Parsing, Lexical Scoping and Incremental Development for a Dependently-Typed Programming Language

Marc-Antoine Ouimet

School of Computer Science

McGill University

Montreal, Quebec, Canada

August, 2024

A thesis submitted to McGill University in partial fulfilment

of the requirements of the degree of

Master of Science

Contents

Abstract

Beluga is a functional programming language and proof assistant for specifying formal systems in contextual LF, an extension of the Edinburgh Logical Framework, and mechanically proving theorems about them using recursive programs. To facilitate the incremental development of proofs with commands and automation tactics, the Harpoon interactive proof environment is subsequently implemented as a read-eval-print loop with structural editing features over Beluga programs. Due to architectural limitations in the implementation of Beluga and Harpoon, top-down and out-of-order proof development sessions can lead to invalid proof states and unsound translated programs.

This thesis reports on technical challenges and solutions to soundly implementing the structural editing of proofs, including the navigation between proof holes, with a main focus on syntactic analysis and the early phases of semantic analysis. Aspects of programming language syntax design are explored to support context-sensitive parsing of user-defined prefix, infix and postfix operators with a two-phase parser. Then, name resolution for Beluga is rectified with the implementation of a uniform referencing environment representation for indexing programs with separate contexts for different classes of variables. Finally, the revised parser and name resolution phases are integrated into Harpoon to ensure the state of identifiers in scope at any given proof hole is sound with respect to where the hole occurs.

Résumé

Beluga est un langage de programmation fonctionnelle et un assistant de preuve pour spécifier des systèmes formels dans LF contextuel, une extension du Edinburgh Logical Framework, et prouver mécaniquement des théorèmes à propos d'eux au moyen de programmes récursifs. Pour faciliter le développement incrémental de preuves, l'environnement interactif de preuve Harpoon est implémenté en tant que read-eval-print loop avec des fonctionnalités d'édition structurelle de programmes écrits dans Beluga. À cause de limitations architecturales dans l'implémentation de Beluga et Harpoon, le développement de preuves vertical ou dans le désordre peuvent entraîner l'invalidation des états de preuve et la génération de programmes incorrects.

Cette thèse présente les défis techniques et les solutions à l'implémentation cohérente d'édition structurelle avec la navigation d'un endroit à l'autre dans des preuves incomplètes. Un analyseur syntaxique contextuel est réalisé pour supporter la définition d'opérateurs préfixes, infixes et postfixes par l'utilisateur. Ensuite, la résolution de noms pour Beluga est rectifiée avec l'implémentation d'un environnement de référencement uniforme pour l'indexation de programmes définis par rapport à classes disjointes de variables. Enfin, ces systèmes sont intégrés dans Harpoon pour garantir que l'état des identifiants visibles à n'importe quel endroit dans une preuve incomplète est cohérent avec son emplacement.

Acknowledgements

I would like to thank all those who have helped me directly or indirectly in the completion of this thesis. Specifically, I want to thank Antoine Gaulin and Johanna Schwartzentruber for their feedback on early versions of chapter 1 and section 3.1 during writing group meetings. I also want to thank my sister Marie-Laurence Ouimet for bearing with my convoluted ramblings and code explanations on the whiteboard, as well as for reading parts of this thesis and providing feedback. Last but not least, I want to thank my advisor, Brigitte Pientka, for her sustained support and trust throughout the development of the changes I pitched for the Beluga system.

Contributions

Marc-Antoine Ouimet contributed the entirety of this thesis, with feedback from peers and his supervisor as outlined in the acknowledgments.

Figure 2.1 is adapted from [20].

Figure 2.2 is adapted from earlier works [30, 23, 11, 21] on BELUGA.

List of Figures

List of Abbreviations

Chapter 1

Introduction

Beluga [43] is a dependently-typed programming language for specifying and proving properties about formal systems using contextual modal type theory [30]. It leverages the Edinburgh Logical Framework (LF) [24], extended with explicit contexts, contextual objects and substitutions [3, 11], which allow for logic reasoning involving open terms. Using higher-order abstract syntax (HOAS) [37], elegant and succinct datatype definitions can be made in LF to subsequently prove theorems pertaining to programming languages and λ-calculi. These proofs are encoded as programs written in a functional style, together with checkers to ensure they always terminate and cover all possible cases [18, 42]. Propositions then correspond to types, and well-founded reasoning by induction corresponds to recursion with totality and coverage checking. With this framework, Beluga has notably been used to mechanically verify proofs for the type safety of System F [1], termination of weak-head normalization for the simply-typed lambda calculus [11], and type preservation for a session-typed system based on classical linear logic [44]. Extensions to Beluga have facilitated the development of proofs by providing interactive tools for manipulating and constructing proof terms.

Interactive theorem proving is the computer-assisted process of finding valid proofs for

propositions in a logic system. Proof assistants have been implemented and successfully leveraged to prove fundamentals of mathematics and type theories. Agda [2, 33, 5], Coq [46, 9], Isabelle [32] and Lean [29] are among the most prevalent proof assistants used in the formalization of programming languages. Functionalities of typical interactive theorem provers range from read-eval-print loops (REPLs) allowing for structural editing and code inspection to automated proof search to generate valid programs.

Harpoon [19] is a command-line frontend to Beluga for proving theorems interactively. It provides a proof development workflow closer to proofs on paper, with built-in commands replicating proof techniques such as case analyses and appeals to induction hypotheses. Additional administrative commands are supported to navigate through the history of input commands, and to checkout different holes in proofs declared elsewhere in Beluga signatures. Upon closing interactive sessions, Harpoon produces verbose yet simple tree-shaped proof scripts which can be translated into well-formed Beluga programs. Harpoon's design is largely inspired by Coq's interactive proof mode using tactics [16] to solve unproven goals. Crucially, Harpoon is a system integrated into Beluga's core functionalities in such a way that the overall proof state does not have to be reconstructed from scratch on every command, which is akin to structural editing.

Structural (or projectional) editing in programming language tooling is the functionality of a software editor that allows the user to manipulate abstract syntax trees (ASTs) directly. This is in contrast with plain text editors in which users may only edit textual representations of their programs, that then subsequently require parsing to be interpreted. The Hazel [35, 34] functional programming environment is an example of a structural editor, notable in the functional programming community for its ability to dynamically provide type information as the user writes programs with holes. This is achieved using carefully designed semantics of moving a text-editing cursor in the concrete syntax and mapping its location to a node in the parsed AST. When it comes to theorem provers, one example of structured editing is the CoqIDE [46], which leverages the Coq `XML` protocol and a state transaction system to

interact with the proof system's kernel by way of messages. This is not too dissimilar to what is done with implementations of the language server protocol to provide advanced editing functionalities for programming languages. In essence, the main objectives for implementing structural editing are to enable and assist in the incremental development of programs. This is in contrast with incremental compilation, which focuses on recompiling minimal sets of programs when they are affected by changes to the source code, which is achieved using dependency analysis and careful handling of cache invalidations.

Structural editors define sets of edit actions the user may execute at given points in their editing session. Typical edit actions include navigating the AST being modified, or constructing new nodes using either a graphical interface or a textual language. These actions are then reflected in the editor state in a predictable way, such that the edited program does not need to be reparsed from scratch. Having a greater control over how a program is edited is also used to prevent or signal edit actions that can invalidate a program. This can be as simple as signalling type errors on-the-fly, or in the case of theorem provers, to reject unsound operations like invalid appeals to induction hypotheses.

Although Beluga's interactive mode and Harpoon are REPLs as opposed to full-fledged structural editors, they do share some functional requirements. Indeed, like in Isabelle, Hazel and Coq, users of Beluga and Harpoon postpone the completion of programs or proof scripts by inserting holes in them. These holes stand for missing expressions, and they are later filled in with the help of a proof assistant that provides typing information for the identifiers in scope for each hole. As such, both REPLs and structural editors are required to construct and update an editor state as the user performs edit actions.

Edit actions in Beluga and Harpoon include navigating between holes in programs and proof scripts respectively, as well as displaying type information for expressions, and performing case analyses on meta-level and computation-level objects identified by (meta-)variables. These actions require surgical manipulation of the editor state to ensure soundness of the AST being edited is preserved, as well as to prevent the undesirable propagation of informa-

tion like typing constraints in unrelated edit locations.

Historically, Beluga was not implemented with the mindset of supporting the level of interactivity that Harpoon purports to have. Indeed, as is the case with prototype implementations of programming languages, Beluga's early development focused on implementing a parser for its concrete syntax and a type-checker for its type theory. The software architectural pattern that arose from this implementation was the pipeline pattern, in which the data processing of programs from its textual to its AST representation is handled in single-responsibility phases that sequentially augment and refine the data for later use. Specifically with respect to Beluga, these phases have evolved to include implicit argument reconstruction, the translation from a named to a nameless representation of binders using de Bruijn indices, term normalization, type-checking, and totality-checking. What transpires from this design is a unilateral flow of information from the concrete syntax to the internal representation of programs in Beluga. Early optimizations were put in place to maximize the performance of this processing pipeline. In particular, a single global mutable representation of the Beluga program being processed was implemented to provide information to later processing phases in a feedforward fashion. This simplified the implementation of new features since globally accessible data does not need intricate data routing procedures like dependency injection. This design was sound only under the assumption that the necessary information for processing Beluga signatures would flow in only one direction. Unfortunately, the introduction of Beluga's interactive mode and subsequently Harpoon inadvertently broke that assumption. Indeed, soundness issues have arisen with regards to features and processes of the two systems that require inspecting only a subset of a signature, and as such using referencing environments built out of order. This issue of locality in references has further been shown to deteriorate the performance of Beluga's logic programming engine used to automate proof search.

This thesis reports on improvements made to the implementation Beluga and Harpoon to address some of the soundness issues in interactive proof development in these two systems.

Overall, the changes listed below have improved the maintainability, resiliency and stability of the affected modules. Better architectural software design patterns have been put in place, and these may be expanded upon in future iterations on the later phases of semantic analysis. The specific contributions of this work are:

1. A complete and formal specification of Beluga and Harpoon's grammars for parsing.

2. A robust implementation of a context-free parser for Beluga and Harpoon, along with a new context-sensitive disambiguation phase to rectify and expand existing features. Most notably, the Beluga feature of fixity pragmas for defining prefix, infix or postfix notations for LF-level constants is reworked to also support computation-level constants.

3. A uniform name resolution algorithm to improve the clarity of Beluga programs, implement namespaces, and to support sound incremental proof development. This fixed soundness issues for Harpoon having to do with its feature of navigation between holes in proofs anywhere in a Beluga signature.

To present these contributions, the thesis is structured as follows:

- Chapter 2 provides an overview of Beluga's design as a language for proving theorems, followed by a discussion about its implementation. Structural editing features of Harpoon are outlined next to motivate the implementation changes discussed in chapter 4. A review of relevant and related work on programming language design, parsing, and incremental proof and program development.

- Chapter 3 describes technical limitations in the legacy implementation of Beluga's parser. These issues are then addressed with a new specification for the language's grammar, along with the introduction of a context-sensitive disambiguation phase that follows context-free parsing.

- Chapter 4 presents correctness issues regarding name resolution in Beluga, both in typical use cases for the language as well as for structural editing with Harpoon. Solutions to these problems are shown, along with a formal proof for a property ensuring that name resolution is sound in structural editing settings in which recoverable user errors occur.

- Chapter 5 concludes with an evaluation of the changes to Beluga and Harpoon, followed by areas of the implementation that will need to be reviewed in future work to further improve the system.

Chapter 2

Background

This chapter presents an overview of Beluga, and Harpoon, whose implementations were modified as part of this thesis. Topics in the design of a programming language's concrete syntax are discussed, along with features found in parsers of languages for proof assistants. Then, the concept of incremental proof and program development is briefly explained to motivate the implementation changes. Finally, the design and implementation of incremental development is examined for various proof assistants.

2.1 The Beluga Language

In Beluga, theorems are encoded using contextual LF, its index language, and proofs are encoded as programs. Logic propositions are represented as LF type families with term-level constants parameterized by β-normal η-long LF terms to construct witnesses for these propositions [30, 39, 3]. To facilitate the implementation of semantic analysis, and because LF kinds, types and terms are strongly normalizing [24], they are required to be encoded in normal form by the user. Beluga extends LF to support explicit contexts and substitutions, which enables mechanized reasoning with respect to assumptions, like in proofs on paper [40].

This means that terms containing free variables can be manipulated and reasoned about explicitly.

To illustrate how BELUGA is leveraged to mechanically prove properties about formal systems, consider the following example adapted from [20] where it is shown that algorithmic equality for an untyped λ-calculus is reflexive. The abstract syntax for this calculus is first defined in figure 2.1a, which is encoded as an LF type constant `term` with term constants `lam`, `app` and `unit` in the `Term` module of figure 2.1d. Then, the algorithmic equality judgment for terms is defined in figure 2.1 and encoded as an LF type family $\equiv$, using an infix notation, in module `Algorithmic_equality`. Note that this definition for equality does not provide constants to construct witnesses of reflexivity, symmetry or transitivity. These properties may instead be derived, like theorem 1 for reflexivity. This corresponds to the `refl` recursive program of figure 2.1d which constructively shows that for any (possibly open) term M, the algorithmic equality $M \equiv M$ holds. The notion that M may contain free variables comes into play when proving reflexivity for terms of the form $\lambda x.M$ since the induction hypothesis requires assumptions about the function parameter x.

$$\text{Terms} \qquad M, N \quad ::= \quad x \mid \lambda x.M \mid M\ N \mid \mathsf{unit}$$

(a) Syntax definition for the untyped λ-calculus used in the running example.

$\boxed{\Gamma \vdash M \equiv N}$: the term M is algorithmically equal to N in context Γ

$$\frac{\Gamma, x : \mathsf{term}, e : x \equiv x \vdash M\ x \equiv N\ x}{\Gamma \vdash \lambda x.M \equiv \lambda x.N}\ \equiv\text{-lam} \tag{2.1}$$

$$\frac{\Gamma \vdash M_1 \equiv N_1 \quad \Gamma \vdash M_2 \equiv N_2}{\Gamma \vdash M_1\ M_2 \equiv N_1\ N_2}\ \equiv\text{-app} \tag{2.2}$$

$$\frac{}{\Gamma \vdash \mathsf{unit} \equiv \mathsf{unit}}\ \equiv\text{-unit} \tag{2.3}$$

(b) Definition of algorithmic equality for terms in this λ-calculus.

Figure 2.1: Running example.

Theorem 1. *If* Γ *is a context where* $(x : \mathsf{term}) \in \Gamma$ *is sufficient for* $(e : x \equiv x) \in \Gamma$*, and* $\Gamma \vdash \overset{\mathcal{D}}{M}\ \mathsf{term}$*, then* $\Gamma \vdash M \equiv M$*.*

Proof. By structural induction on $\mathcal{D}$:

- Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{(x : \mathsf{term}) \in \Gamma}}{\Gamma \vdash x\ \mathsf{term}}$ term-var.

  By the regular context specification on $\mathcal{D}'$, we have $(e : x \equiv x) \in \Gamma$, hence $\Gamma \vdash x \equiv x$.

- Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{\Gamma, x : \mathsf{term} \vdash N\ \mathsf{term}}}{\Gamma \vdash \lambda x.N\ \mathsf{term}}$ term-lam.

  By weakening on $\mathcal{D}'$, we have $\Gamma, x : \mathsf{term}, e : \overset{\mathcal{E}}{x \equiv x} \vdash N\ \mathsf{term}$.

  By the induction hypothesis on $\mathcal{E}$, we have $\Gamma, x : \mathsf{term}, e : \overset{\mathcal{E}'}{x \equiv x} \vdash N\ x \equiv N\ x$.

  Then,

  $$\frac{\overset{\mathcal{E}'}{\Gamma, x : \mathsf{term}, e : x \equiv x \vdash N\ x \equiv N\ x}}{\Gamma \vdash \lambda x.N \equiv \lambda x.N} \equiv\text{-lam}.$$

- Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Gamma \vdash M_1\ \mathsf{term}} \quad \overset{\mathcal{D}_2}{\Gamma \vdash M_2\ \mathsf{term}}}{\Gamma \vdash M_1\ M_2\ \mathsf{term}}$ term-app.

  By the induction hypothesis on $\mathcal{D}_1$, we have $\Gamma \vdash \overset{\mathcal{E}_1}{M_1 \equiv M_1}$.

  By the induction hypothesis on $\mathcal{D}_2$, we have $\Gamma \vdash \overset{\mathcal{E}_2}{M_2 \equiv M_2}$.

  Then,

  $$\frac{\overset{\mathcal{E}_1}{\Gamma \vdash M_1 \equiv M_1} \quad \overset{\mathcal{E}_2}{\Gamma \vdash M_2 \equiv M_2}}{\Gamma \vdash M_1\ M_2 \equiv M_1\ M_2} \equiv\text{-app}.$$

- Case $\mathcal{D} = \dfrac{}{\Gamma \vdash \mathsf{unit}\ \mathsf{term}}$ term-unit.

  This case holds by $\equiv$-unit.

□

(c) On paper proof of algorithmic equality for this λ-calculus.

Figure 2.1: Running example (contd.).

```
module Term = struct
  LF term : type =
  | lam : (term → term) → term
  | app : term → term → term
  | unit : term;
  --name term M.
end

module Algorithmic_equality = struct
  --open Term.
  --infix ≡ none.
  LF ≡ : term → term → type =
  | lam : ({x : term} → x ≡ x → M x ≡ N x) → Term.lam M ≡ Term.lam N
  | app : M1 ≡ N1 → M2 ≡ N2 → Term.app M1 M2 ≡ Term.app N1 N2
  | unit : Term.unit ≡ Term.unit;
end

--open Term.
--open Algorithmic_equality.

schema ctx = block (x : term, eq : x ≡ x);

rec refl : (g : ctx) → {M : [g ⊢ term]} → [g ⊢ M ≡ M] =
  / total d (refl _ d) /
  mlam M ⇒
    case [_ ⊢ M] of
    | [g ⊢ #p.x] ⇒ [g ⊢ #p.eq]
    | [g ⊢ Term.lam \x. F] ⇒
        let [g, b : block (x : term, eq : x ≡ x) ⊢ D] =
          refl [g, b : block (x : term, eq : x ≡ x) ⊢ F[…, b.x]]
        in
        [g ⊢ Algorithmic_equality.lam \x. \eq. D[…, <x; eq>]]
    | [g ⊢ Term.app M1 M2] ⇒
        let [g ⊢ D1] = refl [g ⊢ M1] in
        let [g ⊢ D2] = refl [g ⊢ M2] in
        [g ⊢ Algorithmic_equality.app D1 D2]
    | [g ⊢ Term.unit] ⇒ [g ⊢ Algorithmic_equality.unit];
```

(d) Mechanized proof of algorithmic equality in Beluga for this λ-calculus, adapted from [20].

Figure 2.1: Running example (contd.).

As illustrated in figure 2.1d, a signature in Beluga is the list of all the toplevel constant declarations in a mechanization. This includes the aforementioned LF type-level and term-level constant declarations, but also includes inductive and coinductive type declarations at the computation-level, with their associated constructors and destructors respectively, and computation-level programs. Modules can be declared as well to introduce namespaces, which allows constant names to be reused and offers a way of organizing mechanizations. Though it is not showcased in that example, Harpoon provides an alternative to specifying proofs as programs in the form of a proof script comprised of tactics and their effects on the meta-level and computation-level contexts for variables. A complete specification of the concrete syntax for signature-level declarations is provided in section A.1.12 as part of grammars for parsing. To discuss the statics and dynamics of Beluga, an abstract syntax definition like the following is used instead.

Let x and c range over computation-level variables and constants respectively, X range over meta-object variables, ψ over context variables and g over context schemas. Figure 2.2 then provides an overview of Beluga's core language. LF kinds classify LF types, which classify LF terms. Likewise, computation-level kinds classify computation-level types, which classify computation-level expressions. LF terms are broken down into normal and neutral terms, along with heads and spines, to syntactically enforce canonical forms. The notation $A \to K$ is used for the LF kind $\Pi x{:}A.K$ when x does not appear free in K. The notation $A \to B$ is defined analogously for LF types. Since Beluga's type system is bidirectional, the syntax of computation-level expressions is split into those for which a type can be inferred (type-synthesizing expressions), and those that can be checked against a type (type-checkable expressions). Meta-objects, classified by meta-types, can be embedded in the computation-level by way of boxes that simultaneously bind all variables from LF contexts. Pattern-matching over type-synthesizing expressions (which includes embedded meta-objects) is supported by way of patterns, whose abstract syntax is omitted from figure 2.2 for the sake of conciseness.

LF kinds	K	::=	$\Pi x{:}A.K \mid \mathbf{type}$
LF types	A, B	::=	$\Pi x{:}A.B \mid P$
Atomic LF types	P	::=	$a\ S$
LF normal terms	M	::=	$R \mid \lambda x.M$
LF neutral terms	R	::=	$H\ S \mid u[\sigma]$
LF heads	H	::=	$x \mid c \mid p[\sigma]$
LF spines	S	::=	$\cdot \mid M\ S$
Substitutions	σ	::=	$\cdot \mid \mathsf{id}_\psi \mid \sigma, M \mid \mathsf{id}_\psi[\rho]$
Substitution closures	ρ	::=	$s[\sigma]$
Contexts	Ψ	::=	$\cdot \mid \psi \mid \Psi, x : A$

(a) Internal syntax of contextual LF in Beluga.

Meta-types	U	::=	$\cdots \mid g \mid (\Psi \vdash A) \mid (\Psi \vdash \Psi)$
Meta-objects	C	::=	$\cdots \mid \Psi \mid (\Psi \vdash M) \mid (\Psi \vdash \sigma)$

(b) Internal syntax of Beluga's meta level (excerpt).

Computation-level kinds	κ	::=	$[U] \to \kappa \mid \Pi X{:}U.\kappa \mid \mathbf{ctype}$
Computation-level types	τ	::=	$\cdots \mid c \mid \tau_1 \to \tau_2 \mid \Pi X{:}U.\tau \mid [U] \mid \tau\ [C]$
Type-checkable expressions	e	::=	$\cdots \mid i \mid [C] \mid \mathbf{fn}\ x \Rightarrow e \mid \mathbf{mlam}\ X \Rightarrow e$
		$\mid$	$\mathbf{let}\ x = i\ \mathbf{in}\ e \mid \mathbf{case}\ i\ \mathbf{of}\ \overrightarrow{p \Rightarrow e}$
Type-synthesizing expressions	i	::=	$\cdots \mid x \mid c \mid i\ e \mid e : \tau$

(c) Internal syntax of Beluga's computation level (excerpt).

Figure 2.2: Excerpt of Beluga's internal syntax [30, 23, 11, 21] used in discussions about its theory.

2.2 The Legacy Implementation of Beluga

This section presents an overview of the implementation of Beluga before any contribution detailed in this thesis were made[1]. This initial description serves as the basis for all performance and design comparisons made with respect to Beluga version `1.1`.

Beluga is implemented following the pipeline architectural pattern, whereby processing of a signature is implemented in distinct phases, and data flows in a feed-forward fashion throughout. The compilation of Beluga programs to machine code is not supported, so the implementation only covers the frontend component of compilation, which is responsible for syntactic and semantic analysis of programs. Since Beluga is a dependently-typed language featuring code coverage analysis and termination checking, this semantic analysis process is complex.

As illustrated in figure 2.3, the processing of a Beluga signature starts with syntax analysis, which is comprised of a tokenization and parsing phase that converts the textual representation of the signature to an AST, called the external AST. The model for this AST contains ambiguous nodes, meaning that some AST node variants capture multiple parse trees. Specifically, the application of LF type-level and term-level constants is represented as a list of parsemes, which effectively postpones the parsing of applications featuring user-defined operator notations. Parsing of signature-level declarations also features auxiliary data and routines for mixing or unmixing parsemes to delegate some disambiguation to phases after context-free parsing.

Indexing in Beluga is the process by which the concrete syntax is elaborated to a locally nameless representation called the approximate syntax, wherein bound variables are replaced with their corresponding de Bruijn indices, and constant names are replaced with symbolic identifiers defined in the centralized store of declarations. For instance, the LF

[1]The implementation of Beluga at revision hash 3db1ffd08d4c3bde7ad2ceb924bfb95488eef2b2, available on GitHub.

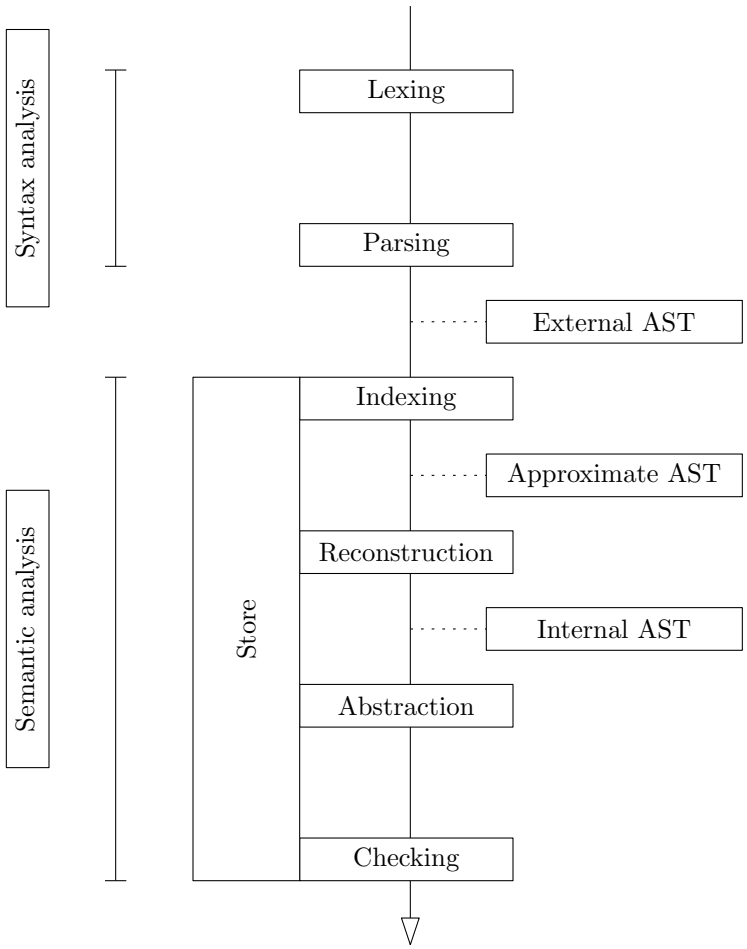


Figure 2.3: Overview of the implementation of Beluga version `1.0`'s processing pipeline. The syntax analysis process converts the textual representation of Beluga programs into an initial AST. The semantic analysis process refines this AST by performing type-directed signature reconstruction using global mutable data structures. This includes a store of constant declarations in the signature, along with auxiliary data for name resolution, fresh name generation and relations between type families.

term $\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z)$ is indexed as $\lambda\ \lambda\ \lambda\ 3\ 1\ (2\ 1)$. As part of BELUGA's design for terse type declarations and programs, some terms may contain free variables, like the variables `M1`, `M2`, `N1` and `N2` in the definition of constant `Algorithmic_equality.app` in figure 2.1d. The computation of de Bruijn indices of those free variables is postponed until the abstraction phase. Indexing is run immediately after parsing, and is additionally responsible for disambiguating the juxtaposition of LF parsemes at the precedence level of applications (which may contain user-defined operators), as well as disambiguating LF types from terms and resolving constants. This design is sensible since computing de Bruijn indices requires a stateful traversal of the AST that accumulates lists of bindings to produce the referencing environment. Using the store, some measure of identifier overloading is supported using a predefined order of lookups in the referencing environment based on the kind of identifier that is expected for a given AST node. For instance, computation-level identifiers are resolved by looking up in order the store of computation-level variables, the store of program constants, and then the store of data type constructors. Since names appearing in the LF level are not part of this name resolution strategy, then an identifier can be overloaded to stand for an LF term as well as a computation-level expression.

After indexing, the reconstruction [41] phase is run to reconstruct holes in types and terms, both at the LF level and the computation level. These holes stand for arguments omitted by the user, and provide an elegant way of abbreviating otherwise tedious aspects of programming with dependent types. For instance, in the `refl` program of figure 2.1d, the context parameter `(g : ctx)` is implicit, meaning that calls like `refl m` actually denote `refl _ m`, with a hole `_` where a context argument is expected. This is admissible because any argument for that parameter can be reconstructed from the argument `m` given for the explicit parameter `{M : [g ⊢ term]}`, which is defined with respect to `g`. At the LF level, approximate types are constructed to partially check the kinding of LF types and the typing of LF terms, as well as for guiding the synthesis of normal terms that check against a given type using typing constraints. Type-driven unmixing of overloaded syntactic forms

occurs at the meta-level, whereby meta-objects are disambiguated from substitutions during reconstruction. The approximate AST provides a disambiguated representation of the overall Beluga signatures, which helps in keeping track of the various changes made during this complicated phase.

A nearly complete internal AST as defined in figure 2.2 is produced at the end of reconstruction. An abstraction phase [23] is run to abstract over free variables, which effectively introduces binders for implicit parameters, and unrolls the contexts of variables used during type-driven reconstruction. For instance, the LF term constant `Algorithmic_equality.app` of figure 2.1d is abstracted to

```
(M1 : term) → (N1 : term) → (M2 : term) → (N2 : term) →
  (_ : M1 ≡ N1) → (_ : M2 ≡ N2) → Term.app M1 M2 ≡ Term.app N1 N2
```

and then de Bruijn indices are recomputed, which yields

$$\Pi_{\texttt{term}}\ \Pi_{\texttt{term}}\ \Pi_{\texttt{term}}\ \Pi_{\texttt{term}}\ \Pi_{4\equiv 3}\Pi_{3\equiv 2}\ \texttt{Term.app}\ 6\ 4\ \equiv \texttt{Term.app}\ 5\ 3$$

This phase completes the desugaring of Beluga programs since it finishes the computation of de Bruijn indices for variables across all levels. The main technical challenge abstraction runs into is determining the order in which introduced binders must appear so that dependencies on terms are preserved in inferred dependent types. This includes identifying and handling circular dependencies in abstracted parameters.

Finally, a semantic checking phase is run to ensure signature reconstruction and abstraction yielded valid programs. This includes performing type-checking, coverage-checking and totality-checking. These processes guarantee, respectively, that LF-level and computation-level expressions are well-typed, that case analyses are exhaustive, and that functions annotated with a totality declaration terminate for all inputs. Typing and coverage constraints generated during theses checking processes are statefully shared throughout this phase. Left-over and unresolved constraints in the state after having fully processed a program unit are used to signal to the user that that program is unsound within Beluga's system.

The flow of data in the implementation of Beluga is complex, like in most software systems. Starting with the indexing phase, data is shared between the phases of signature reconstruction using a global mutable store. This auxiliary data structure is a set of tables mapping constant identifiers to metadata, like in a relational database. Crucially, the referencing environment used during indexing is computed using this store. At any given point during the processing pipeline, the store's state is valid since program units are processed sequentially. New program units may be safely appended afterwards in interactive sessions.

2.3 The Harpoon Interactive Proof Environment

The Beluga system provides two ways to interactively perform queries on an existing Beluga signature, or to augment it with new theorems. These are the legacy REPL and the Harpoon system [19], the latter of which is designed as a replacement for the former. Both systems allow the user to input LF-level terms or computation-level expressions and get their type inferred with respect to an already elaborated Beluga signature. Harpoon further enables the interactive definition and proving of theorems using sets of tactics aimed at automating proof development. Both the REPL and Harpoon depend on the processing phases of Beluga and the flow of information therein as described in the previous section.

Harpoon is a structural editor for proof scripts that can be translated into Beluga programs. To do so, it provides edit actions a user may perform in an interactive proof session. These actions are split into two categories, namely administrative commands and proof tactics.

- Administrative commands affect the ambient state of the REPL by allowing the user to navigate in the Beluga signature and in the history of executed actions. These commands include navigating between incomplete subgoals in a theorem, navigating to incomplete proof scripts, undoing and redoing proof tactics, renaming bound variables,

and persisting the changes made to the signature.

- Proof tactics affect the state of the proof script under edit. These include tactics such as `intros` to universally quantify over the current theorem's premises and `split` to perform a case analysis on the structure of a value. Harpoon also features a `suffices` tactic allowing for backwards reasoning on a given subgoal, as well as tactics to perform automated proof search.

In order for Harpoon edit actions to be sound, they must bring the structural editing session from a valid state to another valid state. For proof tactics, this involves carefully keeping track of induction hypotheses generated during case analyses, as well as maintaining a correct record of identifiers in scope at the node under edit in the proof script. For administrative commands, the state of identifiers in scope has to always be local to the location in the Beluga signature where the current theorem is declared, and undoing edit actions has to restore the current proof script to its previous state. This in particular requires that effects occurring in type-checking and unification be undoable as well, which includes being able to revert the instantiation of unification variables.

In release `v1.0` of Beluga and Harpoon, not all administrative commands are sound, specifically with regards to the state of identifiers in scope. This is explored in chapter 4, with a detailed explanation on the soundness issues and solutions to rectify them.

2.4 Parsing and Programming Language Design

In compiler design, parsing is the process of converting the textual representation of a program into a hierarchical data structure [6, 4], which is typically called a parse tree. When a parse tree captures all the data about the text being processed, including comments and parentheses, then it is referred to as a concrete syntax tree. Otherwise, when parts of the data have been abstracted away, it is instead called an AST.

A program's textual representation is ambiguous with respect to a predefined set of parse rules if the program can be parsed into a parse forest, which is multiple parse trees [6]. By way of analogy, a sentence in a natural language is ambiguous if it can be interpreted in multiple valid ways with respect to its syntactic and grammatical rules. Since programming languages are tools for describing computerized systems, it is necessary that each valid written program has exactly one interpretation [6]. In the strictest of cases, this unicity of interpretation may be enforced at the level of the grammar and parser, such that all syntactically valid programs have exactly one inferable interpretation. This is typically achieved using rules and syntactic conventions that prevent ambiguous programs from being ever written in the first place.

Certain kinds of syntactic ambiguities are unavoidable in programming languages, and are frequently useful to the end user. Indeed, programming languages often reuse or overload syntactic constructs to reduce both the number and the complexity of rules that users have to learn in order to read and write programs. Ambiguous syntax can also make programs more terse, which may improve some workflows [36]. Additional mechanisms need to be put in place as part of the compiler's implementation to detect syntactic ambiguities, and either signal them as errors, or resolve them using additional interpretation rules.

Disambiguation is the process by which a parse forest is filtered down to a single parse tree. It refers to the procedures used to resolve ambiguities in intermediate results of parsing. In parsing systems that do not output parse forests, disambiguation typically involves manipulating the AST representation of a single parse tree that captures ambiguities, meaning that some of its nodes represent the overlapping of syntactic constructs. Functionally, these nodes suspend parsing until more information is available to disambiguate them to the correct node variant.

Increasing the amount of computation required to disambiguate the textual representation of a program negatively impacts the maintainability and readability of that program for the end user. Indeed, some forms of ambiguity in the syntax of a program may prevent the end user from fully understanding it in isolation from the rest of the code sections.

As such, one can argue that the programmatic strategies for disambiguating the syntax of a programming language should be intuitive so that external software is not required for the end user to read a program. For instance, name resolution is sufficiently intuitive for users, whereas searching in a parse forest for a parse tree that type-checks is not necessarily intuitive, especially given the complexity of some type systems. Hence, trade-offs between functionality and ease of understanding must be made when designing a language.

Different kinds of concrete syntax ambiguities may arise during parsing of a programming language. We distinguish three kinds of syntactic ambiguities that come into play in the implementation of Beluga, listed below in increasing degree of cognitive complexity required to solve them:

1. *Static operator ambiguities*: operators and their operands may be interpreted in different orders depending on where they appear in a whitespace-delimited list of terms. For instance, the expression $a + b * c$ is ambiguous with respect to the grammar $e \coloneqq e + e \mid e * e$. These ambiguities are typically resolved by assigning a precedence level, a fixity and an associativity to each operator in the language, and parsing them as keywords.

2. *Name-based ambiguities*: overloaded identifiers may be resolved to different binding sites depending on where they appear in an expression, and fall under different semantic categories as a consequence. For instance, in a dependently-typed language, type-level and term-level variables may be syntactically ambiguous, but they may have distinct syntaxes for binders. Semantic analysis of the AST with respect to a symbol table is usually sufficient to disambiguate those cases. However, overloading of identifiers coupled with more advanced features, like user-defined mixfix operators [13], poses additional challenges since identifiers may not be considered in isolation.

3. *Type-based ambiguities*: the correct interpretation of an expression may only be determined once a type has been inferred for it, or if it is checked against a type. This does

not refer to method overloading in object-oriented programming languages since the syntax for calling a method is not ambiguous. Rather, if the expression being parsed is an identifier, then a type-based ambiguity may be that that expression or a surrounding one falls into a different syntactic category based on the identifier's type or kind. For instance, the syntax `(g, x : t)` may denote a pair of terms, with the variable `x` being checked against the type `t`, or it may denote the context `g` with an added variable declaration `x : t`. In this case, what `(g, x : t)` syntactically denotes depends on the type of `g`. To solve such ambiguities, some type information may be provided by a symbol table if the type ascribed to an identifier is known at the declaration site. In general it is not practical to disambiguate expressions having type-based ambiguities before type-checking, especially when more intricate type inference algorithms need to be applied to determine the type of an identifier. Because of this, languages are typically designed to avoid type-based ambiguities. In the same way that is done in figure 2.2, identifiers like ψ can be reserved to always denote contexts, such that `(`ψ`, x : t)` is unambiguously a context.

The kinds of syntactic ambiguities that arise in the design of a programming language depend on the system used to specify its syntax, which in turn affects the algorithm required to parse it. The concrete syntax of programming languages is typically specified with a context-free grammar (CFG), denoted in extended Backus-Naur form (EBNF). If a language can be specified by a CFG, then it is a context-free language (CFL). CFLs have many advantages, including the fact that there is an abundance of well-vetted parser generators for such languages. OCAMLLEX together with OCAMLYACC [45] is one example of such a parser generator. Crucially, CFLs are easy to parse, both algorithmically and by the end user. As the name suggests, parsers for CFLs do not require additional data in a context specifically intended for disambiguation. This ensures that programs can scale and be readable by the user without having to fully know the context in which programs appear. That is, other

code sections in a CFL do not affect how a given code section is parsed. Static operator ambiguities as mentioned previously can be resolved by rewriting the grammar while keeping it context-free. Name-based and type-based ambiguities however give rise to context-sensitive languages, or even strictly Turing-recognizable languages [12]. Context-sensitive languages necessitate additional data about the context in which parts of a program appear to correctly parse it. This typically means that some semantic analysis is required during parsing in order to complete the syntactic analysis. Consequently, the concrete syntax's design for a programming language is intrinsically linked with the algorithm required to parse it.

A grammar is dynamic (or user-extensible, or featuring syntax extensions) if the way a program is parsed is influenced by directives in the program itself, and otherwise the grammar is static. Languages for specifying logics tend to favour user-extensible grammars over static ones. Indeed, Agda [2, 5] and Isabelle/HOL [49, 47] support mixfix operators, Coq [46] has notation declarations, and Beluga, like Twelf [38], has prefix, infix and postfix operators specified by pragmas. This is justified by the need for more concise and expressive ways to convey the meaning of definitions and lemmas. Such syntax extensions also allow users to develop mechanizations with notations that are closer to what appears on pen-and-paper proofs. However, that design choice negatively impacts the implementation of external tools for the language. Indeed, user-defined syntax extensions complicate the implementation of incremental parsing for efficiently parsing edits in a text editor, as well as indexing for resolving identifiers to their binding site. This forces tooling to be tightly coupled with the core implementation of the language.

2.5 State Management and Incremental Program Development

Users benefit from interacting with the code they are editing by way of auxiliary software that performs actions on it. Incremental program development in programming language tooling is the problem of applying edit actions to programs while only reprocessing a minimal portion of the program under edit. The kinds of edit actions that can be implemented vary from one programming language to the other depending on the language's features. Typically, edit actions include software refactoring commands such as variable renaming, reordering functions in a file, selecting a list of statements from one function and extracting them into a separate function, pretty-printing and formatting the textual representation of code, etc. Sound and efficient handling of these edit actions is instrumental to the productivity of users working on large scale software systems. This problem of incremental program development arises in Beluga and Harpoon because of the interactive tooling they provide for editing proofs, which are represented as programs.

Throughout syntactic and semantic analyses pipelines, data is accumulated during the processing of any given program unit. This data raises one of the more challenging aspects in implementing incremental program development, and the main motivation for reworking Beluga's processing pipeline: *a program unit may only be revisited with the processing state it was defined in.* This means, for instance, that a procedure performing name lookups or mutations on an AST at a given node may only do so using the variables and declarations in scope at that AST node, just as the user does when editing the textual representation of that AST node. As such, edit actions on an AST must preserve its semantic correctness properties so that serializing and subsequently deserializing it produces the same AST and processing state.

A preliminary step to supporting incremental program development is to parameterize

the processing routines with respect to a visitor state. As such, those routines can be used in an out-of-order fashion as opposed to when the program is processed during compilation tasks. This is because the routine can be visited with a different state than the one that is naturally constructed when sequentially processing program units. For instance, a routine like conflict-avoiding variable renaming can be dependent on a referencing environment to perform variable and constant lookups. If such a routine is invoked during an interactive program development session, then a separate routine can be implemented to rebuild the referencing environment without having to reprocess the entire program and its dependencies. In this case, the edit action requires visiting a selected AST node with the referencing environment as visitor state. The issue then becomes how to efficiently construct and preserve the correctness of such visitor states. As it pertains to Beluga and Harpoon, this specifically affects REPL sessions instantiated at program holes.

2.6 Incremental Proof Development in Other Languages

Incremental program development is a broad and open problem in the implementation of interpreters and compilers. The main challenge with implementing such systems is state invalidation, where editing part of a program causes changes elsewhere, both in the code the user wrote and in the state of the interpreter analysing it. While reprocessing the affected parts of the code from scratch is a valid solution to guarantee soundness, this does not scale efficiently to large programs. Structured editing obviates some of the issues with state invalidation and the propagation of changes in incremental development. Indeed, edit actions are scoped to a model of the program as opposed to the program's textual representation itself, which means they have a limited and controlled effect on the structured editor's state. Mechanisms can then be designed for the interpreter's state to improve its runtime performance in soundly responding to edit actions.

Proof assistants providing interactivity through REPLs and tools for text editors each

approach the problem of incremental development in unique ways tailored to their systems. For instance, constraint-based type systems require mechanisms to keep track of the effects of solving said constraints so that those effects can be undone in the event that type-checking fails for a given program. Likewise, context-switching between different parts of a program under edit requires updating the state to correctly reflect what identifiers are in scope, both for providing scope-aware code completion hints and for performing identifier resolution in new terms. Furthermore, command histories may be implemented to support undoing and redoing edit actions, which should be a feature of all proof assistants since users may apply tactics that do not lead to a solution for a given subgoal and decide to undo the application of those tactics. Additionally, incomplete subgoals may be arranged internally as a tree following the structure of a proof to allow navigating between holes. The successful implementation of these structured editing features hinges on careful management of recorded state to ensure soundness.

Abella

The Abella[2] [7] interactive theorem prover has limited support for structurally editing proofs. Indeed, it leverages a single global state comprised of mutable references with a snapshot mechanism to copy the entire state on every undoable command. This state notably includes the signature of declarations, the subordination relation on type families, and the list of subgoals that have yet to be proven in the current lemma under edit. Hence in its interactive mode, Abella users are restricted to adding new declarations at the end of the session's state. Indeed, constructing visiting states to other parts of the signature under edit would require rerunning the processing pipeline from scratch.

[2] Abella version `2.0.8`

Isabelle

Isabelle/Isar[3] implements a system of transitions between immutable state structures to support pure edit actions [49, 48, 47]. Indeed, most of that system's state, internally referred to as contexts [8], is implemented using immutable 2-3 trees to tabulate data, with each context holding a list of all its predecessor states. The definitions, proofs and terms in the buffer under edit are stored in such contexts. Undo operations and modification to the signature then proceed by rolling back the contexts to before the edit was done using their lists of predecessors. Mutable references are leveraged to represent the global state of the kernel, though their usage is limited. This mutable data is split between synchronized and unsynchronized management strategies, the former adding mutual exclusion locks to references that hold mutable data to enable safe concurrent computations. Overall, the extensive use of immutable data structures, coupled with synchronization for mutable data, results in degraded memory usage and runtime performance. Nonetheless, this system allows finer edit actions and ensures that cache invalidations are resolved consistently.

Agda

Agda[4] [2, 33, 5] supports incremental interactions with its Agda mode and its implementation of the language server protocol. Issuing an edit command with those systems directly affects the text editor's buffer, which provides a seamless editing experience. Under the hood, this is supported by Agda's type checking monad, which is an instance of the state monad whose state structure contains most of the mutable and immutable data used throughout the system. This state includes global configuration parameters, scope information, user-defined notations for parsing, typing contexts for variables as association lists, etc. It is an agglomeration of all the data used in Agda's processing pipeline, readily available as a

[3] Isabelle 2023

[4] Agda `v2.6.4`

function parameter as opposed to being represented using a global mutable data structure. This wholly captures the idea of a visiting state over declarations in an Agda signature. The main issues when dealing with this type checking monad are performance overheads, and ensuring that state mutations are sound. To mitigate soundness issues, some interaction kinds have defined scopes to restrict their effect on the type checking state. In case of bugs arising during edit sessions, separate user commands are available to restore the state to earlier checkpoints, or to entirely reload the current editor buffer from scratch.

Coq

Coq[5] and the CoqIDE [46, 9] implement a state transaction machine to efficiently keep track of changes to proofs and their effects on the system's state. This is backed by a version control system whose design is inspired by Git. Internally, a directed acyclic graph (DAG) is created and maintained to represent states as nodes and transactions (or differences) as edges. Committed changes to the state create branches in the version control system which can then be merged, deleted or checked out. The actual data to use at a given state in this system is reconstructed by traversing the DAG from the root node to the target node while applying the effects encoded in each visited edge. Concretely, this version control system is used in Coq's interactive environment when processing vernacular commands, including querying documents, starting proofs, stepping in a proof, and solving a subgoal.

[5] Coq `v8.18.0`

Chapter 3

Implementing a Parser for Beluga

This chapter presents the re-implementation of Beluga's and Harpoon's parsing algorithms to improve the system's maintainability, to expand the user-defined notation feature in a reusable way, and to improve syntax error messages for the user. Aspects of context-sensitive disambiguation of ASTs are showcased, including grammars for parsing user-defined prefix, infix and postfix operators.

3.1 Introduction

Early versions of the Beluga language were context-free and used Camlp4 [15] as parser generator to implement its parser. Beluga became context-sensitive when modifications were made to its syntax, specifically to improve the readability of contextual objects and to add user-defined operators as in Twelf [38] for LF type-level and term-level constants. These changes did not pose a problem with the parser's implementation per se, but it did require the introduction of disambiguation mechanisms as discussed in section 2.4. Specifically, parts of indexing and reconstruction, as illustrated in figure 2.3, and the elaboration steps in between, had to be implemented with AST node variants that mix together syntactically

ambiguous constructs. This complicated the processing pipeline because the semantics of a given Beluga program is only gradually discovered as it is elaborated through the external, approximate and internal syntax representations.

As the Beluga language grew, so did the complexity of its grammar, such that user errors output by the parser generated with Camlp4 were deemed not sufficiently informative. Hence, Beluga version `1.0` featured a new parser implemented using monadic parser combinators, which are higher-order functions for constructing top-down recursive descent parsers [10, 25, 27, 26, 4]. They provide a more declarative way of constructing complex parsers than shift-reduce parsers. Indeed, the combination of smaller parsers (or parselets) in OCaml using operators closely resembles grammar specifications using CFGs, which allows for fast prototyping and better readability of the implementation. The intent of reworking Beluga to use parser combinators was to improve error reporting for the end user, and to improve the overall maintainability of the implementation for developers by making it more modular. These objectives were largely achieved, and as a consequence Harpoon was also implemented using that parsing framework.

Despite the modifications made during the previous parser rewrite from using Camlp4 to using parser combinators, some major flaws remained in the implementation:

1. Error messages produced by the parser did not effectively explain some of the stringent restrictions placed on the syntax of the language as part of its design.

2. A lack of internal documentation with regards to the implemented disambiguation mechanisms hampered modifications to the grammar as well as the reusability of the parser.

3. The user-defined notation feature was not implemented in a reusable way, and its implementation had correctness issues.

To resolve these problems, architectural modifications were made to the parser, with a greater

focus on correctness and usability.

3.2 Beluga Lexing, Parsing and Disambiguation Phases

As part of this thesis, Beluga's parser was reimplemented to address correctness issues, to handle expressions in non-canonical, to improve error reporting, to support user-defined operators at the level of computations, and to improve the implementation's maintainability. Particularly, semantic analysis is introduced as part of a new context-sensitive disambiguation phase to the parser in order to produce a fully unambiguous AST before signature reconstruction. The motivation for these changes to the implementation of Beluga can be summarized in the following points:

1. *Expand the operator pragmas feature to computation-level types and expressions.* By implementing a reusable process for disambiguating function applications, any juxtaposition of terms in the language's grammar can feature user-defined operators.

2. *Improve syntax error messages.* By accepting a wider range of programs during parsing, the subsequent disambiguation phase can more precisely identify the cause for syntax errors and report them with better detail.

3. *Improve the processing pipeline's maintainability.* By decoupling disambiguation from indexing, both phases now respect the single-responsibility principle, which in turn simplifies the information flow at the early stage of signature reconstruction. Dependency injection of mutable states is also introduced to define clear boundaries on effects in the implementation of the disambiguation phase, which enables the development of structured editing procedures.

4. *Provide unambiguous external ASTs to enable debugging of signature reconstruction.* Since the disambiguated external AST of a program fully captures the meaning of what the user wrote, then the internal AST as illustrated in figure 2.3 is no longer the only unambiguous data representation for programs in Beluga. This clear boundary in the system allows for isolated testing of the parser, and provides a point of comparison with the internal syntax AST output from signature reconstruction.

A complete specification of Beluga's and Harpoon's syntax is provided in appendix A to facilitate future changes to the language, and to provide documentation to the end user. The revised parsing algorithm for Beluga signatures is split into three phases, namely: lexing, parsing and disambiguation. These phases are outlined in the next subsections. First, lexing is implemented just like in typical whitespace-agnostic programming languages, except for one lexer hack used to handle some overloading of a static operator. Then, parsing produces an AST close to the concrete syntax and featuring ambiguous nodes, which effectively postpones parsing steps that require external data. Finally, disambiguation converts the parsed AST to a revised external AST without ambiguous nodes.

3.2.1 Lexing

Lexical analysis for Beluga uses regular expressions to convert sequences of characters into tokens to speed up recursive descent parsing. Appendixes A.1.1, A.1.2 and A.1.3 provide the functional requirements for this phase of the implementation. Notably, lexing of nested delimited comments (i.e., those comments enclosed in either `%{` and `}%`, or `%{{` and `}}%`) is achieved by parameterizing the comment-lexing function with respect to the depth of the comment. This ensures that extraneous or missing right-delimiters for comments are detected early.

Where the lexing phase diverges from conventional implementations is with regards to its dot operator. Indeed, usual lexical conventions dictate that whitespace may precede or

follow a dot without affecting a program's semantics. This cannot apply unambiguously in Beluga programs, where a dot denotes one the following:

1. The end of an LF term-level or type-level constant declaration, like in Twelf.
2. The beginning of a computation-level observation application for coinductive objects.
3. The delimiter between the parameter name and the body of an LF λ-abstraction.
4. The projection out of a contextual LF block term.
5. The projection out of a computation-level tuple expression.
6. The projection out of a module acting as a namespace.

This first usage of the dot operator proved particularly difficult to resolve when the operator was overloaded with projections out of namespaces. One approach to solve this issue is to use a different operator for accessing members of modules, which deviates from how ML-style languages are designed. The solution that was ultimately put in place was to instead introduce two additional token kinds $\langle$*dot-identifier*$\rangle$ and $\langle$*dot-integer*$\rangle$ in the lexer for identifiers and numbers immediately prefixed by a dot. These token kinds were reserved to denote projections. During context-free parsing, wherever a dot is expected to not be part of a projection, these two token kinds would still be accepted, but a corresponding identifier or number token would be spliced in the input stream. This effectively means that whitespace cannot follow a dot operator used for projection. Thus, Twelf-style LF term-level and type-level constant declarations could be parsed in Beluga, provided that the trailing dot is followed by whitespace.

This token insertion solution had to be introduced to preserve backward compatibility in the parser, specifically for that Twelf-style of constant declarations. In general, it is inadvisable to mutate the stream of tokens during parsing, especially in the presence of unbounded backtracking since those insertions may have to be reverted.

3.2.2 Parsing

Parsing of Beluga signatures is achieved using $\mathsf{LL}(\infty)$ parsing in pathological cases and $\mathsf{LL}(1)$ parsing in the most frequent cases. The pathological cases are discussed in section 3.4. The parser is implemented using the monadic parser combinator library introduced in Beluga version `1.0`. Unlimited lookahead is enabled only in select cases, or if no input token has been consumed, which performs efficiently enough for large Beluga files.

The main contribution made to the context-free parsing phase of Beluga is the simplification of its grammar. Indeed, the legacy parser was designed following the grammars developed along signature reconstruction and type-checking algorithms, which introduced syntactic classes that ensure some properties always hold by construction for terms in the language. Specifically, the index level LF of Beluga is strongly-normalizing, and LF terms in canonical form cannot feature certain kinds of non-termination in term reduction, then it was decided that the user should only be able to write LF terms and types in canonical form. This required that nonterminals be introduced for normal and neutral terms separately to mirror the syntax of figure 2.2. Additionally, since Beluga's type-checking algorithm is bidirectional, computation-level expressions were split into two syntactic classes, namely type-synthesizing expressions and type-checkable expressions, and this change was also reflected in the parser. Implementation efforts revealed however that these syntactic constraints are too stringent to be imposed during parsing. Indeed, they increased the complexity of the parser, made operator precedence rules opaque, and yielded poor syntax error messages to the user. This is because the pre-terms of Beluga cannot be elegantly partitioned all at once by order of precedence, by whether they are neutral or normal, and by whether they synthesize a type or check against one. This partitioning is required for recursive descent parsing, but those parser rules are too complex to be explained to newcomers to the language, and are also error-prone for developers. The additional constraints further restrict the language recognized by the parser, which in turn prevents some syntactically in-

valid programs from being reported with more precision than simply identifying unexpected tokens. This is in part evidenced by the grammar in figure 3.1 and the example syntax error message in figure 3.2. As such, the revised parser's grammar does not distinguish between neutral and normal terms, nor type-synthesizing and type-checkable expressions. This greatly broadens the scope of programs accepted by the parser, which in turn allows for later phases of processing to report and reject non-canonical terms with greater flexibility for error reporting.

⟨*clf-normal*⟩ ::= ‘`\`’ ⟨*identifier*⟩ ‘`.`’ ⟨*clf-term-application*⟩

| ‘`(`’ ⟨*clf-term-application*⟩ [‘`:`’ ⟨*clf-type*⟩] ‘`)`’

| ⟨*hash-identifier*⟩ [⟨*clf-projection*⟩] [‘`[`’ ⟨*clf-substitution*⟩ ‘`]`’]

| ⟨*qualified-identifier*⟩ [⟨*clf-projection*⟩] [‘`[`’ ⟨*clf-substitution*⟩ ‘`]`’]

| ‘`_`’

| ‘`?`’⟨*identifier*⟩

| ‘`<`’ ⟨*clf-term-application*⟩ (‘`;`’ ⟨*clf-term-application*⟩)+ ‘`>`’

⟨*clf-projection*⟩ ::= ‘`.`’ ⟨*integer*⟩

| ‘`.`’ ⟨*identifier*⟩

⟨*clf-term-application*⟩ ::= ⟨*clf-normal*⟩+

| ⟨*clf-type*⟩

⟨*clf-type*⟩ ::= ‘`{`’ ⟨*identifier*⟩ ‘`:`’ ⟨*clf-type*⟩ ‘`}`’ [‘`->`’] ⟨*clf-type*⟩

| ⟨*clf-type-atomic*⟩ [‘`->`’ ⟨*clf-type*⟩]

⟨*clf-type-atomic*⟩ ::= ⟨*clf-normal*⟩+

| ⟨*identifier*⟩ ⟨*clf-normal*⟩*

| ‘`(`’ ⟨*clf-type*⟩ ‘`)`’

| ‘`block`’ ‘`(`’ ⟨*identifier*⟩ ‘`:`’ ⟨*clf-type*⟩ (‘`,`’ ⟨*identifier*⟩ ‘`:`’ ⟨*clf-type*⟩)+ ‘`)`’

Figure 3.1: The grammar based on figure 2.2 for parsing contextual LF terms in normal form as implemented in the legacy parser, denoted using the lexical conventions of section A.1.3. The definition for the nonterminal ⟨*clf-substitution*⟩ is omitted for brevity. The nonterminals ⟨*clf-term-application*⟩ and ⟨*clf-type-atomic*⟩ showcase a dramatic increase in complexity for parsing contextual LF terms and types.

```
exp : type.
eq : exp → exp → type.
schema ctx = block (x : exp, t : eq x x);
rec reflexivity : {g : ctx} {M : [g ⊢ exp]} [g ⊢ (eq M M)[..]] = ?;
```

(a) Example Beluga theorem statement with a syntactically invalid substitution. The identity substitution `[..]` on the application of an LF type constant in `(eq M M)[..]` is invalid because only canonical forms of LF objects are recognized for signature reconstruction.

```
example.bel, line 4, column 57:
Parse error.
Unexpected token in stream
  Expected token `]'
  Got token `['
```

(b) The syntax error raised during context-free parsing in the legacy parser implementation. Given the stricter grammar of ⟨*clf-type*⟩ in figure 3.1, it is true that the token `[` introducing the identity substitution is unexpected. However, the reason for this error is not obvious from its message, in part because the user unambiguously wrote a substitution.

```
File "example.bel", line 5, column 50:
4 |rec reflexivity : {g : ctx} {M : [g ⊢ exp]} [g ⊢ (eq M M)[..]] = ?;
                                                   ^^^^^^^^^^^^
Error: Substitution terms may not appear as contextual LF types.
```

(c) Improved syntax error message produced during the disambiguation phase instead of during context-free parsing. Using a relaxed grammar for LF objects to support non-canonical forms, the parser accepts the invalid substitution, and then the disambiguation phase identifies that it should be rejected.

Figure 3.2: Example of improved syntax error reporting using less stringent syntactic rules at the level of the parser.

Besides the aforementioned simplifications to the grammar, the revised parser still has to support the syntax of LF kinds, types and terms, which overloads operators. For instance, the $\rightarrow$ operator is overloaded for arrow kinds and types, and the syntaxes for Π-kinds and Π-types are the same in Beluga. This means that an LF kind is syntactically indistinguishable from an LF type until a **type** kind is encountered as the very last operand since LF kinds necessarily contain one. Unrestricted backtracking and infinite lookaheads are not suitable solutions to this disambiguation issue given that those expressions are the most frequent ones in Beluga signatures.

The implemented solution to this sort of problem during context-free parsing is somewhat counterintuitive. The syntactic categories having common syntaxes that appear in ambiguous positions are merged into a single nonterminal in the grammar. We call this blurring the grammar since we effectively lose the distinction between those merged syntactic categories during context-free parsing.

For instance, consider the following subset of Beluga's grammar with an overloading of the ⟨*forward-arrow*⟩ operator.

⟨*lf-kind*⟩ ::= ⟨*lf-type*⟩ ⟨*forward-arrow*⟩ ⟨*lf-kind*⟩ | '`type`'

⟨*lf-type*⟩ ::= ⟨*lf-type*⟩ ⟨*forward-arrow*⟩ ⟨*lf-type*⟩ | ⟨*identifier*⟩

If a parser is in a state where either an ⟨*lf-kind*⟩ or an ⟨*lf-type*⟩ is expected, and the sequence ⟨*lf-type*⟩ ⟨*forward-arrow*⟩ is parsed successfully, then the parser expects again either an ⟨*lf-kind*⟩ or an ⟨*lf-type*⟩ to appear next. While at first left factoring [6] seems usable to rewrite the grammar for predictive parsing, the problem lies in the shared prefix ⟨*lf-type*⟩ ⟨*forward-arrow*⟩ not being in productions of the same nonterminal. A backtracking parser cannot efficiently handle this grammar, and adding memoization to the implementation to avoid redundant parses of the prefix would greatly complicate the parser. Instead, we introduce ⟨*lf-object*⟩ as the merging of the syntactic categories ⟨*lf-kind*⟩ and ⟨*lf-type*⟩ obtained as follows:

1. The production rules of ⟨*lf-kind*⟩ and ⟨*lf-type*⟩ are added to ⟨*lf-object*⟩,
2. Occurrences of ⟨*lf-kind*⟩ and ⟨*lf-type*⟩ in the rules of ⟨*lf-object*⟩ are replaced with ⟨*lf-object*⟩,
3. Duplicate production rules in ⟨*lf-object*⟩ are removed.

This yields a simpler grammar whose language contains the languages of the blurred syntactic categories.

⟨*lf-object*⟩ ::= ⟨*lf-object*⟩ ⟨*forward-arrow*⟩ ⟨*lf-object*⟩ | '`type`' | ⟨*identifier*⟩

This is a viable solution only if there is a way to disambiguate an ⟨*lf-kind*⟩ or ⟨*lf-type*⟩ from a parsed ⟨*lf-object*⟩. In this case, a syntactically valid ⟨*lf-object*⟩ is an ⟨*lf-kind*⟩ if and only if it contains the keyword '`type`'.

As shown in the previous example, a blurred grammar accepts a wider range of expressions than its original grammars. Hence, an additional phase of processing is required to recover a parser for the initial grammars. Section A.2.1 provides the complete example of a blurred grammar used in the implementation, obtained by merging the nonterminals ⟨*lf-kind*⟩, ⟨*lf-type*⟩ and ⟨*lf-term*⟩ of section A.1.5 into one syntactic category, with corresponding nonterminal ⟨*lf-object*⟩. The context-sensitive disambiguation phase outlined in the next section is then tasked with recovering the original LF kind, type or term corresponding to a parsed LF object. We note that a similar technique was used in the implementation of Twelf since its shift-reduce parser could not perform disambiguation during parsing.

3.2.3 Disambiguation

The disambiguation phase of parsing performs an AST transformation from the parser syntax to the external syntax. Given Beluga's features and overloaded syntactic constructs, disambiguation is responsible for recovering the structure specified by the grammars of sec-

tion A.1 using the structure obtained by context-free parsing following the blurred grammars of section A.2. Specifically, it performs disambiguation of:

1. LF kinds, types and terms,
2. contextual LF types and terms,
3. meta-level contexts and substitutions,
4. computation-level kinds and types, and
5. applications throughout the semantic classes (detailed in section 3.3).

Disambiguation of Beluga programs requires that identifiers be resolved to their binding sites in order to determine their sort. This process statefully keeps track of the identifiers in scope at any given point during a traversal of the parser AST. This environment is implemented using a mutable state structure with auxiliary data to deal with patterns, modules and pragmas. Specifically, bindings during this phase are modelled using a stack of scopes, each of which contains a tree mapping fully qualified identifiers to minimal descriptions of what those identifiers are bound to (namely variables or constants). In particular, the referencing environment records the user-defined notation to use for any given identifier. This allows for fixity pragmas like in figure 3.3 to be declared in namespaces, and to later be brought in scope when the namespace is opened. Additionally, notations can be locally redefined since the referencing environment defines the area of effect for those pragmas. Then, applications in the concrete syntax, which were parsed as lists of terms, can be reparsed in a context-sensitive manner during disambiguation.

We postpone the discussion on how the referencing environment is constructed and updated during the AST traversal to section 4.3 because it closely follows the procedure used to compute de Bruijn indices during the indexing phase.

3.3 Parsing User-Defined Operators

In Beluga, users may specify that identifiers should be used as operators in expression applications, as shown in figure 3.3, using `--prefix`, `--infix` or `--postfix` pragmas. Prefix operators are unary and right-associative, postfix operators are unary and left-associative, and infix operators are binary and may be either left-associative, right-associative or non-associative. The priority of operations is specified using integer precedence values. This feature was ported over from Twelf, and using the new parsing architecture, it was subsequently improved to support shadowing of operators by bound variables and other constants. As outlined below, the notation pragmas for user-defined operators in Beluga are more restrictive than that of Agda because Beluga does not support mixfix operators [13].

The legacy Beluga system only supported user-defined operators for LF type-level and term-level applications. As such, only constants in LF could be defined as operators. The implemented revisions allow for user-defined operators to be used in all applications, meaning that computation-level types and expressions may also benefit from prefix, infix and postfix notations. Specifically, all typed constants can now be annotated with a notation pragma. Type annotations are required in this case since disambiguation does not perform type inference. This in turn allows for early error detection for mismatches between the arity expected by the notation pragma and the actual arity of the constant. Using a lookahead mechanism during disambiguation, notation pragmas can be declared before their targets are in scope. As such, their application is postponed until the constants they affect are introduced in scope.

The new implementation for parsing user-defined operators relies on the disambiguation phase for resolving identifiers to constants and their notations. As such, the initial parser suspends parsing of applications, meaning that it represents the juxtaposition of parsemes as lists. Disambiguation then resumes parsing with a recursive descent parser instantiated

```
LF p : type =
| ⊃ : p → p → p                    % Logical implication
| ⊂ : p → p → p                    % Converse implication
| ∧ : p → p → p                    % Logical conjunction
| ∨ : p → p → p                    % Logical disjunction
| ¬ : p → p                        % Logical negation
| ⊤ : p                            % Tautology
| ⊥ : p                            % Contradiction
;
--infix ⊃ 3 right.    % Subsequently treat ⊃ as a right-associative
--infix ⊂ 3 left.     % infix operator with precedence value 3
--infix ∧ 5 right.
--infix ∨ 4 right.
--prefix ¬ 10.

let a = [x : p ⊢ ¬ ¬ ¬ x];
let b = [x : p, y : p ⊢ ¬ x ⊂ y ⊂ ⊥];
let c = [x : p, y : p, z : p ⊢ ⊤ ∨ x ∧ ¬ y ⊃ ¬ z];
let d = [f : p → p, x : p ⊢ ¬ f x ∨ f ⊥];
```

Figure 3.3: Example of user-defined operator definitions in Beluga using pragmas. The subsequent meta-objects a, b, c and d use those notations to construct formulas just like in proofs on paper.

specifically for the operators that appear in the list of parsemes. That is, when disambiguating an application node, not all constant declarations in scope at that point in the Beluga signature need to be taken into account. We only need to build an operator table as in figure 3.4 for the constants appearing in the application and resolved as operators in the referencing environment. This ensures that the complexity of this second phase of parsing for applications scales only with the number of parsemes in the list, and not with the size of the signature. However, this design requires that operators cannot be overloaded, otherwise disambiguation of applications could only be realized when type information is available, which happens much later in signature reconstruction for Beluga.

In Beluga, at the precedence level of expression applications, there are four scenarios

p	Prefix	Infix left-associative	Infix non-associative	Infix right-associative	Postfix
1		$\subset$			
2				$\vee$	
3				$\wedge$	
4	$\neg$				

Figure 3.4: Example operator table for parsing user-defined operators for applications containing only the operators $\neg$, $\wedge$, $\vee$ and $\subset$ as defined in figure 3.3. The rows are arranged in order of relative precedence for those operators. This table would be used to parse $x \subset x \vee y \vee \neg z \wedge w$ as $x \subset (x \vee (y \vee ((\neg z) \wedge w)))$.

to disambiguate:

1. Prefix operators followed by their operands,
2. Left-associative, right-associative or non-associative infix operators preceded and followed by their operands,
3. Postfix operators preceded by their operands, and
4. Juxtaposed expressions having no operator.

During the disambiguation phase of parsing, we only need to determine the syntactic structure of applicands and arguments for parsemes in lists. The implementation of this parsing algorithm is inspired by the general purpose expression parser from the `Parsec` library [27], as well as the parser scheme in figure 3.5, which is adapted from the parser scheme for parsing mixfix operators in [13]. This parsing algorithm proceeds in the following steps:

1. Identify the constants with user-defined operator notations in the list of parsemes.
2. Group those identifiers by precedence, then by fixity and associativity, like in the table in figure 3.4.

3. Construct a parselet for each precedence level in the operator table, such that the precedence climb delegates parsing to the parselet for the next precedence level, in increasing order.

4. Run the parselet handling the least precedence level on the list of parsemes.

This is effectively the same principle as recursive descent parsing for a set of operators statically defined by the language's grammar, but instead implemented over a dynamic set of operators. This set is constructed using only those constants found in the AST node for application. As such, this parsing algorithm for user-defined operators has $O(n^3)$ runtime complexity, where n is the length of the list of parsemes. Figures 3.5a, 3.5b and 3.5c progressively show how to obtain an LL(1) grammar scheme to parse user-defined operators. In Beluga v1.1, the parser is implemented following this last grammar scheme, and using monadic parser combinators just like in the context-free parsing phase.

Coincidentally, the grammar scheme of figure 3.5b was used to reintroduce support for infix left and right arrow operators $\rightarrow$ and $\leftarrow$ in LF types. Support for the $\leftarrow$ operator had been dropped during the rewrite of the parser from Camlp4 to monadic parser combinators. This was due in all likelihood because of the difficulty with amending the grammar of figure 3.1 while respecting the precedence and associativity of operators. As illustrated in figure 3.5, the key takeaway from reintroducing that static operator is that at a given precedence level, we can either have the application of a non-associative operator, successive applications of right-associative operators, or applications of left-associative operators. Those three categories of operators cannot mix at the same precedence level without introducing an unresolvable ambiguity. As such, the recursive descent parselet for $\rightarrow$ and $\leftarrow$ operators only allows one or the other operator to appear.

$$
\begin{array}{rcl}
\langle e \rangle & ::= & \langle \mathsf{op}_{\text{prefix}} \rangle \ \langle e \rangle \\
& | & \langle e \rangle \ \langle \mathsf{op}_{\text{infix}} \rangle \ \langle e \rangle \\
& | & \langle e \rangle \ \langle \mathsf{op}_{\text{postfix}} \rangle \\
& | & \mathbf{a}+
\end{array}
$$

(a) Ambiguous grammar for parsing user-defined operators. The terminal **a** stands for expressions that were already parsed at a higher precedence level than any user-defined operator during the context-free parsing phase. For instance, **a** can stand for variables and projections.

$$
\begin{array}{rcl}
\langle e \rangle & ::= & \langle e \rangle_1 \\
\langle e \rangle_p & ::= & \langle e \rangle_{p+1} \ \langle \mathsf{op}^{\text{N}}_{\text{infix}} \rangle_p \ \langle e \rangle_{p+1} \\
& | & \big(\langle \mathsf{op}_{\text{prefix}} \rangle_p \mid \langle e \rangle_{p+1} \ \langle \mathsf{op}^{\text{R}}_{\text{infix}} \rangle_p \big)+ \ \langle e \rangle_{p+1} \\
& | & \langle e \rangle_{p+1} \ \ \big(\langle \mathsf{op}_{\text{postfix}} \rangle_p \mid \langle \mathsf{op}^{\text{L}}_{\text{infix}} \rangle_p \ \langle e \rangle_{p+1} \big)+ \\
& | & \langle e \rangle_{p+1} \\
\langle e \rangle_{P+1} & ::= & \mathbf{a}+
\end{array}
$$

(b) Initial grammar scheme for parsing user-defined operators. P is the number of distinct precedence levels induced by the user-defined operators appearing in the application, and $\langle e \rangle_p$ is the nonterminal for an expression at precedence level p. Similarly, $\langle \mathsf{op}^{\text{N}}_{\text{infix}} \rangle_p$, $\langle \mathsf{op}^{\text{R}}_{\text{infix}} \rangle_p$ and $\langle \mathsf{op}^{\text{L}}_{\text{infix}} \rangle_p$ stand for infix non-associative, right-associative, and left-associative operators respectively, all at precedence level p.

Figure 3.5: Grammars for parsing user-defined operators in Beluga.

$$
\begin{array}{rcl}
\langle e \rangle & ::= & \langle e \rangle_1 \\
\langle e \rangle_p & ::= & \langle \mathsf{op}_{\text{prefix}} \rangle_p \ \langle e_{\text{R}} \rangle_p \\
& | & \langle e \rangle_{p+1} \ \langle e_{\text{T}} \rangle_p \\
\langle e_{\text{T}} \rangle_p & ::= & \langle \mathsf{op}^{\text{N}}_{\text{infix}} \rangle_p \ \langle e \rangle_{p+1} \\
& | & \langle \mathsf{op}^{\text{R}}_{\text{infix}} \rangle_p \ \langle e_{\text{R}} \rangle_p \\
& | & \langle \mathsf{op}^{\text{L}}_{\text{infix}} \rangle_p \ \langle e \rangle_{p+1} \ \langle e_{\text{L}} \rangle_p \\
& | & \langle \mathsf{op}_{\text{postfix}} \rangle_p \ \langle e_{\text{L}} \rangle_p \\
& | & \varepsilon \\
\langle e_{\text{L}} \rangle_p & ::= & \langle \mathsf{op}_{\text{postfix}} \rangle_p \langle e_{\text{L}} \rangle_p \\
& | & \langle \mathsf{op}^{\text{L}}_{\text{infix}} \rangle_p \langle e \rangle_{p+1} \langle e_{\text{L}} \rangle_p \\
& | & \varepsilon \\
\langle e_{\text{R}} \rangle_p & ::= & \langle \mathsf{op}_{\text{prefix}} \rangle_p \langle e_{\text{R}} \rangle_p \\
& | & \langle e \rangle_{p+1} \langle e'_{\text{R}} \rangle_p \\
\langle e'_{\text{R}} \rangle_p & ::= & \langle \mathsf{op}^{\text{R}}_{\text{infix}} \rangle_p \langle e_{\text{R}} \rangle_p \\
& | & \varepsilon \\
\langle e \rangle_{P+1} & ::= & \mathbf{a}+
\end{array}
$$

(c) Factored grammar scheme derived from figure 3.5b for parsing user-defined operators. Beluga's parser additionally has failure productions interspersed to peek at the next token in the input stream and raise an exception if that token is an operator in an ambiguous position.

Figure 3.5: Grammars for parsing user-defined operators in Beluga (cont.).

3.4 Discussion

Despite the bugfixes and new features, the revised parser architecture is not without its flaws. The additional requirement that parsing must produce an unambiguous AST without resorting to complex disambiguation mechanisms prevents the overloading of some syntax. In particular, meta-objects and meta-types for substitutions now have to be prefixed by `$` to distinguish them from plain meta-objects and meta-types respectively. For the same reason, parameter meta-types have to be prefixed by `#`. Furthermore, overloading of identifiers had to be discarded so that user-defined operators in applications may be disambiguated using only a lookup table. Those breaking changes to the syntax could have been avoided had further semantic analysis been introduced during the disambiguation phase, namely to perform type-driven disambiguation. Should Beluga be extended to support multiple context variables in contexts, then this may be required. Likewise, should mixfix notations be a planned feature, then identifier overloading may need to be reintroduced with type-driven disambiguation since those notations typically require the reuse of names across different operators.

Additionally, further blurring of the grammars in section A.2 would be required to ensure that the context-free parsing phase does not have to perform infinite lookaheads. Indeed, there is a pathological case in parsing where an unbounded lookahead is required because of overlapping syntax in two different categories. Specifically, the opening parenthesis token is accepted by both the parser for meta-level types and the one for computation-level kinds and types. Instead of reworking the grammar to further blur the nonterminals ⟨*meta-thing*⟩ and ⟨*comp-sort-object*⟩ from sections A.2.3 and A.2.4, which could degrade the quality of syntax error messages, unbounded lookahead is enabled when parsing a meta-level type as part of a computation-level kind or type. The performance impact of this lookahead has not been measured, but it appears to be negligible based on the performance of running the

entire processing pipeline on the numerous examples that are part of the system's test suite. Nonetheless, the observation that additional grammar blurring is required to get from an $\mathsf{LL}(\infty)$ parser to an $\mathsf{LL}(1)$ parser for Beluga begs the question whether there should only be one encompassing nonterminal in the grammar of section A.2. This would make the language's grammar uniform, and allow for more flexibility in overloading syntactic constructs. However, since a parselet is required per precedence level in recursive descent parsing, and most static operators in Beluga have distinct precedences, the resulting increase in the recursion depth for parsing could deteriorate the runtime performance of the parser. Arguably, such a design presents more opportunities to improve the precision of syntax error messages during disambiguation.

Chapter 4

Indexing for Incremental Proof Development

This chapter presents implementation concerns with supporting incremental program development in Beluga and Harpoon. Some technical debt in those systems is highlighted to justify the reimplementation of the indexing phase. A formalism and an implementation of identifier resolution for Beluga are presented to showcase how to efficiently and correctly compute de Bruijn indices in an environment featuring multiple indexing contexts.

4.1 Introduction

As an interactive frontend for Beluga, the Harpoon system is designed such that editing a proof script by way of administrative commands or tactics is guaranteed to preserve the soundness of the proof with respect to the rest of the Beluga signature. This includes, among other things, ensuring that input terms are well-typed, that cases analyses cover all possible branches with respect to type family definitions, and that identifiers can be properly resolved once proof scripts are translated into programs. Chiefly, Harpoon's administrative

commands as presented in section 2.3 must work as intended in any valid order of execution so as to ensure that user's work does not get lost. Just like Abella, Isabelle/Isar, Coq and Agda, undoable commands in Harpoon must undo those commands' effects on both the proof script and the editing state. Unlike Abella though, undone commands can be redone, which allows users to move effortlessly forward and backward in the edit history. One feature that is unique to Harpoon is the ability to efficiently checkout the proof state for incomplete theorems occurring anywhere in a Beluga signature. This enables the development of proofs in a top-down fashion by first stating theorems and lemmas and then proving them, fully or partially, in any order that seems most productive. This is stronger than Abella's and Coq's subgoal deferring feature, which Harpoon also supports, in that theorems may be declared with respect to different subsets of the signature in the same proof session.

Unfortunately, case studies revealed that the implementation of Harpoon's theorem-switching feature can lead to invalid Beluga signatures when proof scripts developed out of order are translated into programs. Indeed, upon checking out an incomplete proof script, the signature is not reverted back to the state at which that proof is declared. This effectively means that definitions occurring later in a proof session appear in scope during interactive proof sessions. To soundly support switching between theorems, the Beluga system requires a mechanism to rollback the referencing state to the point where theorems are declared. This prompted the reimplementation of name resolution in Beluga and Harpoon to support visiting states.

4.2 The Legacy Indexing Phase

As outlined in section 2.2, the indexing phase of Beluga's processing pipeline is responsible for converting external syntax trees to corresponding approximate syntax trees. Specifically, references to constants are replaced with their corresponding symbolic identifiers defined in

the signature reconstruction store, and variables are replaced with their corresponding de Bruijn indices. This allows for later phases of semantic analysis (such as type-checking and termination analysis) to efficiently look up constant definitions and resolve variables to their binders without having to keep track of the identifiers in scope. In the implementation, not all variables are replaced with de Bruijn indices during indexing because binders for implicit parameters are not present in the AST; these are introduced during the abstraction phase [23].

The legacy implementation of indexing is additionally responsible for disambiguating the application of user-defined operators since that phase keeps track of constants as they are introduced in the signature. This disambiguation leverages Dijkstra's shunting-yard algorithm [17] as opposed to the recursive descent parsing algorithm presented in section 3.3. In the revised system, this responsibility has been moved to its own disambiguation phase, as presented in section 3.2, which requires its own implementation of name resolution. While this refactoring could have been sufficient in simplifying the processing pipeline, it uncovered significant technical debt and issues having to do with the way identifiers are handled during indexing.

The overarching store illustrated in figure 2.3 records the constant declarations encountered during signature reconstruction. These declarations are arranged in tables, with one table per kind of constant declaration, as illustrated in figure 4.1. Variable declarations, on the other hand, are arranged separately in association lists. This guarantees that there is no overlap between identifiers referring to objects of different kinds since they are not looked up in the same table or association list, which effectively creates distinct namespaces for them. For instance, LF type-level and term-level constants have their own declaration tables, separate from computation-level type constants and constructors. This design aims at ensuring that variables and constants originating from the LF, meta or computation levels do not end up appearing in terms of a different level [23]. Unfortunately, without having a unique table representing an entire referencing environment, name resolution in the presence

of shadowing proves obtuse and leads to unexpected results when coupled with overloading of identifiers. Indeed, when a constant identifier is resolved during indexing, the declaration tables in the store are looked up in a pre-defined order. As such, identifier lookups to not respect the global insertion order of constant and variable declarations. Instead, they are looked up first with respect to the order of identifier kinds, and then by the declaration order within the table for that identifier kind. Consequently, parts of the referencing environment do not exist depending on the expected kind of variable or constant encountered in the AST. For example, it is impossible in Beluga `v1.0` for computation-level coinductive type constant to shadow an inductive one simply because the declaration table for the former kind is always looked up after the declaration table for the latter kind.

Likewise, since identifiers at the meta-level and at the computation-level are kept separate, invalid references to computation-level objects in boxed terms are undetectable as they are instead systematically interpreted as free meta-variables. Counterintuitively, different parts of the referencing environment can also be interleaved in one program with this split table and association lists approach. Indeed, one can introduce a binder for a meta-level variable and another for a computation-level variable, both using the same identifier, and be able to use both meanings for that identifier in meta-level and computation-level expressions respectively because they effectively occupy different namespaces. The contrived but valid example program `f` below illustrates this: the colour-coded identifiers g and x are in Δ, and g and x are in Γ, in such a way that the function's body uses both overloaded meanings of those identifiers. This is admissible because only one namespace is made available at a time during name resolution based on the variant of AST node the identifier lies in.

```
LF tm : type = …;
schema ctx = …;
inductive Ex : ctype = …;
rec f : {g : ctx} → [g ⊢ tm] → Ex → Ex =
  mlam g ⇒ fn g ⇒ fn x ⇒ let [_ ⊢ x] = g in f [g] [g ⊢ x] x;
       1       2       3              4    2       1   1    4  3
```

This overloading does not translate well in proofs on paper since that results in an identifier being used to refer to two distinct objects in the same proof. It also challenges the identifier scoping rules suggested by BELUGA's concrete syntax. An error message like the following should be raised instead to ensure proper lexical scoping of variable declarations.

```
File "example.bel", line 5, column 48:
6 |  mlam g ⇒ fn g ⇒ fn x ⇒ let [_ ⊢ x] = g in f [g] [g ⊢ x] x;
                                                    ^

Error: Expected a context variable.
    File "example.bel", line 5, column 15:
    6 |  mlam g ⇒ fn g ⇒ fn x ⇒ let [_ ⊢ x] = g in f [g] [g ⊢ x] x;
                      ^

    Error: g is a bound computation variable.
```

Additionally, the legacy indexing algorithm is tightly coupled and implicitly dependent on the store of constants depicted in figure 2.3. This means that external mutations to that store affect the output of functions responsible for identifier resolution. As outlined in section 2.5, this design is only suitable for a single pass through the processing pipeline since only then is the store's state is consistent with how the AST is traversed. Indexing is only explicitly parameterized with respect to association lists for variable declarations. This allows HARPOON and REPL sessions instantiated in the very last program declared in a BELUGA signature to stay consistent with identifier resolution during signature reconstruction. Indeed, when navigating to a different HARPOON subgoal, the variable bindings introduced by the current subgoal can be discarded and replaced with those of the new subgoal. Incremental proof development sessions initiated on holes located anywhere else but the last declaration in the signature may result in invalid programs being generated. Since the store of constants is stuck at the state it ends up in after the signature is reconstructed, checking out those holes incorrectly brings later constants in scope. This can lead to shadowing of constants required for the proof development. In order to ensure soundness of incremental proof development with holes in proof scripts located anywhere in a signature, it is necessary for indexing to be explicitly parameterized over the entire referencing environment as opposed to just the

stores for variables.

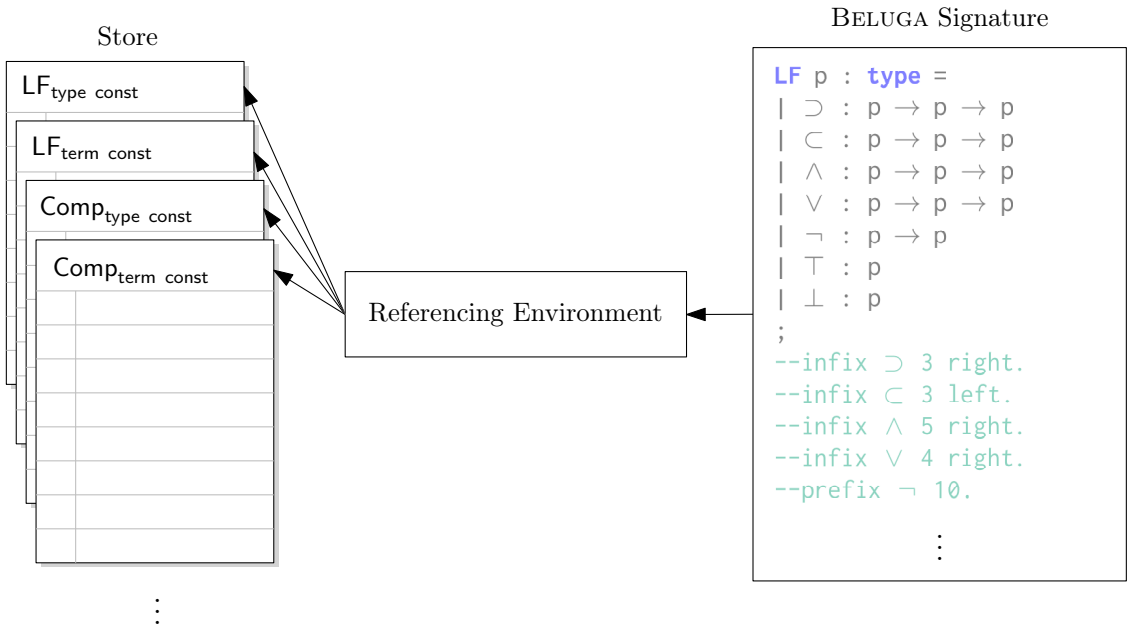


Figure 4.1: Diagram of the referencing environment structure as an AST visitor state and a mediator between the Beluga signature and the reconstruction store for name resolution.

The aforementioned incongruities in name resolution and tight coupling with the signature reconstruction store motivate the reimplementation of the indexing phase to use one uniform namespace while ensuring that identifiers from different domains cannot occur in terms in which they don't belong. Without having types available at this stage of processing, the accidental feature of identifier overloading had to be discarded to avoid postponing ambiguous references to type-checking.

4.3 Uniform Indexing

Uniform indexing is the conversion from a named AST to a locally nameless AST in a single pass and using a single lookup structure for all identifier kinds. By parameterizing procedures dealing with named binders to use such a lookup structure, we can resolve the soundness issue

with holes in programs by explicitly managing the state of that structure to correctly reflect what identifiers are in scope. For Beluga, implementing this requires having a merged view of the declaration tables for constants and variables so as to uniformly know what identifiers are in scope at any given AST node, like in figure 4.1. In particular, variable identifiers in the LF context, the meta-level context, and the computation-level context as illustrated in figure 4.2, which were designed to belong to different identifier worlds [22], now have to be mixed together for name resolution. Subsequent passes of the processing pipeline such as type reconstruction and type-checking may still safely treat those contexts as separate entities using the locally nameless AST. To solve the problem of uniform indexing, some definitions are in order.

LF contexts	Ψ	$::=$	$\cdot \mid \Psi, x : A$
Meta-level contexts	Δ	$::=$	$\cdot \mid \Delta, X : U$
Computation-level context	Γ	$::=$	$\cdot \mid \Gamma, x : \tau$

Figure 4.2: Definition of the indexing contexts in Beluga, with respect to which variables are indexed as de Bruijn indices. This builds upon the definitions of figure 2.2. A variable resolved to be at the LF level is indexed only with respect the LF context. An analogous rule applies to variables at the meta and computation levels.

A *frame* is a lexical region of an AST wherein variable declarations and references can be made. A *scope* is a frame in which name resolution is uniform, meaning that references are resolved to the same declaration site irrespective of the AST node in the scope. Whereas declarations can shadow each other within a frame, all declarations have to be distinct in a scope. A *fully qualified identifier* $x_1.x_2.\cdots.x_n$ is a list of plain identifiers x_1, x_2, …, x_n separated by dots denoting projection out of a namespace. A *referencing environment* in Beluga is the data structure that holds the associations from fully qualified identifiers to definitions in a signature. In essence, a referencing environment provides a window over a

signature for name resolution, and as such its state is restricted to representing one scope at a time. This is in line with its role as an AST visitor state.

During indexing, the referencing environment is updated by adding or removing identifiers in scope as AST nodes are traversed. This presents the challenge of cohesively handling namespaces for modules, patterns for computation-level pattern-matching expressions, and the computation of de Bruijn indices across different *indexing contexts* (Ψ, Δ and Γ). All the while, indexing has to feature some backtracking mechanism to support incremental proof development in Harpoon sessions. It is also worth noting that in the disambiguation phase, where a similar mapping occurs from the parser AST to the external AST, the visitor state has to additionally keep track of user-defined notations. This means that whichever structure is used to model referencing environments has to be extensible.

Every identifier declaration is allowed to shadow previous declarations in Beluga. The only exception to this rule is that identifiers in mutually recursive declarations have to be distinct. Everywhere else, declarations for constants and variables introduce their own scope. This means that a scope graph [31] representation of name resolution for Beluga programs is essentially a line graph, which is computationally inefficient for name resolution. As such, the referencing environment is instead modelled and implemented using frames as a basis.

Concretely, a referencing environment is a stack of frames, as illustrated in figure 4.3. Each frame holds a tree of bindings, wherein each node assigns to an identifier a stack of entries and optionally a subtree. The stacks of bindings provide a means of recalling what earlier definitions were assigned to identifiers in the current frame as the traversal exits a scope. Namespaces are handled uniformly since every binding may introduce one in the form of a subtree of bindings. The type of entry in a referencing environment is extensible, which allows for additional kinds of identifier references to be defined, and for data to be added to support different kinds of algorithms relying on the named AST.

For indexing specifically, the entries are annotated with labels for the various kinds of declarations in Beluga signatures (i.e., $\mathsf{LF}_{\mathsf{term}}$, $\mathsf{Context}$, $\mathsf{Comp}_{\mathsf{term}}$, $\mathsf{LF}_{\mathsf{type\ const}}$, $\mathsf{LF}_{\mathsf{term\ const}}$,

$$
\begin{array}{lrcl}
\text{Referencing environment} & \Xi & ::= & \cdot \mid \Xi; \mathcal{F} \\
\text{Frame} & \mathcal{F} & ::= & \mathbb{F} \mid \mathbb{M} \mid \mathbb{P} \\
\text{Plain frame} & \mathbb{F} & ::= & \mathbb{B} \\
\text{Module frame} & \mathbb{M} & ::= & (\mathbb{B}, \mathcal{T}) \\
\text{Pattern frame} & \mathbb{P} & ::= & (\mathbb{B}_{\mathrm{p}}, \mathbb{B}_{\mathrm{e}}) \\
\text{Bindings} & \mathbb{B} & ::= & (\mathcal{T}, |\Psi|, |\Delta|, |\Gamma|) \\
\text{Binding tree} & \mathcal{T} & ::= & \{x_i \mapsto \mathcal{S}_i\}_{i \in \{1,2,\ldots,n\}} \\
\text{Entry stack} & \mathcal{S} & ::= & \cdot \mid \mathcal{S}; (e, \mathcal{T}) \\
\text{Entry} & e & ::= & \cdots \mid (\mathsf{LF}_{\mathsf{term}}, |\Psi|) \mid (\mathsf{Context}, |\Delta|) \mid (\mathsf{Comp}_{\mathsf{term}}, |\Gamma|) \\
 & & & \quad \mid (\mathsf{LF}_{\mathsf{type\ const}}, \tilde{a}) \mid (\mathsf{LF}_{\mathsf{term\ const}}, \tilde{c})
\end{array}
$$

Figure 4.3: Definition of referencing environments for Beluga programs, assigning stacks of entries to identifiers with a stack of frames.

etc.), as well as indexing context sizes $|\Psi|$, $|\Delta|$ and $|\Gamma|$ for variables, and symbolic identifiers $\tilde{a}$ and $\tilde{c}$ for constants in the signature reconstruction store. Indexing context sizes are also kept track of in the structure for bindings to support computing de Bruijn indices.

The entry stacks and bindings tree structures of figure 4.3 can be flattened into an association list by keeping track of the insertion order for bindings. This is only used for presentation purposes, with $[x_1 : \tau_1, x_2 : \tau_2, \cdots, x_n : \tau_n]_{\mathbb{M}}$ denoting a module frame with bindings $x_1 : \tau_1$, $x_2 : \tau_2$, ..., $x_n : \tau_n$ like in figure 4.6b. To illustrate which identifiers are introduced in scope by a frame, the set of toplevel identifiers for frames and binding trees is defined in figure 4.5.

$$
\begin{array}{lcll}
\mathrm{domain}(\mathbb{F}) & = & \mathrm{domain}(\mathbb{B}) & \text{when } \mathbb{F} = \mathbb{B} \\
\mathrm{domain}(\mathbb{M}) & = & \mathrm{domain}(\mathbb{B}) & \text{when } \mathbb{M} = (\mathbb{B}, \_) \\
\mathrm{domain}(\mathbb{P}) & = & \mathrm{domain}(\mathbb{B}_{\mathrm{p}}) & \text{when } \mathbb{P} = (\mathbb{B}_{\mathrm{p}}, \_) \\
\mathrm{domain}(\mathbb{B}) & = & \mathrm{domain}(\mathcal{T}) & \text{when } \mathbb{B} = (\mathcal{T}, \_, \_, \_) \\
\mathrm{domain}(\mathcal{T}) & = & \{x_1, x_2, \ldots, x_n\} & \text{when } \mathcal{T} = \{x_i \mapsto \mathcal{S}_i\}_{i \in \{1,2,\ldots,n\}}
\end{array}
$$

Figure 4.5: Definition of the domain of frames in a Beluga referencing environment.

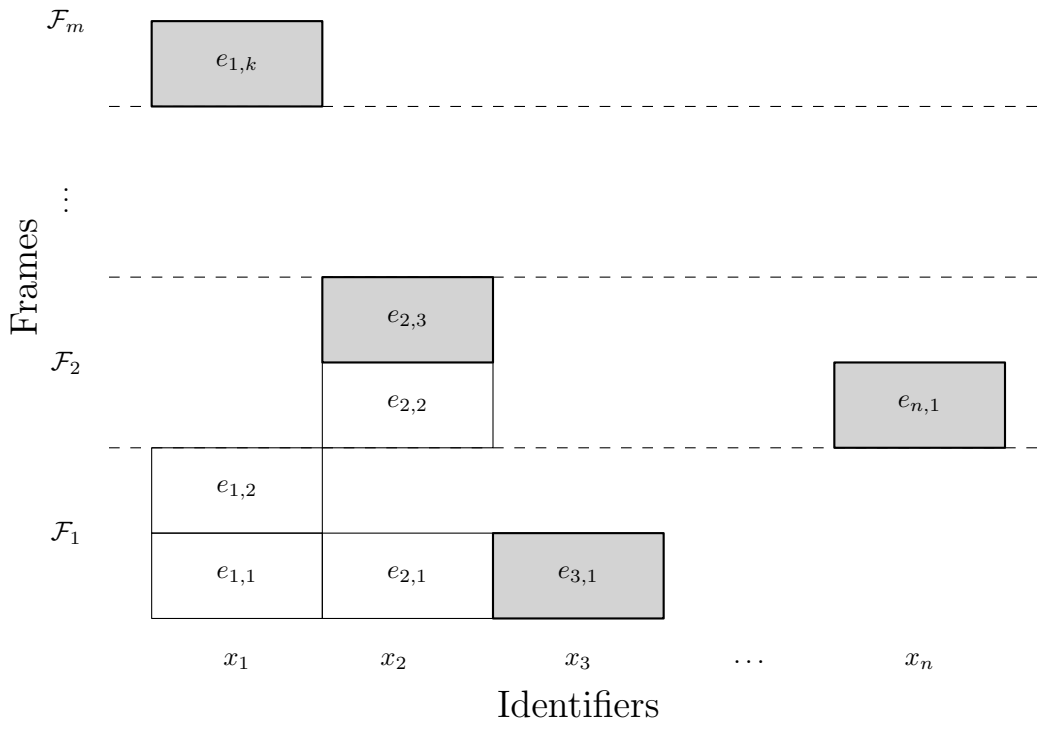


Figure 4.4: Example state for a referencing environment supporting name resolution as defined in figure 4.3. This referencing environment maps identifiers $x_1, x_2, \ldots, x_n$ to entries $e_{i,j}$ in frames $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_m$. Frames are illustrated in a stack growing upwards. Likewise, entries in a given frame are arranged in stacks, like entries $e_{1,1}$ and $e_{1,2}$ bound to x_1 in frame $\mathcal{F}_1$. Had the referencing environment been formulated with respect to scopes, this would not be admissible. The order of insertion for bindings is unknown. The bindings reachable in this state for name resolution are $(x_1, e_{1,k})$, $(x_2, e_{2,3})$, $(x_3, e_{3,1})$, $\ldots$, $(x_n, e_{n,2})$ and shown in gray. These bound entries are the topmost ones for each identifier. Indeed, binding $(x_1, e_{1,k})$ shadows $(x_1, e_{1,2})$, which in turn shadows $(x_1, e_{1,1})$. To support namespaces, each entry is associated with a tree of bindings, like in figure 4.6c.

4.3.1 Frames for Patterns and Modules

Since Beluga supports defining constants in modules, and pattern-matching in computation-level programs, three distinct kinds of frames are designed to operate differently in the way bindings are added or looked up. Figure 4.7 illustrates how these frames are added to and removed from referencing environments to support those features.

1. *Plain frames* $\mathbb{F}$: this kind of frame does not have additional mechanisms. Plain frames are suitable for introducing multiple identifiers in such a way that they can be efficiently removed by discarding the frame entirely.

2. *Module frames* $\mathbb{M}$: constants added to this kind of frame are either private or public. Only public constant declarations are exported from the module they appear in when it is opened, in which case they are brought in scope. This ensures that constants and notations imported from external modules are not reexported when the module is opened elsewhere. To support this, when a constant binding is added to a module frame $(\mathbb{B}, \mathcal{T})$, it is added to both $\mathbb{B}$ for name resolution, and to the tree of exported constants $\mathcal{T}$ for opening modules. This is illustrated in figure 4.6.

3. *Pattern frames* $\mathbb{P}$: variables added to this kind of frame are inner pattern-bound or pattern variables. Inner pattern-bound variables are introduced by binders in the pattern, like c and x in the contextual lambda term pattern `[c : term → term ⊢ Term.lam (\x. c x)]`. Pattern variables on the other hand are free variables in the pattern, which become bound variables in the body of the match case. For instance, the pattern `[g ⊢ Term.app M1 M2]` of example 2.1d introduces `g`, `M1` and `M2` in scope. For a pattern frame $(\mathbb{B}_\mathrm{p}, \mathbb{B}_\mathrm{e})$, the bindings in $\mathbb{B}_\mathrm{p}$ are used for name resolution in the pattern, whereas the bindings in $\mathbb{B}_\mathrm{e}$ are only used for name resolution in the body of the pattern-matching branch. This means that inner pattern-bound variables are added to $\mathbb{B}_\mathrm{p}$, and free pattern variables are added to $\mathbb{B}_\mathrm{e}$.

```
module Nat = struct
  LF nat : type =
  | z : nat
  | s : nat → nat;
  rec plus : [ ⊢ nat] → [ ⊢ nat] → [ ⊢ nat] = ?h1;
end
rec f : [ ⊢ Nat.nat] → [ ⊢ Nat.nat] = ?h2;
--open Nat.
rec g : [ ⊢ nat] → [ ⊢ nat] = ?h3;
```

(a) Excerpt of a BELUGA signature with holes ?h1, ?h2 and ?h3.

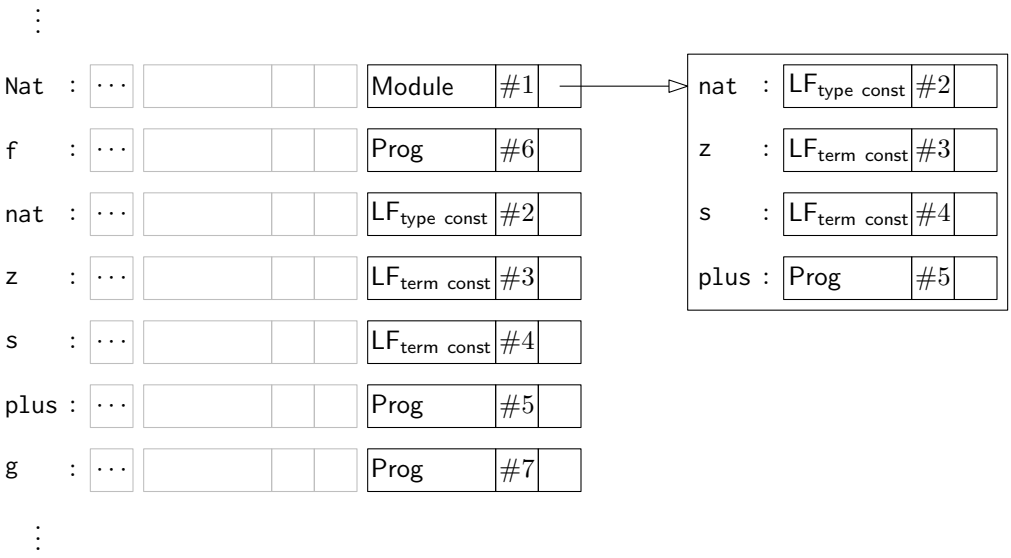
$$\Xi_{?h1} = [\ldots]_{\mathbb{M}}; [\mathsf{nat}^* : \mathsf{LF}_{\mathsf{type\ const}}, \mathsf{z}^* : \mathsf{LF}_{\mathsf{term\ const}}, \mathsf{s}^* : \mathsf{LF}_{\mathsf{term\ const}}, \mathsf{plus}^* : \mathsf{Prog}]_{\mathbb{M}}$$

$$\Xi_{?h2} = [\ldots, \mathsf{Nat}^* : (\mathsf{Module}, \{\mathsf{nat} : \_, \mathsf{z} : \_, \mathsf{s} : \_, \mathsf{plus} : \_\}), \mathsf{f}^* : \mathsf{Prog}]_{\mathbb{M}}$$

$$\Xi_{?h3} = [\ldots, \mathsf{Nat}^* : \_, \mathsf{f}^* : \_, \mathsf{nat} : \_, \mathsf{z} : \_, \mathsf{s} : \_, \mathsf{plus} : \_, \mathsf{g}^* : \mathsf{Prog}]_{\mathbb{M}}$$

(b) The referencing environments $\Xi_{?h1}$, $\Xi_{?h2}$ and $\Xi_{?h3}$ for the holes ?h1, ?h2 and ?h3 in figure 4.6a, denoted using association lists to illustrate the semantics of opening a module as in figure 4.7. Some annotations to the bindings are omitted for brevity. Bindings annotated with an asterisk $*$ denote constants that are exported from the module. BELUGA does not treat files as modules like in OCAML, so all declarations in the toplevel module carry over to the next signature file in the project configuration. Nested modules, however, respect the semantics of exporting declarations.

Figure 4.6: Handling of modules in referencing environments.



(c) Diagram of the referencing environment at hole ?h3 from figure 4.6a as it is implemented in Beluga v1.1 using dictionaries and stacks. The identifiers on the left are part of the toplevel module frame, and are each associated with a stack of bindings, with the most recent binding on the right end of the stack. The numbers #1, #2, ..., #7 denote the symbolic identifiers generated during indexing and associated with the constants in the signature reconstruction store. The aliases introduced by opening module Nat are handled properly, such that constant #3 can be referenced either as Nat.z or z at that point in the signature.

Figure 4.6: Handling of modules in referencing environments (cont.).

$$
\begin{array}{lcll}
\text{enter\_frame}(\Xi;\mathbb{F}) & = & \Xi;\mathbb{F};(\{\},|\Psi|,|\Delta|,|\Gamma|) & \mathbb{F}=(\_,|\Psi|,|\Delta|,|\Gamma|) \\
\text{enter\_frame}(\Xi;\mathbb{M}) & = & \Xi;\mathbb{M};(\{\},|\Psi|,|\Delta|,|\Gamma|) & \mathbb{M}=((\_,|\Psi|,|\Delta|,|\Gamma|),\_) \\
\text{leave\_frame}(\Xi;\mathbb{F}) & = & \Xi & \\
 & & & \\
\text{enter\_module}(\Xi;\mathbb{M}) & = & \Xi;\mathbb{M};((\{\},|\Psi|,|\Delta|,|\Gamma|),\{\}) & \mathbb{M}=((\_,|\Psi|,|\Delta|,|\Gamma|),\_) \\
\text{enter\_module}(\cdot) & = & ((\{\},0,0,0),\{\}) & \\
\text{leave\_module}_x(\Xi;\mathbb{M}) & = & \text{push}\,[x:(\mathsf{Module},\mathcal{T})]\,(\Xi) & \mathbb{M}=(\_,\mathcal{T}) \\
 & & & \\
\text{enter\_pattern}(\Xi;\mathbb{F}) & = & \Xi;\mathbb{F};((\{\},0,0,0),(\{\},|\Psi|,|\Delta|,|\Gamma|)) & \mathbb{F}=(\_,|\Psi|,|\Delta|,|\Gamma|) \\
\text{enter\_pattern}(\Xi;\mathbb{M}) & = & \Xi;\mathbb{M};((\{\},0,0,0),(\{\},|\Psi|,|\Delta|,|\Gamma|)) & \mathbb{M}=((\_,|\Psi|,|\Delta|,|\Gamma|),\_) \\
\text{leave\_pattern}(\Xi;\mathbb{P}) & = & \Xi;\mathbb{B}_{\mathrm{e}} & \mathbb{P}=(\_,\mathbb{B}_{\mathrm{e}})
\end{array}
$$

Figure 4.7: Definition for adding and removing frames in referencing environments. This showcases in particular how indexing context sizes are carried over from outer frames, how constant declarations in modules are kept track of, and how frames are constructed for pattern-matching body expressions.

4.3.2 Operations on Referencing Environments

The previous section described the introduction and elimination principles for frames. This section focusses on the operations on referencing environments specifically used in supporting the incremental indexing of Beluga programs.

As defined earlier, a referencing environment (denoted by Ξ) is a uniform representation of the identifiers in scope at any given node in a named AST representation of a program. An indexing context is a subsequence of a referencing environment in order of insertions, and it is used to compute de Bruijn indices for variables belonging to that context. In Beluga, the LF context, the meta-level context and the computation-level context are disjoint indexing contexts, whose bindings are interleaved in the referencing environment. The abstract data type of referencing environments has the following operations, where Ψ, used as a subscript,

is a generic label referring to an indexing context:

- $\text{push}_\Psi\,[x:\tau]\,(\Xi)$ adds the binding $x:\tau$ onto Ξ as part of the indexing context Ψ. This increments the size of Ψ.
- $\text{pop}_\Psi\,[x]\,(\Xi)$ removes the latest binding of x from Ξ, while assuming it is part of the indexing context Ψ. This decrements the size of Ψ.
- $\text{shift}_\Psi\,(\Xi)$ increments the size of the indexing context Ψ in Ξ.
- $\text{unshift}_\Psi\,(\Xi)$ decrements the size of the indexing context Ψ in Ξ.
- $\text{lookup}\,[x]\,(\Xi)$ is the latest binding $x:\tau$ associated with the identifier x in Ξ.
- $\text{index}_\Psi\,[x]\,(\Xi)$ is the de Bruijn index of x in Ξ with respect to the indexing context Ψ. This only succeeds if $\text{lookup}\,[x]\,(\Xi)$ is a binding in Ψ.

Identifier lookup and index operations are separate since $\text{lookup}\,[x]\,(\Xi)$ considers the entire referencing environment Ξ whereas $\text{index}_\Psi\,[x]\,(\Xi)$ is restricted to the indexing context Ψ in Ξ. As discussed later, the implementation obviates the need for an additional traversal of the referencing environement specifically for the index operation. Popping a binding from the referencing environment is the inverse of pushing that binding, such that $\text{pop}_\Psi\,[x]\,(\text{push}_\Psi\,[x:\tau]\,(\Xi)) = \Xi$. Analogously, $\text{unshift}_\Psi\,(\text{shift}_\Psi\,(\Xi)) = \Xi$. Additionally, the environments constructed as $\text{push}_\Psi\,[x:\mathsf{LF}_{\mathsf{term}}]\,(\Xi)$ and $(\Xi, x:\mathsf{LF}_{\mathsf{term}})$ are equivalent, and so are $\text{shift}_\Psi\,(\Xi)$ and $(\Xi, \_:\mathsf{LF}_{\mathsf{term}})$ since the LF indexing context is only comprised of bindings for LF terms.

By design, the operations that add or remove bindings only affect the topmost frame in the referencing environment so as to respect the nesting structure of frames. The lookup and index operations, however, traverse the entire structure. Lookups are defined intuitively as successive lookups in frames, then lookups in trees of bindings, as illustrated in figure 4.8. This ensures that the topmost entry bound to the identifier is found like in figure 4.4. The

exception to this rule has to do with lookups starting from pattern frames since variable declarations in external frames cannot be referenced in a pattern. Computing de Bruijn indices follows the same lookup procedure to stay consistent with identifier shadowing.

$$
\begin{array}{lcll}
\mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\Xi;\mathbb{F}) & = & \mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\mathbb{F}) & \text{when } x_1 \in \mathrm{domain}(\mathbb{F}) \\
\mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\Xi;\mathbb{F}) & = & \mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\Xi) & \text{when } x_1 \notin \mathrm{domain}(\mathbb{F}) \\
\mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\Xi;\mathbb{M}) & = & \mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\mathbb{M}) & \text{when } x_1 \in \mathrm{domain}(\mathbb{M}) \\
\mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\Xi;\mathbb{M}) & = & \mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\Xi) & \text{when } x_1 \notin \mathrm{domain}(\mathbb{M}) \\
\mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\Xi;\mathbb{P}) & = & \mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\mathbb{P}) & \text{when } x_1 \in \mathrm{domain}(\mathbb{P}) \\
\mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\Xi;\mathbb{P}) & = & \mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\Xi) & \text{when } x_1 \notin \mathrm{domain}(\mathbb{P}) \text{ and} \\
 & & & \mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\Xi) \text{ is not} \\
 & & & \text{a variable} \\
\mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\mathbb{F}) & = & \mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\mathbb{B}) & \text{when } \mathbb{F} = \mathbb{B} \\
\mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\mathbb{M}) & = & \mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\mathbb{B}) & \text{when } \mathbb{M} = (\mathbb{B}, \_) \\
\mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\mathbb{P}) & = & \mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\mathbb{B}_\mathrm{p}) & \text{when } \mathbb{P} = (\mathbb{B}_\mathrm{p}, \_) \\
\mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,((\mathcal{T}, \_, \_, \_)) & = & \mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,(\mathcal{T}) & \\
\mathrm{lookup}\,[x_1.x_2.\cdots.x_n]\,\left(\{y_i \mapsto \mathcal{S}_i\}_{i\in\{1,2,\ldots,m\}}\right) & = & \mathrm{lookup}\,[x_2.x_3.\cdots.x_n]\,(\mathcal{T}) & \text{when } \exists i \in \{1,2,\ldots,m\}, x_i = y_i \\
 & & & \text{and } \mathcal{S}_i = \cdots;(\_, \mathcal{T}) \\
\mathrm{lookup}\,[x]\,\left(\{y_i \mapsto \mathcal{S}_i\}_{i\in\{1,2,\ldots,m\}}\right) & = & e & \text{when } \exists i \in \{1,2,\ldots,m\}, x = y_i \\
 & & & \text{and } \mathcal{S}_i = \cdots;(e, \_) \\
\mathrm{lookup}\,[\_]\,(\_) & = & \lightning &
\end{array}
$$

Figure 4.8: Definition of lookups in referencing environments.

In contrast with association lists, the use of dictionaries in the referencing environment means that the order of insertions for bindings is not readily available. This poses a problem for the computation of de Bruijn indices. Thankfully, it suffices to keep track of the indexing context sizes $|\Psi|$, $|\Delta|$ and $|\Gamma|$ in each frame to record the de Bruijn level of variables as they are inserted in the environment, like in figures 4.3 and 4.7. This recorded level can then be used to recover de Bruijn indices efficiently, as illustrated in figure 4.9. Indeed, if x is a variable in the domain of the indexing context Ψ', and Ψ is the indexing context immediately

before that declaration for x was added, then $|\Psi'| > |\Psi|$, and the de Bruijn index of x with respect to Ψ' is $|\Psi'| - |\Psi|$. This is the same computation as subtracting the de Bruijn level of x from the current abstraction depth [14, 28]. Chiefly, this approach is scalable to arbitrarily many indexing contexts, provided that the context to use for a variable is known at its binding site. Additionally, not having to actually construct Ψ, Δ and Γ simplifies shifting contexts with $\text{shift}_{\Psi}(\Xi)$ in cases like $\mathsf{const} = \lambda x.\lambda\_.\ x$ wherein the identifier introduced by a binder is omitted since it is unused in the abstraction's body. Indeed, it suffices to increase the indexing context's size to shift subsequent de Bruijn indices, so there is no need to add an empty binding in the referencing environment.

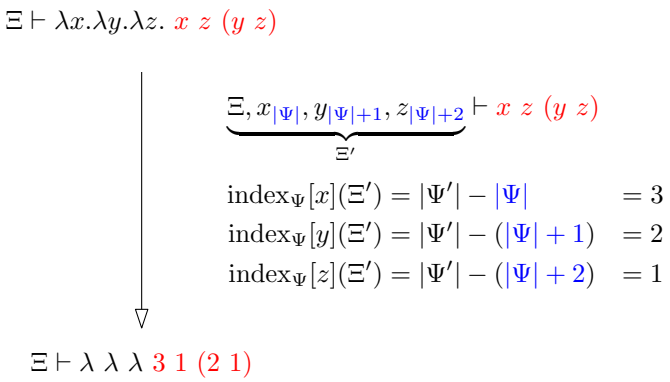


Figure 4.9: Computation of de Bruijn indices for the **S**-combinator using indexing context sizes stored in the lookup table. The ambient LF context Ψ may be non-empty, in which case its bindings are included in the referencing environment Ξ. This ensures that the combinator may be indexed correctly when it is nested under additional abstractions like in $\lambda v.\lambda w.(\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z))$. The indexing context Ψ' is built upon Ψ by adding the bindings for x, y and z introduced by LF abstractions. We only need the context sizes though, and $|\Psi'| = |\Psi| + 3$ in this case, with $|\Psi|$ and $|\Psi'|$ being kept track of in Ξ and Ξ' respectively. Variables in Ξ' are annotated with the size of the LF indexing context before each of them was added to the referencing environment, namely $|\Psi|$ for x, $|\Psi| + 1$ for y and $|\Psi| + 2$ for z. This allows for efficient computation of the de Bruijn indices of x, y and z when $x\ z\ (y\ z)$ is indexed with respect to Ψ' in Ξ'.

4.3.3 Indexing LF Kinds, Types and Terms

Using the definition for referencing environments in Beluga, the indexing phase is defined inductively as a family of translations functions indexed by the input's syntactic class. These are of the form $\Xi \vdash e \rightsquigarrow_s \tilde{e}$, with e and $\tilde{e}$ being the input and output terms respectively, and Ξ being the referencing environment. However, that style of specification implicitly assumes that Ξ is implemented as an immutable data structure. To rectify this, a Hoare-style specification like $\{\mathsf{Env} = \Xi\}\ e \rightsquigarrow_s \tilde{e}\ \{\mathsf{Env} = \Xi'\}$ can be used instead, with Ξ and Ξ'

being the initial and final states of the referencing environment. The translation $\rightsquigarrow_s$ is split following the semantic classes of figure 2.2, with s being one of $K, A, M, U, C, \kappa, \tau, e$ and i, with the addition of Mp, Cp and p for LF patterns, meta-object patterns and computation-level patterns respectively. This subsection focusses on indexing for LF kinds, types and terms, and presents both styles of specifications $\Xi \vdash e \rightsquigarrow_s \tilde{e}$ and $\{\mathsf{Env} = \Xi\}\ e \rightsquigarrow_s \tilde{e}\ \{\mathsf{Env} = \Xi'\}$, then demonstrates their equivalence.

First, the definitions for plain LF kinds, types and terms are recalled below, with a and c standing for type-level and term-level constants respectively, and x ranging over variables.

Plain LF kinds	K	$::=$	$A \to K \mid \Pi x{:}A.K \mid \mathbf{type}$
Plain LF types	A, B	$::=$	$A \to B \mid \Pi x{:}A.B \mid a \mid A\ M_1\ M_2 \ldots M_n$
Plain LF terms	M, N	$::=$	$x \mid c \mid \lambda x.M \mid M\ N_1\ N_2 \ldots N_n \mid M : A$

Then, the nameless counterparts of these LF kinds, types and terms are defined, with $\tilde{a}$ and $\tilde{c}$ denoting the symbolic identifier corresponding to the type and term constants a and c respectively, and ι ranging over de Bruijn indices corresponding to variables.

Nameless LF kinds	$\tilde{K}$	$::=$	$\tilde{A} \to \tilde{K} \mid \Pi_{\tilde{A}}\ \tilde{K} \mid \mathbf{type}$
Nameless LF types	$\tilde{A}, \tilde{B}$	$::=$	$\tilde{A} \to \tilde{B} \mid \Pi_{\tilde{A}}\ \tilde{B} \mid \tilde{a} \mid \tilde{A}\ \tilde{M}_1\ \tilde{M}_2 \ldots \tilde{M}_n$
Nameless LF terms	$\tilde{M}, \tilde{N}$	$::=$	$\iota \mid \tilde{c} \mid \lambda\ \tilde{M} \mid \tilde{M}\ \tilde{N}_1\ \tilde{N}_2 \ldots \tilde{N}_n \mid \tilde{M} : \tilde{A}$

Indexing is first defined with respect to an immutable representation of the referencing environment Ξ as an association list. This algorithm is easy to implement, but comes at the expense of degraded runtime performance for lookups. It is assumed that these computations take place in either a plain or module frame.

$\boxed{\Xi \vdash K \rightsquigarrow_{\mathrm{K}} \tilde{K}}$: in the referencing environment Ξ, the LF kind K is indexed as $\tilde{K}$

$$\frac{\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A} \quad \Xi, \_ : \mathsf{LF}_{\mathsf{term}} \vdash K \rightsquigarrow_{\mathrm{K}} \tilde{K}}{\Xi \vdash A \to K \rightsquigarrow_{\mathrm{K}} \tilde{A} \to \tilde{K}} \tag{4.1}$$

$$\frac{\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A} \quad \Xi, x : \mathsf{LF}_{\mathsf{term}} \vdash K \rightsquigarrow_{\mathrm{K}} \tilde{K}}{\Xi \vdash \Pi x{:}A.K \rightsquigarrow_{\mathrm{K}} \Pi_{\tilde{A}}\ \tilde{K}} \tag{4.2}$$

$$\frac{}{\Xi \vdash \mathbf{type} \rightsquigarrow_{\mathrm{K}} \mathbf{type}} \tag{4.3}$$

$\boxed{\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}}$: in the referencing environment Ξ, the LF type A is indexed as $\tilde{A}$

$$\frac{\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A} \quad \Xi, \_ : \mathsf{LF}_{\mathsf{term}} \vdash B \rightsquigarrow_{\mathrm{A}} \tilde{B}}{\Xi \vdash A \to B \rightsquigarrow_{\mathrm{A}} \tilde{A} \to \tilde{B}} \tag{4.4}$$

$$\frac{\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A} \quad \Xi, x : \mathsf{LF}_{\mathsf{term}} \vdash B \rightsquigarrow_{\mathrm{A}} \tilde{B}}{\Xi \vdash \Pi x{:}A.B \rightsquigarrow_{\mathrm{A}} \Pi_{\tilde{A}}\ \tilde{B}} \tag{4.5}$$

$$\frac{\mathrm{lookup}\,[a]\,(\Xi) = \tilde{a} : \mathsf{LF}_{\mathsf{type\ const}}}{\Xi \vdash a \rightsquigarrow_{\mathrm{A}} \tilde{a}} \tag{4.6}$$

$$\frac{\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A} \quad \left(\Xi \vdash M_k \rightsquigarrow_{\mathrm{M}} \tilde{M}_k\right)_{k \in \{1,2,\ldots,n\}}}{\Xi \vdash A\ M_1\ M_2 \ldots M_n \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \tilde{M}_1\ \tilde{M}_2 \ldots \tilde{M}_n} \tag{4.7}$$

$\boxed{\Xi \vdash M \rightsquigarrow_{\mathrm{M}} \tilde{M}}$: in the referencing environment Ξ, the LF term M is indexed as $\tilde{M}$

$$\frac{\mathrm{lookup}\,[x]\,(\Xi) = x : \mathsf{LF}_{\mathsf{term}} \quad \mathrm{index}_{\Psi}\,[x]\,(\Xi) = \iota}{\Xi \vdash x \rightsquigarrow_{\mathrm{M}} \iota} \tag{4.8}$$

$$\frac{\mathrm{lookup}\,[c]\,(\Xi) = \tilde{c} : \mathsf{LF}_{\mathsf{term\ const}}}{\Xi \vdash c \rightsquigarrow_{\mathrm{M}} \tilde{c}} \tag{4.9}$$

$$\frac{\Xi, x : \mathsf{LF}_{\mathsf{term}} \vdash M \rightsquigarrow_{\mathrm{M}} \tilde{M}}{\Xi \vdash \lambda x.M \rightsquigarrow_{\mathrm{M}} \lambda\ \tilde{M}} \tag{4.10}$$

$$\frac{\Xi \vdash M \rightsquigarrow_{\mathrm{M}} \tilde{M} \quad \left(\Xi \vdash N_k \rightsquigarrow_{\mathrm{M}} \tilde{N}_k\right)_{k \in \{1,2,\ldots,n\}}}{\Xi \vdash M\ N_1\ N_2 \ldots N_n \rightsquigarrow_{\mathrm{A}} \tilde{M}\ \tilde{N}_1\ \tilde{N}_2 \ldots \tilde{N}_n} \tag{4.11}$$

$$\frac{\Xi \vdash M \rightsquigarrow_{\mathrm{M}} \tilde{M} \quad \Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}}{\Xi \vdash M : A \rightsquigarrow_{\mathrm{A}} \tilde{M} : \tilde{A}} \tag{4.12}$$

In the implementation, indexing is imperative and defined with respect to a mutable referencing environment that is passed along for each computation. The fact that computations may mutate the state is reflected formally using a Hoare-style specification.

$\boxed{\{P\}\ K \rightsquigarrow_{\mathrm{K}} \tilde{K}\ \{Q\}}$: the LF kind K is indexed as $\tilde{K}$ with precondition P and postcondition Q

$$\frac{\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi'\} \quad \{\mathsf{Env} = \mathrm{shift}_{\Psi}\,(\Xi')\}\ K \rightsquigarrow_{\mathrm{K}} \tilde{K}\ \{\mathsf{Env} = \Xi''\}}{\{\mathsf{Env} = \Xi\}\ A \to K \rightsquigarrow_{\mathrm{K}} \tilde{A} \to \tilde{K}\ \{\mathsf{Env} = \mathrm{unshift}_{\Psi}\,(\Xi'')\}} \tag{4.13}$$

$$\frac{\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi'\} \quad \{\mathsf{Env} = \mathrm{push}_{\Psi}\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi')\}\ K \rightsquigarrow_{\mathrm{K}} \tilde{K}\ \{\mathsf{Env} = \Xi''\}}{\{\mathsf{Env} = \Xi\}\ \Pi x{:}A.K \rightsquigarrow_{\mathrm{K}} \Pi_{\tilde{A}} \tilde{K}\ \{\mathsf{Env} = \mathrm{pop}_{\Psi}\,[x]\,(\Xi'')\}} \tag{4.14}$$

$$\overline{\{\mathsf{Env} = \Xi\}\ \mathbf{type} \rightsquigarrow_{\mathrm{K}} \mathbf{type}\ \{\mathsf{Env} = \Xi\}} \tag{4.15}$$

$\boxed{\{P\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{Q\}}$: the LF type A is indexed as $\tilde{A}$ with precondition P and postcondition Q

$$\frac{\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi'\} \quad \{\mathsf{Env} = \mathrm{shift}_{\Psi}\,(\Xi')\}\ B \rightsquigarrow_{\mathrm{A}} \tilde{B}\ \{\mathsf{Env} = \Xi''\}}{\{\mathsf{Env} = \Xi\}\ A \to B \rightsquigarrow_{\mathrm{A}} \tilde{A} \to \tilde{B}\ \{\mathsf{Env} = \mathrm{unshift}_{\Psi}\,(\Xi'')\}} \tag{4.16}$$

$$\frac{\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi'\} \quad \{\mathsf{Env} = \mathrm{push}_{\Psi}\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi')\}\ B \rightsquigarrow_{\mathrm{A}} \tilde{B}\ \{\mathsf{Env} = \Xi''\}}{\{\mathsf{Env} = \Xi\}\ \Pi x{:}A.B \rightsquigarrow_{\mathrm{A}} \Pi_{\tilde{A}}\ \tilde{B}\ \{\mathsf{Env} = \mathrm{pop}_{\Psi}\,[x]\,(\Xi'')\}} \tag{4.17}$$

$$\frac{\mathrm{lookup}\,[a]\,(\Xi) = \tilde{a} : \mathsf{LF}_{\mathsf{type\ const}}}{\{\mathsf{Env} = \Xi\}\ a \rightsquigarrow_{\mathrm{A}} \tilde{a}\ \{\mathsf{Env} = \Xi\}} \tag{4.18}$$

$$\frac{\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi_0\} \quad \left(\{\mathsf{Env} = \Xi_{k-1}\}\ M_k \rightsquigarrow_{\mathrm{M}} \tilde{M}_k\ \{\mathsf{Env} = \Xi_k\}\right)_{k \in \{1,2,\ldots,n\}}}{\{\mathsf{Env} = \Xi\}\ A\ M_1\ M_2 \ldots M_n \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \tilde{M}_1\ \tilde{M}_2 \ldots \tilde{M}_n\ \{\mathsf{Env} = \Xi_n\}} \tag{4.19}$$

$\boxed{\{P\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{Q\}}$: the LF term M is indexed as $\tilde{M}$ with precondition P and postcondition Q

$$\frac{\mathrm{lookup}\,[x]\,(\Xi) = x : \mathsf{LF}_{\mathsf{term}} \quad \mathrm{index}_{\Psi}\,[x]\,(\Xi) = \iota}{\{\mathsf{Env} = \Xi\}\ x \rightsquigarrow_{\mathrm{M}} \iota\ \{\mathsf{Env} = \Xi\}} \tag{4.20}$$

$$\frac{\mathrm{lookup}\,[c]\,(\Xi) = \tilde{c} : \mathsf{LF}_{\mathsf{term\ const}}}{\{\mathsf{Env} = \Xi\}\ c \rightsquigarrow_{\mathrm{M}} \tilde{c}\ \{\mathsf{Env} = \Xi\}} \tag{4.21}$$

$$\frac{\{\mathsf{Env} = \mathrm{push}_{\Psi}\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi)\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \Xi'\}}{\{\mathsf{Env} = \Xi\}\ \lambda x.M \rightsquigarrow_{\mathrm{M}} \lambda\ \tilde{M}\ \{\mathsf{Env} = \mathrm{pop}_{\Psi}\,[x]\,(\Xi')\}} \tag{4.22}$$

$$\frac{\{\mathsf{Env} = \Xi\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \Xi_0\} \quad \left(\{\mathsf{Env} = \Xi_{k-1}\}\ N_k \rightsquigarrow_{\mathrm{M}} \tilde{N}_k\ \{\mathsf{Env} = \Xi_k\}\right)_{k \in \{1,2,\ldots,n\}}}{\{\mathsf{Env} = \Xi\}\ M\ N_1\ N_2 \ldots N_n \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \tilde{N}_1\ \tilde{N}_2 \ldots \tilde{N}_n\ \{\mathsf{Env} = \Xi_n\}} \tag{4.23}$$

$$\frac{\{\mathsf{Env} = \Xi\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \Xi'\} \quad \{\mathsf{Env} = \Xi'\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi''\}}{\{\mathsf{Env} = \Xi\}\ M : A \rightsquigarrow_{\mathrm{M}} \tilde{M} : \tilde{A}\ \{\mathsf{Env} = \Xi''\}} \tag{4.24}$$

Neither formulation of the indexing algorithm is total since the identifier and index

lookup operations fail for unbound identifiers. In the immutable setting, recovering from such failures is trivial since no effects are applied to the association list of bindings. On the other hand, the mutable setting does not make any guarantee as to the state of the referencing environment after a computation. This potentially allows insertions to occur in an unpredictable order, and for deletions to unintentionally remove bindings that appear in outer frames of the initial referencing environment. This next theorem certifies that that is not the case: if a binding removal occurs during indexing, then it is immediately preceded by the addition of that same binding.

In fact, the referencing environment is restored to its initial state after successfully indexing an LF object. This means that in the event of a failure during indexing, the environment has at least the bindings of its initial state. Hence, pushing a new frame onto the referencing environment and popping it in case of failure is a sufficient mechanism to undo any and all effects on the state, which is paramount for incremental developments. Additionally, since the ordering of add and remove operations is preserved, it follows that the computation of de Bruijn indices in the mutable setting are correct.

Theorem 2 (Equivalence)**.** *The formalisms for indexing with respect to the immutable and mutable representations of the referencing environment are equivalent.*

1. $\{\mathsf{Env} = \Xi\}\ K \rightsquigarrow_{\mathrm{K}} \tilde{K}\ \{\mathsf{Env} = \Xi'\}$ *if and only if* $\Xi \vdash K \rightsquigarrow_{\mathrm{K}} \tilde{K}$ *and* $\Xi' = \Xi$.

2. $\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi'\}$ *if and only if* $\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}$ *and* $\Xi' = \Xi$.

3. $\{\mathsf{Env} = \Xi\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \Xi'\}$ *if and only if* $\Xi \vdash M \rightsquigarrow_{\mathrm{M}} \tilde{M}$ *and* $\Xi' = \Xi$.

Proof. See appendix B. □

Corollary. *Successfully indexing an LF kind, type or term using the Hoare-style algorithm specification restores the referencing environment to its initial state.*

1. *If* $\{\mathsf{Env} = \Xi\}\ K \rightsquigarrow_{\mathrm{K}} \tilde{K}\ \{\mathsf{Env} = \Xi'\}$, *then* $\Xi' = \Xi$.

2. *If* $\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi'\}$, *then* $\Xi' = \Xi$.

3. *If* $\{\mathsf{Env} = \Xi\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \Xi'\}$, *then* $\Xi' = \Xi$.

4.4 Discussion

As part of the reimplementation of the indexing phase, the referencing environment was first designed using immutable data structures. Specifically, hash array mapped tries (HAMTs) were leveraged for the dictionaries mapping identifiers to stacks of bindings. The indexing functions were then implemented using the state monad so that derived states could be passed on to later operations. Having copies of the referencing environment for free during indexing would have facilitated incremental development because they could be attached to proof declarations to allow state rollbacks like in Isabelle/Isar with its context graphs. In turn, checking out an incomplete theorem in Harpoon would simply result in the referencing environment's copy being used for elaborating commands. Unfortunately, the increased

number of memory allocations, the overhead of insertions and removals in HAMTs and the lack of contiguity in the memory layout for the referencing environment incurred a decrease in runtime performance by a factor of roughly 10 for Beluga's test suite. This meant that running the indexing phase on a signature would take more time than type-checking it. Thankfully, the indexing phase's implementation was loosely coupled with the representation of the referencing environment, and the state monad was used throughout. This made it easy to substitute a mutable data structure for the referencing environment and recover runtime performance comparable to that of the legacy implementation of Beluga. Consequently, checking out incomplete proofs in Harpoon now involves reconstructing the referencing environment using the internal AST's representation of the signature. Because of the language's design, only the identifiers for signature-level and module-level declarations have to be visited, which circumvents having to rerun name resolution over the entire signature.

From a programming language implementation standpoint, the revised indexing phase simplifies the system's flow of information by decoupling it from the signature reconstruction store. This opens up multiple opportunities, including proper unit-testing of the referencing environment structure and indexing functions. Additionally, given Beluga's design for signature-level declaration of constants, the indexing phase can be parallelized. This is because the body of a signature-level declaration is guaranteed not to export non-constant identifiers. This means that the referencing environment right before a signature-level can be constructed while disregarding most of the AST. As such, the declarations in a signature can first be traversed to preallocate symbolic identifiers for toplevel constants and store them in a queue. Concurrent threads can then be assigned non-overlapping ranges of declarations to process, and the initial referencing environment for each can be constructed using only that queue. Should this isolation for signature-level declarations be extended to the later phases of processing, concurrently type-checking programs could improve Beluga's performance in large mechanizations.

Chapter 5

Discussion and Conclusion

The objectives of this thesis were to improve the design and implementation of Beluga and Harpoon in order fix soundness issues with the incremental development of proofs in interactive REPL sessions. This required revising the first few phases of syntactic and semantic analysis for the language in order to make them modular and reusable with respect to different instances of processing states.

The grammar for Beluga and Harpoon was fully formalized in EBNF, along with a specification of syntactic ambiguities and how they are resolved using precedence levels and grammar blurring. The parser was improved with the introduction of a new context-sensitive disambiguation phase. This prevented the propagation of ambiguous AST nodes into the more complex phases of Beluga's processing pipeline. By the same token, the language's grammar was modified to support non-normal terms, which in turn allowed for syntax error messages to be improved. The user-defined notation feature, previously limited to terms in the index language, is now fully supported at the computation-level as well. These changes make the implementation robust, and the language more expressive and cohesive, which in turn improves the user experience.

The indexing phase was fully reimplemented using a new structure of the referencing

environment at all AST nodes in Beluga programs and Harpoon proof scripts. This structure simplified the name resolution algorithm and allowed for the unification of constant declarations and indexing contexts to support simple lexical scoping rules. In tandem, the existing implementation of namespaces for grouping signature-level declarations was rectified. De Bruijn indices computations were implemented, and indexing for the LF part of Beluga was formalized to prove that the stack structure of frames can be safely leveraged to implement a checkpoint mechanism for the bindings in scope. During Harpoon structural editing sessions on proof scripts, the soundness of navigating between holes with respect to name resolution was rectified by ensuring that disambiguation and indexing are explicitly dependent on the referencing environment's state.

The following sections conclude this work with an evaluation of the implemented changes to Beluga and Harpoon, a discussion of future work to rectify long-standing issues in the later phases of semantic analysis, and finally closing remarks on the design and implementation of the system.

5.1 Evaluation

The reimplementation of Beluga and Harpoon's parsing and indexing phases significantly impacted the system. To evaluate these changes, we focus here on the non-functional requirements of runtime performance and maintainability, and then on the functional requirement of soundness.

Runtime Performance

Runtime performance is one of the main concerns for Beluga and Harpoon. Indeed, because dependently-typed programming languages are more difficult to work with than typical languages, assisting the user with data generated during syntactic and semantic

analyses is instrumental to program development. For example, one of Beluga's most used feature is that of displaying type information about holes in programs. Users tend to run type-checking at every step of program development to constantly refresh that information. Consequently, in the implementation of Beluga, a greater emphasis was placed on early error detection and reporting to minimize the latency between requests to the type-checker and the display of useful information.

Signature reconstruction has been sufficiently fast for most of the mechanizations realised in Beluga `v1.0`. Consequently, the new parsing and indexing phases introduced in `v1.1` had to not degrade the runtime performance of the system. This objective was surpassed: on a set of 429 test Beluga signatures totalling approximately 35 000 lines of code, signature reconstruction ran on average in 8.412 ± 0.133 s in `v1.0` and 8.045 ± 0.126 s in `v1.1`, which means there was a 4 % runtime improvement with the revised parsing and indexing phases. These tests were already part of the total 487 example and integration test cases in Beluga `v1.0`'s testing suite. The other 58 tests were filtered out of this benchmark to mitigate the impact of breaking changes to the syntax. The system was compiled using the same version of its dependencies, and the reported results are the average of 15 runs, with 3 warmup runs.

With an identical methodology, a similar test was carried out on a larger mechanization: with an average signature reconstruction runtime of 5.504 ± 0.024 s in `v1.0` and 4.716 ± 0.016 s in `v1.1`, there was a 14 % runtime improvement. This mechanization is mostly comprised of verbose Harpoon proof scripts tallying over 90 000 lines of code, which provides a better estimate for the time required to completely parse, reconstruct and type-check larger projects. No modifications had to be performed on that mechanization since it did not features that are part of the breaking changes in `v1.1`. During the initialization of Harpoon, the referencing environments at each proof hole were constructed using a separate traversal of the reconstructed Beluga signature. This means the soundness fixes had virtually no runtime performance impact on Harpoon's proof navigation commands.

Although there were changes to Beluga between `v1.0` and `v1.1` outside of the parsing, disambiguation and indexing phases, the performance improvement is best explained by the simplification of the grammar for parsing, and the usage of mutable dictionaries instead of association lists for name resolution. Additionally, having a uniform referencing environment structure reduced the number of lookup misses that would occur when sequentially looking up declaration tables.

Maintainability

The Beluga system has accrued significant technical debt throughout its development cycles. This is partly explained by turnover-induced knowledge loss and lack of unit tests. Initiatives were put in place to tackle these issues in `v1.1` during the revision of the parser.

To avoid turnover-induced knowledge loss, the specification for Beluga's and Harpoon's grammar from appendix A has been added to the code repository. Additionally, the grammar manipulations and disambiguation mechanisms required for parsing is documented more thoroughly in the implementation. This ensures that those steps can be repeated in the future, should the grammar be amended to support new features.

As outlined in section 2.2, the parser implemented in `v1.0` returned ASTs containing ambiguous nodes, and would postpone the handling of user-defined operators to the indexing phase. This discouraged the implementation of unit tests for the parser since its outputs were incomplete and more likely to change. In contrast, the parser revised in `v1.1` returns ASTs in an unambiguous external syntax representation. Because of this, 345 unit tests were added to verify the parser in isolation from the rest of the system. These tests check success and failure scenarios for parsing small inputs for the important syntactic categories of the language and with respect to mock disambiguation states. A unit-testing approach this precise could not have been reliably implemented in `v1.0`. Additionally, partial round-trip testing of the parser was added for existing Beluga examples and case studies using a

pretty-printer for signatures in external syntax representation. Complete round-trip testing is not supported since the parser discards inline comments during lexical analysis.

Soundness

Throughout the development of Beluga, a greater emphasis was placed on ensuring the soundness of it's type theory and of it's intricate semantic checks for theorem-proving. Since this work is experimental, there are unresolved soundness issues in those later semantic checks. To facilitate addressing them, the phases preceding those semantic checks also had to be rectified.

Name resolution in interactive Harpoon proof sessions was unsound in `v1.0` as explained in section 4.1. Indeed, the referencing environment was always stuck in its final state at the very end of the Beluga signature. This meant that out-of-order proof sessions would unsoundly have signature-level declarations brought forward in scope despite appearing at later points in the signature. The changes introduced in `v1.1` addressed three soundness issues: fixing the handling of namespaces, allowing the shadowing of signature-level constants, and most importantly fixing the state of referencing environments when navigating to any hole in Harpoon proof scripts. This corrected 4 test cases and generally improved the usability of both Beluga and Harpoon. Top-down proof development in Harpoon is now a viable approach for larger mechanizations.

These changes did come with trade-offs, specifically dropping the support for type-driven disambiguation of meta-level objects and for identifier overloading. However, these additional restrictions improve the readability of Beluga signatures since they are now simpler to disambiguate, and because identifiers refer to one declaration at a time.

5.2 Future Work

With regards to the implementation of Beluga and Harpoon, properly defining referencing environments and implementing indexing with respect to them is a step in the right direction. Indeed, this ensures indexing procedures are reusable and sound when visiting holes in Harpoon proofs in an out-of-order fashion with respect to the Beluga signature. This also enables indexing to be unit-tested independently of the processing pipeline, which offers greatly visibility into the inner workings of the system, as well as provides more fine-grained verifiers of the implementation's correctness.

There are nonetheless many implementation challenges left ahead in order to fully support structural editing of Beluga programs and Harpoon proofs. Indeed, there are areas of the system that still rely on the assumption that signatures are always processed in order of declaration, and hence that the signature reconstruction store only contains data about visible declarations. The future areas of work on the implementation of Beluga and Harpoon include the following:

1. Information flow analysis is required in the later phases of semantic analysis, namely type and term reconstruction, type-checking and unification, to ascertain whether their stateful operations are always handled soundly. Specifically, while there is a trailing mechanism for higher-order unification to keep track of meta-variable instantiations (the assignment of a contextual object to a meta-variable), there are routines during LF reconstruction that ignore this bookkeeping. This can result in unsound programs when users undo edit actions during interactive proof developments. Fixing this issue will require significant refactoring to decouple those phases from global data structures such as the signature reconstruction store.

2. The logic proof search engine which powers Harpoon's automation tactics uses the signature reconstruction store to have a global view of the user-defined constants and

programs that can be used to solve subgoals. As such, synthesized solutions to subgoals may reference declarations that are out of scope. That notwithstanding, taking all constants into account during proof search may result in degraded performance and a non-responsive REPL when using automation tactics on larger projects. Enforcing the constraint that synthesized proofs must be sound with respect to name resolution at the holes they have to be spliced in may prune the search tree.

3. Fresh name generation for the nameless representation of Beluga programs is currently unsound in the implementation. Indeed, that procedure does not wholly take into account the identifiers in scope as it restricts its focus on the declarations in indexing contexts. This has consequences with program synthesis, both for the conversion of Harpoon proof scripts to Beluga programs and for error reporting. Much like was the case with the now resolved soundness issue of navigating to holes in Harpoon proofs, spliced Beluga programs corresponding to Harpoon proof scripts may refer to inadvertedly shadowed constants. That is, a generated variable name, which is assumed to be fresh, may actually clash with a required constant name. To address this, one needs to synthesize programs in a nameless representation and then compute readable and easily distinguishable names for variables, which effectively amounts to reversing indexing. Dynamic programming over the tree structure of the AST would be required to compute sets of referenced identifiers and de Bruijn indices to select appropriate names at the binding sites.

5.3 Final Remarks

In closing, the key lessons learned from this experience with regards to programming language design and implementation are as follows:

1. If a programming language is context-sensitive, separate its syntactic analysis into

a context-free and a context-sensitive phase, if possible. This ensures good runtime performance for the parser, and facilitates the integration of the language with tools that primarily support context-free languages.

2. Keep the syntax accepted during context-free parsing simple, and rely on subsequent processing phases to disambiguate complex syntax overloading and enforce additional syntactic restrictions. This enables error messages to be augmented with some semantic analysis to suggest corrections.

3. Keep disambiguation and name resolution mechanisms simple and intuitive. This ensures that programs written by users will be readable, at least at the syntactic level.

4. Ensure that the programming language's grammar supports desugared forms for all syntactic sugars, as well as syntax for explicitly supplying what are otherwise implicit terms. This facilitates the traceability of programs from their parsed representation to their elaborated representation, it allows users to opt out of using syntactic sugars, and it enables users to observe and verify the results of term and type reconstruction.

5. Avoid the common programming pitfall of relying on global mutable data to capture the implemented system's state. There will always be execution scenarios incompatible with that design, in particular incremental development, unit testing and concurrency.

Appendix A

Beluga Grammar

This appendix describes the CFG for Beluga and Harpoon in their entirety as a specification for the parsing algorithm. The lexical conventions are defined first, then ambiguous CFGs are presented for the various kinds of syntactic structures, and finally blurred CFGs are shown to illustrate how context-free parsing is done before disambiguation.

A.1 Syntax

This section presents the syntax for Beluga signatures using ambiguous CFGs. It is intended to serve as a specification for validating the parser's implementation, as well as to assist in making changes to the parser. Rewriting of the grammar for optimization, left recursion elimination, and disambiguation purposes is outside the scope of this presentation. Section A.2 presents a CFG for Beluga that is suitable for parsing with a separate stage for disambiguation.

The EBNF metasyntax used in the sections hereafter follows this notation:

- $(a)*$ represents the repeated application of a zero or more times.

- $(a)+$ represents the repeated application of a one or more times.
- $[a]$ represents optionally applying a.
- $a - b$ represents the application of a that cannot be the application of b if a and b are single-character length productions.

A.1.1 Comments

Beluga supports three kinds of comments:

- Line comments start with `%` and end with a newline `\n`. They are treated as blank characters, and as such may be interspersed anywhere in the grammar.
- Block comments are enclosed in `%{` and `}%`. They are treated as blank characters, and as such may be interspersed anywhere in the grammar.
- Documentation comments are enclosed in `%{{` and `}}%`. The contents of documentation comments are parsed as `Markdown` following the CommonMark standard version 0.29. These are used in documentation generation to `HTML` or LaTeX. Documentation comments may only appear before or after signature-level declarations.

A.1.2 Keywords

The following identifiers are reserved as keywords in Beluga, and cannot be used otherwise.

`LF`	`fn`	`else`	`world`
`and`	`fun`	`of`	`module`
`impossible`	`mlam`	`schema`	`struct`
`case`	`if`	`some`	`end`
`rec`	`then`	`block`	`trust`

`total`	`inductive`	`proof`	`toshow`
`prop`	`coinductive`	`by`	
`type`	`stratified`	`as`	
`ctype`	`typedef`	`suffices`	

Additionally, the following characters are reserved and cannot be used anywhere in identifiers.

`.`	`%`	`[ ]`
`. . .`	`|`	`{ }`
`,`	`"`	`< >`
`:`	`\`	⊢
`;`	`( )`	

A.1.3 Lexical Conventions

Beluga programs must be encoded in UTF-8. The nonterminal ⟨*white-space*⟩ corresponds to the Unicode property `White_Space` as described in the Unicode Standard Version 15.0.0 Annex #44, *Unicode Character Database*. The nonterminal ⟨*any*⟩ stands for any unicode character.

⟨*forward-arrow*⟩ ::= '`->`' | '→'

⟨*backward-arrow*⟩ ::= '`<-`' | '←'

⟨*thick-forward-arrow*⟩ ::= '`=>`' | '⇒'

⟨*turnstile*⟩ ::= '`|-`' | '⊢'

⟨*turnstile-hash*⟩ ::= ⟨*turnstile*⟩'`#`'

⟨*dots*⟩ ::= '`..`' | '`. . .`'

⟨*non-zero-digit*⟩ ::= ‘1’..‘9’

⟨*digit*⟩ ::= ‘0’ | ⟨*non-zero-digit*⟩

⟨*integer*⟩ ::= ‘0’ | ⟨*non-zero-digit*⟩ ⟨*digit*⟩*

⟨*ascii-control*⟩ ::= ‘\000’ .. ‘\031’ | ‘\127’

⟨*reserved-character*⟩ ::= ‘.’ | ‘,’ | ‘:’ | ‘;’ | ‘%’ | ‘|’ | ‘”’ | ‘\’ | ‘(’ | ‘)’ | ‘[’ | ‘]’
| ‘{’ | ‘}’ | ‘<’ | ‘>’ | ‘⊢’

⟨*identifier-continue*⟩ ::= ⟨*any*⟩ - (⟨*ascii-control*⟩ | ⟨*white-space*⟩ | ⟨*reserved-character*⟩)

⟨*identifier-start*⟩ ::= ⟨*identifier-continue*⟩ - (⟨*digit*⟩ | ‘$’ | ‘#’)

⟨*identifier*⟩ ::= ⟨*identifier-start*⟩ ⟨*identifier-continue*⟩*

⟨*dot-integer*⟩ ::= ‘.’⟨*integer*⟩

⟨*dot-identifier*⟩ ::= ‘.’⟨*identifier*⟩

⟨*hash-identifier*⟩ ::= ‘#’⟨*identifier*⟩

⟨*dollar-identifier*⟩ ::= ‘$’⟨*identifier*⟩

A.1.4 Utilities

⟨*omittable-identifier*⟩ ::= ‘_’ | ⟨*identifier*⟩

⟨*omittable-hash-identifier*⟩ ::= ‘#_’ | ⟨*hash-identifier*⟩

⟨*omittable-dollar-identifier*⟩ ::= ‘$_’ | ⟨*dollar-identifier*⟩

⟨*qualified-identifier*⟩ ::= ⟨*identifier*⟩ ⟨*dot-identifier*⟩*

⟨*dot-qualified-identifier*⟩ ::= ⟨*dot-identifier*⟩+

⟨*meta-object-identifier*⟩ ::= ⟨*identifier*⟩

| ⟨*hash-identifier*⟩

| ⟨*dollar-identifier*⟩

⟨*omittable-meta-object-identifier*⟩ ::= ⟨*omittable-identifier*⟩

| ⟨*omittable-hash-identifier*⟩

| ⟨*omittable-dollar-identifier*⟩

⟨*precedence*⟩ ::= ⟨*integer*⟩

⟨*associativity*⟩ ::= '`left`' | '`right`' | '`none`'

A.1.5 Grammar for LF

The following is the grammar for the definition of LF kinds, types and terms. These syntactic constructs form a weak representational function space without case analysis or recursion.

⟨*lf-kind*⟩ ::= '`{`' ⟨*omittable-identifier*⟩ '`:`' ⟨*lf-type*⟩ '`}`' ⟨*lf-kind*⟩

| ⟨*lf-type*⟩ ⟨*forward-arrow*⟩ ⟨*lf-kind*⟩

| '`type`'

| '`(`' ⟨*lf-kind*⟩ '`)`'

⟨*lf-type*⟩ ::= ⟨*qualified-identifier*⟩

| '`{`' ⟨*omittable-identifier*⟩ '`:`' ⟨*lf-type*⟩ '`}`' ⟨*lf-type*⟩

| ⟨*lf-type*⟩ ⟨*forward-arrow*⟩ ⟨*lf-type*⟩

| ⟨*lf-type*⟩ ⟨*backward-arrow*⟩ ⟨*lf-type*⟩

| ⟨*qualified-identifier*⟩ ⟨*lf-term*⟩

| ⟨*lf-term*⟩ ⟨*qualified-identifier*⟩ ⟨*lf-term*⟩

| ⟨*lf-term*⟩ ⟨*qualified-identifier*⟩

| '`(`' ⟨*lf-type*⟩ '`)`'

⟨*lf-term*⟩ ::= ⟨*identifier*⟩

| ⟨*qualified-identifier*⟩

| '\' '(' ⟨*omittable-identifier*⟩ ':' ⟨*lf-type*⟩ ')' '.' ⟨*lf-term*⟩

| '\' ⟨*omittable-identifier*⟩ '.' ⟨*lf-term*⟩

| ⟨*lf-term*⟩ ⟨*lf-term*⟩+

| ⟨*qualified-identifier*⟩ ⟨*lf-term*⟩

| ⟨*lf-term*⟩ ⟨*qualified-identifier*⟩ ⟨*lf-term*⟩

| ⟨*lf-term*⟩ ⟨*qualified-identifier*⟩

| '_'

| ⟨*lf-term*⟩ ':' ⟨*lf-type*⟩

| '(' ⟨*lf-term*⟩ ')'

A.1.6 Grammar for Contextual LF

The following is the grammar for the extension of LF with substitutions, contexts and pattern-matching.

⟨*clf-type*⟩ ::= ⟨*qualified-identifier*⟩

| '{' ⟨*omittable-identifier*⟩ ':' ⟨*clf-type*⟩ '}' ⟨*clf-type*⟩

| ⟨*clf-type*⟩ ⟨*forward-arrow*⟩ ⟨*clf-type*⟩

| ⟨*clf-type*⟩ ⟨*backward-arrow*⟩ ⟨*clf-type*⟩

| 'block' '(' ⟨*identifier*⟩ ':' ⟨*clf-type*⟩ (',' ⟨*identifier*⟩ ':' ⟨*clf-type*⟩)+ ')'

| 'block' ⟨*identifier*⟩ ':' ⟨*clf-type*⟩ (',' ⟨*identifier*⟩ ':' ⟨*clf-type*⟩)+

| 'block' '(' ⟨*clf-type*⟩ ')'

| 'block' ⟨*clf-type*⟩

| ⟨*qualified-identifier*⟩ ⟨*clf-term*⟩

| ⟨*clf-term*⟩ ⟨*qualified-identifier*⟩ ⟨*clf-term*⟩

| ⟨*clf-term*⟩ ⟨*qualified-identifier*⟩

| ‘(’ ⟨*clf-type*⟩ ‘)’

⟨*clf-term*⟩ ::= ⟨*identifier*⟩

| ⟨*hash-identifier*⟩

| ⟨*qualified-identifier*⟩

| ⟨*clf-term*⟩ ⟨*clf-substitution*⟩

| ‘\’ ‘(’ ⟨*omittable-identifier*⟩ ‘:’ ⟨*clf-type*⟩ ‘)’ ‘.’ ⟨*clf-term*⟩

| ‘\’ ⟨*omittable-identifier*⟩ ‘.’ ⟨*clf-term*⟩

| ⟨*clf-term*⟩ ⟨*clf-term*⟩+

| ⟨*qualified-identifier*⟩ ⟨*clf-term*⟩

| ⟨*clf-term*⟩ ⟨*qualified-identifier*⟩ ⟨*clf-term*⟩

| ⟨*clf-term*⟩ ⟨*qualified-identifier*⟩

| ‘_’

| ‘?’[⟨*identifier*⟩]

| ‘<’ ⟨*clf-term*⟩ (‘;’ ⟨*clf-term*⟩)* ‘>’

| ⟨*clf-term*⟩ ⟨*dot-integer*⟩

| ⟨*clf-term*⟩ ⟨*dot-identifier*⟩

| ⟨*clf-term*⟩ ‘:’ ⟨*clf-type*⟩

| ‘(’ ⟨*clf-term*⟩ ‘)’

⟨*clf-pattern*⟩ ::= ⟨*identifier*⟩

| ⟨*hash-identifier*⟩

| ⟨*qualified-identifier*⟩

| ‘_’

| ‘<’ ⟨*clf-pattern*⟩ (‘;’ ⟨*clf-pattern*⟩)* ‘>’

| ⟨*clf-pattern*⟩ ⟨*dot-integer*⟩

| ⟨*clf-pattern*⟩ ⟨*dot-identifier*⟩

| ‘\’ ‘(’ ⟨*omittable-identifier*⟩ ‘:’ ⟨*clf-type*⟩ ‘)’ ‘.’ ⟨*clf-pattern*⟩

| ‘\’ ⟨*omittable-identifier*⟩ ‘.’ ⟨*clf-pattern*⟩

| ⟨*clf-pattern*⟩ ‘[’ ⟨*clf-substitution*⟩ ‘]’

| ⟨*clf-pattern*⟩ ⟨*clf-pattern*⟩+

| ⟨*qualified-identifier*⟩ ⟨*clf-pattern*⟩

| ⟨*clf-pattern*⟩ ⟨*qualified-identifier*⟩ ⟨*clf-pattern*⟩

| ⟨*clf-pattern*⟩ ⟨*qualified-identifier*⟩

| ⟨*clf-pattern*⟩ ‘:’ ⟨*clf-type*⟩

| ‘(’ ⟨*clf-pattern*⟩ ‘)’

⟨*clf-substitution*⟩ ::= [‘^’]

| ⟨*dollar-identifier*⟩ (‘,’ ⟨*clf-term*⟩)*

| ⟨*dollar-identifier*⟩ ‘[’ ⟨*clf-substitution*⟩ ‘]’ (‘,’ ⟨*clf-term*⟩)*

| ‘..’ (‘,’ ⟨*clf-term*⟩)*

| ⟨*clf-term*⟩ (‘,’ ⟨*clf-term*⟩)*

⟨*clf-substitution-pattern*⟩ ::= [‘^’]

| ⟨*dollar-identifier*⟩ (‘,’ ⟨*clf-pattern*⟩)*

| ⟨*dollar-identifier*⟩ ‘[’ ⟨*clf-substitution*⟩ ‘]’(‘,’ ⟨*clf-pattern*⟩)*

| ‘..’ (‘,’ ⟨*clf-pattern*⟩)*

| ⟨*clf-pattern*⟩ (‘,’ ⟨*clf-pattern*⟩)*

⟨*clf-context*⟩ ::= [‘^’]

| ⟨*omittable-identifier*⟩ (‘,’ ⟨*identifier*⟩ [‘:’ ⟨*clf-type*⟩])*

| ⟨*identifier*⟩ [‘:’ ⟨*clf-type*⟩] (‘,’ ⟨*identifier*⟩ [‘:’ ⟨*clf-type*⟩])*

⟨*clf-context-pattern*⟩ ::= [‘^’]

| ⟨*omittable-identifier*⟩ (‘,’ ⟨*identifier*⟩ ‘:’ ⟨*clf-type*⟩)*

| ⟨*identifier*⟩ ‘:’ ⟨*clf-type*⟩ (‘,’ ⟨*identifier*⟩ ‘:’ ⟨*clf-type*⟩)*

A.1.7 Grammar for the Meta-Level

The following is the grammar for meta-level types (meta-types) and objects (meta-objects). Meta-types classify meta-objects. Meta-objects are contextual LF terms, substitutions and contexts.

Meta-types and meta-objects may be embedded in the computation level by denoting them with boxes. Modifying the box annotation by prefixing it with # or $ syntactically forces it to be a specific variant of meta-type or meta-object. Substitution types and objects are required to be annotated with a $-box to disambiguate them from contextual types and terms respectively. When paired with an identifier in a declaration such as in a meta-context, the prefixes of the identifier and the boxed meta-type must match. Exceptionally, schema meta-types are not boxed.

⟨*schema*⟩ ::= ⟨*qualified-identifier*⟩

| ⟨*schema*⟩ '`+`' ⟨*schema*⟩

| ['`some`' '`[`' ⟨*identifier*⟩ '`:`' ⟨*lf-type*⟩ ('`,`' ⟨*identifier*⟩ '`:`' ⟨*lf-type*⟩)* '`]`'] '`block`' '`(`' ⟨*identifier*⟩ '`:`' ⟨*lf-type*⟩ ('`,`' ⟨*identifier*⟩ '`:`' ⟨*lf-type*⟩)* '`)`'

| ['`some`' '`[`' ⟨*identifier*⟩ '`:`' ⟨*lf-type*⟩ ('`,`' ⟨*identifier*⟩ '`:`' ⟨*lf-type*⟩)* '`]`'] '`block`' ⟨*identifier*⟩ '`:`' ⟨*lf-type*⟩ ('`,`' ⟨*identifier*⟩ '`:`' ⟨*lf-type*⟩)

| ['`some`' '`[`' ⟨*identifier*⟩ '`:`' ⟨*lf-type*⟩ ('`,`' ⟨*identifier*⟩ '`:`' ⟨*lf-type*⟩)* '`]`'] '`block`' ⟨*lf-type*⟩

| ['`some`' '`[`' ⟨*identifier*⟩ '`:`' ⟨*lf-type*⟩ ('`,`' ⟨*identifier*⟩ '`:`' ⟨*lf-type*⟩)* '`]`'] '`block`' ⟨*lf-type*⟩

⟨*meta-type*⟩ ::= ⟨*qualified-identifier*⟩

| '`(`' ⟨*clf-context*⟩ ⟨*turnstile*⟩ ⟨*clf-type*⟩ '`)`'

| '`#(`' ⟨*clf-context*⟩ ⟨*turnstile*⟩ ⟨*clf-type*⟩ '`)`'

| '`$(`' ⟨*clf-context*⟩ ⟨*turnstile*⟩ ⟨*clf-context*⟩ '`)`'

| '`$(`' ⟨*clf-context*⟩ ⟨*turnstile-hash*⟩ ⟨*clf-context*⟩ '`)`'

| '`[`' ⟨*clf-context*⟩ ⟨*turnstile*⟩ ⟨*clf-type*⟩ '`]`'

| '`#[`' ⟨*clf-context*⟩ ⟨*turnstile*⟩ ⟨*clf-type*⟩ '`]`'

| ‘$[’ ⟨*clf-context*⟩ ⟨*turnstile*⟩ ⟨*clf-context*⟩ ‘]’

| ‘$[’ ⟨*clf-context*⟩ ⟨*turnstile-hash*⟩ ⟨*clf-context*⟩ ‘]’

⟨*meta-object*⟩ ::= ‘[’ ⟨*clf-context*⟩ ‘]’

| ‘[’ ⟨*clf-context*⟩ ⟨*turnstile*⟩ ⟨*clf-term*⟩ ‘]’

| ‘$[’ ⟨*clf-context*⟩ ⟨*turnstile*⟩ ⟨*clf-substitution*⟩ ‘]’

| ‘$[’ ⟨*clf-context*⟩ ⟨*turnstile-hash*⟩ ⟨*clf-substitution*⟩ ‘]’

⟨*meta-pattern*⟩ ::= ‘[’ ⟨*clf-context-pattern*⟩ ‘]’

| ‘[’ ⟨*clf-context-pattern*⟩ ⟨*turnstile*⟩ ⟨*clf-pattern*⟩ ‘]’

| ‘$[’ ⟨*clf-context-pattern*⟩ ⟨*turnstile*⟩ ⟨*clf-substitution-pattern*⟩ ‘]’

| ‘$[’ ⟨*clf-context-pattern*⟩ ⟨*turnstile-hash*⟩ ⟨*clf-substitution-pattern*⟩ ‘]’

⟨*meta-context*⟩ ::= [‘^’]

| ⟨*meta-object-identifier*⟩ [‘:’ ⟨*meta-type*⟩]
(‘,’ ⟨*meta-object-identifier*⟩ [‘:’ ⟨*meta-type*⟩])*

A.1.8 Grammar for Computations

The following is the grammar for the computational-level types, expressions and patterns. Atomic patterns are computation-level patterns that may appear as whitespace-delimited lists in pattern-matching functions.

⟨*comp-kind*⟩ ::= ‘{’ ⟨*omittable-meta-object-identifier*⟩ : ⟨*meta-type*⟩ ‘}’ ⟨*comp-kind*⟩

| ‘(’ ⟨*omittable-meta-object-identifier*⟩ : ⟨*meta-type*⟩ ‘)’ ⟨*comp-kind*⟩

| ⟨*meta-type*⟩ ⟨*forward-arrow*⟩ ⟨*comp-kind*⟩

| ‘ctype’

| ‘(’ ⟨*comp-kind*⟩ ‘)’

⟨*comp-type*⟩ ::= ‘{’ ⟨*omittable-meta-object-identifier*⟩ ‘:’ ⟨*meta-type*⟩ ‘}’ ⟨*comp-type*⟩

| ‘(’ ⟨*omittable-meta-object-identifier*⟩ ‘:’ ⟨*meta-type*⟩ ‘)’ ⟨*comp-type*⟩

| ⟨*comp-type*⟩ ⟨*forward-arrow*⟩ ⟨*comp-type*⟩

| ⟨*comp-type*⟩ ⟨*backward-arrow*⟩ ⟨*comp-type*⟩

| ⟨*comp-type*⟩ (‘`*`’ ⟨*comp-type*⟩)+

| ⟨*meta-type*⟩

| ⟨*qualified-identifier*⟩ ⟨*meta-object*⟩*

| ⟨*qualified-identifier*⟩ ⟨*meta-object*⟩

| ⟨*meta-object*⟩ ⟨*qualified-identifier*⟩ ⟨*meta-object*⟩

| ⟨*meta-object*⟩ ⟨*qualified-identifier*⟩

| ⟨*qualified-identifier*⟩ ⟨*meta-object*⟩*

| ⟨*qualified-identifier*⟩ ⟨*meta-object*⟩

| ⟨*meta-object*⟩ ⟨*qualified-identifier*⟩ ⟨*meta-object*⟩

| ⟨*meta-object*⟩ ⟨*qualified-identifier*⟩

| ‘`(`’ ⟨*comp-type*⟩ ‘`)`’

⟨*comp-expression*⟩ ::= ⟨*identifier*⟩

| ⟨*qualified-identifier*⟩

| ⟨*qualified-identifier*⟩

| ‘`fn`’ ⟨*omittable-identifier*⟩+ ⟨*thick-forward-arrow*⟩ ⟨*comp-expression*⟩

| ‘`fun`’ [‘`|`’] ⟨*comp-cofunction-branch*⟩ (‘`|`’ ⟨*comp-cofunction-branch*⟩)*

| ‘`mlam`’ ⟨*omittable-meta-object-identifier*⟩+
⟨*thick-forward-arrow*⟩ ⟨*comp-expression*⟩

| ‘`let`’ [⟨*comp-pattern-meta-context*⟩] ⟨*comp-pattern*⟩
‘`=`’ ⟨*comp-expression*⟩ ‘`in`’ ⟨*comp-expression*⟩

| ⟨*meta-object*⟩

| ‘`impossible`’ ⟨*comp-expression*⟩

| ‘`case`’ ⟨*comp-expression*⟩ [‘`--not`’] ‘`of`’ [‘`|`’]
⟨*comp-case-branch*⟩ (‘`|`’ ⟨*comp-case-branch*⟩)*

| ‘(’ ⟨*comp-expression*⟩ (‘,’ ⟨*comp-expression*⟩)+ ‘)’

| ‘?’ [⟨*identifier*⟩]

| ‘_’

| ⟨*comp-expression*⟩ ⟨*comp-expression*⟩+

| ⟨*qualified-identifier*⟩ ⟨*comp-expression*⟩

| ⟨*comp-expression*⟩ ⟨*qualified-identifier*⟩ ⟨*comp-expression*⟩

| ⟨*comp-expression*⟩ ⟨*qualified-identifier*⟩

| ⟨*comp-expression*⟩ ⟨*dot-qualified-identifier*⟩

| ⟨*comp-expression*⟩ ‘:’ ⟨*comp-type*⟩

| ‘(’ ⟨*comp-expression*⟩ ‘)’

⟨*comp-case-branch*⟩ ::= [⟨*comp-pattern-meta-context*⟩] ⟨*comp-pattern*⟩
⟨*thick-forward-arrow*⟩ ⟨*comp-expression*⟩

⟨*comp-cofunction-branch*⟩ ::= ([⟨*comp-pattern-meta-context*⟩] ⟨*comp-copattern*⟩)+
⟨*thick-forward-arrow*⟩ ⟨*comp-expression*⟩

⟨*comp-pattern-meta-context*⟩ ::= (‘{’ ⟨*meta-object-identifier*⟩ [‘:’ ⟨*meta-type*⟩] ‘}’)+

⟨*comp-pattern*⟩ ::= ⟨*identifier*⟩

| ⟨*qualified-identifier*⟩

| ⟨*meta-pattern*⟩

| ‘(’ ⟨*comp-pattern*⟩ (‘,’ ⟨*comp-pattern*⟩)+ ‘)’

| ⟨*comp-pattern*⟩ ⟨*comp-pattern*⟩+

| ⟨*qualified-identifier*⟩ ⟨*comp-pattern*⟩

| ⟨*comp-pattern*⟩ ⟨*qualified-identifier*⟩ ⟨*comp-pattern*⟩

| ⟨*comp-pattern*⟩ ⟨*qualified-identifier*⟩

| ⟨*comp-pattern*⟩ ‘:’ ⟨*comp-type*⟩

| ‘_’

| ‘(’ ⟨*comp-pattern*⟩ ‘)’

⟨*comp-pattern-atomic*⟩ ::= ⟨*identifier*⟩

| ⟨*qualified-identifier*⟩

| ⟨*meta-pattern*⟩

| '`(`' ⟨*comp-pattern*⟩ ('`,`' ⟨*comp-pattern*⟩)+ '`)`'

| '`_`'

| '`(`' ⟨*comp-pattern*⟩ '`)`'

⟨*comp-copattern*⟩ ::= ⟨*dot-qualified-identifier*⟩ ⟨*comp-pattern-atomic*⟩*

| ⟨*comp-pattern-atomic*⟩

⟨*comp-context*⟩ ::= ['`^`']

| ⟨*identifier*⟩ ['`:`' ⟨*comp-type*⟩] ('`,`' ⟨*identifier*⟩ ['`:`' ⟨*comp-type*⟩])*

A.1.9 Grammar for Harpoon's REPL

The following is the grammar for the commands used in the Harpoon REPL.

⟨*harpoon-command*⟩ ::= '`intros`' ⟨*identifier*⟩*

| '`split`' ⟨*comp-expression*⟩

| '`invert`' ⟨*comp-expression*⟩

| '`impossible`' ⟨*comp-expression*⟩

| '`msplit`' ⟨*identifier*⟩

| '`solve`' ⟨*comp-expression*⟩

| '`by`' ⟨*comp-expression*⟩ '`as`' ⟨*identifier*⟩ [⟨*boxity*⟩]

| '`type`' ⟨*comp-expression*⟩

| '`suffices`' '`by`' ⟨*comp-expression*⟩
  '`toshow`' [('`_`' | ⟨*comp-type*⟩) ('`,`' ('`_`' | ⟨*comp-type*⟩))*]

| '`unbox`' ⟨*comp-expression*⟩ '`as`' ⟨*identifier*⟩

| '`strengthen`' ⟨*comp-expression*⟩ '`as`' ⟨*identifier*⟩

| ‘toggle-automation’ ⟨*automation-kind*⟩ [⟨*automation-change*⟩]

| ‘rename’ (‘comp’ | ‘meta’) ⟨*identifier*⟩ ⟨*identifier*⟩

| ‘defer’

| ‘select’ ⟨*qualified-identifier*⟩

| ‘info’ ‘theorem’ ⟨*qualified-identifier*⟩

| ‘theorem’ ⟨*theorem-subcommand*⟩

| ‘session’ ⟨*session-subcommand*⟩

| ‘subgoal’ ⟨*subgoal-subcommand*⟩

| ‘undo’

| ‘redo’

| ‘history’

| ‘help’

| ‘save’

⟨*boxity*⟩ ::= ‘boxed’ | ‘unboxed’ | ‘strenghtened’

⟨*automation-kind*⟩ ::= ‘auto-intros’ | ‘auto-solve-trivial’

⟨*automation-change*⟩ ::= ‘on’ | ‘off’ | ‘toggle’

⟨*theorem-subcommand*⟩ ::= ‘list’ | ‘defer’ | ‘dump-proof’ ⟨*file-path*⟩ | ‘show-ihs’

| ‘show-proof’

⟨*session-subcommand*⟩ ::= ‘list’ | ‘defer’ | ‘create’ | ‘serialize’

⟨*subgoal-subcommand*⟩ ::= ‘list’ | ‘defer’

A.1.10 Grammar for Harpoon's Proof Scripts

The following is the grammar for the serialized form of Harpoon interactive sessions.

⟨*harpoon-proof*⟩ ::= '`?`'[⟨*identifier*⟩]

| ⟨*harpoon-command*⟩ '`;`' ⟨*harpoon-proof*⟩

| ⟨*harpoon-directive*⟩

⟨*harpoon-directive*⟩ ::= '`intros`' ⟨*harpoon-hypothetical*⟩

| '`solve`' ⟨*comp-expression*⟩

| '`split`' ⟨*comp-expression*⟩ '`as`' ⟨*harpoon-case*⟩*

| '`impossible`' ⟨*comp-expression*⟩

| '`suffices`' '`by`' ⟨*comp-expression*⟩ '`toshow`' ⟨*comp-type*⟩ '`{`' ⟨*harpoon-proof*⟩ '`}`'

⟨*harpoon-case*⟩ ::= '`case`' ⟨*harpoon-case-label*⟩ '`:`' ⟨*harpoon-hypothetical*⟩

⟨*harpoon-case-label*⟩ ::= '`extended`' '`by`' ⟨*integer*⟩

| '`empty`' '`context`'

| ⟨*qualified-identifier*⟩

| '`#`' [⟨*integer*⟩] [⟨*dot-integer*⟩]

| '`head`' '`variable`'

⟨*harpoon-hypothetical*⟩ ::= ⟨*meta-context*⟩ '`|`' ⟨*comp-context*⟩ '`;`' ⟨*harpoon-proof*⟩

A.1.11 Grammar for Pragmas

The following is the grammar for pragmas that adjust parameters for parsing, elaboration or printing. In particular, pragmas allow the user to choose a notation for operators.

⟨*pragma*⟩ ::= '`--prefix`' ⟨*qualified-identifier*⟩ [⟨*precedence*⟩] '`.`'

| '`--infix`' ⟨*qualified-identifier*⟩ [⟨*precedence*⟩] [⟨*associativity*⟩] '`.`'

| '`--postfix`' ⟨*qualified-identifier*⟩ [⟨*precedence*⟩] '`.`'

| '`--name`' ⟨*qualified-identifier*⟩ ⟨*identifier*⟩ [⟨*identifier*⟩] '`.`'

| '`--not`'

| ‘--assoc <associativity> ‘.’

| ‘--open <qualified-identifier> ‘.’

| ‘--abbrev <qualified-identifier> <identifier> ‘.’

| ‘--query’ (‘*’ | ⟨*integer*⟩) (‘*’ | ⟨*integer*⟩) [⟨*identifier*⟩] ‘:’ ⟨*clf-term*⟩ ‘.’

⟨*global-pragma*⟩ ::= ‘--nostrenghten’ | ‘--coverage’ | ‘--warncoverage’

A.1.12 Grammar for Beluga Signatures

The following is the grammar for toplevel code sections in Beluga. Constant declarations are arranged as lists, and may appear in modules for namespacing. This includes the declaration of LF type-level and term-level constants, computation-level type That is, type constants, term constants, programs and proofs are arranged as lists of declarations, and may appear in modules.

⟨*signature*⟩ ::= ⟨*global-pragma*⟩* ⟨*entry*⟩*

⟨*entry*⟩ ::= ⟨*declaration*⟩ | ⟨*pragma*⟩ | ⟨*documentation-comment*⟩

⟨*declaration*⟩ ::= ⟨*module-declaration*⟩

| ⟨*lf-type-constants-declaration*⟩

| ⟨*lf-type-constant-plain-declaration*⟩

| ⟨*lf-term-constant-plain-declaration*⟩

| ⟨*comp-type-constants-declaration*⟩

| ⟨*mutually-recursive-programs-declaration*⟩

| ⟨*schema-declaration*⟩

| ⟨*typedef-declaration*⟩

| ⟨*let-declaration*⟩

⟨*module-declaration*⟩ ::= ‘module’ ⟨*identifier*⟩ ‘=’ ‘struct’ ⟨*entry*⟩* ‘end’ [‘;’]

⟨*lf-type-constants-declaration*⟩ ::= ‘LF’ ⟨*lf-type-constant-declaration*⟩
(‘and’ [‘LF’] ⟨*lf-type-constant-declaration*⟩)* ‘;’

⟨*lf-type-constant-declaration*⟩ ::= ⟨*identifier*⟩ ‘:’ ⟨*lf-kind*⟩
‘=’ [‘|’] [⟨*lf-term-constant-declaration*⟩] (‘|’ ⟨*lf-term-constant-declaration*⟩)*

⟨*lf-term-constant-declaration*⟩ ::= ⟨*identifier*⟩ ‘:’ ⟨*lf-type*⟩

⟨*lf-type-constant-plain-declaration*⟩ ::= ⟨*identifier*⟩ ‘:’ ⟨*lf-kind*⟩ ‘.’

⟨*lf-term-constant-plain-declaration*⟩ ::= ⟨*identifier*⟩ ‘:’ ⟨*lf-type*⟩ ‘.’

⟨*comp-type-constants-declaration*⟩ ::= ⟨*comp-type-constant-declaration*⟩
(‘and’ ⟨*comp-type-constant-declaration*⟩)* ‘;’

⟨*comp-type-constant-declaration*⟩ ::= ⟨*inductive-type-constant-declaration*⟩
| ⟨*coinductive-type-constant-declaration*⟩
| ⟨*stratified-type-constant-declaration*⟩

⟨*inductive-type-constant-declaration*⟩ ::= ‘inductive’ ⟨*inductive-type-constant*⟩

⟨*inductive-type-constant*⟩ ::= ⟨*identifier*⟩ ‘:’ ⟨*comp-kind*⟩
‘=’ [‘|’] [⟨*inductive-expression-constant*⟩] (‘|’ ⟨*inductive-expression-constant*⟩)*

⟨*inductive-expression-constant*⟩ ::= ⟨*identifier*⟩ ‘:’ ⟨*comp-type*⟩

⟨*coinductive-type-constant-declaration*⟩ ::= ‘coinductive’ ⟨*coinductive-type-constant*⟩

⟨*coinductive-type-constant*⟩ ::= ⟨*identifier*⟩ ‘:’ ⟨*comp-kind*⟩
‘=’ [‘|’] [⟨*coinductive-expression-constant*⟩] (‘|’ ⟨*coinductive-expression-constant*⟩)*

⟨*coinductive-expression-constant*⟩ ::= ⟨*identifier*⟩ ‘:’ ⟨*comp-type*⟩ ‘::’ ⟨*comp-type*⟩

⟨*stratified-type-constant-declaration*⟩ ::= ‘stratified’ ⟨*stratified-type-constant*⟩

⟨*stratified-type-constant*⟩ ::= ⟨*identifier*⟩ ‘`:`’ ⟨*comp-kind*⟩
‘`=`’ [‘`|`’] [⟨*stratified-expression-constant*⟩] (‘`|`’ ⟨*stratified-expression-constant*⟩)*

⟨*stratified-expression-constant*⟩ ::= ⟨*identifier*⟩ ‘`:`’ ⟨*comp-type*⟩

⟨*proof-declaration*⟩ ::= ‘`proof`’ ⟨*identifier*⟩ ‘`:`’ ⟨*comp-type*⟩ ‘`=`’ [⟨*totality-declaration*⟩] ⟨*harpoon-proof*⟩

⟨*program-declaration*⟩ ::= ‘`rec`’ ⟨*identifier*⟩ ‘`:`’ ⟨*comp-type*⟩ ‘`=`’ [⟨*totality-declaration*⟩] ⟨*comp-expression*⟩

⟨*mutually-recursive-programs-declaration*⟩ ::= (⟨*proof-declaration*⟩ | ⟨*program-declaration*⟩)
(‘`and`’ (⟨*proof-declaration*⟩ | ⟨*program-declaration*⟩))* ‘`;`’

⟨*totality-declaration*⟩ ::= ‘`/`’ ‘`total`’ ‘`/`’
| ‘`/`’ ‘`total`’ ⟨*named-totality-order*⟩ ‘`(`’ ⟨*identifier*⟩ ⟨*omittable-identifier*⟩+ ‘`)`’ ‘`/`’
| ‘`/`’ ‘`total`’ ⟨*numeric-totality-order*⟩ ‘`/`’
| ‘`/`’ ‘`trust`’ ‘`/`’

⟨*named-totality-order*⟩ ::= ⟨*named-argument-totality-order*⟩
| ⟨*named-lexical-totality-order*⟩
| ⟨*named-simultaneous-totality-order*⟩

⟨*named-argument-totality-order*⟩ ::= ⟨*identifier*⟩

⟨*named-lexical-totality-order*⟩ ::= ‘`{`’ ⟨*named-totality-order*⟩+ ‘`}`’

⟨*named-simultaneous-totality-order*⟩ ::= ‘`[`’ ⟨*named-totality-order*⟩+ ‘`]`’

⟨*numeric-totality-order*⟩ ::= ⟨*numeric-argument-totality-order*⟩
| ⟨*numeric-lexical-totality-order*⟩
| ⟨*numeric-simultaneous-totality-order*⟩

⟨*numeric-argument-totality-order*⟩ ::= ⟨*integer*⟩

⟨*numeric-lexical-totality-order*⟩ ::= ‘`{`’ ⟨*numeric-totality-order*⟩+ ‘`}`’

⟨*numeric-simultaneous-totality-order*⟩ ::= '`[`' ⟨*numeric-totality-order*⟩+ '`]`'

⟨*schema-declaration*⟩ ::= '`schema`' ⟨*identifier*⟩ '`=`' ⟨*schema*⟩ '`;`'

⟨*typedef-declaration*⟩ ::= '`typedef`' ⟨*identifier*⟩ '`:`' ⟨*comp-kind*⟩ '`=`' ⟨*comp-type*⟩ '`;`'

⟨*let-declaration*⟩ ::= '`let`' ⟨*identifier*⟩ '`:`' ⟨*comp-type*⟩ '`=`' ⟨*comp-expression*⟩ '`;`'

⟨*documentation-comment*⟩ ::= '`%{{`' ⟨*any*⟩* '`}}%`'

A.2 Resolving Syntactic Ambiguities

This section highlights some syntactic ambiguities in the syntax for Beluga as presented in section A.1, and presents strategies for resolving them. Ambiguities in the syntax include:

- Fully qualified identifiers, projections, λ-terms, old-style LF declarations and destructors use the same dot operator.
- LF kinds, types and terms share syntactic constructs for arrows, Π's and applications.
- Contextual LF types, terms and patterns share syntactic constructs for arrows and applications.
- Computation kinds and types share syntactic constructs for arrows, Π's and applications.
- Ambiguities associated with the usual operators in expression grammars with their associativities and precedences.
- Ambiguities associated with user-defined operators with their fixities, associativities and precedences.

Resolution for ambiguities in a grammar is either stateless (context-free) using grammar rewriting, or stateful (context-sensitive or data-dependent) using semantic analysis data produced during or after parsing.

Ambiguities for the built-in operators in the syntax may be resolved statelessly by rewriting the grammar to be suitable for recursive-descent parsing.

The remainder of the ambiguities listed above must be resolved statefully since they require complete scope information to correctly resolve operator sorts and the parser settings defined by the user. Furthermore, given that declarations in a Beluga signature may be mutually recursive, resolving these ambiguities is best done in a separate phase after parsing. Indeed, attempting to resolve these ambiguities in a top-down parser would require possibly unbounded lookaheads and more costly backtracking. As such, a Beluga signature is first parsed into an AST with ambiguous nodes, and then a separate disambiguation phase elaborates that AST to the external syntax. By design, the grammar of section A.1 does not require type information for disambiguation. Hence, the symbol table only keeps track of identifiers in scope and their associated sort without types.

The following sections present alternative production rules used in place of those in the grammar of section A.1 obtained by merging nonterminals that lead to ambiguities. Rewriting the grammar for optimization, left recursion elimination, and stateless ambiguity resolution with recursive descent is outside the scope of this presentation.

A.2.1 Resolving Syntactic Ambiguities for LF's Grammar

The grammar for LF kinds, types and terms from section A.1.5 is data-dependent because LF term variables, type constants and term constants are syntactically indistinguishable from one another, and likewise for Π-kinds and Π-types. This next grammar blurs together LF kinds, types and terms with corresponding nonterminals ⟨*lf-kind*⟩, ⟨*lf-type*⟩ and ⟨*lf-term*⟩ for context-free parsing. Occurrences of ⟨*lf-kind*⟩, ⟨*lf-type*⟩ and ⟨*lf-term*⟩ elsewhere in the

grammar are replaced with ⟨*lf-object*⟩. Disambiguation of an LF object as presented below requires:

- a referencing environment to resolve constants and variables, and
- a target sort for the elaboration (i.e. knowing beforehand whether the LF object should be a kind, type or term).

⟨*lf-object*⟩ ::= ⟨*identifier*⟩
| ⟨*qualified-identifier*⟩
| '`type`'
| '`_`'
| '`{`' ⟨*omittable-identifier*⟩ ['`:`' ⟨*lf-object*⟩] '`}`' ⟨*lf-object*⟩
| '`\`' '`(`' ⟨*omittable-identifier*⟩ '`:`' ⟨*lf-object*⟩ '`)`' '`.`' ⟨*lf-object*⟩
| '`\`' ⟨*omittable-identifier*⟩ '`.`' ⟨*lf-object*⟩
| ⟨*lf-object*⟩ ⟨*forward-arrow*⟩ ⟨*lf-object*⟩
| ⟨*lf-object*⟩ ⟨*backward-arrow*⟩ ⟨*lf-object*⟩
| ⟨*lf-object*⟩ '`:`' ⟨*lf-object*⟩
| ⟨*lf-object*⟩ ⟨*lf-object*⟩
| '`(`' ⟨*lf-object*⟩ '`)`'

Additionally, the syntax is disambiguated in order of decreasing precedence as follows:

1. The juxtaposition of LF objects (separated by whitespace to denote application) is left-associative.
2. User-declared infix, prefix (right-associative) and postfix (left-associative) operators.
3. Forward arrows are right-associative, and backward arrows are left-associative, with equal precedence, hence they may not both appear at the same precedence level.

4. Type ascriptions are left-associative.

5. Binders are weak prefix operators, meaning that the identifier they introduce is in scope for the entire LF object on the right.

A.2.2 Resolving Syntactic Ambiguities for Contextual LF's Grammar

The grammar for contextual LF types, terms, patterns, substitutions, contexts and their patterns from section A.1.6 is data-dependent because contextual LF term variables, type constants and term constants are syntactically indistinguishable from one another. This next grammar blurs together contextual LF types, terms and patterns with corresponding nonterminals ⟨*clf-type*⟩, ⟨*clf-term*⟩ and ⟨*clf-pattern*⟩ for context-free parsing. Occurrences of ⟨*clf-type*⟩, ⟨*clf-term*⟩ and ⟨*clf-pattern*⟩ elsewhere in the grammar are replaced with ⟨*clf-object*⟩. Likewise, the nonterminals ⟨*clf-substitution*⟩, ⟨*clf-substitution-pattern*⟩, ⟨*clf-context*⟩ and ⟨*clf-context-pattern*⟩ are blurred together as ⟨*clf-context-object*⟩. Disambiguation of a contextual LF object as presented below requires:

- a referencing environment to resolve constants and variables, and
- a target sort for the elaboration (i.e. knowing beforehand whether the contextual LF object should be a type, term or pattern).

Additionally, contextual LF contexts starting with an untyped identifier are always disambiguated as having a context variable in head position.

⟨*clf-object*⟩ ::= ⟨*identifier*⟩
| ⟨*hash-identifier*⟩
| ⟨*dollar-identifier*⟩
| ⟨*qualified-identifier*⟩

| ‘{’ ⟨*omittable-identifier*⟩ [‘:’ ⟨*clf-object*⟩] ‘}’ ⟨*clf-object*⟩

| ‘\’ ‘(’ ⟨*omittable-identifier*⟩ ‘:’ ⟨*clf-object*⟩ ‘)’ ‘.’ ⟨*clf-object*⟩

| ‘\’ ⟨*omittable-identifier*⟩ ‘.’ ⟨*clf-object*⟩

| ⟨*clf-object*⟩ ⟨*forward-arrow*⟩ ⟨*clf-object*⟩

| ⟨*clf-object*⟩ ⟨*backward-arrow*⟩ ⟨*clf-object*⟩

| ‘block’ ‘(’ [⟨*identifier*⟩ ‘:’] ⟨*clf-object*⟩ (‘,’ [⟨*identifier*⟩ ‘:’] ⟨*clf-object*⟩)* ‘)’

| ‘block’ [⟨*identifier*⟩ ‘:’] ⟨*clf-object*⟩ (‘,’ [⟨*identifier*⟩ ‘:’] ⟨*clf-object*⟩)*

| ⟨*clf-object*⟩ ‘:’ ⟨*clf-object*⟩

| ⟨*clf-object*⟩ ⟨*clf-object*⟩

| ‘_’

| ‘?’[⟨*identifier*⟩]

| ⟨*clf-object*⟩ ‘[’ ⟨*clf-context-object*⟩ ‘]’

| ‘<’ ⟨*clf-object*⟩ (‘;’ ⟨*clf-object*⟩)* ‘>’

| ⟨*clf-object*⟩ ⟨*dot-integer*⟩

| ⟨*clf-object*⟩ ⟨*dot-identifier*⟩

| ‘(’ ⟨*clf-object*⟩ ‘)’

⟨*clf-context-object*⟩ ::= [‘^’]

| ‘..’

| [‘..’ ‘,’] [⟨*identifier*⟩ ‘:’] ⟨*clf-object*⟩ (‘,’ [⟨*identifier*⟩ ‘:’] ⟨*clf-object*⟩)*

Additionally, the syntax is disambiguated in order of decreasing precedence as follows:

1. The juxtaposition of contextual LF objects (separated by whitespace to denote application) is left-associative.

2. User-declared infix, prefix and postfix operators.

3. Projections are left-associative.

4. Substitutions are left-associative.

5. Forward arrows are right-associative, and backward arrows are left-associative, with equal precedence, hence they may not both appear at the same precedence level.

6. Type ascriptions are left-associative.

7. Binders are weak prefix operators, meaning that the identifier they introduce is in scope for the entire contextual LF object on the right.

A.2.3 Resolving Syntactic Ambiguities for the Meta-Level's Grammar

The grammar for context schemas, meta-types, meta-objects, meta-patterns and meta-contexts from section A.1.7 is ambiguous because the contextual LF types and terms therein have ambiguous grammars. This next grammar blurs together meta-types, meta-objects and meta-patterns with corresponding nonterminals ⟨*meta-type*⟩, ⟨*meta-object*⟩ and ⟨*meta-pattern*⟩ for context-free parsing. Occurrences of ⟨*meta-type*⟩, ⟨*meta-object*⟩ and ⟨*meta-pattern*⟩ elsewhere in the grammar are replaced with ⟨*meta-thing*⟩. Likewise, the nonterminals ⟨*schema*⟩ and ⟨*meta-context*⟩ are replaced with ⟨*schema-object*⟩ and ⟨*meta-context-object*⟩ respectively. Disambiguation of a meta-thing as presented below requires:

- a referencing environment to resolve constants and variables, and
- a target sort for the elaboration (i.e. knowing beforehand whether the meta-thing should be a meta-type, meta-object or pattern).

⟨*schema-object*⟩ ::= ⟨*qualified-identifier*⟩
| ⟨*schema-object*⟩ '`+`' ⟨*schema-object*⟩
| ['`some`' '`[`' ⟨*identifier*⟩ '`:`' ⟨*lf-object*⟩ ('`,`' ⟨*identifier*⟩ '`:`' ⟨*lf-object*⟩)* '`]`'] '`block`' '`(`'

[⟨*identifier*⟩ ‘`:`’] ⟨*lf-object*⟩ (‘`,`’ [⟨*identifier*⟩ ‘`:`’] ⟨*lf-object*⟩)* ‘`)`’

| [‘`some`’ ‘`[`’ ⟨*identifier*⟩ ‘`:`’ ⟨*lf-object*⟩ (‘`,`’ ⟨*identifier*⟩ ‘`:`’ ⟨*lf-object*⟩)* ‘`]`’] ‘`block`’ [⟨*identifier*⟩ ‘`:`’] ⟨*lf-object*⟩ (‘`,`’ [⟨*identifier*⟩ ‘`:`’] ⟨*lf-object*⟩)*

⟨*meta-thing*⟩ ::= ⟨*qualified-identifier*⟩

| ‘`(`’ ⟨*clf-context-object*⟩ ⟨*turnstile*⟩ ⟨*clf-context-object*⟩ ‘`)`’

| ‘`#(`’ ⟨*clf-context-object*⟩ ⟨*turnstile*⟩ ⟨*clf-context-object*⟩ ‘`)`’

| ‘`$(`’ ⟨*clf-context-object*⟩ ⟨*turnstile*⟩ ⟨*clf-context-object*⟩ ‘`)`’

| ‘`$(`’ ⟨*clf-context-object*⟩ ⟨*turnstile-hash*⟩ ⟨*clf-context-object*⟩ ‘`)`’

| ‘`[`’ ⟨*clf-context-object*⟩ ‘`]`’

| ‘`[`’ ⟨*clf-context-object*⟩ ⟨*turnstile*⟩ ⟨*clf-context-object*⟩ ‘`]`’

| ‘`#[`’ ⟨*clf-context-object*⟩ ⟨*turnstile*⟩ ⟨*clf-context-object*⟩ ‘`]`’

| ‘`$[`’ ⟨*clf-context-object*⟩ ⟨*turnstile*⟩ ⟨*clf-context-object*⟩ ‘`]`’

| ‘`$[`’ ⟨*clf-context-object*⟩ ⟨*turnstile-hash*⟩ ⟨*clf-context-object*⟩ ‘`]`’

⟨*meta-context-object*⟩ ::= [‘`^`’]

| ⟨*meta-object-identifier*⟩ ‘`:`’ ⟨*meta-thing*⟩ (‘`,`’ ⟨*meta-object-identifier*⟩ ‘`:`’ ⟨*meta-thing*⟩)*

Additionally, in a meta-thing, a fully qualified identifier referring to a context schema constant is ambiguous with a context variable. Such context schema constants are disambiguated from the parsed context object.

A.2.4 Resolving Syntactic Ambiguities in Computations

The grammar for computation-level kinds and types from section A.1.8 is data-dependent because computational base and cobase type constants are syntactically indistinguishable from one another, and because the grammar relies on meta-level objects. This next grammar blurs together computational kinds and types with corresponding nonterminals ⟨*comp-kind*⟩ and ⟨*comp-type*⟩ for context-free parsing. Occurrences of ⟨*comp-kind*⟩ and ⟨*comp-type*⟩ elsewhere

in the grammar are replaced with ⟨*comp-sort-object*⟩. Disambiguation of a computational sort object as presented below requires:

- a referencing environment to resolve constants and variables, and
- a target sort for the elaboration (i.e. knowing beforehand whether the computation-level sort object should be a kind or a type).

⟨*comp-sort-object*⟩ ::= ⟨*identifier*⟩
| ⟨*qualified-identifier*⟩
| '`ctype`'
| '`{`' ⟨*omittable-meta-object-identifier*⟩ ['`:`' ⟨*meta-thing*⟩] '`}`' ⟨*comp-sort-object*⟩
| '`(`' ⟨*omittable-meta-object-identifier*⟩ ['`:`' ⟨*meta-thing*⟩] '`)`' ⟨*comp-sort-object*⟩
| ⟨*comp-sort-object*⟩ ⟨*forward-arrow*⟩ ⟨*comp-sort-object*⟩
| ⟨*comp-sort-object*⟩ ⟨*backward-arrow*⟩ ⟨*comp-sort-object*⟩
| ⟨*comp-sort-object*⟩ '`*`' ⟨*comp-sort-object*⟩
| ⟨*comp-sort-object*⟩ ⟨*comp-sort-object*⟩
| ⟨*meta-thing*⟩
| '`(`' ⟨*comp-sort-object*⟩ '`)`'

Additionally, the syntax is disambiguated in order of decreasing precedence as follows:

1. The juxtaposition of computational sort objects (separated by whitespace to denote application) is left-associative.
2. User-declared infix, prefix (right-associative) and postfix (left-associative) operators.
3. Forward arrows are right-associative, and backward arrows are left-associative, with equal precedence, hence they may not both appear at the same precedence level.
4. Binders are weak prefix operators, meaning that the identifier they introduce is in scope for the entire sort object on the right.

Appendix B

Equivalence of Indexing Specifications

Theorem 2 (Equivalence)**.** *The formalisms for indexing with respect to the immutable and mutable representations of the referencing environment are equivalent.*

1. $\{\mathsf{Env} = \Xi\}\ K \rightsquigarrow_{\mathrm{K}} \tilde{K}\ \{\mathsf{Env} = \Xi'\}$ *if and only if* $\Xi \vdash K \rightsquigarrow_{\mathrm{K}} \tilde{K}$ *and* $\Xi' = \Xi$*.*

2. $\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi'\}$ *if and only if* $\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}$ *and* $\Xi' = \Xi$*.*

3. $\{\mathsf{Env} = \Xi\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \Xi'\}$ *if and only if* $\Xi \vdash M \rightsquigarrow_{\mathrm{M}} \tilde{M}$ *and* $\Xi' = \Xi$*.*

Proof.

$\Rightarrow$ By simultaneous induction:

1. Assume $\{\mathsf{Env} = \Xi\}\ K \overset{\mathcal{D}}{\rightsquigarrow}_{\mathrm{K}} \tilde{K}\ \{\mathsf{Env} = \Xi'\}$.

   By structural induction on $\mathcal{D}$:

   – Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi''\}} \quad \overset{\mathcal{D}_2}{\{\mathsf{Env} = \mathrm{shift}_\Psi\,(\Xi'')\}\ K \rightsquigarrow_{\mathrm{K}} \tilde{K}\ \{\mathsf{Env} = \Xi'''\}}}{\{\mathsf{Env} = \Xi\}\ A \to K \rightsquigarrow_{\mathrm{K}} \tilde{A} \to \tilde{K}\ \{\mathsf{Env} = \mathrm{unshift}_\Psi\,(\Xi''')\}}$.

   By the induction hypothesis on $\mathcal{D}_1$, we have $\Xi \vdash A \overset{\mathcal{E}_1}{\rightsquigarrow}_{\mathrm{A}} \tilde{A}$ and $\Xi'' = \Xi$.

   By the induction hypothesis on $\mathcal{D}_2$, we have $\mathrm{shift}_\Psi\,(\Xi'') \vdash K \overset{\mathcal{E}_2}{\rightsquigarrow}_{\mathrm{K}} \tilde{K}$ and $\Xi''' = \mathrm{shift}_\Psi\,(\Xi'')$.

   Then, $\Xi' = \mathrm{unshift}_\Psi\,(\Xi''') = \mathrm{unshift}_\Psi\,(\mathrm{shift}_\Psi\,(\Xi'')) = \Xi'' = \Xi$, and

   $$\dfrac{\overset{\mathcal{E}_1}{\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}} \quad \dfrac{\overset{\mathcal{E}_2}{\mathrm{shift}_\Psi\,(\Xi) \vdash K \rightsquigarrow_{\mathrm{K}} \tilde{K}}}{\Xi, \_ : \mathsf{LF}_{\mathsf{term}} \vdash K \rightsquigarrow_{\mathrm{K}} \tilde{K}}}{\Xi \vdash A \to K \rightsquigarrow_{\mathrm{K}} \tilde{A} \to \tilde{K}}.$$

– Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi''\}} \quad \overset{\mathcal{D}_2}{\{\mathsf{Env} = \mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi'')\}\ K \rightsquigarrow_{\mathrm{K}} \tilde{K}\ \{\mathsf{Env} = \Xi'''\}}}{\{\mathsf{Env} = \Xi\}\ \Pi x{:}A.K \rightsquigarrow_{\mathrm{K}} \Pi_{\tilde{A}} \tilde{K}\ \{\mathsf{Env} = \mathrm{pop}_\Psi\,[x]\,(\Xi''')\}}$.

By the induction hypothesis on $\overset{\mathcal{E}_1}{\mathcal{D}_1}$, we have $\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}$ and $\Xi'' = \Xi$.

By the induction hypothesis on $\overset{\mathcal{E}_2}{\mathcal{D}_2}$, we have $\mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi'') \vdash K \rightsquigarrow_{\mathrm{K}} \tilde{K}$ and $\Xi''' = \mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi'')$.

Then, $\Xi' = \mathrm{pop}_\Psi\,[x]\,(\Xi''') = \mathrm{pop}_\Psi\,[x]\,(\mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi'')) = \Xi'' = \Xi$, and

$$\dfrac{\overset{\mathcal{E}_1}{\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}} \quad \dfrac{\overset{\mathcal{E}_2}{\mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi) \vdash K \rightsquigarrow_{\mathrm{K}} \tilde{K}}}{\Xi, x : \mathsf{LF}_{\mathsf{term}} \vdash K \rightsquigarrow_{\mathrm{K}} \tilde{K}}}{\Xi \vdash \Pi x{:}A.K \rightsquigarrow_{\mathrm{K}} \Pi_{\tilde{A}} \tilde{K}}.$$

– Case $\mathcal{D} = \overline{\{\mathsf{Env} = \Xi\}\ \mathbf{type} \rightsquigarrow_{\mathrm{K}} \mathbf{type}\ \{\mathsf{Env} = \Xi\}}$.

$\Xi \vdash \mathbf{type} \rightsquigarrow_{\mathrm{K}} \mathbf{type}$ holds trivially.

2. Assume $\overset{\mathcal{D}}{\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi'\}}$.

By structural induction on $\mathcal{D}$:

– Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi''\}} \quad \overset{\mathcal{D}_2}{\{\mathsf{Env} = \mathrm{shift}_\Psi\,(\Xi'')\}\ B \rightsquigarrow_{\mathrm{A}} \tilde{B}\ \{\mathsf{Env} = \Xi'''\}}}{\{\mathsf{Env} = \Xi\}\ A \to B \rightsquigarrow_{\mathrm{A}} \tilde{A} \to \tilde{B}\ \{\mathsf{Env} = \mathrm{unshift}_\Psi\,(\Xi''')\}}$.

By the induction hypothesis on $\overset{\mathcal{E}_1}{\mathcal{D}_1}$, we have $\Xi \vdash A \to B \rightsquigarrow_{\mathrm{A}} \tilde{A} \to \tilde{B}$ and $\Xi'' = \Xi$.

By the induction hypothesis on $\overset{\mathcal{E}_2}{\mathcal{D}_2}$, we have $\mathrm{shift}_\Psi\,(\Xi'') \vdash B \rightsquigarrow_{\mathrm{A}} \tilde{B}$ and $\Xi''' = \mathrm{shift}_\Psi\,(\Xi'')$.

Then, $\Xi' = \mathrm{unshift}_\Psi\,(\Xi''') = \mathrm{unshift}_\Psi\,(\mathrm{shift}_\Psi\,(\Xi'')) = \Xi'' = \Xi$, and

$$\dfrac{\overset{\mathcal{E}_1}{\Xi \vdash A \to B \rightsquigarrow_{\mathrm{A}} \tilde{A} \to \tilde{B}} \quad \dfrac{\overset{\mathcal{E}_2}{\mathrm{shift}_\Psi\,(\Xi) \vdash B \rightsquigarrow_{\mathrm{A}} \tilde{B}}}{\Xi, \_ : \mathsf{LF}_{\mathsf{term}} \vdash B \rightsquigarrow_{\mathrm{A}} \tilde{B}}}{\Xi \vdash A \to B \rightsquigarrow_{\mathrm{A}} \tilde{A} \to \tilde{B}}.$$

– Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi''\}} \quad \overset{\mathcal{D}_2}{\{\mathsf{Env} = \mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi'')\}\ B \rightsquigarrow_{\mathrm{A}} \tilde{B}\ \{\mathsf{Env} = \Xi'''\}}}{\{\mathsf{Env} = \Xi\}\ \Pi x{:}A.B \rightsquigarrow_{\mathrm{A}} \Pi_{\tilde{A}} \tilde{B}\ \{\mathsf{Env} = \mathrm{pop}_\Psi\,[x]\,(\Xi''')\}}$.

By the induction hypothesis on $\overset{\mathcal{E}_1}{\mathcal{D}_1}$, we have $\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}$ and $\Xi'' = \Xi$.

By the induction hypothesis on $\overset{\mathcal{E}_2}{\mathcal{D}_2}$, we have $\mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi'') \vdash B \rightsquigarrow_{\mathrm{A}} \tilde{B}$ and $\Xi''' = \mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi'')$.

Then, $\Xi' = \mathrm{pop}_\Psi\,[x]\,(\Xi''') = \mathrm{pop}_\Psi\,[x]\,(\mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi'')) = \Xi'' = \Xi$, and

$$\dfrac{\overset{\mathcal{E}_1}{\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}} \quad \dfrac{\overset{\mathcal{E}_2}{\mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi) \vdash B \rightsquigarrow_{\mathrm{A}} \tilde{B}}}{\Xi, x : \mathsf{LF}_{\mathsf{term}} \vdash B \rightsquigarrow_{\mathrm{A}} \tilde{B}}}{\Xi \vdash \Pi x{:}A.B \rightsquigarrow_{\mathrm{A}} \Pi_{\tilde{A}} \tilde{B}}.$$

– Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{\mathrm{lookup}\,[a]\,(\Xi) = \tilde{a} : \mathsf{LF}_{\mathsf{type\ const}}}}{\{\mathsf{Env} = \Xi\}\ a \rightsquigarrow_{\mathrm{A}} \tilde{a}\ \{\mathsf{Env} = \Xi\}}$.

Then,

$$\dfrac{\overset{\mathcal{D}'}{\mathrm{lookup}\,[a]\,(\Xi) = \tilde{a} : \mathsf{LF}_{\mathsf{type\ const}}}}{\Xi \vdash a \rightsquigarrow_{\mathrm{A}} \tilde{a}}.$$

– Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_0}{\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi_0\}} \quad \left(\overset{\mathcal{D}_2}{\{\mathsf{Env} = \Xi_{k-1}\}\ M_k \rightsquigarrow_{\mathrm{M}} \tilde{M}_k\ \{\mathsf{Env} = \Xi_k\}}\right)_{k \in \{1,2,\ldots,n\}}}{\{\mathsf{Env} = \Xi\}\ A\ M_1\ M_2 \ldots M_n \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \tilde{M}_1\ \tilde{M}_2 \ldots \tilde{M}_n\ \{\mathsf{Env} = \Xi_n\}}$.

By the induction hypothesis on $\overset{\mathcal{E}_0}{\mathcal{D}_0}$, we have $\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}$ and $\Xi_0 = \Xi$.

For each $k \in \{1, 2, \ldots, n\}$, by the induction hypothesis on $\overset{\mathcal{E}_k}{\mathcal{D}_k}$, we have $\Xi_{k-1} \vdash M_k \rightsquigarrow_{\mathrm{M}} \tilde{M}_k$ and $\Xi_{k-1} = \Xi_k$.

Then, inductively on n, we have $\Xi_n = \Xi_{n-1} = \cdots = \Xi_0 = \Xi$, and

$$\frac{\overset{\mathcal{E}_0}{\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}} \quad \left(\overset{\mathcal{E}_k}{\Xi \vdash M_k \rightsquigarrow_{\mathrm{M}} \tilde{M}_k}\right)_{k \in \{1,2,\ldots,n\}}}{\Xi \vdash A\ M_1\ M_2 \ldots M_n \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \tilde{M}_1\ \tilde{M}_2 \ldots \tilde{M}_n}.$$

3. Assume $\overset{\mathcal{D}}{\{\mathsf{Env} = \Xi\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \Xi'\}}$.

By structural induction on $\mathcal{D}$:

– Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\mathrm{lookup}\,[x]\,(\Xi) = x : \mathsf{LF}_{\mathsf{term}}} \quad \overset{\mathcal{D}_2}{\mathrm{index}_\Psi\,[x]\,(\Xi) = \iota}}{\{\mathsf{Env} = \Xi\}\ x \rightsquigarrow_{\mathrm{M}} \iota\ \{\mathsf{Env} = \Xi\}}$.

Then,

$$\frac{\overset{\mathcal{D}_1}{\mathrm{lookup}\,[x]\,(\Xi) = x : \mathsf{LF}_{\mathsf{term}}} \quad \overset{\mathcal{D}_2}{\mathrm{index}_\Psi\,[x]\,(\Xi) = \iota}}{\Xi \vdash x \rightsquigarrow_{\mathrm{M}} \iota}.$$

– Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{\mathrm{lookup}\,[c]\,(\Xi) = \tilde{c} : \mathsf{LF}_{\mathsf{term\ const}}}}{\{\mathsf{Env} = \Xi\}\ c \rightsquigarrow_{\mathrm{M}} \tilde{c}\ \{\mathsf{Env} = \Xi\}}$.

Then,

$$\frac{\overset{\mathcal{D}'}{\mathrm{lookup}\,[c]\,(\Xi) = \tilde{c} : \mathsf{LF}_{\mathsf{term\ const}}}}{\Xi \vdash c \rightsquigarrow_{\mathrm{M}} \tilde{c}}.$$

– Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{\{\mathsf{Env} = \mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi)\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \Xi''\}}}{\{\mathsf{Env} = \Xi\}\ \lambda x.M \rightsquigarrow_{\mathrm{M}} \lambda\ \tilde{M}\ \{\mathsf{Env} = \mathrm{pop}_\Psi\,[x]\,(\Xi'')\}}$.

By the induction hypothesis on $\overset{\mathcal{E}'}{\mathcal{D}'}$, we have $\mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi) \vdash M \rightsquigarrow_{\mathrm{M}} \tilde{M}$ and $\Xi'' = \mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi)$.

Then, $\Xi' = \mathrm{pop}_\Psi\,[x]\,(\Xi'') = \mathrm{pop}_\Psi\,[x]\,(\mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi)) = \Xi$, and

$$\frac{\frac{\overset{\mathcal{E}'}{\mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi) \vdash M \rightsquigarrow_{\mathrm{M}} \tilde{M}}}{\Xi, x : \mathsf{LF}_{\mathsf{term}} \vdash M \rightsquigarrow_{\mathrm{M}} \tilde{M}}}{\Xi \vdash \lambda x.M \rightsquigarrow_{\mathrm{M}} \lambda\ \tilde{M}}.$$

– Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_0}{\{\mathsf{Env} = \Xi\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \Xi_0\}} \quad \left(\overset{\mathcal{D}_k}{\{\mathsf{Env} = \Xi_{k-1}\}\ N_k \rightsquigarrow_{\mathrm{M}} \tilde{N}_k\ \{\mathsf{Env} = \Xi_k\}}\right)_{k \in \{1,2,\ldots,n\}}}{\{\mathsf{Env} = \Xi\}\ M\ N_1\ N_2 \ldots N_n \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \tilde{N}_1\ \tilde{N}_2 \ldots \tilde{N}_n\ \{\mathsf{Env} = \Xi_n\}}$.

By the induction hypothesis on $\overset{\mathcal{E}_0}{\mathcal{D}_0}$, we have $\Xi \vdash M \rightsquigarrow_{\mathrm{M}} \tilde{M}$ and $\Xi_0 = \Xi$.

For each $k \in \{1, 2, \ldots, n\}$, by the induction hypothesis on $\overset{\mathcal{E}_k}{\mathcal{D}_k}$, we have $\Xi_{k-1} \vdash N_k \rightsquigarrow_{\mathrm{M}} \tilde{N}_k$ and $\Xi_k = \Xi_{k-1}$.

Then, inductively on n, we have $\Xi_n = \Xi_{n-1} = \cdots = \Xi_0 = \Xi$, and

$$\frac{\overset{\mathcal{E}_k}{\Xi_{k-1} \vdash N_k \rightsquigarrow_{\mathrm{M}} \tilde{N}_k} \quad \left(\overset{\mathcal{E}_k}{\Xi \vdash N_k \rightsquigarrow_{\mathrm{M}} \tilde{N}_k}\right)_{k \in \{1,2,\ldots,n\}}}{\Xi \vdash M\ N_1\ N_2 \ldots N_n \rightsquigarrow_{\mathrm{A}} \tilde{M}\ \tilde{N}_1\ \tilde{N}_2 \ldots \tilde{N}_n}.$$

– Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\{\mathsf{Env} = \Xi\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \Xi''\}} \quad \overset{\mathcal{D}_2}{\{\mathsf{Env} = \Xi''\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi'\}}}{\{\mathsf{Env} = \Xi\}\ M : A \rightsquigarrow_{\mathrm{M}} \tilde{M} : \tilde{A}\ \{\mathsf{Env} = \Xi'\}}$.

By the induction hypothesis on $\mathcal{D}_1$, we have $\Xi \vdash M \overset{\mathcal{E}_1}{\rightsquigarrow_{\mathrm{M}}} \tilde{M}$ and $\Xi'' = \Xi$.

By the induction hypothesis on $\mathcal{D}_2$, we have $\Xi'' \vdash A \overset{\mathcal{E}_2}{\rightsquigarrow_{\mathrm{A}}} \tilde{A}$ and $\Xi' = \Xi''$.

Then, $\Xi' = \Xi'' = \Xi$, and

$$\frac{\overset{\mathcal{E}_1}{\Xi \vdash M \rightsquigarrow_{\mathrm{M}} \tilde{M}} \quad \overset{\mathcal{E}_2}{\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}}}{\Xi \vdash M : A \rightsquigarrow_{\mathrm{A}} \tilde{M} : \tilde{A}}.$$

⇐ By simultaneous induction:

1. Assume $\Xi \vdash K \overset{\mathcal{D}}{\rightsquigarrow_{\mathrm{K}}} \tilde{K}$.

   By structural induction on $\mathcal{D}$:

   – Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}} \quad \overset{\mathcal{D}_2}{\Xi, \_ : \mathsf{LF}_{\mathsf{term}} \vdash K \rightsquigarrow_{\mathrm{K}} \tilde{K}}}{\Xi \vdash A \to K \rightsquigarrow_{\mathrm{K}} \tilde{A} \to \tilde{K}}$.

   By the induction hypothesis on $\mathcal{D}_1$, we have $\{\mathsf{Env} = \Xi\}\ A \overset{\mathcal{E}_1}{\rightsquigarrow_{\mathrm{A}}} \tilde{A}\ \{\mathsf{Env} = \Xi\}$.

   By the induction hypothesis on $\mathcal{D}_2$, we have $\{\mathsf{Env} = \Xi, \_ : \mathsf{LF}_{\mathsf{term}}\}\ K \overset{\mathcal{E}_2}{\rightsquigarrow_{\mathrm{K}}} \tilde{K}\ \{\mathsf{Env} = \Xi, \_ : \mathsf{LF}_{\mathsf{term}}\}$.

   Then,

$$\frac{\dfrac{\overset{\mathcal{E}_1}{\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi\}} \quad \dfrac{\overset{\mathcal{E}_2}{\{\mathsf{Env} = \Xi, \_ : \mathsf{LF}_{\mathsf{term}}\}\ K \rightsquigarrow_{\mathrm{K}} \tilde{K}\ \{\mathsf{Env} = \Xi, \_ : \mathsf{LF}_{\mathsf{term}}\}}}{\{\mathsf{Env} = \mathrm{shift}_{\Psi}\ (\Xi)\}\ K \rightsquigarrow_{\mathrm{K}} \tilde{K}\ \{\mathsf{Env} = \mathrm{shift}_{\Psi}\ (\Xi)\}}}{\{\mathsf{Env} = \Xi\}\ A \to K \rightsquigarrow_{\mathrm{K}} \tilde{A} \to \tilde{K}\ \{\mathsf{Env} = \mathrm{unshift}_{\Psi}\ (\mathrm{shift}_{\Psi}\ (\Xi))\}}}{\{\mathsf{Env} = \Xi\}\ A \to K \rightsquigarrow_{\mathrm{K}} \tilde{A} \to \tilde{K}\ \{\mathsf{Env} = \Xi\}}.$$

   – Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}} \quad \overset{\mathcal{D}_2}{\Xi, x : \mathsf{LF}_{\mathsf{term}} \vdash K \rightsquigarrow_{\mathrm{K}} \tilde{K}}}{\Xi \vdash \Pi x{:}A.K \rightsquigarrow_{\mathrm{K}} \Pi_{\tilde{A}}\ \tilde{K}}$.

   By the induction hypothesis on $\mathcal{D}_1$, we have $\{\mathsf{Env} = \Xi\}\ A \overset{\mathcal{E}_1}{\rightsquigarrow_{\mathrm{A}}} \tilde{A}\ \{\mathsf{Env} = \Xi\}$.

   By the induction hypothesis on $\mathcal{D}_2$, we have $\{\mathsf{Env} = \Xi, x : \mathsf{LF}_{\mathsf{term}}\}\ K \overset{\mathcal{E}_2}{\rightsquigarrow_{\mathrm{K}}} \tilde{K}\ \{\mathsf{Env} = \Xi, x : \mathsf{LF}_{\mathsf{term}}\}$.

   Then,

$$\frac{\dfrac{\overset{\mathcal{E}_1}{\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi\}} \quad \dfrac{\overset{\mathcal{E}_2}{\{\mathsf{Env} = \Xi, x : \mathsf{LF}_{\mathsf{term}}\}\ K \rightsquigarrow_{\mathrm{K}} \tilde{K}\ \{\mathsf{Env} = \Xi, x : \mathsf{LF}_{\mathsf{term}}\}}}{\{\mathsf{Env} = \mathrm{push}_{\Psi}\ [x : \mathsf{LF}_{\mathsf{term}}]\ (\Xi)\}\ K \rightsquigarrow_{\mathrm{K}} \tilde{K}\ \{\mathsf{Env} = \mathrm{push}_{\Psi}\ [x : \mathsf{LF}_{\mathsf{term}}]\ (\Xi)\}}}{\{\mathsf{Env} = \Xi\}\ \Pi x{:}A.K \rightsquigarrow_{\mathrm{K}} \Pi_{\tilde{A}} \tilde{K}\ \{\mathsf{Env} = \mathrm{pop}_{\Psi}\ [x]\ (\mathrm{push}_{\Psi}\ [x : \mathsf{LF}_{\mathsf{term}}]\ (\Xi))\}}}{\{\mathsf{Env} = \Xi\}\ \Pi x{:}A.K \rightsquigarrow_{\mathrm{K}} \Pi_{\tilde{A}} \tilde{K}\ \{\mathsf{Env} = \Xi\}}.$$

– Case $\mathcal{D} = \dfrac{}{\Xi \vdash \mathbf{type} \rightsquigarrow_{\mathrm{K}} \mathbf{type}}$.

$\{\mathsf{Env} = \Xi\}\ \mathbf{type} \rightsquigarrow_{\mathrm{K}} \mathbf{type}\ \{\mathsf{Env} = \Xi\}$ holds trivially.

2. Assume $\Xi \vdash A \overset{\mathcal{D}}{\rightsquigarrow_{\mathrm{A}}} \tilde{A}$. By structural induction on $\mathcal{D}$:

– Case $\mathcal{D} = \dfrac{\Xi \vdash A \overset{\mathcal{D}_1}{\rightsquigarrow_{\mathrm{A}}} \tilde{A} \quad \Xi, \_ : \mathsf{LF}_{\mathsf{term}} \vdash B \overset{\mathcal{D}_2}{\rightsquigarrow_{\mathrm{A}}} \tilde{B}}{\Xi \vdash A \to B \rightsquigarrow_{\mathrm{A}} \tilde{A} \to \tilde{B}}$.

By the induction hypothesis on $\mathcal{D}_1$, we have $\{\mathsf{Env} = \Xi\}\ A \overset{\mathcal{E}_1}{\rightsquigarrow_{\mathrm{A}}} \tilde{A}\ \{\mathsf{Env} = \Xi\}$.

By the induction hypothesis on $\mathcal{D}_2$, we have $\{\mathsf{Env} = \Xi, \_ : \mathsf{LF}_{\mathsf{term}}\}\ B \overset{\mathcal{E}_2}{\rightsquigarrow_{\mathrm{A}}} \tilde{B}\ \{\mathsf{Env} = \Xi, \_ : \mathsf{LF}_{\mathsf{term}}\}$.

Then,

$$\dfrac{\dfrac{\{\mathsf{Env} = \Xi\}\ A \overset{\mathcal{E}_1}{\rightsquigarrow_{\mathrm{A}}} \tilde{A}\ \{\mathsf{Env} = \Xi\} \quad \dfrac{\{\mathsf{Env} = \Xi, \_ : \mathsf{LF}_{\mathsf{term}}\}\ B \overset{\mathcal{E}_2}{\rightsquigarrow_{\mathrm{A}}} \tilde{B}\ \{\mathsf{Env} = \Xi, \_ : \mathsf{LF}_{\mathsf{term}}\}}{\{\mathsf{Env} = \mathrm{shift}_\Psi\,(\Xi)\}\ B \rightsquigarrow_{\mathrm{A}} \tilde{B}\ \{\mathsf{Env} = \mathrm{shift}_\Psi\,(\Xi)\}}}{\{\mathsf{Env} = \Xi\}\ A \to B \rightsquigarrow_{\mathrm{A}} \tilde{A} \to \tilde{B}\ \{\mathsf{Env} = \mathrm{unshift}_\Psi\,(\mathrm{shift}_\Psi\,(\Xi))\}}}{\{\mathsf{Env} = \Xi\}\ A \to B \rightsquigarrow_{\mathrm{A}} \tilde{A} \to \tilde{B}\ \{\mathsf{Env} = \Xi\}}.$$

– Case $\mathcal{D} = \dfrac{\Xi \vdash A \overset{\mathcal{D}_1}{\rightsquigarrow_{\mathrm{A}}} \tilde{A} \quad \Xi, x : \mathsf{LF}_{\mathsf{term}} \vdash B \overset{\mathcal{D}_2}{\rightsquigarrow_{\mathrm{A}}} \tilde{B}}{\Xi \vdash \Pi x{:}A.B \rightsquigarrow_{\mathrm{A}} \Pi_{\tilde{A}}\ \tilde{B}}$.

By the induction hypothesis on $\mathcal{D}_1$, we have $\{\mathsf{Env} = \Xi\}\ A \overset{\mathcal{E}_1}{\rightsquigarrow_{\mathrm{A}}} \tilde{A}\ \{\mathsf{Env} = \Xi\}$.

By the induction hypothesis on $\mathcal{D}_2$, we have $\{\mathsf{Env} = \Xi, x : \mathsf{LF}_{\mathsf{term}}\}\ B \overset{\mathcal{E}_2}{\rightsquigarrow_{\mathrm{A}}} \tilde{B}\ \{\mathsf{Env} = \Xi, x : \mathsf{LF}_{\mathsf{term}}\}$.

Then,

$$\dfrac{\dfrac{\{\mathsf{Env} = \Xi\}\ A \overset{\mathcal{E}_1}{\rightsquigarrow_{\mathrm{A}}} \tilde{A}\ \{\mathsf{Env} = \Xi\} \quad \dfrac{\{\mathsf{Env} = \Xi, x : \mathsf{LF}_{\mathsf{term}}\}\ B \overset{\mathcal{E}_2}{\rightsquigarrow_{\mathrm{A}}} \tilde{B}\ \{\mathsf{Env} = \Xi, x : \mathsf{LF}_{\mathsf{term}}\}}{\{\mathsf{Env} = \mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi)\}\ B \rightsquigarrow_{\mathrm{A}} \tilde{B}\ \{\mathsf{Env} = \mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi)\}}}{\{\mathsf{Env} = \Xi\}\ \Pi x{:}A.B \rightsquigarrow_{\mathrm{A}} \Pi_{\tilde{A}}\ \tilde{B}\ \{\mathsf{Env} = \mathrm{pop}_\Psi\,[x]\,(\mathrm{push}_\Psi\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi))\}}}{\{\mathsf{Env} = \Xi\}\ \Pi x{:}A.B \rightsquigarrow_{\mathrm{A}} \Pi_{\tilde{A}}\ \tilde{B}\ \{\mathsf{Env} = \Xi\}}.$$

– Case $\mathcal{D} = \dfrac{\mathrm{lookup}\,[a]\,(\Xi) \overset{\mathcal{D}'}{=} \tilde{a} : \mathsf{LF}_{\mathsf{type\ const}}}{\Xi \vdash a \rightsquigarrow_{\mathrm{A}} \tilde{a}}$.

Then,

$$\dfrac{\mathrm{lookup}\,[a]\,(\Xi) \overset{\mathcal{D}'}{=} \tilde{a} : \mathsf{LF}_{\mathsf{type\ const}}}{\{\mathsf{Env} = \Xi\}\ a \rightsquigarrow_{\mathrm{A}} \tilde{a}\ \{\mathsf{Env} = \Xi\}}.$$

– Case $\mathcal{D} = \dfrac{\Xi \vdash A \overset{\mathcal{D}_0}{\rightsquigarrow_{\mathrm{A}}} \tilde{A} \quad \left(\Xi \vdash M_k \overset{\mathcal{D}_k}{\rightsquigarrow_{\mathrm{M}}} \tilde{M}_k\right)_{k \in \{1,2,\ldots,n\}}}{\Xi \vdash A\ M_1\ M_2 \ldots M_n \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \tilde{M}_1\ \tilde{M}_2 \ldots \tilde{M}_n}$.

By the induction hypothesis on $\mathcal{D}_0$, we have $\{\mathsf{Env} = \Xi\}\ A \overset{\mathcal{E}_0}{\rightsquigarrow_{\mathrm{A}}} \tilde{A}\ \{\mathsf{Env} = \Xi\}$.

For each $k \in \{1, 2, \ldots, n\}$, by the induction hypothesis on $\mathcal{D}_k$, we have $\{\mathsf{Env} = \Xi\}\ M_k \overset{\mathcal{E}_k}{\rightsquigarrow_{\mathrm{M}}} \tilde{M}_k\ \{\mathsf{Env} = \Xi\}$.

Then,

$$\frac{\overset{\mathcal{E}_0}{\{\mathsf{Env} = \Xi\}\ A \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \{\mathsf{Env} = \Xi\}} \quad \left(\overset{\mathcal{E}_k}{\{\mathsf{Env} = \Xi\}\ M_k \rightsquigarrow_{\mathrm{M}} \tilde{M}_k\ \{\mathsf{Env} = \Xi\}}\right)_{k \in \{1,2,\ldots,n\}}}{\{\mathsf{Env} = \Xi\}\ A\ M_1\ M_2 \ldots M_n \rightsquigarrow_{\mathrm{A}} \tilde{A}\ \tilde{M}_1\ \tilde{M}_2 \ldots \tilde{M}_n\ \{\mathsf{Env} = \Xi\}}.$$

3. Assume $\Xi \vdash \overset{\mathcal{D}}{M} \rightsquigarrow_{\mathrm{M}} \tilde{M}$.

   By structural induction on $\mathcal{D}$:

   - Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\mathrm{lookup}\,[x]\,(\Xi) = x : \mathsf{LF}_{\mathsf{term}}} \quad \overset{\mathcal{D}_2}{\mathrm{index}_{\Psi}\,[x]\,(\Xi) = \iota}}{\Xi \vdash x \rightsquigarrow_{\mathrm{M}} \iota}$.

     Then,

     $$\frac{\overset{\mathcal{D}_1}{\mathrm{lookup}\,[x]\,(\Xi) = x : \mathsf{LF}_{\mathsf{term}}} \quad \overset{\mathcal{D}_2}{\mathrm{index}_{\Psi}\,[x]\,(\Xi) = \iota}}{\{\mathsf{Env} = \Xi\}\ x \rightsquigarrow_{\mathrm{M}} \iota\ \{\mathsf{Env} = \Xi\}}.$$

   - Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{\mathrm{lookup}\,[c]\,(\Xi) = \tilde{c} : \mathsf{LF}_{\mathsf{term\ const}}}}{\Xi \vdash c \rightsquigarrow_{\mathrm{M}} \tilde{c}}$.

     Then,

     $$\frac{\overset{\mathcal{D}'}{\mathrm{lookup}\,[c]\,(\Xi) = \tilde{c} : \mathsf{LF}_{\mathsf{term\ const}}}}{\{\mathsf{Env} = \Xi\}\ c \rightsquigarrow_{\mathrm{M}} \tilde{c}\ \{\mathsf{Env} = \Xi\}}.$$

   - Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}'}{\Xi, x : \mathsf{LF}_{\mathsf{term}} \vdash M \rightsquigarrow_{\mathrm{M}} \tilde{M}}}{\Xi \vdash \lambda x.M \rightsquigarrow_{\mathrm{M}} \lambda\ \tilde{M}}$.

     By the induction hypothesis on $\mathcal{D}'$, we have $\{\mathsf{Env} = \Xi, x : \mathsf{LF}_{\mathsf{term}}\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \Xi, x : \mathsf{LF}_{\mathsf{term}}\}$.

     Then,

     $$\frac{\dfrac{\dfrac{\overset{\mathcal{E}'}{\{\mathsf{Env} = \Xi, x : \mathsf{LF}_{\mathsf{term}}\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \Xi, x : \mathsf{LF}_{\mathsf{term}}\}}}{\{\mathsf{Env} = \mathrm{push}_{\Psi}\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi)\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \mathrm{push}_{\Psi}\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi)\}}}{\{\mathsf{Env} = \Xi\}\ \lambda x.M \rightsquigarrow_{\mathrm{M}} \lambda\ \tilde{M}\ \{\mathsf{Env} = \mathrm{pop}_{\Psi}\,[x]\,(\mathrm{push}_{\Psi}\,[x : \mathsf{LF}_{\mathsf{term}}]\,(\Xi))\}}}{\{\mathsf{Env} = \Xi\}\ \lambda x.M \rightsquigarrow_{\mathrm{M}} \lambda\ \tilde{M}\ \{\mathsf{Env} = \Xi\}}.$$

   - Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_0}{\Xi \vdash M \rightsquigarrow_{\mathrm{M}} \tilde{M}} \quad \left(\overset{\mathcal{D}_k}{\Xi \vdash N_k \rightsquigarrow_{\mathrm{M}} \tilde{N}_k}\right)_{k \in \{1,2,\ldots,n\}}}{\Xi \vdash M\ N_1\ N_2 \ldots N_n \rightsquigarrow_{\mathrm{A}} \tilde{M}\ \tilde{N}_1\ \tilde{N}_2 \ldots \tilde{N}_n}$.

     By the induction hypothesis on $\mathcal{D}_0$, we have $\{\mathsf{Env} = \Xi\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \Xi\}$.

     For each $k \in \{1, 2, \ldots, n\}$, by the induction hypothesis on $\mathcal{D}_k$, we have $\{\mathsf{Env} = \Xi\}\ N_k \rightsquigarrow_{\mathrm{M}} \tilde{N}_k\ \{\mathsf{Env} = \Xi\}$.

     Then,

     $$\frac{\overset{\mathcal{E}_0}{\{\mathsf{Env} = \Xi\}\ M \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \{\mathsf{Env} = \Xi\}} \quad \left(\overset{\mathcal{E}_k}{\{\mathsf{Env} = \Xi\}\ N_k \rightsquigarrow_{\mathrm{M}} \tilde{N}_k\ \{\mathsf{Env} = \Xi\}}\right)_{k \in \{1,2,\ldots,n\}}}{\{\mathsf{Env} = \Xi\}\ M\ N_1\ N_2 \ldots N_n \rightsquigarrow_{\mathrm{M}} \tilde{M}\ \tilde{N}_1\ \tilde{N}_2 \ldots \tilde{N}_n\ \{\mathsf{Env} = \Xi\}}.$$

   - Case $\mathcal{D} = \dfrac{\overset{\mathcal{D}_1}{\Xi \vdash M \rightsquigarrow_{\mathrm{M}} \tilde{M}} \quad \overset{\mathcal{D}_2}{\Xi \vdash A \rightsquigarrow_{\mathrm{A}} \tilde{A}}}{\Xi \vdash M : A \rightsquigarrow_{\mathrm{A}} \tilde{M} : \tilde{A}}$.

By the induction hypothesis on $\mathcal{D}_1$, we have $\{\mathsf{Env} = \Xi\}\ M \overset{\mathcal{E}_1}{\rightsquigarrow_{\mathrm{M}}} \tilde{M}\ \{\mathsf{Env} = \Xi\}$.
By the induction hypothesis on $\mathcal{D}_2$, we have $\{\mathsf{Env} = \Xi\}\ A \overset{\mathcal{E}_2}{\rightsquigarrow_{\mathrm{A}}} \tilde{A}\ \{\mathsf{Env} = \Xi\}$.
Then,

$$\frac{\{\mathsf{Env} = \Xi\}\ M \overset{\mathcal{E}_1}{\rightsquigarrow_{\mathrm{M}}} \tilde{M}\ \{\mathsf{Env} = \Xi\} \quad \{\mathsf{Env} = \Xi\}\ A \overset{\mathcal{E}_2}{\rightsquigarrow_{\mathrm{A}}} \tilde{A}\ \{\mathsf{Env} = \Xi\}}{\{\mathsf{Env} = \Xi\}\ M : A \rightsquigarrow_{\mathrm{M}} \tilde{M} : \tilde{A}\ \{\mathsf{Env} = \Xi\}}\ .$$

□

Bibliography

[1] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. POPLMark reloaded: Mechanizing proofs by logical relations. *Journal of Functional Programming*, 29:e19, 2019.

[2] Andreas Abel, Thierry Coquand, and Ulf Norell. Connecting a Logical Framework to a First-Order Logic Prover. In Bernhard Gramlich, editor, *Frontiers of Combining Systems*, pages 285–301, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[3] Andreas Abel and Brigitte Pientka. Explicit Substitutions for Contextual Type Theory. In Karl Crary and Marino Miculan, editors, *Proceedings of the 5th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP)*, volume 34 of *EPTCS*, pages 5–20, 2010.

[4] Ali Afroozeh and Anastasia Izmaylova. *Practical General Top-down Parsers*. PhD thesis, University of Amsterdam, 2019.

[5] Agda Team. Agda 2.6.4, 2023. `https://wiki.portal.chalmers.se/agda/pmwiki.php`.

[6] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson/Addison Wesley, 2nd edition, 2007.

[7] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2):1–89, 2014.

[8] Clemens Ballarin. Interpretation of locales in Isabelle: Theories and proof contexts. In *Proceedings of the 5th International Conference on Mathematical Knowledge Management (MKM)*, pages 31–43. Springer, 2006.

[9] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[10] William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975.

[11] Andrew Cave and Brigitte Pientka. First-class Substitutions in Contextual Type Theory. In *Proceedings of the 8th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP)*, pages 15–24, 2013.

[12] Noam Chomsky. Three models for the description of language. *IRE Transactions on information theory*, 2(3):113–124, 1956.

[13] Nils Anders Danielsson and Ulf Norell. Parsing Mixfix Operators. In *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages (IFL)*, pages 80–99. Springer, 2008.

[14] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[15] Daniel de Rauglaudre. Camlp4 reference manual. `https://caml.inria.fr/pub/docs/manual-camlp4/index.html`, 2003. Version 3.07.

[16] David Delahaye. A tactic language for the system Coq. In *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning (LPAR)*, pages 85–95. Springer, 2000.

[17] Edsger Wybe Dijkstra. Algol 60 Translation: An Algol 60 Translator for the X1 and Making a Translator for Algol 60. *Stichting Mathematisch Centrum. Rekenafdeling*, (MR 34/61), 1961.

[18] Jana Dunfield and Brigitte Pientka. Case Analysis of Higher-Order Data. In *Proceedings of the 3rd International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP)*, volume 228, pages 69–84. Elsevier Science Publishers B. V., 2009.

[19] Jacob Errington, Junyoung Jang, and Brigitte Pientka. Harpoon: Mechanizing Metatheory Interactively: (System Description). In *Proceedings of the 28th International Conference on Automated Deduction (CADE)*, pages 636–648. Springer, 2021.

[20] Amy Felty and Brigitte Pientka. Reasoning with Higher-Order Abstract Syntax and Contexts: A Comparison. In *Proceedings of the 1st International Conference on Interactive Theorem Proving (ITP)*, pages 227–242. Springer, 2010.

[21] Francisco Ferreira, Stefan Monnier, and Brigitte Pientka. Compiling Contextual Objects: Bringing Higher-Order Abstract Syntax to Programmers. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification (PLPV)*, pages 13–24, 2013.

[22] Francisco Ferreira Ruiz. A Compiler for the dependently typed language Beluga. Master's thesis, Université de Montréal, 2012.

[23] Renaud Germain. Implementation of a dependently typed functional programming language. Master's thesis, McGill University, 2010.

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

[25] Graham Hutton and Erik Meijer. Monadic parser combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[26] Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. Practical, general parser combinators. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, page 1–12, New York, NY, USA, 2016. Association for Computing Machinery.

[27] Daan Leijen and Erik Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical Report UU-CS-2001-27, Universiteit Utrecht, 2001.

[28] Pierre Lescanne and Jocelyne Rouyer-Degli. Explicit Substitutions with de Bruijn's Levels. volume 914 of *LNCS*, pages 294–308, 1995.

[29] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Proceedings of the 28th International Conference on Automated Deduction (CADE)*, pages 625–635, Cham, 2021. Springer.

[30] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual Modal Type Theory. *ACM Transactions on Computational Logic (TOCL)*, 9(3):1–49, 2008.

[31] Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A Theory of Name Resolution. In *Proceedings of the 24th European Symposium on Programming Languages and Systems (ESOP)*, pages 205–231. Springer, 2015.

[32] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283 of *LNCS*. Springer Science & Business Media, 2002.

[33] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[34] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. Live Functional Programming with Typed Holes. In *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, volume 3, pages 14:1–14:32, 2019.

[35] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 86–99. Association for Computing Machinery, 2017.

[36] Viktor Palmkvist, Elias Castegren, Philipp Haller, and David Broman. Resolvable ambiguity. *Computing Research Repository*, abs/1911.05672, 2019.

[37] F. Pfenning and C. Elliott. Higher-Order Abstract Syntax. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, page 199–208. Association for Computing Machinery, 1988.

[38] Frank Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE)*, pages 202–206. Springer, 1999.

[39] Brigitte Pientka. A Type-Theoretic Foundation for Programming with Higher-Order Abstract Syntax and First-Class Substitutions. *35th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 371–382, 2008.

[40] Brigitte Pientka. Programming inductive proofs: A new approach based on contextual types. In *Verification, Induction, Termination Analysis*, pages 1–16. Springer, 2010.

[41] Brigitte Pientka. An insider's look at LF type reconstruction: everything you (n)ever wanted to know. *Journal of Functional Programming*, 23(1):1–37, 2013.

[42] Brigitte Pientka and Andrew Cave. Inductive Beluga: Programming Proofs. In *Proceedings of the 25th International Conference on Automated Deduction (CADE)*, page 272–281. Springer, 2015.

[43] Brigitte Pientka and Jana Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *Proceedings of the 5th International Joint Conference on Automated Reasoning (IJCAR)*, pages 15–21. Springer, 2010.

[44] Chuta Sano, Ryan Kavanagh, and Brigitte Pientka. Mechanizing Session-Types using a Structural View: Enforcing Linearity without Linearity. In *Proceedings of the ACM SIGPLAN on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, volume 7. Association for Computing Machinery, 2023.

[45] Joshua B. Smith. OCamllex and OCamlyacc. *Practical OCaml*, pages 193–211, 2007.

[46] The Coq Development Team. The Coq Proof Assistant Reference Manual. `http://coq.inria.fr`, 2023. Version 8.18.0.

[47] Makarius Wenzel. The Isabelle system manual. `https://isabelle.in.tum.de/doc/system.pdf`, 2023.

[48] Makarius Wenzel, Clemens Ballarin, Stefan Berghofer, Jasmin Blanchette, Timothy Bourke, Lukas Bulwahn, Amine Chaieb, Lucas Dixon, Florian Haftmann, Brian Huffman, Lars Hupel, Gerwin Klein, Alexander Krauss, Ondřej Kunčar, Andreas Lochbihler, Tobias Nipkow, Lars Noschinski, David von Oheimb, Larry Paulson, Sebastian Skalberg, Christian Sternagel, and Dmitriy Traytel. The Isabelle/Isar reference manual. `https://isabelle.in.tum.de/doc/isar-ref.pdf`, 2023.

[49] Makarius Wenzel, Stefan Berghofer, Florian Haftmann, and Larry Paulson. The Isabelle/Isar implementation. `https://isabelle.in.tum.de/doc/implementation.pdf`, 2023.