scope)
Computational context:
  x : [ |- oft M A]
  x1 : [ |- step M M']

--------------------------------
[ |- oft M' A]

>
```

The subgoal prompt shows the prover state at the subgoal. This state contains three key pieces of information.

1. The subgoal label. Every subgoal in Harpoon is identified by a *subgoal label*. This label indicates where in the proof this subgoal is located. In the example, the label is `intros` at the very top, and demonstrates that this subgoal is right after having introduced the assumptions of the theorem.
2. The contexts. Harpoon uses Beluga's *indexed type system* in which one distinguishes between metavariables and computational variables. Metavariables belong to the meta-context and computational variables belong to the computational context. Notice that the metavariables in the example are all marked `(not in scope)`. This annotation is presented for implicit parameters: recall that in the statement of the theorem, the parameters `M`, `A` and `M'` appeared free.
3. The goal. Below the line, the type of the subgoal appears. As tactics are applied and new subgoals are introduced, one can expect the goal type to change. Broadly speaking, one's objective is to construct a term of this type.

Administrative tactics

There are a number of tactics in Harpoon that do not contribute directly to the development of the proof, but are used to manipulate the state of the prover. To distinguish these from the *proof tactics*, we call these *administrative tactics*. Despite not contributing to the development of the proof, administrative tactics are nonetheless entered into the subgoal prompt.

See *here* for the complete list of administrative tactics.

3.2 Proof automation

Harpoon provides rudimentary automation for basic tasks during proofs. Each automation type can be controlled via the `toggle-automation` *tactic*.

Automations run at the moment that a new subgoal is created.

Actions performed by automatic tactics can be *undone*.

3.2.1 auto-intros

This automation introduces assumptions into the context whenever the subgoal has a function type.

3.2.2 auto-solve-trivial

This automation solves subgoals in which the context contains an assumption whose type is convertible with goal type.

It will never solve the last remaining subgoal in a theorem, as this makes certain theorems impossible to prove using Harpoon. For example, this is essential for implementing a function such as `double : [|- nat] -> [|- nat]`: `auto-intros` will bring `x : [|- nat]` into the context and `auto-solve-trivial` would immediately finish the theorem before the user ever sees a subgoal prompt.

actions capabilities.

Current limitations:

- no paramater variables
- no substitution variables
- no case splits (>1 case produced)
- no pair type
- no splitting on contexts

3.2.3 auto-invert-solve

This automation attempts to solve the subgoal if no variable splitting (other than inversions) is required to solve the goal. It performs 2 iterations of depth-first proof-search, once on the computation level, and once again on the LF level. Use `auto-invert-solve INT` to specify the maximum depth you want your search tree to reach. Depth is incremented when we attempt to solve a subgoal.

For example, if we want to solve `Ev [|- S (S (S (S z)))]` this would require a depth bound of at least 2::

```
inductive Ev : [ nat] → ctype =
| ZEv : Ev [ z]
| SEv : Ev [ N] → Ev [ S (S N)]

depth = 0
solve for Ev [|- S (S (S (S z)))] -> focus on SEv --->

depth = 1
solve for Ev [|- S (S z)] -> focus on SEv --->

depth = 2
solve for Ev [|- z] -> focus on zEv

Note: If a goal has more than one subgoal, depth only increments by 1.

For example, if we want to solve ``Less_Than [|- S z] [|- S (S (S z))]`` this
would require depth bound of 3:

inductive Less_Than : [ nat] → [ nat] → ctype =
| ZLT : Less_Than [ z] [ S N]
| LT : Less_Than [ N] [ M] → Less_Than [ S N] [ S M]
| Trans_LT : Less_Than [ N] [ M]
```

(continues on next page)

(continued from previous page)

```
               → Less_Than [ M] [ K] → Less_Than [ N] [ K]

depth = 0
solve for Less_Than [|- S z] [|- S (S (S z))] -> focus on Trans_LT --->

   depth = 1
   solve for Less_Than [|- M] [|- S (S (S z))] -> focus on LT --->

      depth = 2
      solve for Less_Than [|- M'] [|- S (S z)] -> focus on LT --->

         depth = 3
         solve for Less_Than [|- M''] [|- S z] -> focus on ZLT

   ---> found LT LT ZLT : Less_Than [|- S (S z)] [|- S (S (S z))]

   depth = 1
   solve for Less_Than [|- S z] [|- S (S z)] -> focus on LT --->

      depth = 2
      solve for Less_Than [|- z] [|- S z] -> focus on ZLT

   ---> found LT ZLT : Less_Than [|- S z] [|- S (S z)]

-> found Trans_LT (LT ZLT) (LT LT ZLT) : Less_Than [|- S z] [|- S (S (S z))]
```

3.2.4 `inductive-auto-solve`

This automation will perform a case split on the user-specified variable then call auto-invert-solve on each sub case. Use `inductive-auto-solve INT` to specify the maximum depth you want your search tree to reach. Depth is incremented as above.

3.3 Interactive Command Reference

- *session create*
- *session serialize*
- *save*
- *subgoal list*
- *subgoal defer*
- *select*
- *rename*
- *toggle-automation*
- *type*
- *info*

- *Proof tactics*
  - *intros*
  - *split*
  - *msplit*
  - *invert*
  - *impossible*
  - *by*
  - *unbox*
  - *strengthen*
  - *solve*
  - *suffices*
- *auto-invert-solve*
- *inductive-auto-solve*

3.3.1 Administrative tactics

Administrative tactics are used to navigate the proof, obtain information about functions or constructors, or to prove a lemma in the middle of another proof.

undo

Undoes the effect of a previous *proof tactic*. See *Undo*.

redo

Undoes the effect of a previous `undo`. See *Undo*.

history

Displays the `undo` history. See *Undo*.

theorem list

Lists all theorems in the current session.

theorem defer

Moves the current theorem to the bottom of the theorem stack, selecting the next theorem.

See *select* for a more flexible way to select a theorem.

theorem show-ihs

Display the induction hypotheses available in the current subgoal.

Note: This is a debugging command, and the output is not particularly human-readable.

---

theorem dump-proof PATH

Records the current theorem's partial proof to `PATH`.

theorem show-proof

Displays the current theorem's partial proof.

session list

Lists all active sessions together with all theorems within each session.

session defer

Moves the current session to the bottom of the session stack and selects the next one.

See *select* for a more flexible way to select a theorem.

session create

Creates a new session. This command will start the *Session configuration wizard* for setting up the theorems in the new session.

session serialize

Saves the current session as partial proofs to the signature. In other words, any work done interactively will be reflected back into the loaded signature.

save

This command is a shortcut for `session serialize`.

subgoal list

Lists all remaining subgoals in the current theorem.

subgoal defer

Moves the current subgoal to the bottom of the subgoal stack and selects the next one.

select

`select NAME` selects a theorem by name for proving. See the *session list* command.

Note: When selecting a theorem from another session, be aware of the consequences this has on scoping. See *Changing sessions*.

---

rename

Note: Renaming is poorly supported at the moment.

The resulting Harpoon proof script that is generated by interactive proving will not contain the renaming, and this could lead to accidental variable capture.

Renames a variable. Use `rename meta SRC DST` to rename a metavariable and `rename comp SRC DST` to rename a program variable.

toggle-automation

Use `toggle-automation AUTO [STATE]` to change the state of proof automation features. See *Proof automation* for available values for `AUTO`.

Valid values for `STATE` are `on`, `off`, and `toggle`. If unspecified, `STATE` defaults to `toggle`.

type

Use `type EXP` to display the computed type of the given synthesizable expression `EXP`.

info

Use `info KIND OBJ` to get information on the `KIND` named `OBJ`.

Valid values for `KIND` are

- `theorem`: displays information about the Beluga program or Harpoon proof named `OBJ`.

3.3.2 Proof tactics

intros

Use `intros [NAME...]` to introduce assumptions into the context.

Restrictions:

- The current goal type is either a simple or dependent function type.

For Pi-types, the name of the assumption matches the name used in the Pi. For arrow-types, names will be taken from the given list of names, in order. If no names are given explicitly, then names are automatically generated.

On success, this tactic will replace the current subgoal with a new subgoal in which the assumptions are in the context.

Note: It is uncommon to use this tactic directly due to *automation*.

---

split

Use `split EXP` to perform case analysis on the synthesizable expression `EXP`.

Restrictions:

- The expression `EXP` and its synthesized type may not contain uninstantiated metavariables.

On success, this tactic removes the current subgoal and introduces a new subgoal for every possible constructor for `EXP`.

msplit

Use `msplit MVAR` to perform case analysis on the metavariable `MVAR`.

This command is syntactic sugar for `split [_ |- MVAR]`.

invert

Use `invert EXP` to perform inversion on the synthesizable expression `EXP`. This is the same as using `split EXP`, but `invert` will check that a unique case is produced.

impossible

Use `impossible EXP` to eliminate the uninhabited type of the synthesizable expression `EXP`. This is the same as using `split EXP`, but `impossible` will check that zero cases are produced.

by

Use `by EXP as VAR [MODIFIER]` to invoke a lemma or induction hypothesis represented by the synthesizable expression `EXP` and bind the result to the name `VAR`. The optional parameter `MODIFIER` specifies at what level the binding occurs.

Valid values for `MODIFIER` are

- `boxed` (default): the binding is made as a computational variable.

- `unboxed`: the binding is made as a metavariable.
- `strengthened`: the binding is made as a metavariable, and its context is strengthened according to *LF Subordination*.

Restrictions:

- The defined variable `VAR` must not already be in scope.
- `EXP` and its synthesized type may not contain uninstantiated metavariables.
- (For `unboxed` and `strengthened` only.) The synthesized type must be a boxed contextual object.

On success, this tactic replaces the current subgoal with a subgoal having one additional entry in the appropriate context.

Tip: LF terms whose contexts contain blocks are not in principle eligible for strengthening. But such a context is equivalent to a flat context, and Beluga will automatically flatten any blocks when strengthening. Therefore, `strengthened` has a secondary use for flattening.

---

unbox

The command `unbox EXP as X` is syntactic sugar for `by EXP as X unboxed`. See also *by*.

strengthen

The command `strengthen EXP as X` is syntactic sugar for `by EXP as X strengthened`. See also *by*.

solve

Use `solve EXP` to complete the proof by providing an explicit checkable expression `EXP`.

Restrictions:

- The expression `EXP` must check against the current subgoal's type.

On success, this tactic removes the current subgoal, introducing no new subgoals.

suffices

Use `suffices by EXP toshow TAU...` to reason backwards via the synthesizable expression `EXP` by constructing proofs for each type annotation `TAU`.

This command captures the common situation when a lemma or computational constructor can be used to complete a proof, because its conclusion is (unifiable with) the subgoal's type. In this case, it *suffices* to construct the arguments to the lemma or constructor.

The main restriction on `suffices` is that the expression `EXP` must synthesize a type of the form

```
{X1 : U1} ... {Xn : Un} tau_1 -> ... -> tau_k -> tau
```

Thankfully, this is the most common form of type one sees when working with Beluga.

Restrictions:

- The expression `EXP` must synthesize a compatible type, as above.

- Its target type `tau` must unify with the current goal type.
- Each type `tau_i` must unify with the `i` th type annotation given in the command.
- After unification, there must remain no uninstantiated metavariables.

Tip: Sometimes, not all the type annotations are necesary to pin down the instantiations for the Pi-bound metavariables. Instead of a type, you can use _ to indicate that this type annotation should be uniquely inferrable given the goal type and the other specified annotations. It is not uncommon to use `suffices by i toshow _`.

Tip: `suffices` eliminates both explicit and implicit leading Pi-types via unification. It can sometimes be simpler to manually eliminate leading explicit Pi-types via partial application: `suffices by i [C] ... toshow ...`. When explicit Pi-types are manually eliminated, the need for a full type annotation is less common.

On success, one subgoal is generated for each `tau_i`, and the current subgoal is removed.

In principle, this command is redundant with `solve` because one could just write `solve EXP` to invoke the lemma directly, but this can be quite unwieldy if the arguments to the lemma are complicated. Furthermore, the arguments would need to be written as Beluga terms rather than interactively constructed.

Note: The user-provided type annotations `TAU...` are allowed to refer to metavariables marked `(not in scope)`. However, it is an error if an out-of-scope metavariable appears in the instantiation for an explicitly Pi-bound metavariable.

---

actions capabilities.

Current limitations:

- no paramater variables
- no substitution variables
- no case splits (>1 case produced)
- no pair type
- no splitting on contexts

3.3.3 `auto-invert-solve`

This automation attempts to solve the subgoal if no variable splitting (other than inversions) is required to solve the goal. It performs 2 iterations of depth-first proof-search, once on the computation level, and once again on the LF level. Use `auto-invert-solve INT` to specify the maximum depth you want your search tree to reach. Depth is incremented when we attempt to solve a subgoal.

For example, if we want to solve `Ev [|- S (S (S z)))]` this would require a depth bound of at least 2::

```
inductive Ev : [ nat] → ctype =
  | ZEv : Ev [ z]
  | SEv : Ev [ N] → Ev [ S (S N)]

depth = 0
solve for Ev [|- S (S (S z)))] -> focus on SEv --->

depth = 1
```

(continues on next page)

(continued from previous page)

```
solve for Ev [|- S (S z)] -> focus on SEv --->

depth = 2
solve for Ev [|- z] -> focus on zEv

Note: If a goal has more than one subgoal, depth only increments by 1.

For example, if we want to solve ``Less_Than [|- S z] [|- S (S (S z))]`` this
would require depth bound of 3:

inductive Less_Than : [ nat] → [ nat] → ctype =
  | ZLT : Less_Than [ z] [ S N]
  | LT : Less_Than [ N] [ M] → Less_Than [ S N] [ S M]
  | Trans_LT : Less_Than [ N] [ M]
               → Less_Than [ M] [ K] → Less_Than [ N] [ K]

depth = 0
solve for Less_Than [|- S z] [|- S (S (S z))] -> focus on Trans_LT --->

    depth = 1
    solve for Less_Than [|- M] [|- S (S (S z))] -> focus on LT --->

      depth = 2
      solve for Less_Than [|- M'] [|- S (S z)] -> focus on LT --->

          depth = 3
          solve for Less_Than [|- M''] [|- S z] -> focus on ZLT

    ---> found LT LT ZLT : Less_Than [|- S (S z)] [|- S (S (S z))]

    depth = 1
    solve for Less_Than [|- S z] [|- S (S z)] -> focus on LT --->

      depth = 2
      solve for Less_Than [|- z] [|- S z] -> focus on ZLT

    ---> found LT ZLT : Less_Than [|- S z] [|- S (S z)]

-> found Trans_LT (LT ZLT) (LT LT ZLT) : Less_Than [|- S z] [|- S (S (S z))]
```

3.3.4 `inductive-auto-solve`

This automation will perform a case split on the user-specified variable then call auto-invert-solve on each sub case. Use `inductive-auto-solve INT` to specify the maximum depth you want your search tree to reach. Depth is incremented as above.

3.4 Undo

Harpoon includes an `undo` command to revert previous actions. Undo history is stored on a per-theorem basis, so ensure that the correct theorem is selected when executing `undo`. Only commands that effect a change to the subgoal list can be undone. Concretely, this means that *Administrative tactics* cannot be undone, since they do not introduce nor eliminate subgoals.

Harpoon also includes a command `redo` that will undo the effect of the last undo command. If a change to the subgoal list is effected, the redo history is purged. In other words, the history is stored linearly and no "undo tree" is available.

To see the state of the history, use the `history` command. This will show the history of commands that can be undone as well as whether any commands that can be redone.

3.4.1 Limitations

Undo cannot undo things such as creating new sessions. It also cannot undo the command that completes a proof, as beyond that point there is no more subgoal prompt through which the user could type `undo`.

Serializing the current Harpoon state will also cause the undo history to be lost. This is because serialization will cause Harpoon to reload all state from the signature. From Harpoon's point of view, the list of subgoals it then has comes simply from the signature, not from a history of tactics.

To work around these limitations, we currently suggest manually undoing the offending actions by editing the proof script.

CHAPTER 4

Funding

This research has been funded through: NSERC (Natural Science and Engineering Research Council), FQRNT Recherche d'Equipe, PSR-SIIRI Projets conjoints de recherche et d'innovation and 63e session de la Commission permanente de coopération franco-québécoise by Ministère du Développement économique, de l'Innovation et de l'Exportation au Quebec.

search