Mechanizing Metatheory Interactively

Jacob Thomas Errington

Submitted in partial fulfillment of the requirements for the degree of
Master of Science

School of Computer Science
McGill University
Montreal, Quebec, Canada

Abstract

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. Our implementation is a part of Beluga on GitHub at https://github.com/Beluga-lang/Beluga and the reference manual is available at https://beluga-lang.readthedocs.io/.

Résumé

Beluga est un assistant de preuves basé sur le cadre logique (logical framework) LF et donnant une infrastructure quant à la représentation de systèmes formels et de preuves concernant ceux-ci. Par conséquence, ces preuves metathéoriques sont précises et compactes. Cependant, on développe une preuve avec Beluga en tant que programme récursif total, ce qui est difficile et encombrant.

Nous présentons alors la conception de Harpoon, une extension à Beluga, avec lequel on fait le développement interactif de preuves. L'utilisateur de Harpoon construit une preuve en utilisant un petit ensemble fermé *d'actions*. L'exécution de celles-ci élabore un *texte de preuve* (proof script) qui reflète la structure des sous-objectifs qui se présentent dans la preuve. Nous modélisons les preuves incomplètes en utilisant des variables contextuelles pour représenter les sous-objectifs qui restent à résoudre. Nous établissons de plus une manière de traduire les textes de preuve en programmes Beluga traditionels, ce qui permet l'exécution de ces preuves. Les textes de preuve et les programmes Beluga intéragissent facilement, donc l'utilisateur peut choisir quel méthode de preuve lui convient le mieux.

Nous avons évalué Harpoon sur de nombreux exemples, en passant par des théorèmes de préservation de typage et de progrès pour MiniML ainsi que des théorèmes de normalisation tels que le récent défi POPLMark Reloaded.

Le code que nous avons écrit fait maintenant partie de Beluga sur GitHub à https://github.com/Beluga-lang/Beluga et le manuel de référence est disposible à https://beluga-lang.readthedocs.io/.

Contents

Preface

Ackowledgements

Although a thesis is something one writes primarily alone, it documents work that would be impossible without the large support structure that comes with a research group in a university. In my case, I am truly grateful to Complogic group at McGill, and I am thankful that so many of its members have helped me through the years of work poured into my research.

I want to especially acknowledge David Thibodeau, desk neighbour and colleague extraordinaire. His insight in resolving both practical implementation concerns with Harpoon as well as more theoretical issues was invaluable. I very much appreciated his persistent readiness to hear out my crazy ideas and to offer his own crazy ideas in return. I look back fondly on the summer reading group we put together for Homotopy Type Theory and the many times we got sidetracked messing around with Agda. I will consider myself very lucky should I ever have an officemate so equally both fun and intellectually stimulating as him.

Next I want to thank Clare Jang and Marcel Goh for their help in more directly developing Harpoon. Clare took charge of several tricky implementation tasks such as Harpoon's ability to overwrite existing holes in proof scripts with new proof fragments. Marcel helped to develop the theory for translation from proof scripts to Beluga programs as part of an undergraduate research course, and he wrote the preliminary code for typechecking and

translation proof scripts.

Of course, I want to recognize next the tremendous support of my advisor, Brigitte Pientka. I vividly recall still to this day a time when I told a friend in first year that I would be so happy to do a summer research project with Brigitte. Little did I know then that I would find her so engaging and stimulating that I would do an entire master's degree under her supervision. Beluga is a quite the large project, so it is entirely due to her guidance that I was able to understand it so deeply and to make improvements to it. Her careful attention to detail and her principled approach to programming language theory taught me a lot about how one should conduct research. I admire her involvement in the research community, and I aspire to myself be so involved in the communities I should find myself in.

Finally, I want to thank my partner Eric. I especially want to thank him for being so understanding of the time commitments that graduate school involves. I reminisce often about our walks, talking about logic, proofs, and theory. Few people outside this specific research community have much interest in what I do, so it was crucial for me to have an outsider to bounce ideas of off and to keep me sane and grounded.

Contribution of authors

The main technical exposition in Chap. 4 is primarily my own original work. Prof. Pientka gave me a lot of guidance early on for ideas on how to define the structured proof script language and she helped me to develop its type system. She also guided me in the implementation as I brushed up against several complex features of Beluga.

I developed the formalization of the interactive actions themselves, in Sec. 4.3 which led me to extend the syntax of proof scripts with subgoal variables. I then developed the notion of subgoal context. I noticed that these subgoal variables were unusual in that they are checkable expressions,

and that the upshot of this is that the subgoal context is understood as an output of the typing judgment. The statements of all the theorems and their proofs were worked out by me.

Marcel Goh helped to develop the translation presented in Sec. 4.4 and wrote some preliminary code for typechecking and translating proof scripts. I was the primary implementor of the code, with some help from Clare Jang who implemented the system of automatic tactics and the code for overwriting proof scripts with their more refined forms. Clare also was responsible for porting a few of the examples listed in Sec. 5.1 and Marcel wrote a few of the smaller tests for Harpoon.

The running example discussed in Chap. 3 was designed by Prof. Pientka and me, and I owe the overall structure of that chapter to her. I also owe it to her for how to properly motivate this work in Chap. 1.

The literature review given in Chap. 2 and the comparative analysis of other systems in Sec. 5.2 is entirely my own work.

Funding

The research conducted during this degree was funded by the Fonds de Recherche du Québec, Nature et Technologies (FRQNT) scholarship B1X #258099. I also want to recognize supplemental funding by my advisor, Brigitte Pientka.

Chapter 1

Introduction

Properties upheld by a formal system such as a logic or a programming language are called its *metatheory*. Developing the metatheory of formal systems such as logics and programming languages is central to establishing trust in those systems. Such trust is essential since quite sophisticated programming languages, having huge numbers of features, are used to create the vast bodies of software that surround us in the modern world. For programmers to have any hope of creating software free of bugs, they must rely on guarantees made to them by the programming language they use. If that language makes safety promises that turn out to be broken, then code believed be bug-free may in fact contain bugs. Even worse, these bugs may lead to security vulnerabilities in applications.

Sadly, as a programming language becomes more complex, so does its metatheory. Developing a proof on paper, although an excellent strategy for gaining an intuition for the proof's correctness, does not provide bulletproof assurance of its correctness. Instead, one can seek a higher standard of correctness by *mechanizing* the metatheory. That is, one can verify a proof using software called a proof assistant.

Alas, using a proof assistant is not as simple as one might hope. One must first *encode* the system of study in the assistant. Next, one must formulate the

statements of the theorems and their proofs in a way that the assistant can automatically verify. How one goes about resolving both of these concerns depends greatly on the chosen proof assistant.

Proof assistants come in many shapes and flavours, as building such an assistant requires that one make several impactful choices. What underlying theory will it use? How will a user interact with it? What sorts of proofs is it specialized for, if any? When mechanizing metatheory, a key question is: how to represent variables, (simultaneous) substitutions, assumptions, derivations that depend on assumptions, and the proof state? These decisions greatly influence how easy (or how cumbersome) it might be to encode various formal systems and to reason about them.

Beluga [41, 39] is a proof assistant which provides sophisticated infrastructure for implementing formal systems based on the logical framework LF [22]. 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 for managing 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. Given that the proof is represented as a program, proof checking amounts to type-checking the program. Beluga hence follows in the foot steps of proof checkers such as Automath [30], Agda [31], and specifically Twelf [35].

While writing a proof as a dependently typed program is a beautiful idea, it can be quite challenging and cumbersome. In a dependently typed language, the values of terms can influence the types of other terms. Keeping track mentally of this rich type information is nothing short of a tall order. This limits the widespread use of dependently typed programming languages for mechanizing proofs in general. Hence, many proof assistants in this domain provide some form of interactivity: for example, Agda [31] supports

leaving holes (question marks) and writing partial programs which can later be refined using a fixed limited set of interactions. These holes crucially allow the user to inspect the types of all in-scope identifiers, which relieves them of the mental tax of dependent types. However a clear specification and theoretical foundation of how these interactions transform programs is largely missing. In Coq [4] users interactively develop a proof using *tactics*. Behind the scenes, a sequence of tactic applications is elaborated into a dependently typed program. 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 [13] are mostly unconstrained; it is Coq's typechecker that verifies post hoc that the program generated by the tactics is valid. The later Mtac system [24] makes a marked improvement by introducing a static type system for tactics. Although a tactic is guaranteed to produce a term of a known type (if it terminates), the tactic type says nothing of the *context* in which the generated term is meaningful. A common additional caveat of tactic languages, statically typed or otherwise, is that often, the resulting proof script is brittle and unreadable.

This thesis presents the design and implementation of Harpoon, an interactive proof environment built on top of Beluga, where programmers develop proofs using a fixed set of tactics. The user invokes an action on a subgoal in order to eliminate it, possibly introducing new subgoals in doing so. Our fixed set of tactics is largely inspired by similar systems such as Abella [17] and Coq, 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, a tactic can also refer to a Beluga program 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 appli-

cation 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 reflects 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 thesis 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 meta-theoretic 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 demonstrate Harpoon to interactively develop a proof in Chap. 3 by way of giving a whirlwind tour of the main supported tactics in Harpoon.

- In Chap. 4, we describe a logical foundation for interactive proof development. To that end, we explain first in Sec. 4.2 a proof script language that reflects the proof structure laid out by the user. This language clearly separates forwards and backwards reasoning. Then, we explain formally the interactive tactics and their connection to proof scripts. We prove soundness of interactive proof construction, and give a translation from proof scripts into Beluga programs, showing that this translation is type-preserving. This justifies that proof scripts indeed represent proofs and form a valid way to develop metatheory, and also allows proof scripts to not only be typechecked, but also executed as programs.

- We characterize and reason about incomplete programs using *contextual types*. A variable of such a type, which we call a *subgoal variable*, 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 to successively applying contextual substitutions to eliminate a subgoal variable, while possibly introducing new ones.

- Harpoon is implemented as part of Beluga and is documented together with Beluga at https://beluga-lang.readthedocs.io/. We have used Harpoon 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 [8], benchmarks for reasoning about binders [15, 16], 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, substitutions, and variables.

Chapter 2

Background

Before presenting my work, I review in this chapter some of the foundations upon which it rests. First, I broadly discuss tactic languages in Sec. 2.1 before moving on to languages used for defining tactics in Sec. 2.2. Next, since Harpoon can be viewed also as a form of structured editor for proofs, I survey the literature on structured editing in Sec. 2.3. Finally, I discuss the Beluga project at large since Harpoon builds directly on Beluga.

2.1 Tactics and tacticals

2.1.1 Origins: the LCF system

Now a common word in today's proof assistant jargon, the notion of *tactic* as we know it was introduced by Robin Milner nearly 40 years ago as part of the Logic for Computable Functions (LCF) system [28, 21]. This system is specialized for reasoning in a logic called PPLAMBDA, a polymorphic predicate lambda-calculus, based on Dana Scott's Logic *of* Computable Functions [44]. Although the way we use tactics has evolved somewhat since then, the core idea remains remarkably the same: a tactic is a function applied to a goal to eliminate it, producing zero or more new goals. This view of tactics is

natural if one understands an inference rule as a function from theorems to theorems: a primitive tactic is merely the inverse of an inference rule, mapping the conclusion of that rule to its necessary premises. As not every tactic is applicable to every goal, Milner defines a tactic specifically as a partial function. A tactic produces also as output a function that Milner calls a *validation*. This function accepts a list of theorems – the eventual solutions to the new goals generated by the tactic – and produces a new theorem. This is due to the crucially backwards orientation of the tactics-based approach: tactics act on a *goal* (a desired end state), so validations are composed together once the proof is complete to form a forwards proof.

For example, a tactic representing the rule for $\wedge$-introduction applied to a goal of the form $A \wedge B$ generates two subgoals, A and B. The validation generated by the tactic expects a list of two theorems, one for A and one for B, and constructs the theorem $A \wedge B$ using the function representing the $\wedge$-introduction rule, of type $\texttt{theorem} \rightarrow \texttt{theorem} \rightarrow \texttt{theorem}$.

The core tactics in LCF are simply the inference rules of PPLAMBDA, but having only these would be quite limiting. Constructing even small proofs would be a hugely laborious task in that case. To make the tactic-based approach to proving more practical, Milner introduces tactic combinators called *tacticals*. These can take advantage of tactics' failure for backtracking and repetition. As an example of the former, the tactic t_1 Orelse t_2, constructed using the Orelse tactical, executes t_1 and if it fails, then t_2. As an example of the latter, consider the tactic Repeat t. It executes the tactic t: on failure, nothing happens; on success, Repeat t is executed on every subgoal generated by the successful application of t.

A strength of the LCF system is that users may define their own tactics, but this comes also at a cost. Users must also therefore define the validation associated to their tactic. Granting totally unrestricted power to the user in constructing theorems would be a huge blow to any potential soundness guarantee for the system. Instead, only limited power is given to the user,

as Milner says that "the only operations for generating [theorems] are the basic inference rules ... and the rules derived from them." But this isn't quite enough either. Nothing (statically) prevents a user from defining what Milner calls an "invalid" tactic. He writes in [21],

> Validity is clearly a necessary condition for a tactic to be useful; indeed we may deny that invalid tactics are tactics at all. But it is hard to see how to design a programming language so that all definable objects of type `tactic` are valid, or how to gain this effect by a type discipline. At most we can adopt a style which encourages the programming of valid tactics; this can be done with tacticals.

Milner shows on paper that given valid tactics, his predefined tacticals generate valid tactics. But to reiterate, the overall soundness of the system is undermined by the user's ability to define and use invalid tactics and tacticals.

2.1.2 Tactics in Isabelle

In the early 80's, Lawrence Paulson assists Milner in the development of the LCF system in Edinburgh. Once Paulson's time in Scotland ends, he travels south to England, taking with him the insights from the LCF system to create Cambridge LCF, which goes on to become the Isabelle-86 and later the Isabelle-88 system [33].

Both Isabelle-88 and Edinburgh LCF use ML as a form of interactivity with the user. That is, theorems can be proven interactively purely via the ML read-eval-print loop (REPL). The main downside to this approach is that anything is possible: this REPL allows general-purpose programming in ML, of which one potential application is the development of proofs. In other words, Isabelle-88 is an ML library rather than an application per se. The functions representing tactics must be applied manually to values rep-

resenting goals to build new goal values. These entities must all be managed directly by the user.

To alleviate some of this manual labour, a helper library called the *goal stack package* is bundled together with Isabelle-88 that automates much of the goal management. It provides a notion of a current goal state, an undo mechanism, and a helper for invoking a tactic on the current state. This helper adds any new goals generated by the tactic to the goal stack.

Isabelle-88 builds substantially on Edinburgh LCF. First, whereas LCF is specialized for reasoning about a particular logic called PPLAMBDA, Isabelle-88 aims to be a *generic* theorem prover, capable of reasoning about several logics that one would encode in it[1]. Second, as for the tactic languages, one key development in Isabelle-88 is the removal of validations, those functions used to construct a forwards proof once the tactic-based backwards proof is complete. Recall that these validations are a source of concern in Edinburgh LCF as users can define invalid tactics, whose validations construct a proof for the the wrong theorem. A shift in the representation technique for inference rules is what enables explicit, manual validations to be eliminated. Rather than represent inference rules as opaque functions, they are represented directly as data in the system. From this representation, both forwards reasoning functions as in LCF and backwards reasoning tactics can be derived. Moreover, the application of a tactic automatically derived from an inference rule performs the work of LCF's validations behind the scenes as what Paulson calls a "meta-inference". Since a new object-logic is encoded by extending Isabelle-88's meta-logic $\mathcal{M}$ with additional axioms representing as (meta-)implications the inference rules of the object-logic, reasoning in the metalanguage $\mathcal{M}$ amounts to reasoning in the object-logic. In fact, an entire intermediate *proof state* for a theorem in an object language is represented as a *theorem* in $\mathcal{M}$. Hence, it is by checking inferences *in* $\mathcal{M}$ that the system

[1]A similar drive to represent represent and reason about a broader class of logics also takes place in Edinburgh, culminating with the development of the Edinburgh logical framework LF [22].

ensures that every proof state is arrived at correctly. Users may define new tactics, but these, by virtue of ultimately using the primitive tactics representing inference rules, effectively represent *derived* inference rules in the object language.

Although the underlying details of tactics are quite different, the tactics and tacticals themselves are mostly the same, and remain quite powerful. Tacticals for repetition, backtracking, and so on are all present in Isabelle-88. Paulson demonstrates the power of tacticals by using them to build a proof search strategy for classical first-order logic. Paulson's strategy is to divide inference rules into two categories that he calls "safe" and "unsafe". A rule that can be applied eagerly, without affecting the provability of the overall statement, is safe; else it is unsafe. Then, one applies as many safe rules as possible until applying one unsafe rule, and repeating.

Now it seems that there are two different views of tactics. Some tactics represent (derived) inference rules in the object-logic, whereas others represent proof search and automation techniques. It is not immediately clear that these two views can be reconciled. Inference rules have clear conditions under which they may apply, and their outcomes are predictable, whereas proof search techniques have less clear applicability conditions and their outcomes can be uncertain. These two different views were already present in LCF, and we will see that these two views appear in later systems, too.

2.1.3 Tactics in NuPRL

While Europe is certainly abuzz with research into proof assistants, so is Cornell University in New York. There, Robert Constable and his collaborators develop the NuPRL system [11]. The tactics and surface-level architecture of NuPRL resemble Isabelle-88's: the system is implemented in ML, and tactics are created by writing them in ML. Again as in Isabelle, the tactics are safe in the sense that invalid proofs cannot be produced by a tactic, so they similarly do not suffer from that shortcoming of the original LCF tactics.

The distinguishing feature of NuPRL is that it is an *extensional* type system, unlike Isabelle-88 discussed above and Coq discussed below. That is, NuPRL admits a typing rule called equality reflection, by which propositional equality is included within definitional equality. The upshot is that definitional equality is undecidable, so there is no proper typechecking algorithm for NuPRL. Instead, the user presents a type to the system and interactively constructs a derivation of that type using tactics.

One of NuPRL's uses is formalized mathematics, in which the proof term that witnesses the theorem is often irrelevant. In that case, the user instructs the system to begin a proof for the statement A that is the encoded form of the theorem they wish to prove. The use of the tactics generates behind the scenes the proof term a and the explicit typing derivation witnessing that a has type A.

In contrast, another of NuPRL's uses is to verify algorithms. This proceeds in two steps. First, one implements the algorithm a in a familiar ML-like programming language. Second, one instructs the system to begin a proof of $a \in A$, where A is a type that encodes the correctness guarantees to be established about a. (For example, one might want to show that a list sorting algorithm's output is an ordered permutation of its input; the type A would encode this property.)

In sum, although undecidability of typechecking appears to be a blow to the system's ergonomics, it does also provide a benefit: the algorithms one designs must be cleanly separated from the proofs about those algorithms. Moreover, in the common case of establishing theorems whose proof terms one doesn't care about, one can simply concentrate on the formulas being manipulated without regard for the proof term. Let the tactics take care of the tedious work of constructing the proof term and its typing derivation!

2.2 Languages for defining tactics

NuPRL, Isabelle-88, and LCF do not have proper languages for defining tactics. Instead, one defines tactics essentially by extending the proof assistant itself. This works reasonably well in those settings because the assistant is interpreted, and we can think of the assistant not as an application itself, but rather as a library implemented in ML. The upshot is that one can use the ML REPL to develop proofs interactively.

When a proof assistant is its own, separate application, then the development of new tactics cannot reasonably proceed by extending the assistant itself. Therefore, we see a drive to design languages for defining the new tactics.

2.2.1 Tactic languages for Coq

In the mid-80s, as LCF and Isabelle are developed in Scotland and England, Thierry Coquand and Gérard Huet develop the Calculus of Constructions in France [12]. This is a higher-order constructive logic possessing all forms of quantification (in the sense of Barendregt's lambda cube [3]). Furthermore, they developed an implementation of this logic, called Coq[2]. What distinguishes Coq from LCF and Isabelle is that it is designed with *program extraction* in mind. This technique recovers from a proof object its computational component as an executable program in a conventional programming language, e.g. ML.

As a highly simplified but concrete example, consider a formula of the form $\forall x{:}\tau_1.\exists y{:}\tau_2.P$. In constructive logic, existential claims actually contain an object that witnesses the claimed existence. Then it seems possible to transform this logical statement into a program that, given an $x{:}\tau_1$, computes the $y{:}\tau_2$ that witnesses the claim. That is, one can recover an ordinary

[2]The system was itself called CoC and then CONSTR for several years before being renamed, but I will refer to it only as Coq for simplicity.

function of type $\tau_1 \to \tau_2$.

This extraction process enables the development of *verified algorithms*. First, one uses the system to prove a theorem, e.g. "for any list l, there exists a list l' such that l' contains exactly the elements of l and such that l''s elements are in ascending order". Then, one uses program extraction to obtain from this theorem a proper sorting algorithm together with a guarantee of this algorithm's correctness.

Similar to LCF and Isabelle, Coq features a number of tactics and tacticals to aid in proof development. The original tactics are much like LCF's and Isabelle's, so I will not describe them. An important difference, however, between Coq and the other systems I have described, is that Coq is a proper application written in ML[3], with its own syntax, parser, etc. In other words, users do not interact with Coq via the ML REPL. This has two unfortunate downsides for the extensibility of Coq's tactic language. First, it is challenging to add new tactics to Coq, as one must obtain the source code for Coq, implement the new tactics in ML, and recompile the system as a whole. Second, the distribution of custom tactics is difficult: should one ask that one's domain-specific tactic be included in the core system, or should one circulate a patch file to be applied to the system's source tree? Neither of these distribution strategies is particularly satisfying. Domain specific tactics should probably not be included in the core distribution of the system, else users eventually find themselves drowning in an abundance of highly specialized tactics unnecessary to their specific problem domain. The circulation of patches, on the other hand, requires that one keep their patches constantly up to date with the current distributed version of the system.

At last, a solution appears at the turn of the millenium as David Delahaye announces the Ltac language [13]. Ltac is a domain-specific language embedded within Coq for defining new tactics. Essentially, scripts in Ltac

[3]Coq has been implemented in many different ML languages over the years, first in CAML, then in Caml-light, and finally (and still to this day) in OCaml. To simplify, I will say that Coq is merely implemented in ML.

are interpreted to execute commands in Coq's kernel. The benefits of Ltac are that it is high-level compared to implementing tactics directly in ML and that new tactics can be defined alongside proofs. In fact, Delahaye reports significant code size reductions (and even sometimes speedups!) in tactics ported from ML to Ltac. Since custom tactics can be defined together with one's proofs, tactic and even proof distribution are vastly simplified. Indeed, Pierre Pédrot later writes in [43], "the Ltac tactic language is probably one of the major ingredients of the success of [Coq], as it allows to write proofs in an incremental, more efficient and more robust way that the state of the art at that time."

To illustrate one particularly high-level construct from Ltac called `match goal`, let us consider an example from Delahaye's paper, in which he proves that the natural numbers have more than two elements. This statement can be expressed formally as

$$\neg(\exists x{:}\mathbb{N}.\exists y{:}\mathbb{N}.\forall z{:}\mathbb{N}.\, x = z \vee y = z)$$

The proof stategy is to assume the negated formula and arrive at a contradiction. The assumed existential claim can be decomposed, assuming the existence of such an x and such a y. Then, it suffices to instantiate the assumption $\forall z{:}\mathbb{N}.x = z \vee y = z$ with three distinct numbers, say 1, 2, and 3. At this point, we have three relevant assumptions: $\mathcal{H}_1 :: x = 1 \vee y = 1$, $\mathcal{H}_2 :: x = 2 \vee y = 2$, and $\mathcal{H}_3 :: x = 3 \vee y = 3$. Successively eliminating these assumptions can be accomplished quite easily with the ; (semicolon) tactical, which applies the tactic on the right to each subgoal generated by the tactic on the left: `elim` $\mathcal{H}_1$; `elim` $\mathcal{H}_2$; `elim` $\mathcal{H}_3$. In detail, the first elimination generates two subgoals, and *for each* of these, the second elimination generates two subgoals, *and for each of those*, the third elimination generates two subgoals. This exponential blowup results in eight subgoals, each of which contains a pair of assumptions of the form $x = a$ and $x = b$, or $y = a$ and $y = b$ for distinct constants a, b.

```
Lemma nat_card : ¬(∃ x:nat, ∃ y:nat, ∀ z:nat, x = z ∨ y = z).
Proof.
  intros A; elim A. intros x H; elim H. intros y J.
  specialize (J 0) as H1. specialize (J 1) as H2. specialize (J 2) as H3.
  elim H1; elim H2; elim H3; intros;
  match goal with
    | [_ : ?x = ?a, H : ?x = ?b |- _] ⇒
      subst x; discriminate H
  end.
Qed.
```

Figure 2.1: A simple theorem on the cardinality of natural numbers, to illustrate the `match goal` construct in Ltac.

Ltac's `match goal` construct is a form of pattern matching for extracting parts of the active subgoal. One can match on available assumptions as well as on the current goal. The pattern Delahaye uses is `[_ : ?x = ?a, H : ?x = ?b |- _]`. The turnstile separates the hypothesis pattern from the goal pattern. The question mark syntax expresses metavariables, and the nonlinear appearance of `?x` will involve unification during matching. This pattern can match all eight of the subgoals that we have generated, as each of them contain (at least) the assumptions of the written forms. Then, in the body of this branch, we eliminate the first equality using `subst x`, refining the type of `H` to evidently be uninhabited, as it states an equality between two syntactically unequal, closed terms. Then we eliminate the absurd assumption `H` using the `discriminate` tactic to produce the required contradiction. Since we want to perform this general analysis on all eight subgoals, we sequentially compose the `match goal` construct with the eliminations using `;`. See Fig. 2.1 for the full example.

The beauty of Ltac is that this basic recipe could be further generalized to prove that the natural numbers have more than n elements for a closed n. First eliminate the n nested existentials. Second, eliminate the universal $n + 1$ times, with distinct naturals, e.g. 0 through n. Finally, end the proof

in the same way, using `match goal` to find a pair of equalities that can be used to produce a contradiction.

The main downside to Ltac is that it is ill-specified. Indeed, Delahaye does not give any semantics for tactics defined in Ltac, and upon Ltac's release, it was not statically typed. The lack of a typing discipline would not, however, be a dealbreaker, given that Ltac runs during typechecking anyway: any runtime error during execution of an Ltac script becomes a type error at its invocation site. Overall, the lack of specification is acceptable at the time of Ltac's release as the language eliminates a growing problem with custom tactics by allowing users to define simple custom tactics while encouraging them to implement larger, more sophisticated ones in ML.

Alas, it is all too common for domain-specific languages to outgrow their original scope, and Ltac is no exception. Over the following two decades, several other approaches to programming tactics in Coq are proposed: Mtac [53] / Mtac2 [24], Rtac [25], Template-Coq [2] / MetaCoq [46], and Ltac2 [43] to name a few. I briefly discuss each of these lines of work.

Mtac / Mtac2

The Mtac project broadly aims to give a static type system to tactics, encapsulating the effects that tactics wish to use inside a monad. Thus, a tactic of type `M A` is guaranteed to produce a term of type `A` if it terminates, and it may use effects from the monad `M`. The constructors for this type family `M` include the usual monadic operations, together with many useful combinators for tactic programming, e.g. fixed points, exception handling, pattern matching, and more. The primitive tactic execution construct **`run`** `t` has type `A` assuming that `t` has type `M A`. This primitive is eliminated during the type inference stage, so the trusted Coq kernel does not need to be extended to handle **`run`**[4]. Like in Ltac, an Mtac program is "untrusted" in the

[4]This is a common theme in Coq: earlier typechecking stages are often extended in order to keep the trusted kernel as simple as possible.

sense that it must be executed to generate a term and that term must be typechecked. The downside to this is that very large proof terms could be generated by seemingly simple tactics (as a result of proof search), which negatively impacts the performance of typechecking.

Rtac

The Rtac project [25] of Gregory Malecha et al. seeks to address performance issues with Ltac and belongs to a family of approaches called *reflective metaprogramming*. Such an approach provides primitives for quoting and unquoting terms from the proof assistant itself: a Coq term is *reified* (quoted) into a value of a concrete Coq datatype, transformed according to the implementation of the tactic, and then *reflected* (unquoted) back.

Rtac distinguishes itself from other reflective metaprogramming techniques (e.g. in Agda [50] or in Template-Coq) by allowing the user to define a custom concrete syntax that is the target of reification. Hence, a tactic needs only to consider the cases for that syntax instead of the syntax for the entire assistant.

Rtac's main motivation is not ease-of-use or safety, but actually *performance*. To use an example from [25], consider the problem of checking equality in a commutative monoid, e.g. checking that $x \oplus 2 \oplus 3 \oplus 4 = 4 \oplus 3 \oplus 2 \oplus x$. One could write an Ltac program prove this directly, using transitivity of equality to witness a series of permutations of elements on the left until they match those on the right. Such an Ltac program is not particularly hard to write, but the generated proof following this strategy is generally quadratic in the size of the input. A more clever proof would flatten these expressions into lists and check that one is a permutation of the other. Malecha describes this technique in detail in [25] using Rtac, and the resulting proof is linear in the input size. On large enough input sizes (8 elements in the commutative monoid equivalence example), these improved asymptotics outweigh the cost of reification and reflection.

Alas, there are some additional requirements to make this approach work: the user must prove a soundness lemma for their tactic. This lemma expresses that if the tactic decides that the two reified terms are syntactically equal, then their reflections are also equal. Malecha addresses this concern by providing a number of tacticals such that tactics built using them can have their soundness lemmas automatically derived. However, more sophisticated tactics that cannot be built out of the provided tacticals will require soundness lemmas to be proven manually. Further limitations of the Rtac system is that its internal language is simply-typed, so Rtac cannot reify dependently-typed terms.

Template-Coq / MetaCoq

The recent MetaCoq [46] project is quite large, having several goals. Among them are reification and reflection of terms as well as a monadic interpreter for scripting tactics. (Its goals less relevant to us, but nonetheless quite impressive, include a full specification of Coq's typing and operational semantics as well as a correctness proof of a functional typechecker for Coq.) To achieve these goals, MetaCoq combines ideas from the Mtac2 and Template-Coq projects: it keeps Mtac's idea of using a monad to capture effects that tactics need (such as nontermination, backtracking, state, etc.) but it replaces Mtac's shallow embedding of Coq terms with Template-Coq's deep embedding.

In more detail, MetaCoq's `TemplateMonad` is defined as a free monad in Coq. This is essentially a syntactic device that happens to be a monad, but must later be interpreted to have any effect. The interpretation is defined in OCaml as a Coq plugin. The free monad definition includes functions for quoting various constructs in Coq, such as terms, inductive types, universes, and constants.

The concrete syntax into which terms are reified is a simple type `term` and nothing prevents a MetaCoq program from manipulating these in ways

that violate scoping, let alone typing. This is in fact intended by the authors, as the alternative (as developed in Agda [50]) is to use sophisticated concepts such as inductive-recursive (IR) and quotient inductive-inductive types (QIITs) to obtain an intrinsically-typed representation of the syntax. Significant extensions to Coq's metatheory would be required to accommodate IR and QIITs.

Given that MetaCoq's concrete representation of terms is untyped, there are two forms of unquoting available: `tmUnquote t` and `tmUnquoteTyped A t`. Both of these infer some type for the unquoted term. In the former, the inferred type is existentially quantified, so there is no way to know the type[5]. In the latter, one specifies an expected type `A` to `tmUnquoteTyped` and the system requires that the inferred type for `t` unify with `A`. Recall that all of these operations take place within `TemplateMonad`, so any type inference failure (or unification failure in `tmUnquoteTyped`) will cause an exception in the monad.

Although Ltac and Mtac programs can generate only terms, Template-Coq and by extension MetaCoq are capable of generating full toplevel definitions, as in Template Haskell [45]. This makes it possible to do such things as defining new inductive types and to generically generate lemmas by inspecting the shapes of inductive types. The authors do not discuss at length how one defines new tactics using MetaCoq, since this is not their primary aim. They claim, however, that MetaCoq can act as a foundation for eventually building domain-specific tactic languages in the style of Ltac and Mtac.

Ltac2

All competitors to Ltac surveyed so far offer a brand new approach entirely, namely by using an explicit monad or reflection (or both in the case of MetaCoq). For a traditional Ltac user, it can be challenging to adopt these techniques since they differ quite a lot from Ltac. The Ltac2 project [43]

[5]Since pattern matching on types is forbidden.

addresses this by explicitly aiming to resemble Ltac, maintaining a strong degree of backwards compatibility.

Although it resembles Ltac on the surface, Ltac2 is a proper ML language. It is "call-by-value and effectful, supports algebraic datatypes and allows prenex polymorphism". The ability to define datatypes in Ltac2 is a huge improvement, as it allows writing tactics that use data structures such as lists and dictionaries. Like Ltac, Ltac2 uses runtime checks to ensure that generated terms are well-typed.

Ltac2 features a macro system of "notations" together with a metaprogramming system of quotations and antiquotations for Coq terms. This is similar to the notation system in Ltac, so it should be familiar to users. The key difference is that each notation is associated with a *scope*, which is a function that expands syntax on the fly. The notion of scope is made higher-order by introducing *scope combinators*. For example, the `ident` scope for parsing identifiers can easily be made into a scope for parsing a list of identifiers using the `list0` scope transformer. This leverages the existence of data structures, generating a *list* of identifiers.

2.2.2 VeriML

Although the Coq ecosystem sees a lot of development regarding tactic languages, some have thought to distance themselves from the Calculus of Constructions to bring new insights. VeriML [47, 48] is an ML-like language for developing proof automation. It is very expressive, having references and general recursion.

A key idea in this system is to clearly separate the logical language $\lambda\mathsf{HOL}_{ind}$ from the computation language: propositions and proofs are explicitly embedded into the computation language using an angle-bracket syntax $\langle\cdot\rangle$. VeriML features a form of dependent types in which objects representing proofs are indexed by the proposition they prove. Furthermore, following [29, 40], logical terms come packaged with the context in which they

are meaningful. Abstraction over contexts is also possible, to express that a function can work in any context.

The language $\lambda\mathsf{HOL}_{ind}$ is a higher-order logic augmented with inductive types and the ability to define total recursive functions. It can be seen as a common core between the Calculus of Inductive Constructions[6] [34] (CIC) and systems in the HOL family such as Isabelle. Rather than use a pattern matching construct, which is quite complex in the presence of dependent types (see [20]), $\lambda\mathsf{HOL}_{ind}$ uses eliminators. That is, when one defines a new inductive type, the system synthesizes a special constant called an *eliminator* for that type. The eliminator represents the induction principle for the type. This is essentially a function from the newly defined type into some other type family of one's choosing, sometimes called a *motive* [27]. To use the eliminator, one must also provide a number of *methods*: given that an inductive type is a sum of products, each method handles one branch of the sum. Recursive occurrences of the inductive type within a branch become replaced by the motive. From a proof-theoretic point of view, these replacements correspond to the induction hypotheses. As a concrete example, consider the type $\mathbb{N}$ of natural numbers. The natural eliminator one obtains for $\mathbb{N}$ is

$$\mathsf{elim}_{\mathbb{N}} : P\ \mathsf{z} \to (\Pi n{:}\mathbb{N}.\ P\ n \to P\ (\mathsf{s}\ n)) \to \Pi n{:}\mathbb{N}.\ P\ n$$

where the motive P is a type family indexed by $\mathbb{N}$.

The benefit of using eliminators over the more familiar pattern matching and explicit recursion is that eliminators greatly simplify totality checking. In fact, the type of the eliminator itself is constructed so that totality is guaranteed. The downside to eliminators is that the selection of a motive can be quite tricky, making programming using eliminators quite challenging.

The computation language of VeriML manipulates contextual $\lambda\mathsf{HOL}_{ind}$ terms. These terms are $\lambda\mathsf{HOL}_{ind}$ terms together with a context in which they are meaningful: a contextual term $[\phi]t$ may mention in t free variables

[6] This is the Calculus of Constructions augmented with inductive families [14].

that are listed in the context ϕ. Although $\lambda\mathsf{HOL}_{ind}$ uses eliminators to define total functions, the computation language of VeriML uses the usual pattern matching and explicit recursion of functional programming. The need to use eliminators to simplify totality checking is unnecessary in the computation language as this language admits and embraces nontermination. The construct `holcase` T `of` $\cdots$ is used to perform dependent pattern matching on the contextual $\lambda\mathsf{HOL}_{ind}$ term T. Given that this matching is dependent, the branches may have different types, according to the refinement generated by matching the term T against the pattern of the branch.

The expressivity of VeriML makes it possible to define sophisticated decision procedures and proof automation without worrying about termination and by using powerful imperative data structures. In [47], the author uses these features to implement for example a decision procedure for the theory of equality with uninterpreted functions (EUF). This theory is generated by the usual axioms of equality (reflexivity, symmetry, and transitivity) together with the rule $x = y \implies f\,x = f\,y$. The functions f are uninterpreted in the sense that they do not have any associated reduction behaviour, unlike e.g. in the theory of arithmetic where the function + has such a behaviour. A problem in EUF asks whether a given equation is true in a context of assumed equations. The standard technique for deciding this theory is to use a union-find data structure to find equivalence classes of terms according to the assumed equations. Then it suffices to check whether the two terms of interest belong to the same equivalence class. Although persistent implementations of the union-find data structure have been developed (see [10]), the traditional implementation is imperative. The imperative features of VeriML make it easy to port this implementation straight from standard algorithms textbooks.

2.3 Structured editing

Although Harpoon uses a tactic language for interfacing with the user, these tactics are not really of primary interest, since they are not recorded. Instead, applying a tactic elaborates a (partial) *structured proof script*, which is recorded to a file. Therefore, we can see Harpoon as a high-level editor for proof scripts. An editor aware of the structure of the document being edited is called a *structured editor*[7].

Usually, editing source code means writing text which is then parsed according to the grammar of the language being written. Nothing prevents writing text that is unmeaningful: one can easily write in Emacs some text that does not parse. In a structured editor, edit actions are restricted so they respect the grammar of the language, making it impossible to construct syntactically incorrect programs. These editors have a notion of "hole", which stands for a node in the abstract syntax tree (AST) that remains to be constructed by the user.

Perhaps the biggest success of structured editors today is in teaching programming to children: the Scratch language [26], designed to be user-friendly and using a GUI for editing, is a syntactic structure editor. The benefit in an educational setting is that learners can focus immediately on computational thinking, bypassing any struggle with program syntax.

Another advantage of structured editors is that they can more easily provide editing services such as syntax highlighting, type-aware code completion, and automated refactoring. These operations require that the program being analyzed or transformed be syntactically or semantically meaningful, so traditional editors must selectively disable them or employ ad hoc heuristics when a malformed program is encountered. This is a nonissue in a structured editor, which needs not even concern itself with malformed program text.

However, some editing services require not only syntactically meaningful

[7]Other names include *structural editor*, *syntax-directed editor*, and *projectional editor*.

programs, but also *semantically* meaningful programs. The Hazelnut system [32] gives a theoretical foundation for a structured editor in which every edit state is semantically meaningful. That is, every edit state is well-typed.

The challenge in this development is to preserve some degree of user-friendliness: naively enforcing typing would require the user to construct programs in a very rigid, outside-in form. For instance, the user would have to identify that they wish to construct a function *application* before they identify the function they wish to apply! Even worse, in a functional language encouraging currying, due to left-associativity, the programmer would have to construct the appropriate *number* of application nodes before specifying the function to call!

The solution to this is one of Hazelnut's key innovations: a hole in a program may be *nonempty*. In this case, the hole is understood not as a mere missing AST node, but as an internalized type mismatch. The content of the hole must internally be well-typed, but crucially it can have *any* type.

For example, consider a hole of type B. We wish to fill it with a function application $f\ a$ for some $f : A \to B$ and $a : A$. We first construct a reference to the function f. This is ill-typed: the expected type is B but the inferred type of f is $A \to B$. Thus the variable f is placed *inside* the hole, witnessing the type mismatch. Then, inside the hole we construct the application node, placing the variable f as the left child and specify the argument a as the right child. The type of the expression inside the hole is now B, matching the expected type outside the hole, so the hole vanishes, becoming replaced by its contents.

To properly separate the notions of inferred and expected type, Hazelnut is presented as a bidirectional type system. Both empty holes $(\!|\,|\!)$ and nonempty holes $(\!|e|\!)$ are classified as synthesizable expressions. A nonempty hole's body e must synthesize some type τ, so the hole must be internally well-typed, but both an empty and a nonempty hole synthesize the hole type $(\!|\,|\!)$. Note that although *expression holes* may be nonempty, *type holes* may

not be nonempty[8].

In a traditional bidirectional type system, the rule for switching from the checking mode to the synthesis mode has the additional premise that the synthesized type be convertible (or made convertible via unification) with the type being checked against. In Hazelnut, the notion of type equality used in this situation is called *type consistency*, and crucially it accounts for the hole type $(\!|\,|\!)$ being "equal" to any other type. As for checking against the hole type, additional consideration is necessary. Consider the situation $\Gamma \vdash \lambda x.\ e \Longleftarrow (\!|\,|\!)$. The hole type $(\!|\,|\!)$ is made to expand into $(\!|\,|\!) \to (\!|\,|\!)$ so that the context is extended with the declaration $x{:}(\!|\,|\!)$ and the body of the lambda is further checked against $(\!|\,|\!)$. Extending the system with additional type formers (e.g. product types or sum types) requires corresponding new judgments to handle such hole type expansions.

Hazelnut's metatheory is rich and interesting: not only are the traditional progress and preservation theorems proven, but also the authors show several results concerning the interactive nature of their system, which they call a *structured editor calculus*. Recall that Hazelnut has a notion of cursor that is superimposed onto expressions. The programmer manipulates the cursor and constructs AST nodes using interactive commands called *actions*. Here is a summary of the main results concerning actions. All this metatheory is mechanized in Agda.

- The program is invariant under movement actions, i.e. movement actions affect only the cursor.
- From any cursor position, a sequence of movement actions exists to bring the cursor to any other position.
- For any well-typed expression, there exists a sequence of actions to construct it interactively.

[8]One might imagine that they can be nonempty in a potential extension of this system to dependent types.

- Any expression constructed interactively is well-typed.

Hazelnut's edit actions bear a resemblance to tactics. A key difference is that in tactic-based systems, the cursor is less explicit and can be focused only on a subgoal, whereas in Hazelnut the cursor can be moved around anywhere in the expression. Hazelnut's edit actions include also a command for deleting a node, replacing it with a subgoal. This is more general than a mere undo mechanism as it appears in tactic-based systems.

The notion of nonempty holes in Hazelnut gives a uniform way to handle backwards reasoning. This distinguishes Hazelnut from traditional tactic-based systems, which employ special tactics for backwards reasoning. However, any backwards reasoning performed in a nonempty hole is obscured in the final expression: by merely seeing a function application, for example, one cannot know whether it was constructed in a forwards or backwards style.

2.4 Beluga

Harpoon is an interactive frontend for Beluga [6]. One develops a proof interactively using Harpoon, and this proof can either remain as a proof script, or it can be translated into a Beluga program. It is this translation that justifies our claim that a Harpoon proof script truly does represent a proof. Considering this, it is essential to first describe Beluga. The main technical exposition of Beluga is given in Sec. 4.1 before we describe Harpoon's metatheory, so this section focuses on a high-level overview of Beluga.

2.4.1 Contextual Modal Type Theory

Beluga is based on Contextual Modal Type Theory (CMTT) [29]. Although Beluga is dependently-typed, I discuss here the simply-typed Contextual Modal Type Theory so we can concentrate solely on the contextual ideas.

This theory can be obtained from simple type theory by internalizing the notion of *context*. First, the syntax of types is extended to include the form $[\Psi]A$, called a *boxed type*. A term of this type represents a hypothetical derivation depending on the context Ψ and concluding A. Second, the typing judgment $\Delta; \Gamma \vdash t : A$ mentions two contexts: Γ contains ordinary assumptions whereas Δ contains contextual assumptions. A contextual assumption has the form $u : A[\Psi]$ in which the contextual variable u expresses that A is true in the context Ψ. Contextual variables are also called *metavariables*. One must take care to distinguish an ordinary assumption of a boxed type $x : [\Psi]A$ from a contextual assumption $u : A[\Psi]$. The introduction and elimination rules for boxed types are the following.

$$\frac{\Delta; \Psi \vdash M : A}{\Delta; \Gamma \vdash \mathsf{box}(\Psi.\ M) : [\Psi]A} \qquad \frac{\Delta; \Gamma \vdash M : [\Psi]A \quad \Delta, u{:}A[\Psi]; \Gamma \vdash N : C}{\Delta; \Gamma \vdash \mathsf{letbox}(M, u.\ N) : C}$$

In the introduction rule, M is a hypothetical derivation concluding A and depending on the assumptions Ψ. In the elimination rule, we wish to prove C using the hypothetical derivation M depending on Ψ. The context Δ becomes extended with a new contextual variable, and we conduct the proof N of C in this extended context. In other words, rather than demand upfront that the assumptions Ψ be instantiated with some terms, this is delayed until the moment that the contextual variable is used. To use a contextual variable, one provides an explicit substitution that maps the context of that assumption into the current context.

$$\frac{\Delta, u{:}A[\Psi]; \Gamma \vdash \sigma : \Psi}{\Delta, u{:}A[\Psi]; \Gamma \vdash \mathsf{clo}(u, \sigma) : A}$$

2.4.2 Contextual types in Beluga

Now let us see how these ideas scale up to dependent types and are applied in Beluga. In the presentation of CMTT above, the system is homogeneous

in the sense that the language of terms inside a box is the same as the language of terms outside a box. In principle this means one can nest boxes arbitrarily deep. Beluga, in contrast, limits this nesting to one level, given that the language inside a box differs from the language one writes outside a box. Inside a box, one uses LF [22] to uniformly represent the syntax and the derivations of an encoded language. In general, the language used inside a box is called an *index language* and it could be natural numbers, strings, or lists [9, 52], but given Beluga's focus on mechanized metatheory, LF is a good choice as it gives concise and elegant representations of binding structures. (We discuss concretely in Chap. 3 how to encode a language in Beluga, using the simply-typed lambda calculus as an example.)

Beluga goes slightly beyond pure contextual LF for its index language, by allowing quantification over contexts. These quantifiers bind *context variables*. Context variables in particular are key to expressing even basic metatheorems about formal systems. For example, consider the type uniqueness theorem for the simply-typed lambda calculus (in which the parameter of an abstraction is equipped with a type annotation).

$$\text{If } \Gamma \vdash M : A \text{ and } \Gamma \vdash M : B, \text{ then } A = B.$$

This theorem cannot be proven without somehow generalizing over contexts, as the case for abstractions $M = \lambda x{:}A.\ M'$ requires extending Γ with an additional declaration to use the induction hypothesis. We state this theorem as the following type in Beluga, using the function space $\rightarrow$ to model implications as is often the case in proof assistants based on type theory.

```
unique : (g:ctx) [g ⊢ oftype M A[]] → [g ⊢ oftype M B[]] → [⊢ eq A B]
```

Whereas terms are classified by types, contexts are classified by *schemas*. In the above function type, the syntax `(g : ctx)` binds a context variable `g` with the schema `ctx`. Note that this schema is not built in to Beluga: one must separately define it, specifying the form of the declarations it is allowed to contain. In this example, we would define `ctx` as containing declarations

that associate a variable to a type. This represents another departure from basic CMTT, in which the internalized contexts are precisely the contexts of CMTT itself. Since Beluga aims to model various formal systems and since the notion of context can vary from system to system, Beluga must therefore also support modelling these contexts.

What Beluga retains from CMTT, however, is the notion of metavariable. Consider again the type of `unique` and observe that the variables `M`, `A`, and `B` are free. Beluga automatically abstracts over these and infers their type by a process called type reconstruction. The internal type generated by type reconstruction is

```
(g:ctx) (M : (g ⊢ tm)) (A : (⊢ tp)) (B : (⊢ tp))
[g ⊢ oftype M A[]] → [g ⊢ oftype M B[]] → [⊢ eq A B]
```

where we use `( )` to express an implicit Pi-type [9]. As in CMTT, to *use* a metavariable, we must associate it with an explicit substitution to mediate between its context and the context at the use site. This is seen explicitly in `[g ⊢ oftype M A[]]`, where the explicit weakening substitution `[]` is associated to `A`. On the other hand, no substitution seems to be associated to `M`: as a programming convenience, omitting a substitution is syntactic sugar for the identity substitution, `[..]`. Notice that since metavariables in Beluga stand for contextual LF terms, they can appear only within boxes, whereas in CMTT, being homogeneous, they can appear anywhere provided that an appropriate substitution is given.

2.4.3 Programming with contextual types

Now let us shift our focus to Beluga's programming language, in which one manipulates these contextual objects. Beluga is a dependently-typed functional programming language. Beluga's form of dependent types differs from that in languages such as Coq or Agda. One can quantify over any type in those systems, whereas in Beluga quantification is over contextual types,

[9] A Pi-type is the type-theoretic analogue of a universal quantifier.

as we have seen in the reconstructed type of `unique`. Following the Curry-Howard correspondence, a proof is represented as a total recursive function understood as mapping the premises of a theorem to its conclusion. These total functions are written by using (dependent) pattern matching on boxed objects, representing derivations.

Dependent pattern matching has two essential components in Beluga. First, the type of the pattern is unified with the type of the scrutinee. Second, in the case of matching on a contextual object, the pattern itself is unified with the scrutinee. These unifications produce a *refinement substitution* θ, one for each branch. This susbtitution refines the types of other assumptions as well as the goal type: if the case expression as a whole is to be a term of type τ, then the ith branch is to have type $[\theta_i]\tau$.

To illustrate the effect of dependent pattern matching, consider again the type of `unique`. Now suppose we perform a case analysis on the metavariable `M : (g ⊢ tm)` and suppose further that we have the assumptions `d1 : [g ⊢ oft M A[]]` and `d2 : [g ⊢ oft M B[]]`. One branch of this case analysis is for `M = app M1 M2`, so the refinement substitution in that branch is $\theta =$ `app M1 M2 / M`. This means that in the branch, the types of the assumptions `d1` and `d2` have changed: e.g. `d1 : [g ⊢ oft (app M1 M2) B[]]`.

Given that Beluga uses recursion and pattern matching rather than eliminators, the totality checking puzzle is more complicated. First, all case expressions must be covering. This is ensured by a coverage analysis [38]. Second, all recursive calls must be well-founded. To ensure this, each function must be annotated with an *induction order*. The simplest such order specifies that induction is on a specific premise. Then, when splitting on that premise using pattern matching, each subterm in a pattern is used to generate an induction hypothesis. The full list of these induction hypotheses is carried around during typechecking, so when a recursive call is encountered in the program, it suffices to check that it is among the list of available induction hypotheses.

We end the discussion of Beluga for now, returning with a more detailed and technical description in Sec. 4.1. In the next chapter, we see how to encode a language in Beluga and how to use Harpoon to prove properties about it.

Chapter 3

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.

3.1 Initial setup: encoding the language

We begin by defining the simply-typed lambda-calculus in the logical framework LF [22] 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. Given that LF does not have case analysis or recursion, its function space is a weak, representational one: only genuine lambda terms can be represented.

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

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

Free variables such as T1 and T2 are implicity universally quantified (see [37]) 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 =
  | s_app  : step M M' →
             step (app M N) (app M' N)
  | s_beta : step (app (lam M) N) (M N);
```

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

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;
```

3.2 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 [36, 40, 29]. 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 meta-context; second, the assumptions `s : [⊢ step M M']` and `h : [⊢ halts M']` are added to the program context. Observe that since `M` and `M'` have type `tm T`, `intros` also adds `T` to the meta-context, although it is implicit in the definitions of `step` and `halts` and is not visible. (See HARPOON example 1 Step 1.)

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 meta-context. (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 meta-context 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.)

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 : ( ⊢ steps 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 : ( ⊢ steps 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.

3.3 Setup continued: reducibility

We define normalization using reducibility candidates [19]. That is, for each type `T` we define a set of terms *reducible* at that type. All reducible terms are required to halt, and terms reducible at an arrow type are required to produce reducible output given reducible input. Formally, a term `M` is reducible at type `(arr T1 T2)`, if for all terms `N:tm T1`, if `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 `Reduce` as a *stratified type* (see [23]). This can be seen as a restricted form of type-level function, defined inductively on the type `T` used as an index.

```
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];
```

3.4 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 meta-variables `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 of 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. 3.2) 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 HAR-

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`).

POON Proof 3 Step 5).

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

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

and the weak normalization theorem. For a more comprehensive description of this proof in Beluga see [7, 8].

3.5 Additional features

Besides the tactics already discussed in this section, our implementation supports several others. Two 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 [51]. This tactic is essential in the completeness proof for algorithmic equality [8].

We also support a number tactics that do not contribute to the elabora-

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

tion 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.

Chapter 4

A Logical Foundation for Interactive Theorem Proving

In this chapter we give a logical foundation for interactive command-driven theorem proving in Beluga using Harpoon. In particular, we describe Harpoon's interactive commands, their relationship to proof scripts, and the translation from proof scripts into Beluga programs.

4.1 Background: proofs as programs

We have already described programming in Beluga in Sec. 2.4, but that discussion remained at a very high level to give intuition about the system. Here we describe formally Beluga's programming language where we can describe (inductive) proofs as total recursive programs. From a logical perspective, a Beluga program witnesses a theorem in a first-order logic equipped with induction over a specific index domain. Recall that in general, there are many possible index domains to choose from. Beluga chooses contextual LF together with first-class contexts as its index language. 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 examples from Chap. 3, 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 pair those objects (and their types) together with the context in which they are meaningful. In the grammar above, U refers to such a contextual type and C denotes a contextual object, for example `( ⊢ arr unit unit)` is the contextual type of `( ⊢ lam λx. x)`. Contextual objects may contain index variables that are declared in Δ. For example, `( ⊢ steps M M)` is meaningful in the index context $\Delta =$ `A:( ⊢ tp), M:( ⊢ tm A)`.

We do not describe here the index language in full detail – it is quite technical, it has been described elsewhere, and such a level of detail is not crucial for the understanding or our design of Harpoon. Instead, we refer the interested reader to [49, 23]. 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 meta-context 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 [38]).

Below we describe the core fragment of Beluga as a bidirectional type system. Terms are separated into those that we check against a given type and those for which we synthesize a type. To keep the presentation simple, we model (co)inductive and stratified types as constants. Types include simple functions $\tau_1 \to \tau_2$, dependent functions $\Pi X{:}U.\,\tau$, boxed types $[U]$, and constants $\mathbf{b}\ \overrightarrow{C}$ used to model (co)inductive and stratified types. Here $\mathbf{b}$ stands for an indexed type family and recall U stands for an index type.

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

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

Checkable terms include simple and dependent function abstraction (`fn` and `mlam` respectively), boxed index objects $[C]$, and a `case` expression. We also include for convenience two different let-expressions $\mathtt{let}\ x = i\ \mathtt{in}\ e$ and $\mathtt{let\,box}\ X = i\ \mathtt{in}\ e$, although in principle each could be defined using a function (`fn` and `mlam` respectively) that is immediately eliminated. Such a translation must be type-directed in our bidirectional setting, as functions are checkable terms which cannot appear on the left-hand side of a function elimination. Broadly speaking, such a translation would synthesize the type for i in order to construct a type annotation for the anonymous function to

embed it as the subject of the function elimination.

Last, the syntax of checkable expressions contains contextual variables $\bar{g}$ following [29, 5], 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 that checks against τ in the index context Δ with the assumptions in Γ. 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.

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. 4.1 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.

Most of the typing rules in Fig. 4.1 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 the covering set using the primitive $\mathsf{cov}\ (\Delta; \Gamma; \beta)$ which in turn relies on the coverage primitive $\mathsf{cov}\ (\Delta; U)$ for index objects. Similar to the coverage primitive for index types, the coverage primitive for computation-level base types also generates well-founded recursive calls and includes in Γ_k. Concretely, Γ_k is an extension of $[\theta_k]\Gamma$ that includes any program variables bound by the pattern as well as any well-founded recursive calls. Finally, we check each branch considering the index susbtitution θ_k that accounts for any index variable refinements induced by the pattern. We omit the typing rules for patterns: given that patterns are a subset of expressions, their typing mirrors expression typing.

As for the subgoal context Ω, every rule's conclusion collects all premise

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

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

$$\frac{\Omega \mid \Delta; \Gamma \vdash i \Longrightarrow \tau}{\Omega \mid \Delta; \Gamma \vdash i \Longleftarrow \tau} \qquad \frac{}{\bar{g} : (\Delta; \Gamma \vdash \tau) \mid \Delta; \Gamma \vdash \bar{g} \Longleftarrow \tau}$$

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

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

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

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

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

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

$$\frac{\Omega_1 \mid \Delta; \Gamma \vdash i \Longrightarrow \tau_1 \to \tau_2 \quad \Omega_2 \mid \Delta; \Gamma \vdash e \Longleftarrow \tau_1}{\Omega_1, \Omega_2 \mid \Delta; \Gamma \vdash i\ e \Longrightarrow \tau_2}$$

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

$$\frac{}{\vdash \cdot\ \mathsf{ctx}} \qquad \frac{\vdash \Omega\ \mathsf{ctx} \quad \Delta; \Gamma \vdash \tau : \mathsf{type}}{\vdash (\Omega, g : (\Delta; \Gamma \vdash \tau))\ \mathsf{ctx}}$$

Figure 4.1: Beluga's bidirectional type system, and well-formedness of subgoal contexts.

subgoal contexts to propagate the subgoals downwards. Note that all subgoal variables are distinct and occur exactly once: we allow neither weakening nor contraction for Ω. We distinguish between subgoal variables $\bar{g}$ that stand for programs from those g that stand for *proofs* described in Sec. 4.2. We call these *program* subgoal variables and *proof* subgoal variables, respectively. Observe that a proof subgoal variable cannot appear within a program, so the subgoal context Ω in Fig. 4.1 only ever contains program subgoal variables, of the form $\bar{g}$. As expected, one can define a notion of substitution $[e'/\bar{g}]e$ that eliminates a subgoal variable, satisfying the following theorem.

Theorem 1 (Subgoal Substitution Property 1)**.** *If* $\Omega,\ \bar{g}{:}(\Delta';\Gamma' \vdash \tau') \mid \Delta;\Gamma \vdash e \Longleftarrow \tau$ *and* $\Omega' \mid \Delta';\Gamma' \vdash e' \Longleftarrow \tau'$*, then* $\Omega,\Omega' \mid \Delta;\Gamma \vdash [e'/\bar{g}] \Longleftarrow \tau$*.*

Proof. By induction on the first typing derivation. □

This property together with a corresponding one for proof subgoal variables is central to the soundness of interactive proof development in Sec. 4.3.

We omit the kinding rules and the well-formedness rules for Δ and Γ. However, 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. 4.1.

4.2 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 it while possibly introducing new subgoals. Eliminating a subgoal with exactly one new subgoal can be understood as transforming the initial subgoal.

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

We only consider here a subset of the tactics we support in HARPOON; others and new ones can be added following the same principles: `intros` introduces all available assumptions; `solve` provides an explicit term to close the current subgoal; `by` allows to refer to a lemma, introduce an intermediate result, or use an induction hypothesis, binding the result to a new program variable; `unbox` is the same as `by`, but it binds its result to a new index variable; `split` generates a covering set of cases to consider for a given scrutinee; `suffices` allows programmers to reason backwards via a lemma or a constructor.

Behind the scenes, executing a tactic builds a *proof script* that represents it, and substitutes this script for the subgoal that the tactic eliminates. A proof script very closely reflects the structure of the proof, and the core constructs of the proof script language closely resemble the syntax of actions.

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

We give the typing rules for proof scripts and directives in Fig. 4.2. In its simplest form, a proof script P is either a proof subgoal variable g or a directive D that describes how to prove a given goal. The understanding of a subgoal variable here is the same as in the previous section: it is a contextual variable of type $(\Delta; \Gamma \vdash \tau)$, representing that it remains to show τ in the index context Δ with assumptions Γ. As before, proof subgoal variables cannot depend on any other subgoal variables. Given that expressions appear embedded within a proof script and that program subgoal variables may appear within these expressions, we have that the subgoal context Ω computed from a proof script may contain both forms of subgoal variable. We lift susbtitution of an expression for a program subgoal variable to proof scripts, writing $[e/\bar{g}]P$, and we define substitution of a proof script for a proof subgoal variable, written $[P'/g]P$. These forms of substitution admit a soundness property.

Theorem 2 (Subgoal Substitution Property 2)**.**

1. *If* $\Omega' \mid \Delta'; \Gamma' \vdash e \Longleftarrow \tau'$ *and* $\Omega, \bar{g}{:}(\Delta'; \Gamma' \vdash \tau') \mid \Delta; \Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau$*, then* $\Omega, \Omega' \mid \Delta; \Gamma \vdash_{\mathbf{P}} [e/\bar{g}]P \Longleftarrow \tau$*.*

2. *If* $\Omega' \mid \Delta'; \Gamma' \vdash_{\mathbf{P}} P' \Longleftarrow \tau'$ *and* $\Omega, g{:}(\Delta'; \Gamma' \vdash \tau') \mid \Delta; \Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau$*, then* $\Omega, \Omega' \mid \Delta; \Gamma \vdash_{\mathbf{P}} [P'/g]P \Longleftarrow \tau$*.*

Proof. 1. By induction on the typing for P, using Subgoal Substitution Property 1 when we arrive at an embedded expression e. 2. By induction on the typing for P. □

We extend a proof script using **by** or **unbox** to introduce new assumptions. 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. The **unbox** construct is identical to **by**, but it binds a new index variable, requiring that the term being unboxed synthesize a boxed contextual type $[U]$. Often, one unboxes an assumption from Γ, but it is convenient to allow directly unboxing the result of an appeal to a lemma or an induction hypothesis, without first requiring that this result be bound to a program variable in Γ.

To check well-foundedness of appeals to induction hypotheses, we adopt the approach used internally by Beluga. When the user states a theorem, they must explicitly give a termination measure. Then, when splitting on a variable that participates in the termination measure, we can generate all valid induction hypotheses in advance and store them in the context Γ (see [38]). Finally, when we encounter an appeal to an induction hypothesis, it suffices to check whether it is compatible with any of the precomputed ones in Γ.

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

$\boxed{\Omega \mid \Delta; \Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau}$ Proof script P corresponds to theorem τ

$$\frac{\cdot \mid \Delta; \Gamma \vdash i \Longrightarrow \tau' \quad \Omega \mid \Delta; \Gamma, x{:}\tau' \vdash_{\mathbf{P}} P \Longleftarrow \tau}{\Omega \mid \Delta; \Gamma \vdash_{\mathbf{P}} \textbf{by } i \textbf{ as } x; P \Longleftarrow \tau}$$

$$\frac{\cdot \mid \Delta; \Gamma \vdash i \Longrightarrow [U] \quad \Omega \mid \Delta, X{:}U; \Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau}{\Omega \mid \Delta; \Gamma \vdash_{\mathbf{P}} \textbf{unbox } i \textbf{ as } X; P \Longleftarrow \tau}$$

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

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

$$\frac{\cdot \mid \Delta; \Gamma \vdash e \Longleftarrow \tau}{\cdot \mid \Delta; \Gamma \vdash_{\mathbf{D}} \textbf{solve } e \Longleftarrow \tau}$$

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

$$\frac{\begin{array}{c}\cdot \mid \Delta; \Gamma \vdash i \Longrightarrow \beta \quad \mathsf{cov}\ (\Delta; \Gamma \vdash \beta) = \overrightarrow{(\_; \theta_k; \Delta_k; \Gamma_k)} \\ \text{for all } k.\ \Omega_k \mid \Delta_k; \Gamma_k \vdash_{\mathbf{P}} P_k \Longleftarrow [\theta_k]\tau\end{array}}{\bigcup_k \Omega_k \mid \Delta; \Gamma \vdash_{\mathbf{D}} \textbf{split } i \textbf{ as } \overrightarrow{\{\Delta_k; \Gamma_k \vdash P_k\}} \Longleftarrow \tau}$$

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

Figure 4.2: The type system for HARPOON proofs and directives

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. 4.3. This process stops once we reach a base type β. As the type is unrolled, the judgment also constructs a partial program that is used for the translation in Sec. 4.4. Understood algorithmically, the unrolling judgment encodes a total function whose output respects an expected soundness property.

Theorem 3 (Soundness of Unrolling)**.** *1. For any Δ and Γ and any type τ, there exists a unique term e such that $g : (\Delta'; \Gamma' \vdash \beta) \mid \Delta; \Gamma \vdash \tau \rightsquigarrow e$, where Δ' and Γ' are extensions of Δ and Γ, respectively.*

2. If $g : (\Delta'; \Gamma' \vdash \beta) \mid \Delta; \Gamma \vdash \tau \rightsquigarrow e$, then $g : (\Delta'; \Gamma' \vdash \beta) \mid \Delta; \Gamma \vdash e \Longleftarrow \tau$.

Proof. 1. By induction on τ; 2. by induction on the given derivation. □

$$\boxed{g : (\Delta'; \Gamma' \vdash \beta) \mid \Delta; \Gamma \vdash \tau \rightsquigarrow e}$$

Beluga type τ unrolls to β in the extended meta-context Δ' and computation context Γ'.

$$\frac{}{g : (\Delta; \Gamma \vdash \beta) \mid \Delta; \Gamma \vdash \beta \rightsquigarrow g}$$

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

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

Figure 4.3: Unrolling a Beluga type.

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 [38]). Each computed 4-tuple contains the pattern p_k (unused here, but used and explained in Sec. 4.4), 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 from by current goal and introduces new proof obligations based on what it would take 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.

4.3 Interactive Proof Development

We now describe the construction of a proof script based on the actions in Fig. 4.4. This relationship is by design both immediate and straightforward: each action generates a well-typed proof fragment, and these fragments are assembled together by subgoal substitutions to form the overall proof script.

Each action is simply elaborated into its corresponding construct in the proof script language, using subgoal variables where appropriate to explicitly model the remaining 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.

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

The proof script P that results from a session $\bar{\alpha}$ is well-typed. To see this, we first establish that the partial proof script generated by a single action indeed solves the subgoal in which it is executed.

$\boxed{\Delta; \Gamma \vdash \alpha : \tau \longrightarrow \Omega \vdash P}$ Action α applied to subgoal $(\Delta; \Gamma \vdash \tau)$ produces a proof script P with subgoals Ω

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

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

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

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

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

Figure 4.4: Typing of interactive actions and elaboration into proof scripts.

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

$$\frac{}{\Omega \vdash P \xrightarrow{\cdot} \Omega \vdash P}\ \text{I-Refl}$$

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

Figure 4.5: Rules for sequencing interactive Harpoon actions. Note that we can reorder Ω which allows us in principle to work on any subgoal in Ω in the I-Single rule.

Theorem 4 (Action Soundness)**.**

If $\Delta; \Gamma \vdash \alpha : \tau \longrightarrow \Omega \vdash P$*, then* $\Omega \mid \Delta; \Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau$*.*

Proof. By case analysis on the given derivation. □

Then, given that individual actions produce well-typed proof fragments, it suffices to use the fact that subgoal substitution preserves types to establish that a whole session $\bar{\alpha}$ generates a well-typed proof script.

Theorem 5 (Session Soundness)**.**

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

Proof. By induction using the previous theorem and the subgoal substitution property. □

What one would like to establish next is a completeness result: any *provable* statement τ admits a session $\bar{\alpha}$ whose elaboration generates a proof script P such that $\cdot \mid \Delta; \Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau$. As a proxy for provability and in light of the translation presented in the following section, one can consider statements τ having a Beluga term e that check against that type. Sadly, we cannot establish completeness for now, since Beluga supports deep pattern

matching (in which a pattern consists of nested constructors), whereas the **split** directive can only capture 1-deep patterns.

4.4 Translation

The translation in Fig. 4.6 from proofs to Beluga programs is now straightforward. An **unbox** becomes a `let box` construct in Beluga. Similarly, **by** i **as** x translates into a `let`-expression. The translation of directives is also 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. The following soundness property follows immediately.

Theorem 6 (Translation Soundness)**.**

1. *For any proof script P, if $\Omega \mid \Delta; \Gamma \vdash_{\mathbf{P}} P \Longleftarrow \tau$, then there exists a unique expression e such that $\Omega \mid \Delta; \Gamma \vdash_{\mathbf{P}} P \rightharpoonup e \Longleftarrow \tau$.*

2. *For any directive D, if $\Omega \mid \Delta; \Gamma \vdash_{\mathbf{D}} D \Longleftarrow \tau$, then there exists a unique expression e such that $\Omega \mid \Delta; \Gamma \vdash_{\mathbf{D}} D \rightharpoonup e \Longleftarrow \tau$.*

Moreover, in both cases e is such that $\Omega \mid \Delta; \Gamma \vdash e \Longleftarrow \tau$.

Proof. By induction on the translation derivation. □

$$\boxed{\Omega \mid \Delta; \Gamma \vdash_{\mathbf{P}} P \longrightarrow e \Longleftarrow \tau}$$

Proof P is translates to Beluga term e

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

$$\frac{\cdot \mid \Delta; \Gamma \vdash i \Longrightarrow \tau' \quad \Omega \mid \Delta; \Gamma, x : \tau' \vdash_{\mathbf{P}} P \longrightarrow e \Longleftarrow \tau}{\Omega \mid \Delta; \Gamma \vdash_{\mathbf{P}} \textbf{by } i \textbf{ as } x; P \longrightarrow \texttt{let } x = i \texttt{ in } e \Longleftarrow \tau}$$

$$\frac{\cdot \mid \Delta; \Gamma \vdash i \Longrightarrow [U] \quad \Omega \mid \Delta, X{:}U; \Gamma \vdash_{\mathbf{P}} P \longrightarrow e \Longleftarrow \tau}{\Delta; \Gamma \vdash_{\mathbf{P}} \textbf{unbox } i \textbf{ as } X; P \longrightarrow \texttt{let box } X = i \texttt{ in } e \Longleftarrow \tau}$$

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

$$\frac{\cdot \mid \Delta \vdash; \Gamma \vdash e \Longleftarrow \tau}{\cdot \mid \Delta; \Gamma \vdash_{\mathbf{D}} \textbf{solve } e \longrightarrow e \Longleftarrow \tau}$$

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

$$\frac{\begin{array}{c}\cdot \mid \Delta; \Gamma \vdash i \Longrightarrow \beta \quad \mathsf{cov}\ (\Delta; \Gamma; \beta) = \overrightarrow{(p_k; \theta_k; \Delta_k; \Gamma_k)} \\ \text{for all } k.\ \Omega_k \mid \Delta_k; \Gamma_k \vdash_{\mathbf{P}} P_k \longrightarrow e_k \Longleftarrow [\theta_k]\tau\end{array}}{\bigcup_k \Omega_k \mid \Delta; \Gamma \vdash_{\mathbf{D}} \textbf{split } i \textbf{ as } \overrightarrow{\{\Delta_k; \Gamma_k \vdash P_k\}} \longrightarrow \texttt{case } i \texttt{ of } \overrightarrow{p_k \Rightarrow e_k} \Longleftarrow \tau}$$

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

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

Figure 4.6: The translation from a HARPOON proof script to a Beluga program.

Chapter 5

Conclusion

5.1 Evaluation

One should be able to use Harpoon to prove anything that one could prove in Beluga. As mentioned in Sec. 4.3, we do not prove a completeness theorem, so instead we have replicated a number of case studies originally proven as functional programs in Beluga. (See table 5.1.)

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 [8]	Large development

Table 5.1: 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.

Recreating these case studies in Harpoon was straightforward, since most constructs from Beluga have clear analogues in Harpoon. One challenge was to decide when to use the `suffices` tactic, since most reasoning in Beluga is purely forwards.

Porting these case studies provides us with insight as to future work regarding automation. In the syntactic case studies, proofs 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 general recipe could be automated in whole or in part to simplify the development of similar simple proofs.

5.2 Related work

Although we devoted Chap. 2 to discussing the prior work related to Harpoon, we wish to now make a critical comparison between Harpoon and some of that work.

The Hazelnut system is similar to Harpoon in that its metatheory formally describes partial programs and the user interactions that construct such a program [32]. 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. In Hazelnut, there is a notion of cursor that is absent in Harpoon. The focus of the Harpoon user is always on an (unsolved) subgoal, whereas in Hazelnut, one can use certain actions to move to arbitrary locations in the constructed expression. Harpoon distinguishes clearly between expressions containing holes and expressions free of holes, thanks to the subgoal context Ω that is part of

the typing judgments. This context is also used to drive the selection of the next hole to work on when considering an interactive session $\bar{\alpha}$. The typing judgment for Hazelnut, in contrast, does not show the presence of subgoals in the expression. A context analoguous to Ω is not necessary as a means of finding the next subgoal to work on since the system is equipped with general movement actions.

Abella is similar to the Beluga project more broadly in that it is a domain-specific language using HOAS for mechanizing metatheory [17, 18]. Its theoretical basis differs from Beluga's, however, and 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 VeriML system is grounded in Contextual Modal Type Theory, as is Beluga. However, it differs in its goal, as it seeks to provide a very expressive language for defining new tactics. In our work, we restrict ourselves to a finite set of tactics with clearly defined semantics, whereas in VeriML one can use effects such as state and nontermination in order to build complex decision procedures in addition to more basic tactics. Both VeriML and Beluga follow a two-layer approach: whereas Beluga's object language is contextual LF, VeriML's is $\lambda\mathsf{HOL}_{ind}$, which the authors see as a common core between Coq and HOL provers such as Isabelle and HOL-light. The management of metatheoretic concerns such as substitutions and contexts is low-level: one much explicitly model them, e.g. using lists.

Although much of the discussion in Chap. 2 focused on tactic languages for Coq, all of these differ from Harpoon in their aims. Harpoon seeks simply to provide a form of user interaction together with a well-specified semantics

for those interactions based on an explicit notion of partial program. Ltac, on the other hand, is meant as a language for defining *new* tactics, and these tactics may implement decision procedures or restricted proof search techniques. Given Beluga's focus on mechanized metatheory, it is not clear yet that one needs the ability to define decision procedures or use proof search techniques. This is in fact the reason why we avoid using the word "tactic" to refer to the interactive actions available in Harpoon: the notion of tactic is much broader that what we aim for in Harpoon. The MetaCoq system, in contrast with Ltac, is more foundational. It seeks to provide a formalization of the Coq system within Coq itself. Moreover, it appears more akin to a metaprogramming system than an interactivity layer: one can write MetaCoq programs that are able to look up definitions and inspect them, and whose effects generate new toplevel declarations. Harpoon is humbler, in that it does not try to formalize Beluga within itself, nor does it provide any way to generate new definitions. Instead Harpoon restricts its actions to those that generate proof fragments (terms).

5.3 Final remarks

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 which we have presented shows that all intermediate partial proofs are well-typed with respect to a context of outstanding subgoals to resolve. We have also developed the metatheory of proof scripts, giving a sound translation 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 [42] to enable users to define custom tactics together with correctness guarantees about them.

Appendix A

Harpoon Commands Reference

The information in this appendix is available online at https://beluga-lang.readthedocs.io/en/latest/harpoon/interactive-reference.html but is reproduced here as a way of capturing a snapshot of the features of Harpoon at the time of writing.

This appendix gives a complete list of the interactive commands supported by Harpoon. These commands are divided into two categories: administractive actions (Sec. A.1). and proof actions (Sec. A.2) The former category's actions are used to obtain information from the system, to select different subgoals, and to manipulate the history of proof actions. The latter category's actions solve subgoals and contribute to the construction of proof scripts.

Harpoon is structured hierarchically: a user session consists of a number of proof sessions, each of which contains a number of theorems, each of which contains a number of subgoals. Theorems within a proof session are proven by mutual induction and may refer to each other, but theorems belonging to different sessions may not refer to each other.

A.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 the previous proof tactic.

`redo`

Undoes the effect of a previous `undo`.

`history`

Displays the `undo` history.

`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.

Note: this will drop the current undo history.

`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` ⟨`cmd-session-list`⟩ command.

Note: when selecting a theorem from another session, be aware of the consequences this has on scoping.

`rename`

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

Warning: 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.

`toggle-automation`

Use `toggle-automation AUTO [STATE]` to change the state of proof automation features.

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`.

A.2 Proof actions

These are the actions that manipulate subgoals. In general, they solve the current subgoal, possibly replacing it with zero or more subgoals. In case the action generates exactly one subgoal, it can be understood as transforming the subgoal it operates on.

`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 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.

Note. 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.

Warning. 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.

Bibliography

[1] Andreas Abel, Guillame Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. POPLMark Reloaded: Mechanizing Proofs by Logical Relations. *J. Funct. Program.*, 29:e19, 2019.

[2] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards Certified Meta-Programming with Typed Template-Coq. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving*, volume 10895, pages 20–39. Springer International Publishing, Cham, 2018.

[3] H. P. Barendregt. *Lambda Calculi with Types*, page 117–309. Oxford University Press, Inc., USA, 1993.

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

[5] Mathieu Boespflug and Brigitte Pientka. Multi-level contextual modal type theory. In Gopalan Nadathur and Herman Geuvers, editors, *6th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP'11)*, volume 71 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pages 29–43, 2011.

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

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

[8] Andrew Cave and Brigitte Pientka. Mechanizing Proofs with Logical Relations – Kripke-style. *Mathematical Structures in Computer Science*, 28(9):1606–1638, 2018.

[9] James Cheney and Ralf Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, 2003.

[10] Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In *Proceedings of the 2007 Workshop on Workshop on ML*, ML '07, page 37–46, New York, NY, USA, 2007. Association for Computing Machinery.

[11] Robert L. Constable, Stuart F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, NJ, 1986.

[12] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[13] David Delahaye. A tactic language for the system coq. In Michel Parigot and Andrei Voronkov, editors, *7th International Conference on Logic*

for Programming and Automated Reasoning (LPAR'00), volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000.

[14] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1997.

[15] Amy F. Felty, Alberto Momigliano, and Brigitte Pientka. Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions. *Math. Struct. in Comp. Science*, 28(9):1507–1540, 2018.

[16] Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2 - a survey. *Journal of Automated Reasoning*, 55(4):307–372, 2015.

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

[18] Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *Journal of Automated Reasoning*, 49(2):241–273, 2012.

[19] Jean-Yves Girard. *Proofs and types*. Number 7 in Cambridge tracts in theoretical computer science. Cambridge University Press, Cambridge [England] ; New York, 1989.

[20] Healfdene Goguen, Conor McBride, and James McKinna. Eliminating Dependent Pattern Matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, pages 521–540. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[21] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[22] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, January 1993.

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

[24] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. Mtac2: typed tactics for backward reasoning in Coq. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–31, July 2018.

[25] Gregory Malecha and Jesper Bengtson. Extensible and Efficient Automation Through Reflective Tactics. In Peter Thiemann, editor, *Programming Languages and Systems*, volume 9632, pages 532–559. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[26] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Trans. Comput. Educ.*, 10(4), November 2010.

[27] Conor McBride. Elimination with a Motive. In Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack, and Robert Pollack, editors, *Types for Proofs and Programs*, pages 197–216, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[28] R. Milner and R. S. Bird. The use of machines to assist in rigorous proof [and discussion]. *Philosophical Transactions of the Royal Society of*

London. Series A, Mathematical and Physical Sciences, 312(1522):411–422, 1984.

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

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

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

[32] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: A bidirectionally typed structure editor calculus. *SIGPLAN Not.*, 52(1):86–99, January 2017.

[33] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, September 1989.

[34] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, pages 209–228, New York, NY, 1990. Springer-Verlag.

[35] Frank Pfenning and Carsten Schürmann. 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), pages 202–206. Springer, 1999.

[36] Brigitte Pientka. 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)*, pages 371–382. ACM Press, 2008.

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

[38] Brigitte Pientka and Andreas Abel. Structural recursion over contextual objects. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA'15)*, pages 273–287. Leibniz International Proceedings in Informatics (LIPIcs) of Schloss Dagstuhl, 2015.

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

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

[41] Brigitte Pientka and Joshua Dunfield. Beluga: a framework for programming and reasoning with deductive systems (System Description). In Jürgen Giesl and Reiner Haehnle, editors, *5th International Joint Conference on Automated Reasoning (IJCAR'10)*, Lecture Notes in Artificial Intelligence (LNAI 6173), pages 15–21. Springer, 2010.

[42] Brigitte Pientka, David Thibodeau, Andreas Abel, Francisco Ferreira, and Rébecca Zucchini. 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*, pages 1–13. IEEE, 2019.

[43] Pierre-Marie Pédrot. Ltac2: Tactical Warfare. In *CoqPL*, page 3, 2019.

[44] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1-2):411–440, December 1993.

[45] Tim Sheard and Simon L. Peyton Jones. Template meta-programming for haskell. *SIGPLAN Notices*, 37(12):60–75, 2002.

[46] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020.

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

[48] Antonis Stampoulis and Zhong Shao. Static and user-extensible proof checking. In John Field and Michael Hicks, editors, *39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, pages 273–284. ACM, 2012.

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

[50] Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in agda. In Ralf Hinze, editor, *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*, volume 8241 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 2012.

[51] Roberto Virga. *Higher-order rewriting with dependent types.* PhD thesis, PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, 1999.

[52] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *30th ACM Symposium on Principles of Programming Languages (POPL'03)*, pages 224–235. ACM Press, 2003.

[53] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in Coq. *Journal of Functional Programming*, 25:e12, 2015.