Semi-Automation of Meta-Theoretic Proofs in Beluga

Johanna Schwartzentruber
McGill University
Montreal, Canada
johanna.schwartzentruber@mail.mcgill.ca

Brigitte Pientka
McGill University
Montreal, Canada
bpientka@cs.mcgill.ca

We present a sound and complete focusing calculus for the core of the logic behind the proof assistant Beluga as well as an overview of its implementation as a tactic in Beluga's interactive proof environment Harpoon. The focusing calculus is designed to construct uniform proofs over contextual LF and its meta-logic in Beluga: a dependently-typed first-order logic with recursive definitions. The implemented tactic is intended to complete straightforward sub-cases in proofs allowing users to focus only on the interesting aspects of their proofs, leaving tedious simple cases to Beluga's theorem prover. We demonstrate the effectiveness of our work by using the tactic to simplify proving weak-head normalization for the simply-typed lambda-calculus.

1 Introduction

To establish trust in a software system, we need to begin with establishing trust in the programming language (PL) that is used in its implementation. To model formal systems like PLs, we first need to specify them in a specification logic, and then prove statements about the behaviour of the specified system within another logic, called the reasoning logic. A key challenge in specifying languages is how to model variable bindings, variable renaming, α-conversion, substitution, and reasoning under assumptions. Higher-order abstract syntax (HOAS) offers a solution to these issues, reducing the amount of overhead infrastructure users must construct. On the reasoning side, proofs about these language specifications for even small languages can still involve long and tedious proofs where many sub-cases are trivial, and the only challenging aspect pertains to one or two interesting parts. Our goal is to ease the burden on specifying and proving properties by providing users with automated support based on proof theoretic foundations to discharge trivial, straightforward cases automatically.

There are several HOAS-based systems that offer varying degrees of automation. Abella [14] allows users to develop proofs interactively using a small set of tactics which offer little automation. Hybrid [2, 18] adds HOAS support to the general proof assistants Coq [6] and Isabelle/HOL [20] and consequently is able to harness their automation and tooling powers, at the cost of added work to specify systems [13, 12]. Twelf offers full automation; although users are not able to interact with its prover which does not support backtracking. Most importantly, Twelf and Abella do not produce proof terms and therefore offer no way to verify their proofs independently.

We investigate proof automation within the HOAS-based proof assistant, Beluga [25]. In Beluga, users formalize their systems within the logical framework LF [15] and subsequently prove properties about LF objects in Beluga's reasoning logic: a dependently typed first-order logic. Beluga takes a functional approach; modelling inductive proofs about LF objects

Preliminary Report. Final version to appear in:
LFMTP 2023

as recursive dependently-typed programs, following the Curry-Howard isomorphism. To ease proof developments, a tactic-based prover HARPOON was recently added [11]. Users may now construct proofs interactively using a small set of high-level actions similar to those in Abella, and, upon completion of a proof, will be presented with a proof script which can be translated to a BELUGA program that may be independently type-checked. It is still the case however that much human interaction is required for HARPOON proof developments.

In this paper we present a focusing calculus designed to build uniform proofs over the core of BELUGA's two-level logic, and provide an overview of its implementation as a HARPOON tactic. The tactic is meant to solve simple lemmas and simple cases of proofs, allowing users to concentrate on the interesting aspects of a proof. We have proven that the focusing calculus, presented in Section 2.3, is sound and complete with respect to the cut-free sequent calculus for BELUGA, presented in Section 2.2.2. We have used our tactic on a number of interesting case studies in PL theory, which we summarize in Section 3.2, including type preservation and value soundness for MiniML (without fix points), weak-head normalization for the simply-typed lambda-calculus, and the Church-Rosser theorem for the untyped lambda-calculus. We have seen that they allow for automatic completion of many of the simpler lemmas and subcases of these theorems. We believe automating proof search over the core of BELUGA results in simpler proof developments, making it more appealing to users looking to verify formal systems.

2 Introduction to Beluga

In this section, we present an overview of the BELUGA system. We begin with an informal, followed by a formal, description of its logic, concluding with a presentation of the focusing calculus that we implement.

2.1 Encoding (meta-)theories

We introduce BELUGA informally by demonstrating how theories and their meta-theories are encoded within the system. Throughout this paper we will focus on the proof development of a key lemma that is used to prove weak-head normalization for the simply-typed lambda-calculus using logical relations. The lemma states that reducibility is closed under expansion, i.e. if term `M` steps to term `N` and `N` reduces to type `A`, then `M` does as well. A detailed description of the full mechanization may be found at [9].

We begin by encoding the theory of the simply-typed lambda-calculus within BELUGA's specification logic, the logical framework LF [15] making use of HOAS. We choose an intrinsically-typed representation to simplify our mechanization.

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

```
LF term : tp → type =
| app : term (arr A B) → term A → term B
| abs : tp → (term A → term B) → term (arr A B)
| c : term b ;
```

Next, we define the operational semantics of our lambda terms. For simplicity we do not reduce under abstractions. We observe that object-level substitution is modelled by LF application, as in the type of `beta`.

```
LF step : term A → term A → type =
| beta : step (app (abs A M) N) (M N)
| stepapp : step M M'
            → step (app M N) (app M' N) ;
```

```
LF steps : term A → term A → type =
| id   : steps M M
| sstep : step M M'
           → steps M' M'' → steps M M'' ;
```

To complete the theory's encoding, we also define what it means for a term to halt, i.e. it steps to a value.

```
LF val : term A → type =
| val/c : val c
| val/abs : val (abs A M) ;
```

```
LF halts : term A → type =
| halts/m : steps M M' → val M' → halts M ;
```

To reason about LF objects, we embed them within the computation logic using a modal box (necessity) operator [23]. Users also have the ability to reason about "open" LF objects in Beluga. To do this, we pair each LF object together with the LF context in which it is meaningful [23, 24]. This concept is internalized as a contextual type $\lceil\Psi \vdash P\rceil$ [19]. This contextual type describes a computation-level expression $\mathsf{box}\ (\widehat{\Psi} \vdash R)$ where R is an LF object of type P in the context Ψ. In other words, $\widehat{\Psi}$ describes the free variables in R and corresponds to the erased typing context Ψ.

We present a trivial result about our specified theory to demonstrate how meta-theorems are expressed in Beluga.

```
rec halts_step : [ ⊢step M M'] → [ ⊢halts M'] → [ ⊢halts M] =
fn s, h =>
   let [ ⊢halts/m MS V] = h in let [ ⊢S] = s in [ ⊢halts/m (sstep S MS) V] ;
```

We leave the contextual variables `M`, `M'`, and `M''` implicitly universally quantified as Beluga is able to reconstruct their type. We use Beluga's simple function space to formalize our implication statement. Recall proofs are programs in Beluga, therefore proof development proceeds in a functional manner.

We begin by giving the theorem name and statement, prefixed with the keyword **`rec`**. The proof starts by peeling off the implication antecedents (`fn s, h =>`). Working backwards, we know we must use `halts/m` to construct our desired term as it is currently the only constructor for terms of type `halts`, therefore we must solve its subgoals, namely that there is a value that `M` steps to. We first invert `h` as it has only one possible constructor. This reveals that it is actually the contextual object `[ ⊢ halts/m MS V]` where `MS` and `V` are LF terms of type `steps M' N` and `val N` (for some implicit meta-variable `N`) respectively. It may appear we have every piece of the puzzle required to solve our goal: we have a value that our term `M` steps too. However once we transition to the LF level to build our LF proof term, we do not have access to our computation-level context, in which `s` resides. Therefore, we must first *unbox* said assumption.

The free variables appearing in all the specifications above are treated as implicitly quantified. Beluga infers their types during type reconstruction. As such, users do not supply arguments for such parameters.

2.2 Theoretical foundation

We give a formal presentation of the core of Beluga's two-level logic beginning with its grammar followed by its two-level proof system described using two cut-free sequent calculi. For a full description of the logic, readers may refer to past works [8, 23].

2.2.1 Grammar

Beluga's two-level logic is based on dependent contextual modal type theory (CMTT) [19]. At the core of the specification logic is the logical framework LF [15] which supports encodings using HOAS. We give here a definition that characterizes only canonical (normal) objects as these are the only ones that are meaningful in our setting.

We separate terms into two categories, neutral and normal. We characterize neutral terms to be those that do not cause beta-redexes when they are applied in function application. Terms are classified by types, which are either type constants $\mathbf{a}$ that may be indexed by terms $M_1, \ldots, M_n$, or dependent types.

Atomic Types	P, Q	$::=$	$\mathbf{a}\overrightarrow{M}$	Substitutions	σ	$::=$	$\cdot \mid \mathsf{wk}_\psi \mid \sigma, M$
Types	A, B	$::=$	$P \mid \Pi x{:}A.\, B$	Contexts	Ψ, Φ	$::=$	$\cdot \mid \Psi, x : A \mid \psi$
Neutral Terms	R	$::=$	$x \mid \mathbf{c} \mid R\ N \mid u[\sigma]$	Contextual Variables	X	$::=$	$u[\sigma] \mid \psi$
Normal Terms	M, N	$::=$	$R \mid \lambda x.\, M$				

To support pattern matching on LF objects, we further extend LF with two kinds of contextual variables: meta-variables, written as $u[\sigma]$ and context variables, written as ψ. Context variables allow for abstraction over contexts which is required for recursion over HOAS specifications. Meta-variables allow us to describe "holes" in an LF object. They describe possibly open objects that are paired with a postponed simultaneous substitution σ (by convention written to the right of a term) that is applied as soon as we know what u stands for.

Simultaneous substitutions σ provide a mapping from one context of variables Φ to another Ψ. We do not always make the domain of the substitution explicit, but one can think of the i-th element of σ corresponding to the i-th declaration in Φ. We assume all substitutions are hereditary substitutions [33].

Variables in a contextual LF expression may be bound by one of two contexts. There is the LF context Ψ that holds typings for ordinary variables, and there is the meta-context Δ (introduced below) which holds typings for contextual variables, uniformly denoted by X. Contextual variables include meta-variables and context variables ψ.

In order to uniformly abstract over meta-objects in the computation logic, we lift contextual LF objects to meta-types U and meta-terms C.

Meta Terms	C	$::=$	$(\hat{\Psi} \vdash R) \mid \Psi$	Context Schemas	G	$::=$	$\exists\overrightarrow{(x{:}A_o)}.\ A \mid G + \exists\overrightarrow{(x{:}A_o)}.\ A$
Meta Types	U	$::=$	$(\Psi \vdash P) \mid G$	Meta Contexts	Δ	$::=$	$\cdot \mid \Delta, X : U$
Meta Substitutions	θ	$::=$	$\cdot \mid \theta, C/X$				

The core of our meta language's terms include contextual terms as well as LF contexts. The meta-type $(\Psi \vdash P)$ denotes the type of a meta-variable u and stands for a contextual term. For simplicity, we restrict context schemas G to be constructed from schema elements $\exists\overrightarrow{(x{:}A_o)}.A$ using $+$, where A is an LF type.

We write the single meta-substitution as $[\![C/X]\!]$. In most cases X stands for a meta-variable u and C stands for a contextual object $(\hat{\Psi} \vdash R)$. In this case the substitution gets pushed through λ-expressions until we reach a meta-variable $u[\sigma]$. We then apply the meta-substitution to its associated substitution to obtain σ' before eagerly applying σ' to R. The full definition of meta-substitutions may be found at [8, 23].

On top of contextual LF we have the computational layer, which is used to describe programs that operate on data. The computation types include atomic box-types $\lceil \Psi \vdash P \rceil$, computation level function abstraction, as well as abstraction over contextual objects.

Types	τ	$::=$	$\lceil \Psi \vdash P \rceil \mid \tau_1 \to \tau_2 \mid \Pi^{\Box} X{:}U.\,\tau$
Expressions	E	$::=$	$y \mid \mathsf{box}\ (\hat{\Psi} \vdash R) \mid \mathsf{let\ box}\ X = E_1\ \mathsf{in}\ E_2$
			$\mid \mathsf{fn}\ y.E \mid E_1\ E_2 \mid \lambda^{\Box} X.E \mid E\ (C)$
Computation-level Contexts	Γ	$::=$	$\cdot \mid \Gamma, y : \tau$

Ordinary functions are created using $\mathsf{fn}\ y.E$, while we write $\lambda^{\Box} X.E$ for dependent functions that abstract over meta-objects. We overload the application operation. We write $E_1\ E_2$ for applying the expression E_1 of function type $\tau_1 \to \tau_2$ to an expression E_2. We also write $E\ (C)$ for applying the expression E of type $\Pi^{\Box} X{:}U.\,\tau$ to a contextual object C.

2.2.2 Sequent calculi

We now present the sequent calculus for contextual LF, which is based on the sequent calculus for intuitionistic contextual modal logic presented in [19]. Following the logic programming interpretation of LF, a proof of a proposition encoded as an LF type is an LF term which inhabits said type. We exploit the fact that if B does not depend on x in $\Pi x{:}A.B$, we interpret it as an implication $A \to B$. If x does occur in B, then we treat $\Pi x{:}A.B$ as universal quantification, written as $\Pi \hat{x}{:}A.B$.

All sequents have access to the global signature Σ which we keep implicit. The sequent $\Delta; \Psi \Longrightarrow M : A$ states that M is a proof of the proposition A, or M has type A, using assumptions from the meta-context Δ and the LF context Ψ. As a consequence, we keep in the LF context Ψ not only proof-relevant assumptions that arise from implications (in particular $\to R$), but we also add parameter assumptions that come from ΠR. We distinguish between these two assumptions by writing $x{:}A$ for the former, and $\hat{x} : A$ for the latter. This is emphasized in the init^{Ψ} rule, where only proof-relevant assumptions are used to finish proofs.

In addition to the sequent $\Delta; \Psi \Longrightarrow M : A$, we also have $\Delta; \Psi \Longrightarrow \sigma : \Phi$ which states that the substitution σ witnesses a proof of the propositions in Φ using assumptions from Δ and Ψ. Intuitively, Φ is an LF context containing assumptions $x_i{:}A_i$ and $\hat{x}_j{:}A_j$. Note that we need to construct a proof for the former, but for the latter we are able to determine the witness via unification in practice.

The rules for the sequent calculus for contextual LF are presented in Figure 1, on the next page. The right rules introduce variable declarations into the local context Ψ. However, those introduced via the ΠR rule are simply parameters that are not used during proof search, unlike those introduced via $\to R$. To use a universally quantified assumption (i.e. a dependent function type) as in ΠL we require that M checks against type A, written $\Delta; \Psi \vdash M \Leftarrow A$, in the appropriate contexts. In practice, we do not search for the term M but introduce meta-variables for such universally quantified variables which are later instantiated via unification. In contrast, using an assumption of ordinary function type, as in $\to L$, involves searching for a proof term of type A. Note that in the left rules we replace a neutral term by a neutral term, thus still constructing normal proofs.

In the reflect rule we may use the contextual assumption $(\Phi \vdash P)$ to deduce P in the context Ψ if we can verify Φ. In order to verify Φ we need to find a substitution which maps all the variables in Φ to terms that make sense in Ψ. There are several ways to construct such a substitution, depending on the shape of Φ. If it is empty, we simply use an empty substitution (as P is closed). If it is a context variable ψ and we simply want to use $(\psi \vdash P)$ in a weaker context, we apply the weakening substitution. Otherwise, Φ contains two different kinds of

$\boxed{\Delta; \Psi \Longrightarrow M : A}$ Object M is a proof term for the proof of A in the sequent calculus

$$\frac{\mathbf{c} : A \in \Sigma}{\Delta; \Psi \Longrightarrow \mathbf{c} : A}\ \text{init}^{\Sigma} \qquad \frac{}{\Delta; \Psi, x : A \Longrightarrow x : A}\ \text{init}^{\Psi} \qquad \frac{\Delta; \Psi, \hat{x} : A \Longrightarrow M : B}{\Delta; \Psi \Longrightarrow \lambda x.M : \Pi \hat{x}{:}A.\, B}\ \Pi R$$

$$\frac{\Delta; \Psi, x_1 : \Pi \hat{x}{:}A.\ B \vdash M \Leftarrow A \quad \Delta; \Psi, x_1 : \Pi \hat{x}{:}A.\, B, x_2 : [M/\hat{x}]B \Longrightarrow N : A'}{\Delta; \Psi, x_1 : \Pi \hat{x}{:}A.\, B \Longrightarrow [x_1 M/x_2]N : A'}\ \Pi L$$

$$\frac{\Delta; \Psi, x : A \Longrightarrow M : B}{\Delta; \Psi \Longrightarrow \lambda x.M : A \to B} \to R \qquad \frac{\Delta; \Psi, x_1 : A \to B \Longrightarrow M : A \quad \Delta; \Psi, x_1 : A \to B, x_2 : B \Longrightarrow N : A'}{\Delta; \Psi, x_1 : A \to B \Longrightarrow [x_1 M/x_2]N : A'} \to L$$

$$\frac{\Delta, u : (\Phi \vdash P); \Psi \Longrightarrow \sigma : \Phi \quad \Delta, u : (\Phi \vdash P); \Psi, x : [\sigma]P \Longrightarrow M : A}{\Delta, u : (\Phi \vdash P); \Psi \Longrightarrow [u[\sigma]/x]M : A}\ \text{reflect}$$

$\boxed{\Delta; \Psi \Longrightarrow \sigma : \Phi}$ Object σ is a substitution that witnesses the proof of Φ in the sequent calculus

$$\frac{}{\Delta; \Psi \Longrightarrow \cdot : \cdot}\ \text{sub}_{\text{empty}} \qquad \frac{}{\Delta; \psi, \Psi \Longrightarrow \mathsf{wk}_{\psi} : \psi}\ \text{sub}_{\mathsf{wk}}$$

$$\frac{\Delta; \Psi \Longrightarrow \sigma : \Phi \quad \Delta; \Psi \Longrightarrow N : [\sigma]B}{\Delta; \Psi \Longrightarrow (\sigma, N) : (\Phi, x{:}B)}\ \text{sub}_p \qquad \frac{\Delta; \Psi \Longrightarrow \sigma : \Phi \quad \Delta; \Psi \vdash M \Leftarrow [\sigma]B}{\Delta; \Psi \Longrightarrow (\sigma, M) : (\Phi, \hat{x}{:}B)}\ \text{sub}_u$$

Figure 1: Sequent calculus for contextual LF.

variable declarations: a declaration $x{:}B$ requires proof search in order to find a term N in Ψ of type $[\sigma]B$ to replace x in P; a declaration $\hat{x}{:}A$ stands for a universally quantified variable and does not require proof search. In practice, we can determine it via unification. This different treatment of assumptions reflects their different roles.

We turn our attention to proof search over computations (see Figure 2). Our inference rules are mostly standard for a first-order logic. The $\Box R$ rule is the transition rule between contextual LF and computation-level proofs. In $\Box L$ we *unbox* a boxed assumption, adding it to Δ. Using computation assumptions as in $\Pi^{\Box} L$ and $\to L$ is similar to how contextual LF assumptions are used. To use a universally quantified assumption, as in $\Pi^{\Box} L$, we require that C checks against U. Again, this term C is not explicitly constructed but found instead through unification. We note that meta-objects may only depend on the meta-context, hence we only require that C check against type U in Δ.

We show in [31] that cut and contextual cut are admissible in the computation logic. These results are shown for contextual LF in [19]. Using the admissibility of cut results, we can deduce invertibility of some of the rules in our calculi. Interestingly, the box constructor $\Box$ has an invertible *left* rule, which will have implications in the focusing calculus. We omit the proof terms for readability here.

Lemma 1 (Invertibility in the sequent calculi)**.**
a) (ΠR) If $\Delta; \Psi \Longrightarrow \Pi \hat{x}{:}A.B$ then $\Delta; \Psi, \hat{x} : A \Longrightarrow B$
b) ($\to R$) If $\Delta; \Psi \Longrightarrow A \to B$ then $\Delta; \Psi, x : A \Longrightarrow B$
c) ($\Pi^{\Box} R$) If $\Delta; \Gamma \Longrightarrow \Pi^{\Box} X{:}U.\tau$ then $\Delta, X : U; \Gamma \Longrightarrow \tau$
d) ($\to R$) If $\Delta; \Gamma \Longrightarrow \tau_1 \to \tau_2$ then $\Delta; \Gamma, y : \tau_1 \Longrightarrow \tau_2$
e) ($\Box L$) If $\Delta; \Gamma, y : \lceil \Psi \vdash P \rceil \Longrightarrow \tau$ then $\Delta, X : (\Psi \vdash P); \Gamma, y : \lceil \Psi \vdash P \rceil \Longrightarrow \tau$

$\boxed{\Delta; \Gamma \Longrightarrow E : \tau}$ Object E is a proof term for the proof of τ in the sequent calculus

$$\frac{\Delta, X : U; \Gamma \Longrightarrow E : \tau}{\Delta; \Gamma \Longrightarrow \lambda^{\Box} X.E : \Pi^{\Box} X{:}U.\tau}\ \Pi^{\Box}R \qquad \frac{\Delta \Vdash C \Leftarrow U \quad \Delta; \Gamma, y_1 : \Pi^{\Box} X{:}U.\tau', y_2 : [\![C/X]\!]\tau' \Longrightarrow E : \tau}{\Delta; \Gamma, y_1 : \Pi^{\Box} X{:}U.\tau' \Longrightarrow [y_1\ (C)/y_2]E : \tau}\ \Pi^{\Box}L$$

$$\frac{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \Longrightarrow E' : \tau_1 \quad \Delta; \Gamma, y_1 : \tau_1 \to \tau_2, y_2 : \tau_2 \Longrightarrow E : \tau}{\Delta; \Gamma, y_1 : \tau_1 \to \tau_2 \Longrightarrow [y_1\ E'/y_2]E : \tau}\ \to L$$

$$\frac{\Delta; \Gamma, y : \tau_1 \Longrightarrow E : \tau_2}{\Delta; \Gamma \Longrightarrow \mathsf{fn}\ y.\ E : \tau_1 \to \tau_2}\ \to R \qquad \frac{}{\Delta; \Gamma, y : \tau \Longrightarrow y : \tau}\ \text{init} \qquad \frac{\Delta; \Psi \Longrightarrow R : P}{\Delta; \Gamma \Longrightarrow \mathsf{box}\ (\hat{\Psi} \vdash R) : \lceil \Psi \vdash P \rceil}\ \Box R$$

$$\frac{\Delta, X{:}(\Psi \vdash P); \Gamma, y{:}\lceil \Psi \vdash P \rceil \Longrightarrow E : \tau}{\Delta; \Gamma, y{:}\lceil \Psi \vdash P \rceil \Longrightarrow \mathsf{let\ box}\ X = y\ \mathsf{in}\ E : \tau}\ \Box L$$

Figure 2: Sequent calculus for the computation logic.

2.3 Focused proof system

As a foundation for automating proof search in Beluga we develop a focused sequent calculus over Beluga's two-level logic. For the description of this focused proof system, we omit proof terms when permitted to ease readability. This logic formalizes the proof search procedure that we implement. The loop is fully automatic and therefore requires that non-determinism be handled with ease. The sequent calculus previously presented does not suffice as the rules do not provide any inherent direction for proof search. The rules of the following calculi guide better proof development. The calculus builds uniform proofs [17] by applying all invertible rules first. We then handle non-invertible rules systematically through focusing [3].

The focusing calculus for contextual LF consists of two main phases: a uniform and focusing phase, and is mostly straightforward (see Figure 3). The uniform proof phase consists of applying the invertible rules until we reach an atomic goal (P). During this phase, parameters and assumptions are collected and placed in the LF context. We then try to find a solution by focusing on assumptions from the different contexts. In particular, we iterate through assumptions (in the meta and LF contexts), decomposing each one into the atoms it defines without utilizing any other assumption.

In the transition$^{\Delta}$ rule, using an assumption $(\Phi \vdash Q)$ from Δ to complete a proof requires a simultaneous substitution (σ) to be constructed so that Q makes sense in the current LF context Ψ. We find such a substitution through proof search. When focusing on assumptions from Ψ of function type, we search for a proof of A if our assumption is of non-dependent function type. Otherwise the assumption is of dependent function type, in which case M is found via unification.

Proof search over the reasoning layer proceeds similarly to LF (see Figure 4). We first perform all invertible rules, then we must make a choice on what to focus on. Unlike in LF proof search, proof search over computations requires two separate phases of inversions since the box connective has an invertible left rule. Further, in addition to focusing on the left, we also focus on the right, which corresponds to proof search in LF.

We begin with a uniform right phase which ends with an atomic goal formula, $\lceil \Psi \vdash P \rceil$. From

$\boxed{\Delta;\Psi \overset{u}{\Longrightarrow} A}$ There is a uniform proof of A in the focusing calculus

$$\frac{\Delta;\Psi,\hat{x}{:}A \overset{u}{\Longrightarrow} B}{\Delta;\Psi \overset{u}{\Longrightarrow} \Pi\hat{x}{:}A.\ B}\ \Pi R \qquad \frac{\Delta;\Psi, x : A \overset{u}{\Longrightarrow} B}{\Delta;\Psi \overset{u}{\Longrightarrow} A \to B}\ \to R$$

$$\frac{\Delta(X) = (\Phi \vdash Q) \quad \Delta;\Psi \overset{u}{\Longrightarrow} \sigma : \Phi \quad [\sigma]Q = P}{\Delta;\Psi \overset{u}{\Longrightarrow} P}\ \text{transition}^{\Delta} \qquad \frac{\Psi(x) = A \quad \Delta;\Psi > x : A \rightrightarrows P}{\Delta;\Psi \overset{u}{\Longrightarrow} P}\ \text{transition}^{\Psi}$$

$\boxed{\Delta;\Psi \overset{u}{\Longrightarrow} \sigma : \Phi}$ Object σ is a substitution that witnesses a uniform proof of Φ in the focusing calculus

$$\frac{}{\Delta;\Psi \overset{u}{\Longrightarrow} \cdot : \cdot}\ \text{empty} \qquad \frac{}{\Delta;\psi,\Psi \overset{u}{\Longrightarrow} \mathsf{wk}_{\psi} : \psi}\ \text{sub}_{\mathsf{wk}}$$

$$\frac{\Delta;\Psi \overset{u}{\Longrightarrow} \sigma : \Phi \quad \Delta;\Psi \overset{u}{\Longrightarrow} N : [\sigma]B}{\Delta;\Psi \overset{u}{\Longrightarrow} (\sigma, N) : (\Phi, x{:}B)}\ \text{sub}_p \qquad \frac{\Delta;\Psi \overset{u}{\Longrightarrow} \sigma : \Phi \quad \Delta;\Psi \vdash M \Leftarrow [\sigma]B}{\Delta;\Psi \overset{u}{\Longrightarrow} (\sigma, M) : (\Phi, \hat{x}{:}B)}\ \text{sub}_u$$

$\boxed{\Delta;\Psi > x : A \rightrightarrows P}$ There is a focused proof of P with focus on x in the focusing calculus

$$\frac{}{\Delta;\Psi > x : P \rightrightarrows P}\ \text{init}^{\Psi} \qquad \frac{\Delta;\Psi \overset{u}{\Longrightarrow} A \quad \Delta;\Psi > x' : B \rightrightarrows P}{\Delta;\Psi > x : A \to B \rightrightarrows P}\ \to L$$

$$\frac{\Delta;\Psi \vdash M \Leftarrow A \quad \Delta;\Psi > x' : [M/\hat{x}]B \rightrightarrows P}{\Delta;\Psi > x : \Pi\hat{x}{:}A.\ B \rightrightarrows P}\ \Pi L$$

Figure 3: Focusing calculus for contextual LF.

there we transition to the uniform left phase where we unbox all box-type assumptions in Γ, moving them to Δ. This is done because when we shift levels to LF proof search we only bring with us assumptions that are true across all levels (those in Δ) and computation assumptions do not make sense on the LF level. The sequent for the uniform left phase is novel. We use the symbol $\gg$ as a way to distinguish assumptions that may be of box-type (to the right of $\gg$) from ones that are not (to the left of $\gg$). Recall that the order of assumptions in Γ does not matter, therefore it is acceptable that the order reverses each time we complete a uniform left phase.

Similarly to focusing in LF proof search, if we focus on a universally quantified assumption then we find C through unification. Focusing on the right, i.e. LF proof search, can only be applied if the goal is of box-type. Focusing on the left is standard and commences once we have decomposed the focused formula into its atoms as in the blur rule. At this point, we add the atomic formula to Γ and restart the process from the uniform left stage (to unbox the atomic formula if necessary). In practice, we implement backtracking when focusing. If for example, we cannot find a proof while focusing on the right, we backtrack and try focusing on the left. In practice we also support recursive types, which we treat as atomic computation-level types. These goal types may only be solved by focusing on the left.

We show in [31] that the focusing calculi in Figures 3 and 4 are sound and complete with respect to the sequent calculi presented in Section 2.2.2. The completeness result in particular is

$\boxed{\Delta;\Gamma \overset{R}{\Longrightarrow} \tau}$ There is a uniform right proof of τ in the focusing calculus

$$\frac{\Delta;\Gamma,y:\tau_1 \overset{\mathrm{R}}{\Longrightarrow} \tau_2}{\Delta;\Gamma \overset{\mathrm{R}}{\Longrightarrow} \tau_1 \to \tau_2} \to R \qquad \frac{\Delta,X:U;\Gamma \overset{\mathrm{R}}{\Longrightarrow} \tau}{\Delta;\Gamma \overset{\mathrm{R}}{\Longrightarrow} \Pi^{\Box} X{:}U.\tau} \Pi^{\Box} R \qquad \frac{\Delta;\cdot \gg \Gamma \overset{\mathrm{L}}{\Longrightarrow} \lceil \Psi \vdash P \rceil}{\Delta;\Gamma \overset{\mathrm{R}}{\Longrightarrow} \lceil \Psi \vdash P \rceil} \text{left to right}$$

$\boxed{\Delta;\Gamma \gg \Gamma' \overset{L}{\Longrightarrow} \lceil \Psi \vdash P \rceil}$ There is a uniform left proof of $\lceil \Psi \vdash P \rceil$ in the focusing calculus

$$\frac{\Delta,X:(\Phi \vdash Q);\Gamma \gg \Gamma' \overset{\mathrm{L}}{\Longrightarrow} \lceil \Psi \vdash P \rceil}{\Delta;\Gamma \gg \Gamma',y:\lceil \Phi \vdash Q \rceil \overset{\mathrm{L}}{\Longrightarrow} \lceil \Psi \vdash P \rceil} \Box L \qquad \frac{\tau \neq \lceil \Phi \vdash Q \rceil \quad \Delta;\Gamma,y:\tau \gg \Gamma' \overset{\mathrm{L}}{\Longrightarrow} \lceil \Psi \vdash P \rceil}{\Delta;\Gamma \gg \Gamma',y:\tau \overset{\mathrm{L}}{\Longrightarrow} \lceil \Psi \vdash P \rceil} \text{shift}$$

$$\frac{\Delta;\Psi \overset{u}{\Longrightarrow} P}{\Delta;\Gamma \gg \cdot \overset{L}{\Longrightarrow} \lceil \Psi \vdash P \rceil} \text{level} \qquad \frac{\Gamma(y) = \tau \quad \Delta;\Gamma > y:\tau \Rightarrow \lceil \Psi \vdash P \rceil}{\Delta;\Gamma \gg \cdot \overset{L}{\Longrightarrow} \lceil \Psi \vdash P \rceil} \text{focus to uniform}$$

$\boxed{\Delta;\Gamma > y:\tau \Rightarrow \lceil \Psi \vdash P \rceil}$ There is a focused proof of $\lceil \Psi \vdash P \rceil$ with focus on y in the focusing calculus

$$\frac{\Delta \Vdash C \Leftarrow U \quad \Delta;\Gamma > y':[\![C/X]\!]\tau \Rightarrow \lceil \Psi \vdash P \rceil}{\Delta;\Gamma > y:\Pi^{\Box} X{:}U.\tau \Rightarrow \lceil \Psi \vdash P \rceil} \Pi^{\Box} L$$

$$\frac{\Delta;\Gamma \overset{R}{\Longrightarrow} \tau_1 \quad \Delta;\Gamma > y':\tau_2 \Rightarrow \lceil \Psi \vdash P \rceil}{\Delta;\Gamma > y:\tau_1 \to \tau_2 \Rightarrow \lceil \Psi \vdash P \rceil} \to L \qquad \frac{\Delta;\cdot \gg \Gamma,y':\lceil \Phi \vdash Q \rceil \overset{L}{\Longrightarrow} \lceil \Psi \vdash P \rceil}{\Delta;\Gamma > y':\lceil \Phi \vdash Q \rceil \Rightarrow \lceil \Psi \vdash P \rceil} \text{blur}$$

Figure 4: Focusing calculus for the computation logic.

interesting as it requires an intermediate result stating that it does not matter in which context box-type assumptions appear. That is, if there is a proof in our sequent calculus of $\lceil \Psi \vdash P \rceil$ (possibly) using some assumption $y:\lceil \Phi \vdash Q \rceil$ in Γ then there is also a proof of $\lceil \Psi \vdash P \rceil$ where y is omitted but under the added assumption $X:(\Phi \vdash Q)$ in Δ. Given a Δ and Γ, $\Gamma^{-}_{\Delta,\Gamma}$ denotes Γ without any assumptions of box-type, and $\Delta^{+}_{\Delta,\Gamma}$ denotes Δ extended with the (unboxed) boxed assumptions from Γ.

Theorem 1 (Soundness)**.**
a) If $\Delta;\Psi \overset{u}{\Longrightarrow} A$ *then* $\Delta;\Psi \Longrightarrow A$
b) If $\Delta;\Psi \overset{u}{\Longrightarrow} \sigma:\Phi$ *then* $\Delta;\Psi \Longrightarrow \sigma:\Phi$
c) If $\Delta;\Psi > x:A \rightrightarrows P$ *then* $\Delta;\Psi,x:A \Longrightarrow P$
d) If $\Delta > X:U;\Psi \rightrightarrows P$ *then* $\Delta,X:U;\Psi \Longrightarrow P$
e) If $\Delta;\Gamma \overset{R}{\Longrightarrow} \tau$ *then* $\Delta;\Gamma \Longrightarrow \tau$
f) If $\Delta;\Gamma \gg \Gamma' \overset{L}{\Longrightarrow} \lceil \Psi \vdash P \rceil$ *then* $\Delta;\Gamma,\Gamma' \Longrightarrow \lceil \Psi \vdash P \rceil$
g) If $\Delta;\Gamma > y:\tau \Rightarrow \lceil \Psi \vdash P \rceil$ *then* $\Delta;\Gamma,y:\tau \Longrightarrow \lceil \Psi \vdash P \rceil$

Theorem 2 (Completeness)**.**
a) If $\Delta;\Psi \Longrightarrow A$ *then* $\Delta;\Psi \overset{u}{\Longrightarrow} A$
b) If $\Delta;\Psi \Longrightarrow \sigma:\Phi$ *then* $\Delta;\Psi \overset{u}{\Longrightarrow} \sigma:\Phi$

c) If $\Delta;\Psi \Longrightarrow P$ *then* $\Delta;\Psi > x : A \rightrightarrows P$ *for some* $A \in \Psi$ *or* $\Delta > X : U;\Psi \rightrightarrows P$ *for some* $U \in \Delta$
d) If $\Delta;\Gamma \Longrightarrow \tau$ *then* $\Delta^+_{\Delta,\Gamma};\Gamma^-_{\Delta,\Gamma} \overset{R}{\Rightarrow} \tau$
e) If $\Delta;\Gamma \Longrightarrow \lceil \Psi \vdash P \rceil$ *then* $\Delta^+_{\Delta,\Gamma};\Gamma^-_{\Delta,\Gamma} \gg \cdot \overset{L}{\Rightarrow} \lceil \Psi \vdash P \rceil$
f) If $\Delta;\Gamma \Longrightarrow \lceil \Psi \vdash P \rceil$ *then* $\Delta^+_{\Delta,\Gamma};\Psi \overset{u}{\Rightarrow} P$ *or* $\Delta^+_{\Delta,\Gamma};\Gamma^-_{\Delta,\Gamma} > y : \tau \Rightarrow \lceil \Psi \vdash P \rceil$ *for some* $y : \tau \in \Gamma^-_{\Delta,\Gamma}$

The soundness proofs proceed by mutual structural induction on the given derivations and are straightforward. Proving completeness however is more involved and depends on several minor lemmas, consisting mostly of postponement results and a variation of the reflect rule for computation-level focusing, all proven with straightforward induction. The completeness proof proceeds again by mutual structural induction on the given derivations. In part a) the base case (i.e. the derivation is init$^\Psi$) is simplified by showing that the init$^\Psi$ rule is admissible under the addition of the rule $\Delta;\Psi, x : P \Longrightarrow P$. The remaining cases are then straightforward. Part b) is trivial. Part c) must be done by case analysis on the induction hypothesis but is otherwise straightforward. The base case in part d) is also made simpler by showing that the init$^\Gamma$ rule is admissible under the added assumption $\Delta;\Gamma, y : \lceil \Psi \vdash P \rceil \Longrightarrow \lceil \Psi \vdash P \rceil$. The cases in part d) where the derivation is either $\rightarrow R$, $\rightarrow L$, or $\Pi^\square L$ must be done by case analysis on whether or not τ_1, τ_2, or $[\![C/X]\!]\tau'$ (respectively) are atomic box-types. This is because the result of $\Delta^+_{\Delta,\Gamma}$ and $\Gamma^-_{\Delta,\Gamma}$ in the induction hypotheses depends on whether or not these assumptions are boxed. If they are boxed they will appear unboxed in $\Delta^+_{\Delta,\Gamma}$ and omitted from $\Gamma^-_{\Delta,\Gamma}$, otherwise they remain in $\Gamma^-_{\Delta,\Gamma}$. Part e) also involves case analysis on some subgoals, depending on if the assumptions in Γ are boxed or not. Some cases in part f) involve multiple case analyses- once on the result of the induction hypothesis, and once on the shape of the assumptions in Γ. The entire proof is given in [31].

3 Automation tactic

We present an overview of our automation tactic which implements the focusing calculi from Section 2.3. We begin with a walk-through of the theorem prover followed by a summary of its performance on various case studies from PL theory.

3.1 Example

To demonstrate the capabilities of the (meta-)theorem provers we examine how Beluga automatically proves that reduction is closed under expansion, a key lemma needed to show weak-head normalization for the simply-typed lambda-calculus. Specifically, we prove that if term `M` steps to term `N` and `N` reduces to type `A`, then `M` does as well.

Building upon the theory presented in Section 2.1, we encode the notion of reducibility using a logical relation by categorizing terms by the type they reduce to. We formalize the predicate in Beluga's reasoning logic, as it requires a strong, computational function space unlike the weak function space of LF. Inductive properties about contextual objects are defined using indexed recursive types [8]. We encode the set of reducible terms using the recursive type `Reduce`, which is stratified by its index `tp` [16].

```
stratified Reduce : {A:[ ⊢ tp]}{M:[ ⊢ term A]} ctype =
| I   : [ ⊢ halts M] → Reduce [ ⊢ b ] [ ⊢ M]
| Arr : [ ⊢ halts M]
          → ({N:[ ⊢ term A]} Reduce [ ⊢ A] [ ⊢ N] → Reduce [ ⊢ B ] [ ⊢ app M N])
            → Reduce [ ⊢ arr A B ] [ ⊢ M]
;
```

To begin our theorem, we load the Beluga file containing our LF and computation-level signatures into Harpoon. Harpoon then takes as input a theorem name and statement, and index of induction variable specified by its position in the overall goal, counted from left-to-right:

```
Name of theorem: bwd_closed
Statement of theorem: {A:[ ⊢ tp]} {M: [ ⊢ term A]} {M’: [ ⊢ term A]} [ ⊢ step M M’]
                          → Reduce [ ⊢ A] [ ⊢ M’] → Reduce [ ⊢ A] [ ⊢ M]
Induction order (empty for none): 1
```

Once loaded, users may simply invoke the tactic `auto`[1] to prove the lemma. Behind the scenes, Beluga gives the specified induction variable to our solver which then performs induction on said variable and generates the respective induction hypotheses for each subgoal. It then performs a round of inversions and bounded depth-first proof search on each produced subgoal. Users have the option of choosing their own depth bound by providing an argument to `auto`, otherwise it is set to 3. The proof search algorithm closely implements the focusing calculus presented in Section 2.3. As it searches for a proof, the solver also constructs proof terms in the form of LF and computation-level terms, and LF substitutions. If the solver can prove each case, the constructed proof term is presented to the user. If it cannot prove all subgoals, users will have to manually split on the specified variable (with the `split` tactic) and then call `auto` on each applicable case. In these instances, since no induction variable is specified, the solver will only perform inversions and bounded search. We show below the proof script that is being generated by invoking only the `auto` tactic.

```
proof bwd_closed : {A : ( ⊢ tp)} {M : ( ⊢ term A)}{M’ : ( ⊢ term A)}
                     [ ⊢ step M M’] → Reduce [ ⊢ A] [ ⊢ M’] → Reduce [ ⊢ A] [ ⊢ M] =
/ total 1 /
intros
{ A : ( ⊢ tp), M : ( ⊢ term A), M’ : ( ⊢ term A)
| x : [ ⊢ step M M’], x1 : Reduce [ ⊢ A] [ ⊢ M’]
; solve
    let [ ⊢ Y] = x in
    case [ ⊢ A] of
    | [ ⊢ b] =>
      let (I x2 : Reduce [ ⊢ b] [ ⊢ M’]) = x1 in
      let ([ ⊢ halts/m X3 X4] : [ ⊢ halts M’]) = x2 in
      let [ ⊢ val/c] = [ ⊢ X4] in I [ ⊢ halts/m (sstep Y X3) val/c]
    | [ ⊢ arr X X1] =>
      let (Arr x2 x3 : Reduce [ ⊢ arr X X1] [ ⊢ M’]) = x1 in
      let ([ ⊢ halts/m X5 X6] : [ ⊢ halts M’]) = x2 in
      let [ ⊢ val/abs ] = [ ⊢ X6] in
      Arr [ ⊢ halts/m (sstep Y X5) (val/abs )]
        (mlam N => fn y =>
          bwd_closed [ ⊢ X1] [ ⊢ app M N] [ ⊢ app M’ N] [ ⊢ stepapp Y] (x3 [ ⊢ N] y)) } ;
```

[1]In Harpoon, this tactic is invoked using `inductive-auto-solve`. See https://beluga-lang.readthedocs.io/en/latest/harpoon/proof-automation.html#inductive-auto-solve for a description on how to use the tactic.

The Beluga program that our solvers construct is given in lines 8 - 20, which gets spliced into the overall proof script. From the given proof script, we can extract a complete Beluga program [11]. Previously, this lemma could be proved with no less than 10 Harpoon tactic calls (see also [11]).

The proof begins with the automatic application of the `intros` tactic, which performs the uniform right phase. The rest of the proof is constructed using Beluga's theorem provers. It first unboxes `x` (line 8), concluding the uniform left phase. Since an induction variable was specified, the algorithm immediately splits on the variable. After a split, there is a round of inversions for each produced subgoal followed by bounded focused proof search. In the first case (`A = b`) after the inversions (line 13), the computation-level solver focuses on the constructor `I` and solves its subgoal by providing the appropriate boxed LF proof term (`halts/m (sstep Y X3) val/c`) which it finds again through focused proof search, but this time on the LF level. In the second case (`A = arr X X1`), the solver focuses on `Arr` which requires solving two subgoals, one of which is of function type. It is here where the solver focuses on the induction hypothesis to solve a subgoal. If the solver focuses on the "wrong" assumption, it will execute backtracking and continue focusing until it has exhausted all possible branches on the bounded search tree.

3.2 Evaluation

The proof-search procedures behind our automation tactic is the beginning of the implementation of the focusing calculi presented in Section 2.3. There are several areas of incompleteness that may be improved in the future [31]. Nevertheless, the tactic is able to prove a number of interesting theorems both semi- and fully-automatically. We provide a summary of these case studies here.

Case study	Automation	Difficulty	Interesting proof features
MiniML/fix type preservation	Full	Advanced	Solving substitutions, I.H. appeal, inversions
MiniML/fix value soundness	Full	Basic	I.H. appeal
STLC weak-head normalization lemmas	Full	Intermediate	Inversions, I.H. appeal, higher-order solving
STLC type uniqueness	Partial	Basic	I.H. appeal, inversions
Untyped λ-calculus reduction lemmas	Full/Partial	Basic	I.H. appeal

Table 1: Overview of case studies.

We categorize our case studies by the amount of automation that may be successfully applied. We say full automation is applied when only `auto` is used to complete a proof (as in the previous example). Partial automation is used on induction proofs when not all the subgoals fall within the prover's applicable subset. In these instances, `auto` is used after a variable split has been manually made, and only on a portion of the subgoals. All proofs proceed by induction along with various features that we have outlined.

We use our tactic to prove key lemmas required to show weak-head normalization for the simply-typed lambda-calculus. The backwards closed lemma is particularly interesting as it requires higher-order function type solving. Type uniqueness for the simply-typed lambda-calculus can be proven semi-automatically as two of its cases involve parameter variables and context block schemas. We prove all lemmas regarding ordinary reduction for the untyped lambda-calculus automatically, and all but one lemma regarding parallel reduction automatically. These lemmas are used to prove equivalence of the ordinary and parallel reductions, and ultimately the Church-Rosser theorem for each reduction procedure. For MiniML without fixpoints, we are able to prove type preservation and value soundness fully-automatically. Preservation in particular showcases the solvers ability to solve for substitutions [31].

Beluga's proving power does not yet surpass that of Twelf's. However, Beluga is able to reason directly using logical relations, unlike Twelf. Certain properties, like normalization theorems, are most commonly proven using logical relations. In Twelf, users must find alternative proof methods [30, 1], which may be conceptually different from on-paper formulations and require more work from the user to construct additional machinery. In Beluga, such logical relations may be directly translated from on-paper formulations and their proofs become simplified with the use of our automation tactic.

4 Related work

4.1 Proof environments based on HOAS

Twelf [22] currently provides the most automation out of all HOAS-based proof assistants designed to formalize PLs. It fully automatically proves many theorems such as type-preservation for MiniML, Church-Rosser for the simply typed lambda-calculus, and cut-admissibility for first-order logic [29, 28]. It essentially has a simple loop that splits on an assumption based on a heuristic, generates induction hypothesis, and tries to prove a given goal by bounded depth-first proof search. If the last step fails, it will again split on an assumption heuristically picking one. This has proven remarkably effective. However, the kind of properties that can be stated is more restrictive. For example, Twelf's meta-logic permits only theorems expressed as Π_2 statements and lacks support for recursive types. Therefore it cannot, for example, directly deal with proofs that proceed via logical relations. More importantly, Twelf's prover does not produce proof terms and therefore provides no way to verify its proofs. Even worse, if the meta-theorem prover fails to find a proof automatically, there is no possibility for the user to pick up the pieces and manually proceed.

Abella is another HOAS-based proof assistant with the capabilities to mechanize meta-theoretic proofs about PLs [14]. Semi-automation exists in Abella provided in the form of tactics that are similar to the ones previously implemented in Harpoon. Contexts and (simultaneous) substitutions in Abella are not treated as first-class like they are in Beluga. This means that in order to use such constructs, users must manually define them and any properties they wish to utilize about them (e.g. context weakening). These can further add to the complexity of formal theorem proving. As Twelf, Abella also does not construct proof objects during proof development. Moreover, the tactics used in constructing proofs in Abella concentrate on "small" steps and there is presently no analogue to the proof search tactic that we describe in this paper. However, we believe that similar semi-automation could be added to Abella to discharge straightforward cases.

4.2 General proof environments

The Hybrid system aims to bring full HOAS support to general proof environments like Coq and Isabelle/HOL [2, 18]. These systems often have more users compared to specialized systems and thus more reason to provide tooling and automation. Both systems employ the use of tacticals, which allow users to create their own tactics specific to their mechanizations.

In Coq, users write decision procedures in a tactic language like LTac [10]. There are general automated proof search tactics (auto, eauto, iauto and jauto) but they do not conduct any case analysis (including inversions), inductions, or rewritings, and are intended to finish a proof instead of complete it entirely [26]. Coq also offers many libraries to assist with building and verifying infrastructure needed in PL mechanizations. Autosubst may be used as a library for example to automatically generate parallel substitution operations for a custom type and prove the related substitution lemmas [27]. The libraries Ott [32] and LNgen [4] may be used together to generate locally nameless definitions from a specification and provide the corresponding recursion schemes and infrastructure lemmas. However these tools are limited as they only automate trivial lemmas.

For much of its automation, Isabelle elicits the help of several external solvers. Sledgehammer [21], for example, takes a goal and heuristically chooses from Isabelle's libraries containing various lemmas, definitions, and axioms, a few hundred applicable ones to perform search over. Then, translates the goal and each of these assumptions to SMT (first-order logic) and sends the query off to an external SMT or resolution-based solver. In its own system, Isabelle performs various general-purpose proof search methods which help discharge simple parts of a proof allowing users to focus on the main ones [7]. They also have several strengthened endgame tactics which are meant to finish a proof but provide no hints upon failing.

Despite the abundance of tooling in these systems, some specialized systems (Beluga and Twelf for example) offer more automation for PL mechanizations. This is because these systems have fixed specification logics, so automatic proof procedures can be more intricate. Hybrid on the other hand allows for encodings of various specification logics, therefore proof procedures are more difficult as they must be customized to work for different logics.

5 Conclusion

In closing, we have presented the theorem and meta-theorem provers behind Beluga, a specialized proof assistant with sophisticated built-in support for specifying formal systems. These provers perform two-leveled proof search over a core subset of Beluga's logic which allows for the automatic completion of many simple lemmas and cases of PL theory proofs. Users of Beluga may now bypass simple proofs and focus their energy on the interesting cases. Along with our implementation, we provide a theoretical foundation for our solvers in the form of a cut-free sequent calculus, which is easy to understand, and a sound and complete focusing calculus, which closely reflects our implementation. These provide us with a way to study our implementation and ensure its correctness.

Our next steps are to expand the solver so that its proving capabilities are equivalent to that of the logic presented in this paper [31]. After that we plan to add support for context block schemas, and substitution and parameter variables, which should bring its proving power up to that of Twelf's. Finally, it would be interesting to extend the focusing calculus (and implementation) with automatic induction, similar to [5].

References

[1] Andreas Abel (2008): *Normalization for the Simply-Typed Lambda-Calculus in Twelf.* In: *LFM '04*, 199, pp. 3–16, doi:10.1016/j.entcs.2007.11.009.

[2] Simon Ambler, Roy L. Crole & Alberto Momigliano (2002): *Combining Higher Order Abstract Syntax with Tactical Theorem Proving and (Co)Induction.* In: *Proc. TPHOLs '02*, *LNCS* 2410, Springer-Verlag, p. 13–30, doi:10.1007/3-540-45685-6_3.

[3] Jean-Marc Andreoli (1992): *Logic Programming with Focusing Proofs in Linear Logic.* *J. Log. Comput.* 2(3), pp. 297–347, doi:10.1093/logcom/2.3.297.

[4] Brian E. Aydemir & Stephanie Weirich (2010): *LNgen: Tool Support for Locally Nameless Representations.* Technical Report, University of Pennsylvania. Available at `https://repository.upenn.edu/cis_reports/933`.

[5] David Baelde, Zachary Snow & Dale Miller (2010): *Focused Inductive Theorem Proving.* In Jürgen Giesl & Reiner Haehnle, editors: *IJCAR'10*, LNAI 6173, Springer, pp. 278–292, doi:10.1007/978-3-642-14203-1_24.

[6] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series (TTCS), Springer Berlin, Heidelberg, doi:10.1007/978-3-662-07964-5.

[7] Jasmin Christian Blanchette, Lukas Bulwahn & Tobias Nipkow (2011): *Automatic Proof and Disproof in Isabelle/HOL.* In Cesare Tinelli & Viorica Sofronie-Stokkermans, editors: *FroCoS*, *LNAI* 6989, Springer Berlin Heidelberg, pp. 12–27, doi:10.1007/978-3-642-24364-6_2.

[8] Andrew Cave & Brigitte Pientka (2012): *Programming with Binders and Indexed Data-Types.* In: *Proc. POPL '12*, ACM, p. 413–424, doi:10.1145/2103656.2103705.

[9] Andrew Cave & Brigitte Pientka (2018): *Mechanizing proofs with logical relations – Kripke-style.* *Math. Struct. Comput. Sci.* 28(9), pp. 1606 – 1638, doi:10.1017/S0960129518000154.

[10] David Delahaye (2000): *A Tactic Language for the System Coq.* In Michel Parigot & Andrei Voronkov, editors: *LPAR*, Springer Berlin Heidelberg, pp. 85–95, doi:10.1007/3-540-44404-1_7.

[11] Jacob Errington, Junyoung Jang & Brigitte Pientka (2021): *Harpoon: Mechanizing Metatheory Interactively: (System Description).* In André Platzer & Geoff Sutcliffe, editors: *Automated Deduction - CADE 28*, *LNAI* 12699, Springer, Cham, pp. 636–648, doi:10.1007/978-3-030-79876-5_38.

[12] Amy Felty, Alberto Momigliano & Brigitte Pientka (2015): *The Next 700 Challenge Problems for Reasoning with Higher-Order Abstract Syntax Representations.* *J. Autom. Reasoning* 55(4), p. 307–372, doi:10.1007/s10817-015-9327-3.

[13] Amy Felty & Brigitte Pientka (2010): *Reasoning with Higher-Order Abstract Syntax and Contexts: A Comparison.* In: *Proc. ITP '10*, *LNCS* 6172, Springer-Verlag, p. 227–242, doi:10.1007/978-3-642-14052-5_17.

[14] Andrew Gacek (2008): *The Abella Interactive Theorem Prover (System Description).* In: *IJCAR'08*, *LNAI* 5195, Springer Berlin Heidelberg, pp. 154–161, doi:10.1007/978-3-540-71070-7_13.

[15] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A Framework for Defining Logics.* *Journal of the ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

[16] Rohan Jacob-Rao, Brigitte Pientka & David Thibodeau (2018): *Index-Stratified Types.* In Hélène Kirchner, editor: *FSCD 2018*, *LIPIcs* 108, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 19:1–19:17, doi:10.4230/LIPIcs.FSCD.2018.19.

[17] Dale A. Miller, Gopalan Nadathur, Frank Pfenning & Andre Scedrov (1991): *Uniform Proofs as a Foundation for Logic Programming.* *Ann. Pure Appl. Log.* 51, pp. 125–157, doi:10.1016/0168-0072(91)90068-W.

[18] Alberto Momigliano, Alan J. Martin & Amy P. Felty (2008): *Two-Level Hybrid: A System for Reasoning Using Higher-Order Abstract Syntax.* In: *Proc. LFMTP '07*, *ENTCS* 196, pp. 85–93, doi:10.1016/j.entcs.2007.09.019.

[19] Aleksandar Nanevski, Frank Pfenning & Brigitte Pientka (2008): *Contextual Modal Type Theory. TOCL* 9(3), doi:10.1145/1352582.1352591.

[20] Lawrence C. Paulson (1994): *Isabelle: A Generic Theorem Prover.* Springer-Verlag, doi:10.1007/BFb0030541.

[21] Lawrence C. Paulson & Kong Woei Susanto (2007): *Source-Level Proof Reconstruction for Interactive Theorem Proving.* In Klaus Schneider & Jens Brandt, editors: *TPHOLs '07*, *LNCS* 4732, Springer Berlin Heidelberg, pp. 232–245, doi:10.1007/978-3-540-74591-4_18.

[22] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf — A Meta-Logical Framework for Deductive Systems.* In H. Ganzinger, editor: *CADE-16*, *LNAI* 1632, Springer, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[23] Brigitte Pientka (2008): *A Type-Theoretic Foundation for Programming with Higher-Order Abstract Syntax and First-Class Substitutions.* In George C. Necula & Philip Wadler, editors: *Proc. POPL '08*, 43, ACM, p. 371–382, doi:10.1145/1328897.1328483.

[24] Brigitte Pientka & Jana Dunfield (2008): *Programming with proofs and explicit contexts.* In: *Proc. PPDP '08*, PPDP '08, ACM, pp. 163–173, doi:10.1145/1389449.1389469.

[25] Brigitte Pientka & Jana Dunfield (2010): *Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description).* In Jürgen Giesl & Reiner Hähnle, editors: *IJCAR '10*, *LNAI* 6173, Springer Berlin, Heildelberg, pp. 15–21, doi:10.1007/978-3-642-14203-1_2.

[26] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach & Brent Yorgey (2022): *Programming Language Foundations.* *Software Foundations* 2, Electronic textbook. Available at `http://softwarefoundations.cis.upenn.edu`. Version 6.2.

[27] Steven Schäfer, Tobias Tebbi & Gert Smolka (2015): *Autosubst: Reasoning with de Bruijn Terms and Parallel Substitutions.* In Christian Urban & Xingyuan Zhang, editors: *ITP*, Springer International Publishing, pp. 359–374, doi:10.1007/978-3-319-22102-1_24.

[28] Carsten Schürmann (2000): *Automating the Meta Theory of Deductive Systems.* Ph.D. thesis, Carnegie Mellon University. Available at `https://www.cs.cmu.edu/~rwh/students/schuermann.pdf`.

[29] Carsten Schürmann & Frank Pfenning (1998): *Automated Theorem Proving in a Simple Meta-Logic for LF.* In Claude Kirchner & Hélène Kirchner, editors: *Proc. CADE-15*, Springer-Verlag LNCS, pp. 286–300, doi:10.1007/BFb0054266.

[30] Carsten Schürmann & Jeffrey Sarnat (2008): *Structural Logical Relations.* In: *LICS*, IEEE Computer Society, pp. 69–80, doi:10.1109/LICS.2008.44.

[31] Johanna Schwartzentruber (2023): *Semi-Automation of Meta-Theoretic Proofs.* Master's thesis, McGill University.

[32] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar & Rok Strniša (2007): *Ott: Effective Tool Support for the Working Semanticist.* In: *Proc. ICFP '07*, ICFP '07, ACM, p. 1–12, doi:10.1145/1291151.1291155.

[33] Kevin Watkins, Iliano Cervesato, Frank Pfenning & David Walker (2004): *A Concurrent Logical Framework: The Propositional Fragment.* In Stefano Berardi, Mario Coppo & Ferruccio Damiani, editors: *TYPES*, Springer Berlin Heidelberg, pp. 355–377, doi:10.1007/978-3-540-24849-1_23.