Typechecking Proof Scripts: Making Interactive Proof Assistants Robust

Marcel K. Goh

McGill University

Abstract. The vast majority of software contains bugs, and various methods have been devised to find bugs and prevent their creation. Formalising programming languages and proving theorems about them is one way of verifying the soundness of programs. Proof assistants provide an interactive medium for constructing such proofs and they are widely used in programming-language research as well as other domains.

Harpoon is a proof assistant built upon the logical framework of Beluga, a functional programming language. User-supplied tactics as well as automated proof-search techniques are applied directly to the internal syntax of a source program, gradually refining subgoals until a correct proof is found. This paper describes an algorithm for typechecking Harpoon proof scripts. In addition, rules are given for the translation of Harpoon proof scripts to equivalent Beluga programs. These typechecking and translation phases act as a layer of verification in the development of a proof.

Note. *This report was written as part of a COMP 400 Honours Project in Computer Science in Fall 2019. The author is indebted to Jacob Errington and Prof. Brigitte Pientka for their patience and extensive guidance.*

1. INTRODUCTION

For as long as we have trusted computers to perform our routine tasks, the gap between the specification of programs and their actual implementation has contributed risk to software projects in nearly every industry. Empirical evidence has suggested that the number of bugs in software is at least proportional to the number of lines it contains [14], meaning that the average industrial codebase contains an immense amount of bugs. Furthermore, once a bug is created, it can silently affect users for a long period of time before it is found. For instance, the average lifespan of a bug in the Linux 2.6 kernel was found to be approximately 1.5 years [18]. In general, users accept buggy software as a fact of life and are happy to use such software so long as it largely meets their needs.

However, in certain sensitive contexts (e.g. network security), system faults are less likely to be dismissed or ignored; and in the most critical of applications, error-laden code can be deadly. In March 2019, all Boeing 737 Max planes were grounded after bugs in the aircraft's anti-stall software were linked to two fatal crashes in the nine months prior [11]. Simply replacing faulty code with different but equally unverified code cannot solve these problems. And as E. W. Dijkstra famously put it, "Program testing can be used to show the presence of bugs, but never to show their absence!" The only way to ensure that a program behaves exactly as specified is to prevent the actual creation of bugs and to prove that the resulting programs are indeed bug-free. This can be done using a proof assistant whose engine is small and proven to work correctly. If we model a source language in the language of a proof assistant, we can show that entire classes of bugs (e.g. non-termination) cannot occur.

Consider the problem of compiling a structured programming language to assembly code. A compiler is a program like any other, and it is easy to see why finding bugs in compilers is critical. After all, even the most formally proven of source programs are of no use if one cannot trust the compiler to preserve the semantics of the program when generating the corresponding assembly code. This concern is not without basis. A three-year study by researchers at the University of Utah found more than 325 previously undetected bugs in

several C compilers using Csmith, a program that randomly generates C programs [23]. Even the widely-used GCC and Clang compilers were found to crash unexpectedly and, more troublingly, silently create incorrect assembly code for valid source programs. The only compiler that produced correct code was CompCert, a verified C compiler written in Coq [12]. In fact, roughly six CPU-years were spent trying to find a bug in CompCert, and outside of unproven parts of the compiler (e.g. the parser), not a single bug was found [23].

The CompCert project is evidence that proving properties about languages can result in robust software. However, constructing proofs in a proof language can be difficult. One must keep in mind all available assumptions and these do not always lead to the desired theorem statement in a straightforward manner. For this reason, proof assistants have been designed as a way to construct proofs without needing to type out the whole proof all at once. Instead, the user can enter a partial proof and the proof assistant will sketch out the progress towards the goal statement, presenting the current proof environment and subgoal in a human-readable manner. (In this way, "proof assistant" is an apt name.) The user can then select one of a number of tactics to apply, which brings the proof one step closer to fruition.

In many conventional proof assistants (e.g. Coq), this interactive loop relies heavily on integration with a text editor. Incomplete source programs are fed into the compiler and elaborated into internal syntax. The proof assistant then identifies holes and user-invoked tactics are applied to an internal representation of the partial program. The resulting proof object is then erased into ordinary code and the source program is modified to reflect the outcome of the tactic. If the final goal is solved, then this loop terminates with a program that accurately corresponds to the proof. The alternation between internal and external syntax is clumsy and repeated modification of source code provides an opportunity for errors in implementation to affect the validity of proofs.

Harpoon, an interactive proof assistant for Beluga, approaches the problem in a different fashion by separating the proof construction from the external source program. Beluga code is parsed into Harpoon internal syntax only once, and upon invocation of a tactic, only this internal representation is modified. All of this happens in a standalone program, allowing the user to focus on the theorem at hand, instead of the program that is generated as a by-product. When a correct proof is found, success is announced to the user and a proof script that represents the generated proof is created. This script is an improvement over the fragile alternative of simply recording user commands because it is an independent representation that contains all the information required to reconstruct the proof or the program that corresponds to it.

The central problem that this paper addresses is this task of translating this proof script into its corresponding Beluga program. It is important not to generate code that contains type errors (otherwise no improvement would have been made over the conventional methods previously outlined). A typechecking algorithm must therefore be devised to ensure that only well-typed proofs are translated. The typechecking will be done separately and only after it is complete should the Harpoon syntax tree be translated into Beluga internal syntax, which can more safely be converted into source code.

2. GRAMMAR

Before we can begin developing the rules by which we will typecheck and translate Harpoon proofs, it is worthwhile to acquaint ourselves with the structure of Beluga and Harpoon's syntax as well as some of the notation that will be used in the rules we devise.

2.1 Basic Notions and Notation

We write $x : A$ to indicate that the term represented by the variable x is of type A. A *context* is a snoc-list of such type declarations: it is either empty or consists of a type-declaration appended to another context. When the validity of the type A is dependent on a context Ψ, we write $(\Psi \vdash A)$ and this construction is itself a kind of type, called a *contextual type*. The context Δ consists of type declarations of the form $X : U$, where U is a contextual type.

Computational types come in three different forms. The first is a box-type $[U]$, where U is a contextual type. The second is a function type, $\tau_1 \to \tau_2$, where τ_1 and τ_2 are themselves computational types. Lastly, a computational type can be of the form $\Pi^{\square} X : U. \tau'$. (Because these are the only Π-types that concern us,

the square superscript will be omitted from now on.) This can be thought of as a universal quantification of the variable X over all contextual types U. The context Γ is a list of computational type-declarations.

2.2. Beluga

Given a type τ and a Beluga term t, one can make the judgement

$$\Delta;\ \Gamma \vdash t \Longleftarrow \tau \qquad \textit{In contexts } \Delta \textit{ and } \Gamma\textit{, Beluga term } t \textit{ checks against type } \tau.$$

to typecheck the term t against τ. If t is a synthesisable term (e.g. a variable, an application), its type can be looked up in the relevant context and the synthesis judgement

$$\Delta;\ \Gamma \vdash t \Longrightarrow \tau \qquad \textit{In contexts } \Delta \textit{ and } \Gamma\textit{, Beluga term } t \textit{ synthesises type } \tau.$$

can be made. We introduce various Beluga terms and the rules by which Beluga terms are judged to be well-typed. These are a subset of the rules found in [21].

First, we have `mlam`-expressions, which abstract over a metavariable:

$$\frac{\Delta, X : U;\ \Gamma \vdash t \Longleftarrow \tau}{\Delta;\ \Gamma \vdash \texttt{mlam}\ X \Rightarrow t \Longleftarrow \Pi X : U.\, \tau}\ (\textsc{bel-mlam})$$

The inference rule should be read as follows: If $t \Longleftarrow \tau$ in contexts $\Delta, X : U$ and Γ, then $\texttt{mlam}\ X \Rightarrow t \Longleftarrow \Pi X : U.\, \tau$ in contexts Δ and Γ. Rules will be presented in this format throughout this paper. The rule for `fn`-expressions, which abstract over an data-level variable, is very similar:

$$\frac{\Delta;\ \Gamma, x : \tau_1 \vdash t \Longleftarrow \tau_2}{\Delta;\ \Gamma \vdash \texttt{fn}\ x \Rightarrow t \Longleftarrow \tau_1 \to \tau_2}\ (\textsc{bel-fn})$$

To typecheck `case`-expressions, we assume the existence of a function coverage that returns a finite map $\mathcal{B}$ from constructor names to quadruples $(\Delta_i;\ \Gamma_i;\ \theta_i;\ p_i)$. Here the contexts Δ_i and Γ_i are newly generated and θ_i is a refinement substitution that moves a type τ from $\Delta;\ \Gamma$ to $\Delta_i;\ \Gamma_i$. The last element p_i of the tuple is the pattern that appears in the actual Beluga term.

$$\frac{\begin{array}{rr}\Delta;\ \Gamma \vdash t \Longrightarrow [U] & \mathsf{coverage}(\Delta \vdash U) = \mathcal{B} \\ \mathcal{B}(c_i) = (\Delta_i;\ \Gamma_i;\ \theta_i;\ p_i) & \Delta_i;\ \Gamma_i \vdash t_i \Longleftarrow [\![\theta_i]\!]\tau\end{array}}{\Delta;\ \Gamma \vdash \texttt{case}\ t\ \texttt{of}\ \overrightarrow{\texttt{|}\ p_i \Rightarrow t_i} \Longleftarrow \tau}\ (\textsc{bel-case})$$

Semantically, a `let`-expression is equivalent to a `case`-expression with only one case. It is convenient for our purposes, however, to establish a separate inference rule for it.

$$\frac{\Delta;\ \Gamma \vdash t \Longrightarrow \tau'' \qquad \Delta;\ \Gamma, x : \tau'' \vdash t' \Longleftarrow \tau}{\Delta;\ \Gamma \vdash \texttt{let}\ x = t\ \texttt{in}\ t' \Longleftarrow \tau}\ (\textsc{bel-let})$$

Likewise, we have the following typechecking rule for the unboxing of a variable within a `let`-expression.

$$\frac{\Delta;\ \Gamma \vdash t \Longrightarrow [\Psi \vdash A] \qquad \Delta, X : [\Psi \vdash A];\ \Gamma \vdash t' \Longleftarrow \tau}{\Delta;\ \Gamma \vdash \texttt{let}\ [\Psi \vdash X] = t\ \texttt{in}\ t' \Longleftarrow \tau}\ (\textsc{bel-letbox})$$

2.3. Harpoon

The proof language Harpoon consists of a small number of tactics. These are split into two mutually recursive syntactic categories: proof descriptions P and directives D:

i) A proof description P consists of a single directive or a tactic $\mathsf{By}\ I\ t\ \mathsf{as}\ x$; followed by another proof P'. Here the invocation I can be one of Lemma, IH, or Unboxing. The first two both invoke another theorem (IH indicates that the invocation is inductive) and Unboxing extracts a contextual type from a box-type and assigns it a metavariable name.

ii) To introduce all available assumptions, we use the directive $\mathsf{intros}\{\Delta;\ \Gamma \vdash P\}$.

iii) The Split directive splits a term into its cases: $\mathsf{Split}\ t\ \overrightarrow{\mathsf{Case}\ c_i\ H_i}$.

iv) A subgoal is solved by means of the directive $\mathsf{solve}\ t$.

The subset of Beluga and Harpoon's grammars that concerns us can be summarised as follows:

$$
\begin{array}{lll}
\text{LF Substitution} & \sigma & ::= \cdot \mid \sigma, M/x \\
\text{LF Context} & \Psi & ::= \cdot \mid \Psi, x\colon A \\
\text{LF Term} & M & ::= \lambda x\colon A.\, M \mid M_1\ M_2 \mid X[\sigma] \\
\text{LF Type} & A & ::= \Pi x\colon A.\, B \mid a\ \overrightarrow{M} \\
\text{LF Kind} & K & ::= \mathrm{type} \mid \Pi x\colon A.\, K \\
\text{Contextual Object} & C & ::= \Psi \mid (\Psi \vdash M) \\
\text{Contextual Type} & U & ::= \mathrm{ctx} \mid (\Psi \vdash A) \\
\text{Type} & \tau & ::= \tau_1 \to \tau_2 \mid \Pi^{\square} X\colon U.\, \tau \mid [U] \\
\text{Beluga Term} & t & ::= \mathtt{mlam}\ X \Rightarrow t \\
 & & \quad \mid \mathtt{fn}\ x \Rightarrow t \\
 & & \quad \mid \mathtt{case}\ x\ \mathtt{of}\ \overrightarrow{\mid\ p_i \Rightarrow t_i} \\
 & & \quad \mid \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 \\
 & & \quad \mid \mathtt{let}\ [\Psi \vdash X] = t_1\ \mathtt{in}\ t_2 \\
 & & \quad \mid t_1\ t_2 \mid t\ C \mid [C] \\
 & & \quad \mid \ldots \\
\text{Proof Desc.} & P & ::= D \mid \mathsf{By}\ I\ t\ \mathsf{as}\ x; P \\
\text{Invocation} & I & ::= \mathsf{Lemma} \mid \mathsf{IH} \mid \mathsf{Unboxing} \\
\text{Hypothetical} & H & ::= \{\Delta;\ \Gamma \vdash P\} \\
\text{Directives} & D & ::= \mathsf{intros}\ H \\
 & & \quad \mid \mathsf{Split}\ t\ \overrightarrow{\mathsf{Case}\ c_i\ H_i} \\
 & & \quad \mid \mathsf{solve}\ t
\end{array}
$$

3. TYPECHECKING AND TRANSLATING HARPOON PROOFS

We now investigate how Harpoon proof descriptions P and directives D can be typechecked and subsequently translated into well-typed Beluga terms t. First we must develop typing rules for Harpoon proofs, analogous to the ones given in Section 2.2 for Beluga terms. Then we can devise additional rules to formalise the translation step.

3.1 Typechecking

Harpoon proofs and directives are typed by the following judgements:

$\Delta;\ \Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau$ *In contexts* Δ *and* Γ, *Harpoon proof* P *checks against type* τ.

$\Delta;\ \Gamma \vdash_{\mathbf{D}} D \Longleftarrow \tau$ *In contexts* Δ *and* Γ, *Harpoon directive* D *checks against type* τ.

Typechecking proofs. These inference rules describe how proofs will be typechecked. The primary objective is to show that typechecking of Harpoon proofs is decidable. The broad strategy is to reduce the

problem of typechecking proofs to typechecking Beluga terms, since we already know this typechecking to be decidable.

When typechecking By, the case that the invocation I is an induction hypothesis or a lemma differs from the case in which I unboxes a term. If $I \in \{\mathsf{IH}, \mathsf{Lemma}\}$, we have

$$\frac{\Delta;\ \Gamma \vdash t \Longrightarrow \tau' \quad \Delta;\ \Gamma, x : \tau' \vdash_{\mathbf{P}} P \Longleftarrow \tau}{\Delta;\ \Gamma \vdash_{\mathbf{P}} \mathsf{By}\ I\ t\ \mathsf{as}\ x; P \Longleftarrow \tau}\ (\textsc{check-p-lemma})$$

So to verify that $\mathsf{By}\ I\ i\ \mathsf{as}\ x; P$ checks against type τ, we need to verify that t synthesises type τ' and that the proof P checks against type τ in contexts Δ and Γ extended with $x : \tau'$. If By invokes an unboxing of a Beluga term, then this rule changes slightly:

$$\frac{\Delta;\ \Gamma \vdash t \Longrightarrow [\Psi \vdash A] \quad \Delta, X : (\Psi \vdash A);\ \Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau}{\Delta;\ \Gamma \vdash_{\mathbf{P}} \mathsf{By}\ \mathsf{Unboxing}\ t\ \mathsf{as}\ X; P \Longleftarrow \tau}\ (\textsc{check-p-unbox})$$

Here we check that the term t synthesises a contextual type U. Then we may extend Δ with $X : U$ and show that the proof P checks against type τ. This shows that an $\mathsf{Unboxing}$ expression is well-typed.

The last case arises when the proof P is a Harpoon directive D.

$$\frac{\Delta;\ \Gamma \vdash_{\mathbf{D}} D \Longleftarrow \tau}{\Delta;\ \Gamma \vdash_{\mathbf{P}} D \Longleftarrow \tau}\ (\textsc{check-p-dir})$$

Here we defer to the typechecking rules for Harpoon directives, which will be given next.

Typechecking directives. The simplest directive is $\mathsf{solve}\ t$, which only requires that one typecheck the Beluga term t.

$$\frac{\Delta;\ \Gamma \vdash t \Longleftarrow \tau}{\Delta;\ \Gamma \vdash_{\mathbf{D}} \mathsf{solve}\, t \Longleftarrow \tau}\ (\textsc{check-d-solve})$$

To typecheck the intros directive, we describe a recursive function unroll that allows us to peel assumptions off a type one by one, until a box-type is found:

$$\begin{aligned}
&\mathsf{unroll}(\Delta;\ U) = U \\
&\mathsf{unroll}(\Delta;\ \Pi X : U.\, \tau) = \mathsf{unroll}(\Delta, X : U;\ \tau) \\
&\mathsf{unroll}(\Delta;\ \tau_1 \to \tau_2) = \mathsf{unroll}(\Delta;\ \tau_2)
\end{aligned}$$

Then the intros directive can be typechecked as follows:

$$\frac{\mathsf{unroll}(\Delta;\ \Gamma;\ \tau) = \tau' \quad \Delta';\ \Gamma' \vdash_{\mathbf{P}} P \Longleftarrow \tau'}{\Delta;\ \Gamma \vdash_{\mathbf{D}} \mathsf{intros}\{\Delta';\ \Gamma' \vdash_{\mathbf{P}} P\} \Longleftarrow \tau}\ (\textsc{check-d-intros})$$

The last case to consider is the one in which a term is split into its cases. Using the $\mathsf{coverage}$ function described in Section 2.2, we can typecheck the case of each constructor c_i by means of the following inference rule.

$$\frac{\Delta;\ \Gamma \vdash t \Longrightarrow [U] \quad \mathsf{coverage}(\Delta \vdash U) = \mathcal{B} \quad \mathcal{B}(c_i) = (\Delta_i;\ \Gamma_i;\ \theta_i;\ \_) \quad \Delta_i;\ \Gamma_i \vdash_{\mathbf{P}} P_i \Longleftarrow [\![\theta_i]\!]\tau}{\Delta;\ \Gamma \vdash_{\mathbf{D}} \mathsf{split}\, t\ \overrightarrow{\mathsf{Case}\, c_i\{\Delta_i;\ \Gamma_i \vdash_{\mathbf{P}} P_i\}} \Longleftarrow \tau}\ (\textsc{check-d-split})$$

The entire split expression typechecks if and only if this is satisfied for all i.

3.2 Translation

After the Harpoon proof has been typechecked, it may be translated into its corresponding Beluga program. The translation judgements are as follows:

$\Delta;\ \Gamma \vdash_{\mathbf{P}} P \longrightarrow t \Longleftarrow \tau$	*In contexts Δ and Γ, Harpoon proof P is translated into Beluga term t, checking against type τ.*

$\Delta;\ \Gamma \vdash_{\mathbf{D}} D \longrightarrow t \Longleftarrow \tau$ — *In contexts Δ and Γ, Harpoon directive D is translated into Beluga term t, checking against type τ.*

Translating proofs. Since the validity of induction hypotheses has already been verified during the typechecking phase, the translation of a By tactic only splits into two cases now. If the invocation $I \in \{\mathsf{IH}, \mathsf{Lemma}\}$, the translation rule looks like this:

$$\frac{\Delta;\ \Gamma \vdash t \Longrightarrow \tau'' \quad \Delta;\ \Gamma, x : \tau'' \vdash_{\mathbf{P}} P \longrightarrow t' \Longleftarrow \tau}{\Delta;\ \Gamma \vdash_{\mathbf{P}} \mathsf{By}\ I\ t\ \mathsf{as}\ x; P \longrightarrow \mathtt{let}\ x = t\ \mathtt{in}\ t' \Longleftarrow \tau}\ (\textsc{trans-p-lemma})$$

On the other hand, if an unboxing occurs, we have the rule

$$\frac{\Delta;\ \Gamma \vdash t \Longrightarrow [\Psi \vdash A] \quad \Delta, X : (\Psi \vdash A);\ \Gamma \vdash_{\mathbf{P}} P \longrightarrow t' \Longleftarrow \tau}{\Delta;\ \Gamma \vdash_{\mathbf{P}} \mathsf{By\ Unboxing}\ t\ \mathsf{as}\ X; P \longrightarrow \mathtt{let}\ [\Psi \vdash X] = t\ \mathtt{in}\ t' \Longleftarrow \tau}\ (\textsc{trans-p-unbox})$$

If the proof P is a directive D, we translate the directive:

$$\frac{\Delta;\ \Gamma \vdash_{\mathbf{D}} D \longrightarrow t \Longleftarrow \tau}{\Delta;\ \Gamma \vdash_{\mathbf{P}} D \longrightarrow t \Longleftarrow \tau}\ (\textsc{trans-p-dir})$$

Translating directives. The base case is a $\mathsf{solve}\, t$ expression. Since this directive has already been typechecked, it may simply be translated into the term t:

$$\frac{}{\Delta;\ \Gamma \vdash_{\mathbf{D}} \mathsf{solve}\, t \longrightarrow t \Longleftarrow \tau}\ (\textsc{trans-d-solve})$$

To give a translation rule for intros, we make use of the unroll function described previously. We also observe that if we have $\mathsf{unroll}(\Delta;\ \Gamma;\ \tau) = \tau'$ and $\Delta';\ \Gamma' \vdash_{\mathbf{P}} P \longrightarrow t' \Longleftarrow \tau'$, then τ and τ' have the same underlying box-type and we can easily produce a Beluga term t that checks against type τ. Algorithmically, this by done by adding the appropriate `mlam`- and `fn`-expressions to t'. These are prepended in the *opposite* order that that we peeled assumptions away from Δ and Γ to produce τ' in the first place. Then we see that t' checked against type τ' only if t checks against τ. This reasoning justifies the following rule:

$$\frac{\mathsf{unroll}(\Delta;\ \Gamma;\ \tau) = \tau' \quad \Delta';\ \Gamma' \vdash_{\mathbf{P}} P \longrightarrow t' \Longleftarrow \tau' \qquad \begin{array}{c}\overline{\Delta';\ \Gamma' \vdash t' \Longleftarrow \tau'} \\ \vdots \\ \Delta;\ \Gamma \vdash t \Longleftarrow \tau\end{array}}{\Delta;\ \Gamma \vdash_{\mathbf{D}} \mathsf{intros}\{\Delta';\ \Gamma' \vdash_{\mathbf{P}} P\} \longrightarrow t \Longleftarrow \tau}\ (\textsc{trans-d-intros})$$

Finally, we translate split expressions given the same conditions previously described in the corresponding typechecking rule. The same coverage function that provides the contexts Δ_i and Γ_i and the refinement substitution θ_i also provides the pattern p_i for the branch of the Beluga `case`-expression.

$$\frac{\begin{array}{rl}\Delta;\ \Gamma \vdash t \Longrightarrow [U] & \mathsf{coverage}(\Delta \vdash U) = \mathcal{B} \\ \mathcal{B}(c_i) = (\Delta_i;\ \Gamma_i;\ \theta_i;\ p_i) & \Delta_i;\ \Gamma_i \vdash_{\mathbf{P}} P_i \longrightarrow t_i \Longleftarrow [\![\theta_i]\!]\tau\end{array}}{\Delta;\ \Gamma \vdash_{\mathbf{D}} \mathsf{split}\, t\ \overrightarrow{\mathsf{Case}\, c_i\{\Delta_i;\ \Gamma_i \vdash_{\mathbf{P}} P_i\}} \longrightarrow \mathtt{case}\ t\ \mathtt{of}\ \overrightarrow{\mid\ p_i \Rightarrow t_i} \Longleftarrow \tau}\ (\textsc{trans-d-split})$$

4. SOUNDNESS OF TRANSLATION

The goal of this section is to prove that if a Harpoon proof script has been successfully typechecked, then the translation procedure only outputs Beluga programs that are well-typed. Concretely, we want to prove that in contexts Δ and Γ, if a Harpoon proof P is successfully typechecked against type τ and then translated into some Beluga term t, then the term t checks against type τ in the same contexts. The analogous statement for Harpoon directives will also be proved. Because the structures of Harpoon proofs and directives are mutually recursive, both statements must be proved simultaneously. Using the notation we established for judgements, the statement can be expressed as a theorem. The proof is by induction and examines each tactic separately.

Theorem T. *The following both hold:*

i) *If* $\Delta;\, \Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau$ *and* $\Delta;\, \Gamma \vdash_{\mathbf{P}} P \longrightarrow t \Longleftarrow \tau$, *then* $\Delta;\, \Gamma \vdash t \Longleftarrow \tau$.

ii) *If* $\Delta;\, \Gamma \vdash_{\mathbf{D}} D \Longleftarrow \tau$ *and* $\Delta;\, \Gamma \vdash_{\mathbf{D}} D \longrightarrow t \Longleftarrow \tau$, *then* $\Delta;\, \Gamma \vdash t \Longleftarrow \tau$.

Proof. First we prove statement (i). Let

$$\mathcal{D} ::= \Delta;\, \Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau$$
$$\mathcal{E} ::= \Delta;\, \Gamma \vdash_{\mathbf{P}} P \longrightarrow t \Longleftarrow \tau$$

We will do a case analysis on the three possibilities for P.

Case 1. If P invokes a By tactic on a lemma or induction hypothesis, then

$$\mathcal{D} ::= \Delta;\, \Gamma \vdash_{\mathbf{P}} \mathsf{By}\ I\ t\ \mathsf{as}\ x; P' \Longleftarrow \tau$$
$$\mathcal{E} ::= \Delta;\, \Gamma \vdash_{\mathbf{P}} \mathsf{By}\ I\ t\ \mathsf{as}\ x; P \longrightarrow \mathtt{let}\ x = t\ \mathtt{in}\ t' \Longleftarrow \tau$$

Then by inversion on the rules CHECK-P-LEMMA and TRANS-P-LEMMA, we have

$$\frac{\mathcal{D}_1 \qquad\qquad \mathcal{D}_2}{\Delta;\, \Gamma \vdash t \Longrightarrow \tau^* \quad \Delta;\, \Gamma, x : \tau^* \vdash_{\mathbf{P}} P' \Longleftarrow \tau}\ (\textsc{check-p-lemma})$$
$$\mathcal{D} ::= \Delta;\, \Gamma \vdash_{\mathbf{P}} \mathsf{By}\ I\ t\ \mathsf{as}\ x; P' \Longleftarrow \tau$$

and

$$\frac{\mathcal{E}_1 \qquad\qquad \mathcal{E}_2}{\Delta;\, \Gamma \vdash t \Longrightarrow \tau^{**} \quad \Delta;\, \Gamma, x : \tau^{**} \vdash_{\mathbf{P}} P' \longrightarrow t' \Longleftarrow \tau}\ (\textsc{trans-p-lemma})$$
$$\mathcal{E} ::= \Delta;\, \Gamma \vdash_{\mathbf{P}} \mathsf{By}\ I\ t\ \mathsf{as}\ x; P' \longrightarrow \mathtt{let}\ x = t\ \mathtt{in}\ t' \Longleftarrow \tau$$

Note that by uniqueness of synthesised types, $\tau^* = \tau^{**}$. So we can apply the induction hypothesis on $\mathcal{D}_2$ and $\mathcal{E}_2$ to get

$$\mathcal{F} ::= \Delta;\, \Gamma, x : \tau^* \vdash t' \Longleftarrow \tau$$

Lastly, we use $\mathcal{D}_1$ and $\mathcal{F}$ with the typing rule BEL-LET to show that the $\mathtt{let}$-expression is well-typed:

$$\frac{\mathcal{D}_1 \qquad\qquad \mathcal{F}}{\Delta;\, \Gamma \vdash t \Longrightarrow \tau^* \quad \Delta;\, \Gamma, x : \tau^* \vdash t' \Longleftarrow \tau}\ (\textsc{bel-let})$$
$$\Delta;\, \Gamma \vdash \mathtt{let}\ x = t\ \mathtt{in}\ t' \Longleftarrow \tau$$

Case 2. If P unboxes a term, then we have

$$\mathcal{D} ::= \Delta;\, \Gamma \vdash_{\mathbf{P}} \mathsf{By\ Unboxing}\ t\ \mathsf{as}\ X; P' \Longleftarrow \tau$$
$$\mathcal{E} ::= \Delta;\, \Gamma \vdash_{\mathbf{P}} \mathsf{By\ Unboxing}\ t\ \mathsf{as}\ X; P' \longrightarrow \mathtt{let}\ [\Psi \vdash X] = t\ \mathtt{in}\ t' \Longleftarrow \tau$$

and inversion of the rules CHECK-P-UNBOX and TRANS-P-UNBOX produces

$$\frac{\mathcal{D}_1 \qquad\qquad \mathcal{D}_2}{\Delta;\, \Gamma \vdash t \Longrightarrow [\Psi \vdash A] \quad \Delta, X : (\Psi \vdash A);\, \Gamma \vdash_{\mathbf{P}} P' \Longleftarrow \tau}\ (\textsc{check-p-unbox})$$
$$\mathcal{D} ::= \Delta;\, \Gamma \vdash_{\mathbf{P}} \mathsf{By\ Unboxing}\ t\ \mathsf{as}\ X; P' \Longleftarrow \tau$$

and

$$\frac{\mathcal{E}_1 \qquad\qquad \mathcal{E}_2}{\Delta;\, \Gamma \vdash t \Longrightarrow [\Psi \vdash A] \quad \Delta, X : (\Psi \vdash A);\, \Gamma \vdash_{\mathbf{P}} P \longrightarrow t' \Longleftarrow \tau}\ (\textsc{trans-p-unbox})$$
$$\mathcal{E} ::= \Delta;\, \Gamma \vdash_{\mathbf{P}} \mathsf{By\ Unboxing}\ t\ \mathsf{as}\ X; P' \longrightarrow \mathtt{let}\ [\Psi \vdash X] = t\ \mathtt{in}\ t' \Longleftarrow \tau$$

By means of the induction hypothesis on $\mathcal{D}_2$ and $\mathcal{E}_2$, we obtain

$$\mathcal{F} ::= \Delta, X : [\Psi \vdash A];\, \Gamma \vdash t' \Longleftarrow \tau,$$

which can be used in the rule BEL-LETBOX to get what we want:

$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{F} \\ \Delta;\, \Gamma \vdash t \Longrightarrow [\Psi \vdash A] & \Delta, X : [\Psi \vdash A];\, \Gamma \vdash t' \Longleftarrow \tau\end{array}}{\Delta;\, \Gamma \vdash \mathtt{let}\ [\Psi \vdash X] = t\ \mathtt{in}\ t' \Longleftarrow \tau}\ (\textsc{bel-letbox})$$

Case 3. When P is a directive D, we invert the rules CHECK-P-DIR and TRANS-P-DIR to get

$$\mathcal{D}' ::= \Delta;\, \Gamma \vdash_{\mathbf{D}} D \Longleftarrow \tau$$
$$\mathcal{E}' ::= \Delta;\, \Gamma \vdash_{\mathbf{D}} D \longrightarrow t \Longleftarrow \tau$$

and appeal to the induction hypothesis with $\mathcal{D}'$ and $\mathcal{E}'$.

So much for translating proofs. Now we prove statement (ii) concerning directives. Let

$$\mathcal{D} ::= \Delta;\, \Gamma \vdash_{\mathbf{D}} D \Longleftarrow \tau$$
$$\mathcal{E} ::= \Delta;\, \Gamma \vdash_{\mathbf{D}} D \longrightarrow t \Longleftarrow \tau$$

We will do a case analysis on the three possibilities for D.

Case 1. The base case of the induction is when D is a solve directive. In this case,

$$\mathcal{D} ::= \Delta;\, \Gamma \vdash_{\mathbf{D}} \mathsf{solve}\, t \Longleftarrow \tau$$

and we can invert the rule CHECK-D-SOLVE to get $\Delta;\, \Gamma \vdash t \Longleftarrow \tau$, exactly what was to be shown.

Case 2. If D is an intros directive, we have

$$\mathcal{D} ::= \Delta;\, \Gamma \vdash_{\mathbf{D}} \mathsf{intros}\{\Delta';\, \Gamma' \vdash_{\mathbf{P}} P\} \Longleftarrow \tau$$
$$\mathcal{E} ::= \Delta;\, \Gamma \vdash_{\mathbf{D}} \mathsf{intros}\{\Delta';\, \Gamma' \vdash_{\mathbf{P}} P\} \longrightarrow t \Longleftarrow \tau$$

and we invert on the rules CHECK-D-INTROS and TRANS-D-INTROS to obtain

$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ \mathsf{unroll}(\Delta;\, \Gamma;\, \tau) = \tau' & \Delta';\, \Gamma' \vdash_{\mathbf{P}} P' \Longleftarrow \tau'\end{array}}{\Delta;\, \Gamma \vdash_{\mathbf{D}} \mathsf{intros}\{\Delta';\, \Gamma' \vdash_{\mathbf{P}} P'\} \Longleftarrow \tau}\ (\textsc{check-d-intros})$$

and

$$\frac{\begin{array}{ccc}\mathcal{E}_1 & \mathcal{E}_2 & \dfrac{\mathcal{E}_3}{\Delta';\, \Gamma' \vdash t' \Longleftarrow \tau'} \\ \mathsf{unroll}(\Delta;\, \Gamma;\, \tau) = \tau' & \Delta';\, \Gamma' \vdash_{\mathbf{P}} P' \longrightarrow t' \Longleftarrow \tau' & \Delta;\, \Gamma \vdash t \Longleftarrow \tau\end{array}}{\Delta;\, \Gamma \vdash_{\mathbf{D}} \mathsf{intros}\{\Delta';\, \Gamma' \vdash_{\mathbf{P}} P'\} \longrightarrow t \Longleftarrow \tau}\ (\textsc{trans-d-intros})$$

(The triples returned under $\mathcal{D}_1$ and $\mathcal{E}_1$ are identical because the unrolling operation is well-defined and deterministic.) Now we appeal to the induction hypothesis on $\mathcal{D}_2$ and $\mathcal{E}_2$ to get

$$\mathcal{F} ::= \Delta';\, \Gamma' \vdash t' \Longleftarrow \tau'$$

Now with $\mathcal{E}_3$ and $\mathcal{F}$, we can perform a substitution to get $\Delta;\, \Gamma \vdash t \Longleftarrow \tau$.

Case 3. The final case is when D is split directive. Then we have

$$\mathcal{D} ::= \Delta;\, \Gamma \vdash_{\mathbf{D}} \mathsf{split}\, t\ \overrightarrow{\mathsf{Case}\, c_i\{\Delta_i;\, \Gamma_i \vdash_{\mathbf{P}} P_i\}} \Longleftarrow \tau$$
$$\mathcal{E} ::= \Delta;\, \Gamma \vdash_{\mathbf{D}} \mathsf{split}\, t\ \overrightarrow{\mathsf{Case}\, c_i\{\Delta_i;\, \Gamma_i \vdash_{\mathbf{P}} P_i\}} \longrightarrow \mathtt{case}\ t\ \mathtt{of}\ \overrightarrow{\mid p_i \Rightarrow t_i} \Longleftarrow \tau$$

Then inversion on the rules CHECK-D-SPLIT and TRANS-D-SPLIT gives

$$\frac{\begin{array}{rr}\mathcal{D}_1 ::= \Delta;\ \Gamma \vdash t \Longrightarrow [U] & \mathcal{D}_2 ::= \mathsf{coverage}(\Delta \vdash U) = \mathcal{B} \\ \mathcal{D}_3 ::= \mathcal{B}(c_i) = (\Delta_i;\ \Gamma_i;\ \theta_i;\ p_i) & \mathcal{D}_4 ::= \Delta_i;\ \Gamma_i \vdash_{\mathbf{P}} P_i \Longleftarrow [\![\theta_i]\!]\tau\end{array}}{\mathcal{D} ::= \Delta;\ \Gamma \vdash_{\mathbf{D}} \mathsf{split}\, t\ \overrightarrow{\mathsf{Case}\, c_i\{\Delta_i;\ \Gamma_i \vdash_{\mathbf{P}} P_i\}} \Longleftarrow \tau}\ (\textsc{check-d-split})$$

and

$$\frac{\begin{array}{rr}\mathcal{E}_1 ::= \Delta;\ \Gamma \vdash t \Longrightarrow [U] & \mathcal{E}_2 ::= \mathsf{coverage}(\Delta \vdash U) = \mathcal{B} \\ \mathcal{E}_3 ::= \mathcal{B}(c_i) = (\Delta_i;\ \Gamma_i;\ \theta_i;\ p_i) & \mathcal{E}_4 ::= \Delta_i;\ \Gamma_i \vdash_{\mathbf{P}} P_i \longrightarrow t_i \Longleftarrow [\![\theta_i]\!]\tau\end{array}}{\mathcal{E} ::= \Delta;\ \Gamma \vdash_{\mathbf{D}} \mathsf{split}\, t\ \overrightarrow{\mathsf{Case}\, c_i\{\Delta_i;\ \Gamma_i \vdash_{\mathbf{P}} P_i\}} \longrightarrow \mathtt{case}\ t\ \mathtt{of}\ \overrightarrow{\mid p_i \Rightarrow t_i} \Longleftarrow \tau}\ (\textsc{trans-d-split})$$

We apply the inductive hypothesis on $\mathcal{D}_4$ and $\mathcal{E}_4$, obtaining

$$\mathcal{F} ::= \Delta_i;\ \Gamma_i \vdash t_i \Longleftarrow [\![\theta_i]\!]\tau$$

This is the last piece we need to prove that the Beluga `case`-expression is well-typed.

$$\frac{\begin{array}{rr}\mathcal{D}_1 ::= \Delta;\ \Gamma \vdash t \Longrightarrow [U] & \mathcal{D}_2 ::= \mathsf{coverage}(\Delta \vdash U) = \mathcal{B} \\ \mathcal{D}_3 ::= \mathcal{B}(c_i) = (\Delta_i;\ \Gamma_i;\ \theta_i;\ p_i) & \mathcal{F} ::= \Delta_i;\ \Gamma_i \vdash t_i \Longleftarrow [\![\theta_i]\!]\tau\end{array}}{\Delta;\ \Gamma \vdash \mathtt{case}\ t\ \mathtt{of}\ \overrightarrow{\mid p_i \Rightarrow t_i} \Longleftarrow \tau}\ (\textsc{bel-case})$$

In all three cases, the Harpoon directive is translated into a well-typed Beluga term. ∎

5. RELATED WORK

History. The first computer languages designed to express mathematical proofs in a machine-verifiable fashion were created during the Automath project, started in 1967 by N. G. de Bruijn [3]. The project culminated in the complete translation and verification of E. Landau's classic text *Grundlagen der Analysis* in 1976 [2]. This required the formalisation of the classical logic used in conventional mathematics. A generalisation arose in 1988 in the form of the Edinburgh Logical Framework (LF), which provides a powerful means of formally presenting arbitrary logical systems [10]. In 1999, LF was implemented in the form of the Twelf logic programming language [20], which uses higher-order abstract syntax (HOAS) to represent name-binding in an object language [20]. Beluga extends the techniques used in Twelf with a type-system based on Contextual Modal Type Theory [16], which allows the direct manipulation of open terms by treating contexts as first-class objects [22].

Naturally, proof assistants have been useful to mathematical researchers as well. In 1976, K. Appel and W. Haken famously proved the four-colour theorem of graph theory by means of an exhaustive computerised analysis of hundreds of different cases [1]. This method was somewhat controversial, because the computer produced a proof in a long and technical format. Complete inspection of such a proof by a human mathematician was infeasible; one needed to have complete faith in the program that constructed the proof. The essence of the dispute hinged on the question of whether empirical testing of a computer program is equivalent to a mathematical proof. Judging by the sheer amount of bugs that routinely riddle industry software, we know that the answer is a definitive no. More recently however, the four-colour theorem has been reformalised and proven using Coq [8], and because the metatheory of Coq is trusted, this program is a far less controversial certificate of the theorem's correctness.

One does not normally think of mathematics as a discipline that lacks rigour and formality; however, proving mathematical concepts in a proof language requires more care and precision than proving them on paper. For example, it took a large group of researchers six years to formalise the Feit-Thompson Odd Order Theorem of group theory [9]. The reward is a fully-verified proof that can be used to prove further results in the future. Mathematical progress is highly incremental and new theorems are often built upon

dozens of prior ones. This suggests that the field would greatly benefit from a large-scale, mechanically-proven database of all known mathematical truths. Indeed, this is what the anonymously-authored "QED Manifesto" proposed in 1994 [24].

Tactic languages. During the 1970s, the LCF (Logic for Comuputable Functions) automated theorem prover was developed at Stanford University. Its implementation enabled users to build proofs step-by-step using a language of commands [15]. In addition, a metalanguage called ML could be used to write tactics for the system. This language was highly influential and many modern functional programming languages take inspiration from ideas first seen in ML.

The proof assistant Coq began life at INRIA in 1984 as an implementation of the Calculus of Constructions by T. Coquand and G. Huet [4]. An extensive description of Coq's modern tactic language is given by D. Delahaye in [6]. Just as in Harpoon, there is a duality between theorems proven in Coq and dependently-typed functional programs. The flexibility of proof assistants like Coq allows researchers to investigate the validity of mathematical statements in different logical frameworks. As just one example of this, a Coq program due to L. Cruz-Filipe proved all the theorems of elementary real analysis up to and including the Fundamental Theorem of Calculus constructively, i.e. in a logic where proof by contradiction is not allowed [5].

Harpoon is strictly simpler than the tactic language of Coq, because it does not allow users to define their own tactics. Instead, the tactics that Harpoon supports are the ones that come up most often in the development of proofs in Beluga. The fixed number of tactics makes the correctness of the Harpoon system significantly easier to prove.

6. CONCLUSION

The contributions described in this paper form a layer of verification in the development of proofs in Beluga's logical framework. Harpoon's proof script format provides a way to save completed proofs for subsequent use, and the typechecking algorithm ensures these scripts are a robust representation of an interactively-constructed proof. With the addition of the translation procedure, Harpoon provides an alternative method to write Beluga programs.

Programs are usually written in the form of plain text. However, for a large portion of the time it takes to write a program, the partial source text is not syntactically well-formed and thus meaningless to the computer. Various methods have been devised to divorce the construction of programs from the text that represents it. For example, to write programs in Scratch, a language designed to teach children the foundations of programming, users drag and drop atomic statements and expressions in user-friendly interface that illustrates the constraints of the language's abstract syntax [13]. In a stronger vein, there are programs known as structure editors that allow programmers to directly build the abstract syntax tree of their programs. An example is Hazelnut, which is based on a bidirectionally-typed lambda calculus and allows editing over statically-meaningful incomplete terms. Hazelnut's metatheory has been formalised with help from the Agda proof assistant [17].

The formal typechecking and translation of Harpoon proofs closes the gap between proof scripts and functional programs. These pieces allow Harpoon to be more than just a read-eval-print loop; it can become a vital tool in the development of Beluga programs.

BIBLIOGRAPHY

[1] K. Appel, W. Haken, and J. Koch, "Every planar map is four colourable. Part 1: Discharging," *Illinois J. Math.* **21** (1977), 429–490.

[2] L. S. van Benthem Jutting, "Checking Landau's 'Grundlagen' in the Automath System," *Mathematisch Centrum* **83**. Amsterdam, 1979.

[3] N. G. de Bruijn, "A Survey of the Project Automath," *Studies in Logic and the Foundations of Mathematics* **133** (1994), 141–161.

[4] T. Coquand and G. Huet, "The Calculus of Constructions," *Information and Computation* **76** (1988), 95–120.

[5] L. Cruz-Filipe, "A Constructive Formalization of the Fundamental Theorem of Calculus," *Proceedings of the 2002 International Conference on Types for Proofs and Programs* (2002), 108–126.

[6] D. Delahaye, "A Tactic Language for the System Coq," *Proceedings of Logic for Programming and Automated Reasoning (Lecture Notes in Computer Science)* (2000), 85–95.

[7] E. Dijkstra, "Notes on Structured Programming (EWD249)" (1970), 7.

[8] G. Gonthier, "Formal proof—The Four Color Theorem," *Notices of the American Mathematical Society* **55** (2008), 1382–1393.

[9] G. Gonthier et al., "A Machine-Checked Proof of the Odd Order Theorem," *Proceedings of the 4th International Conference on Interactive Theorem Proving* (2013), 163–179.

[10] R. Harper, F. Honsell, and G. Plotkin, "A Framework for Defining Logics," *Journal of the Association for Computing Machinery* **40**,1 (1993), 143–184.

[11] L. Josephs and K. Rooney, "The FAA set to sign off on Boeing 737 Max software fix in 10 days, shares rise," CNBC, March 15, 2019.

[12] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM* **52** (2009), 107–115.

[13] J. Maloney, M. Resnick, N. Rusk, B. Silverman, E. Eastmond, "The Scratch Programming Language and Environment," *ACM Transactions on Computing Education* **10**,4 (2010), Article No. 16.

[14] S. McConnell, *Code Complete*. Microsoft Press, 2004.

[15] R. Milner, "LCF: A Way of Doing Proofs with a Machine," *Mathematical Foundations of Computer Science (Lecture Notes in Computer Science)* **74** (1979), 146–159.

[16] A. Nanevski, F. Pfenning, and B. Pientka, "Contextual Modal Type Theory," *ACM Transactions on Computational Logic*, **9**,3 (2008), 1–49.

[17] C. Omar, I. Voysey, M. Hilton, J. Aldritch, and M. Hammer, "Hazelnut: A Bidirectionally Typed Structure Editor Calculus," *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*, (2017), 86–99.

[18] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in Linux: Ten Years Later," *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011), 305–318.

[19] F. Pfenning, C. Elliott, "Higher Order Abstract Syntax," *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (1988), 199–208.

[20] F. Pfenning, C. Schürmann, "System Description: Twelf — A Meta-Logical Framework for Deductive Systems," *16th Inteernational Conference on Automated Deduction (CADE-16)* (1999), 202–206.

[21] B. Pientka, "A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions," *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)* (2008), 371–382.

[22] B. Pientka and J. Dunfield, "Programming with Proofs and Explicit Contexts," *10th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '08)* (2008), 163–173.

[23] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and Understanding Bugs in C Compilers,", *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)* (2011), 283–294.

[24] "The QED Manifesto," *12th International Conference on Automated Deduction (CADE-12)* (1994), 238–251.